Table of Contents

PROOF

The Parallel ROOT Facility, PROOF, is an extension of ROOT enabling interactive analysis of large sets of ROOT files in parallel on clusters of computers or many-core machines. More generally PROOF can parallelize tasks that can be formulated as a set of independent sub-tasks (embarrassingly or ideally parallel).
The *main design goals* for the PROOF system are:

- **Transparency**: there should be as little difference as possible between a local ROOT based analysis session and a remote parallel PROOF session. Typically analysis macros should work unchanged.
- **Scalability**: the basic architecture should not put any implicit limitations on the number of computers that can be used in parallel.
- **Adaptability**: the system should be able to adapt itself to variations in the remote environment (changing load on the cluster nodes, network interruptions, etc.).

PROOF is primarily meant as an alternative to batch systems for Central Analysis Facilities and departmental work groups (Tier-2's and Tier-3's) in particle physics experiments. However, thanks to a multi-tier architecture allowing multiple levels of masters, it can be easily adapted to a wide range of virtual clusters distributed over geographically separated domains and heterogeneous machines (GRID's).
A side project called PoD (Proof-On-Demand) allows to enable PROOF on any kind of resources, dedicated or shared. PoD automatically provides sandboxing, easier administration and self-servicing, and efficient multi-user scheduling. PoD is the recommended way to install PROOF.
The PROOF technology has also proven to be quite efficient in exploiting all the CPU's provided by many-core processors. A dedicated version of PROOF, PROOF-Lite, provides an out-of-the-box solution to take full advantage of the additional cores available in today desktops or laptops.

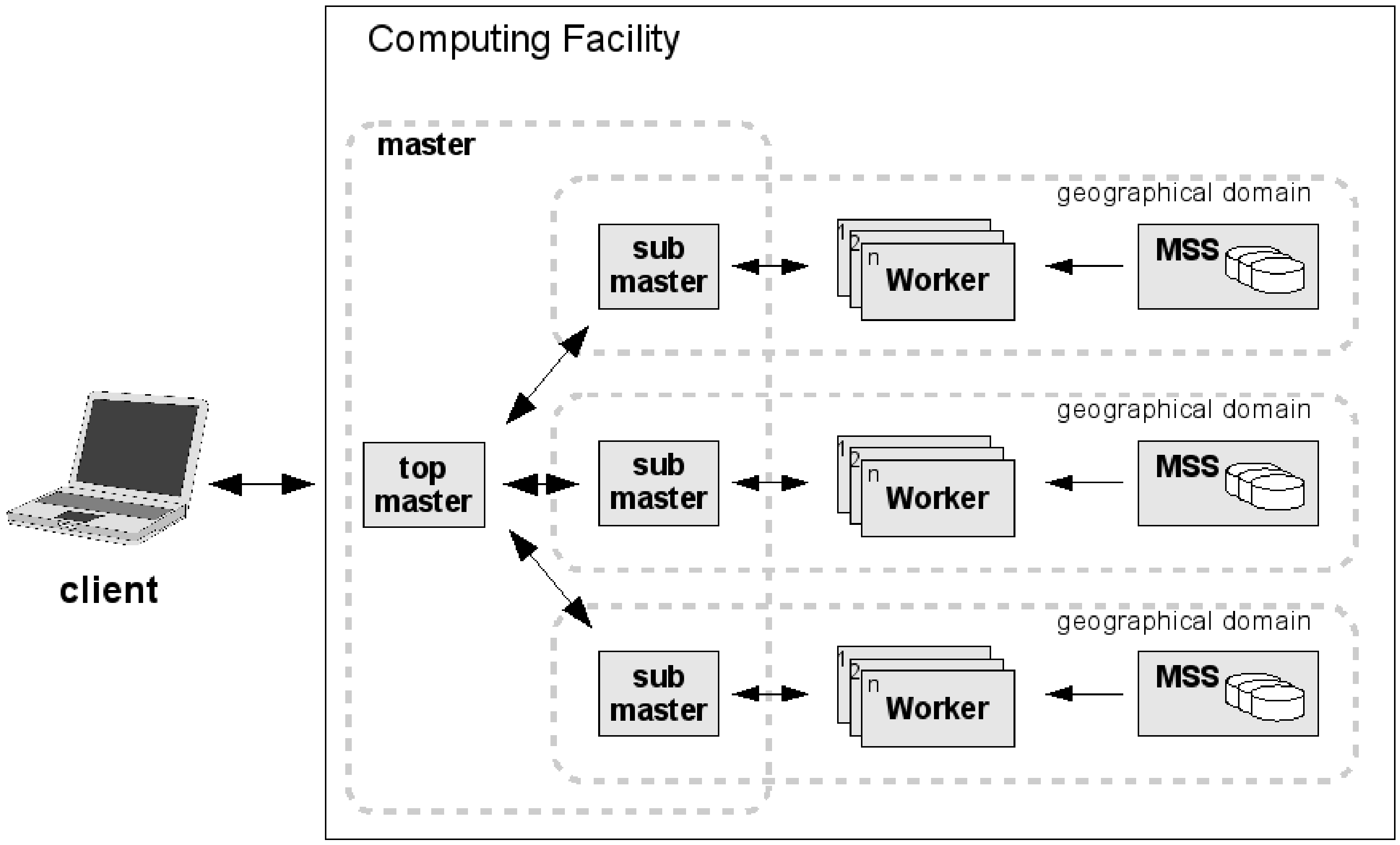
More about PROOF

The PROOF system uses the multi-process approach to parallelism. This approach allows to adapt easily to heterogeneous scenarios and also puts less requirements on users' code. In this section we describe in some more detail soem of the key issues of the PROOF approach.

Multi-Tier Master-Worker Architecture

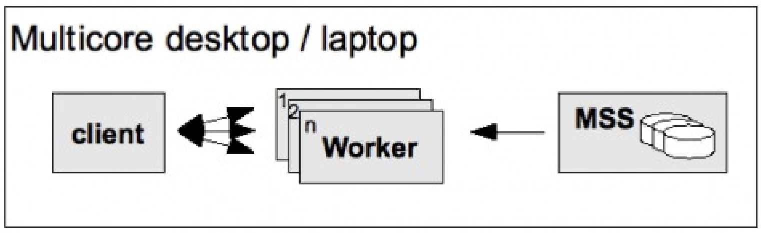
Multi-Tier Master-Worker Architecture

The PROOF system implements a *multi-tier architecture* as shown in the figure:



The *client* is the user that wants to use the resources to perform a task. The *master* is the entry point to the computing facility: it parses the client requests, it distributes the work to the *workers*, it collects and merges the results. The master tier can be multi-layered. This allows, for example, to federate geographically separated clusters by optimizing the access to auxilliary resources, like mass storages. It also allows to distribute the distribution and merging work, which may become the bottle-neck in the case of many workers.

PROOF-Lite, the version of PROOF dedicated to multicore desktops, implements a *two-tier architecture* where the master is merged into the client, the latter being in direct control of the workers:

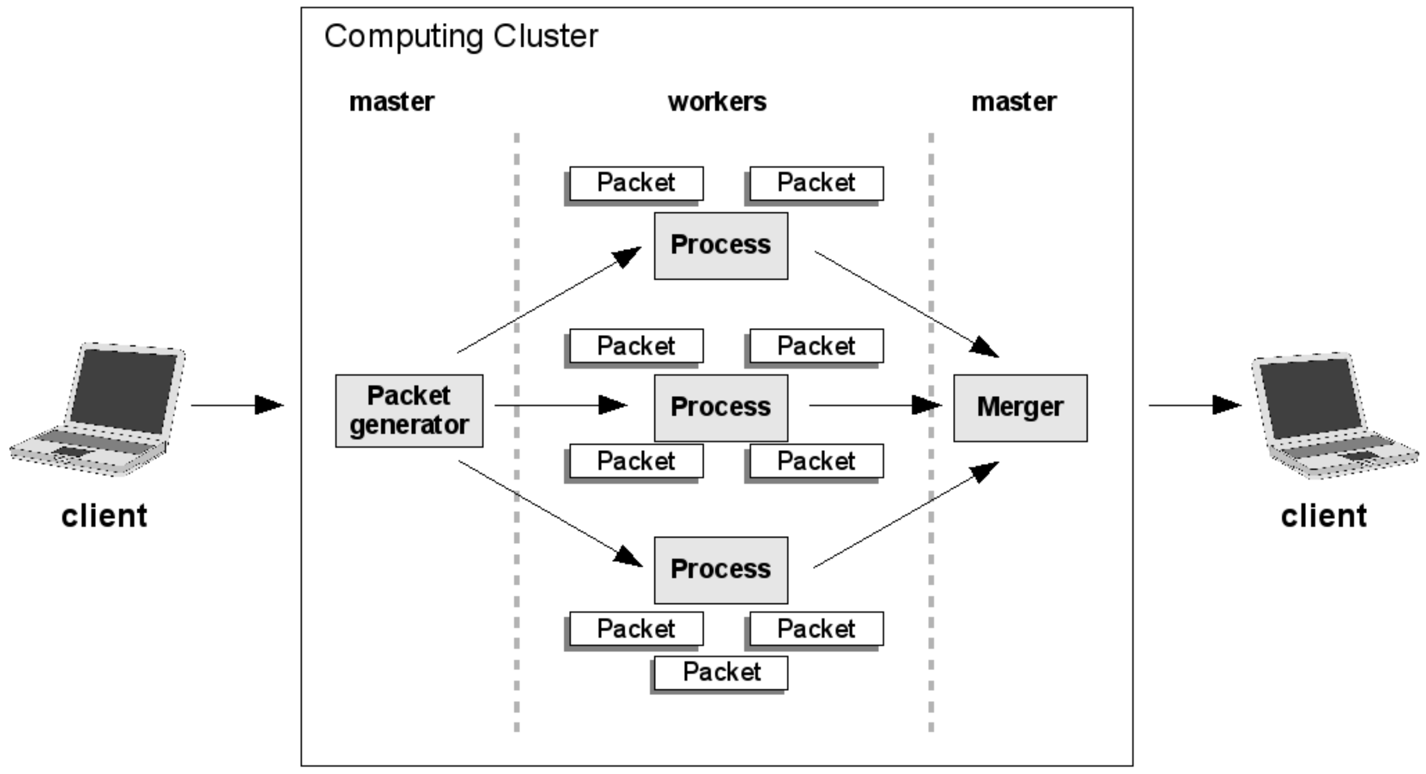


Event Level Parallelism

Event-level parallelism

One of the ideas idea behind PROOF is to minimize the execution time by having all contributing workers terminating their assigned task at the same time. This is achieved by using fine-grained work distribution, where the amount of work assigned to a worker, also called a *packet*, is determined dynamically following the real-time performance of the workers. In principle, the packet can be as small as the basic unit, *the event*.

A schematic view of the execution flow is given in the picture:



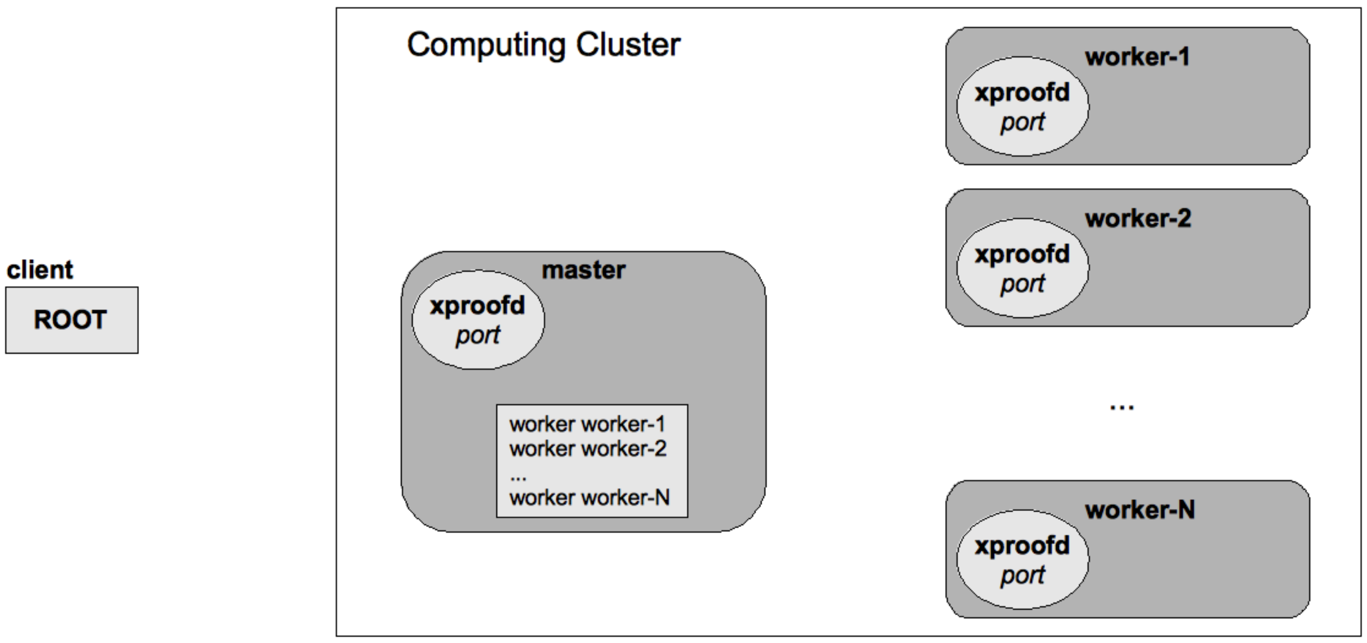
PROOF implementation details

In this section we give more technical details about the implementation of the PROOF system.

To use PROOF a user (*the client*) needs to start a *PROOF session*. Practically this means creating an instance of a the TProof class, which represents (maps) the PROOF session.

The first argument of TProof is an URL representing the network address of the PROOF master node: this is a machine where a xproofd daemon runs and accepts connections on a given port (default 1093). The same, as we will see below, isrequired on the worker nodes. A prerequisite to start PROOF sessions is therefore the availability of a set of machines with xproofd running on it (or xrootd with the XrdProofdProtocol enabled on it). This situation is depicted in Figure 1.

Figure 1. PROOF cluster in idle state; the daemons are ready to accept connections on the specified ports.

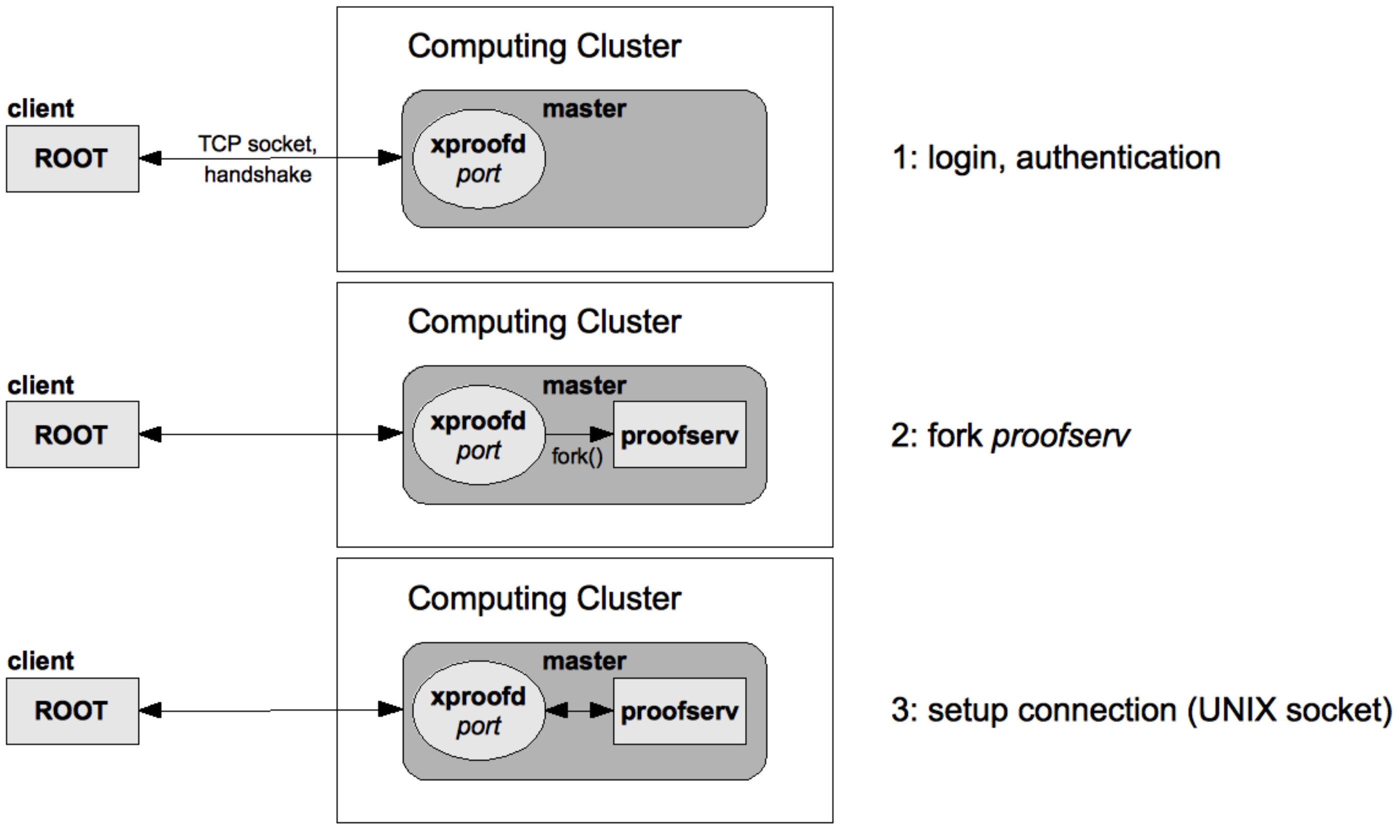


To start the session the user issues the command

```
root[] TProof *p = TProof::Open("master")
```

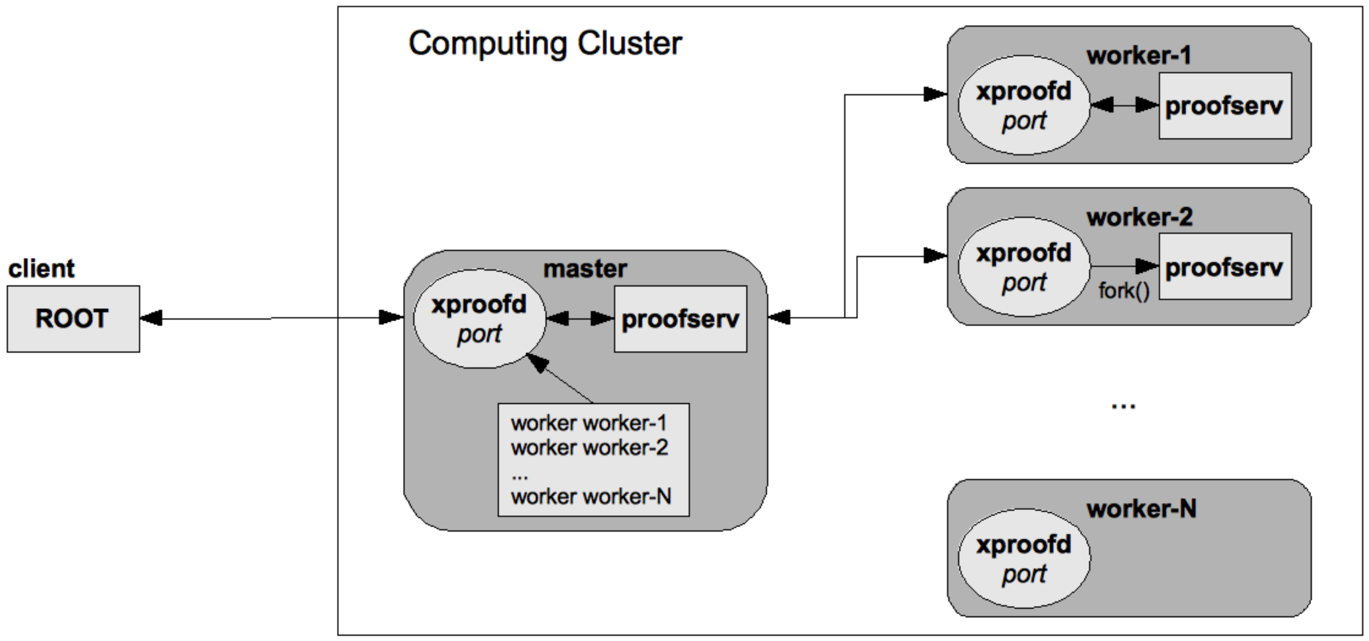
This opens a TCP socket to the master and logins the client to the master xproofd (by running the authentication handshake, if required). This is depicted in the top part of Figure 2. Once logged in, the master process (proofserv in master mode) is forked by xproofd (step 2 in the middle of Figure 2). Finally, when proofserv is up it calls back xproofd to establish the second component of the connection (a UNIX socket) used to communicate with the client.

Figure 2. Setting up the master: 1) login; 2) proofserv fork; 3) UNIX connection.



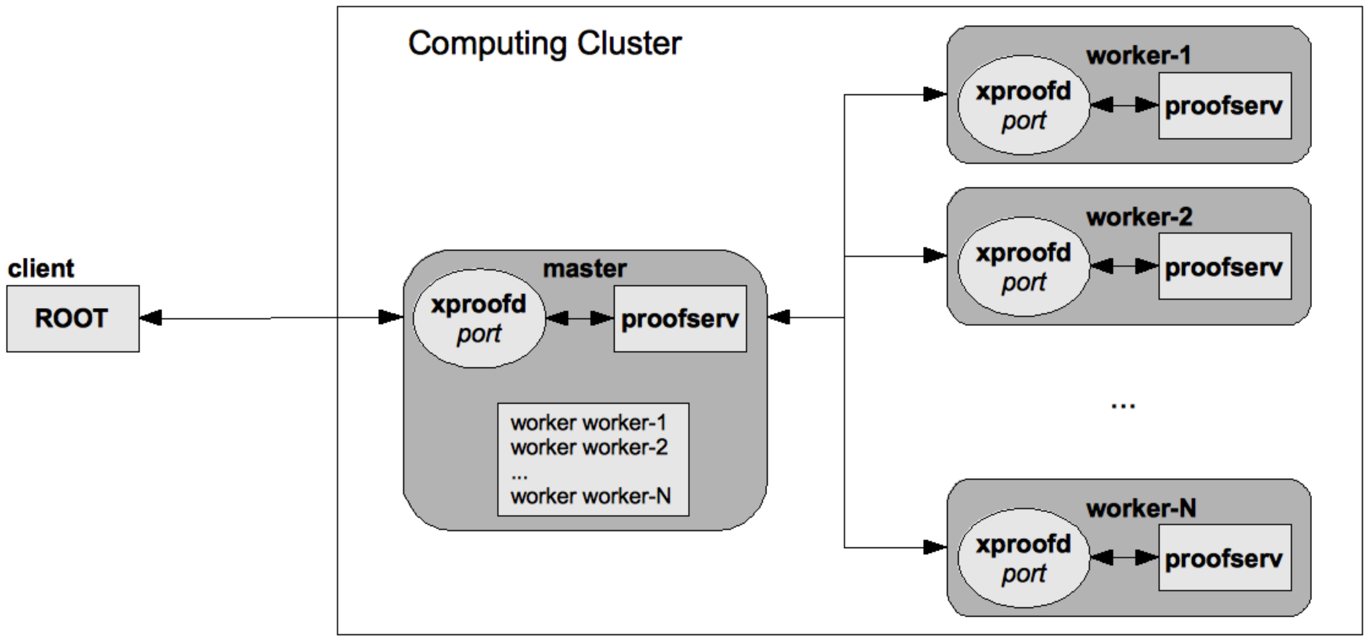
Once the connection with the client is up, the master proofserv proceeds to setup the rest of the session, namely the workers. The list of the workers is obtained from xproofd, where it is determined by the scheduler instance on the base of the configuration file or the dedicated worker configuration file (proof.conf). Then a TProof instance, in master mode, is created; this instance, which, during construction, opens the connections to the worker machines to setup the worker processes. This is done serially and each step is the exact replica of the master process creation (TCP socket, login, fork and UNIX connection setup; see Figure 3).

Figure 3. Setting up the workers: the steps described in Figure 2 are repeated serially for each worker.



Once the whole process is finished the situation looks like the one depicted in Figure 4, with the client connected with the master process and the master process connected to the worker processes.

Figure 4. PROOF session up and ready to process the client commands.



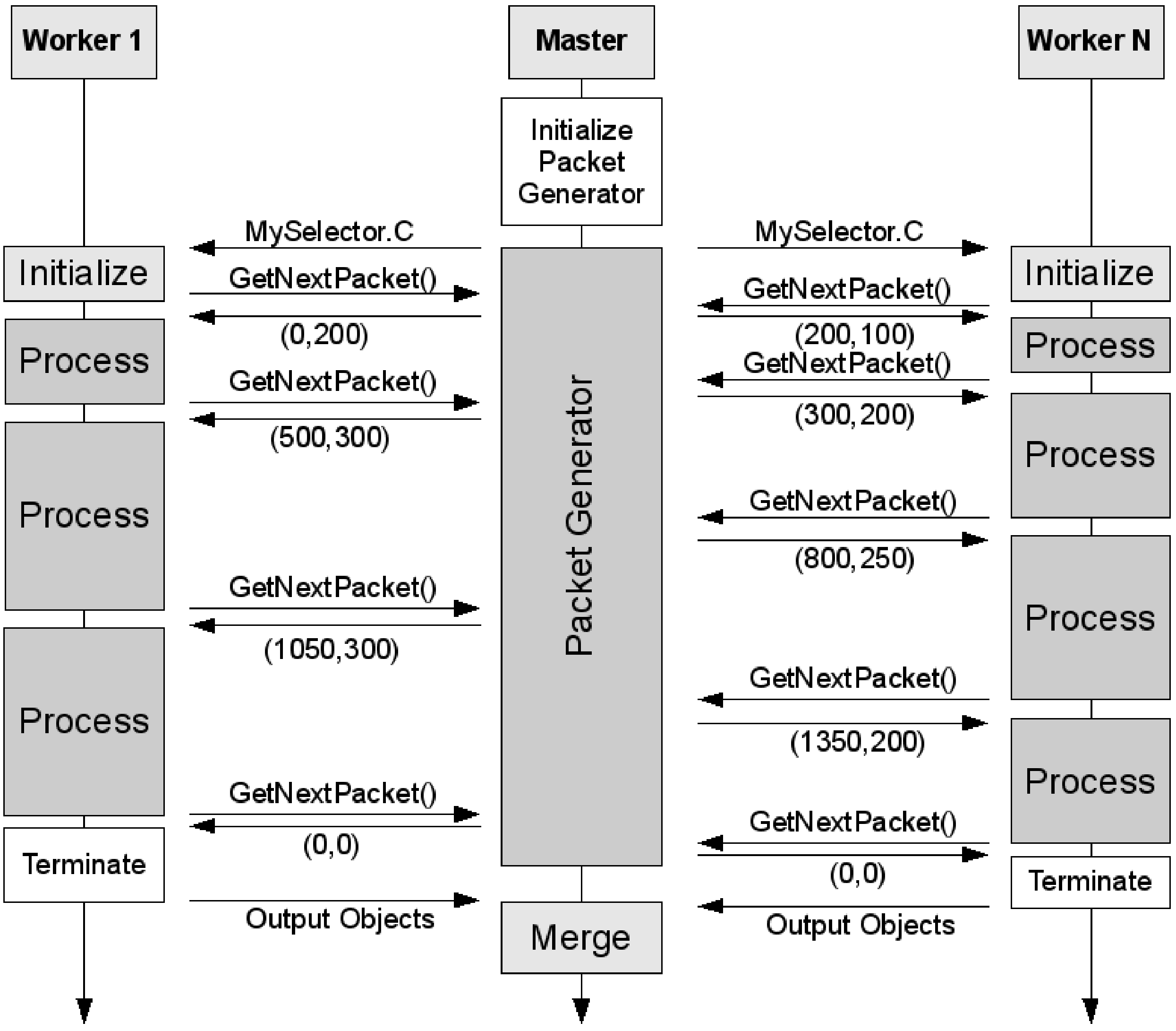
The Packetizer

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 *a packet* - should be processed. An instance of the packetizer is created on the master node. In case of a multi-master configuration, there is one packetizer created for each of the sub-masters. Therefore, when looking at 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: when workers are ready for further processing they ask the packetizer for a next packet.

The Pull Architecture



The different packetizers and their strategies are described in this paper.

The Selector Framework

The Selector framework

To be able to perform event-level parallelism, PROOF needs to be in charge of the *event-loop*, i.e. the execution flow steering the job. This requires that the code to be executed must have a predefined, though flexible structure. In ROOT this is provided by the *Selector framework*, defined by the abstract class TSelector, which defines *three logical steps*:

1. *Begin*, where the job definition (parameters, input data, outputs) is given; executed on the client and the workers;
2. *Process*, where the actual job is done; called for each event, on the workers;
3. *Terminate*, where the results are finally manipulated (fitted, visualized,...); called on the client and the workers.

Process is the part that can be parallelized for the class of problems addressed by PROOF, i.e. those embarassingly or ideally parallel.

Scheduling

The Scheduling options in PROOF

NB: scheduling options described in this section were experiemntal and are depercated. Scheduling in PROOF is provided via Proof-On-Demand(PoD) and the chosen underlying resource management system.

The task of the **PROOF scheduler** is to **assign a set of workers** to each query submitted to the system in accordance with the **scheduling policy** or to enqueue the job (if the policy allows that).

The purpose of scheduling in PROOF is to efficiently utilize resources with the best response time possible on systems ranging from multi-core machines to big clusters.

Scheduling Policies

The scheduling policy determines the algorithm of the scheduler. It's defined by the schedparam directive in the config file and by the mode of scheduling. Below some of the options are described:

Load based scheduling

This option enables the load based scheduler which decides on the number of workers assigned to a user/query based on the current load of the PROOF system, user priority and the scheduling policy.

Priorities and fair-share

The number of workers assigned to a session can also depend on the priority of the user or group. The priorities can be static or calculated dynamically in cooperation with a monitoring system like MonALISA. The interface through which PROOF scheduler reads those priorities, is the group file. Example of the group file is distributed with ROOT in:

```
$ROOTSYS/etc/proof/xpd.groups.sample
```

Currently, to implement a history-based calculation of priorities, one needs a standalone cron job or a demon which will read the history from the monitoring database and edit the group file.

Configuration

This option can be enabled with schedparam directive in the config file. For example:

```
xpd.schedparam selopt:load fraction:0.7
```

Queueing

Even though PROOF is designed for interactive work, the system must foresee the situation of **congestion** where the volume of submitted jobs exceeds the processing capability. Queueing is one way of handling that situation. The alternative is to reject new queries until the system load decreases. PROOF scheduler offers the FIFO queue which can be used in conjunction with the load based scheduling or the static scheduling with a limit on the maximum number of running jobs/sessions set. For details see the schedparam directive.

Configuration

To enable the FIFO queue with the load-based scheduling use the

```
schedparam selopt:load queue:fifo
```

To enable the round robin worker assignment with maximum number of concurrent jobs/sessions set to k and a FIFO queue, use:

```
schedparam selopt:roundrobin queue:fifo mxrun:k
```

Scheduling mode (per job or per session)

Dynamic per-job worker startup (available since ROOT version: 5-21-04)

In order to refine the resource utilization in a multi-user environment, it is advised to use the dynamic per-job scheduling.

In this mode the PROOF session is started only on the master node and workers are assigned when a query is submitted for processing (i.e. when TProof::Process is called).

Configuration

To enable use of this mode add this line to the config file:

```
xpd.putrc Proof.DynamicStartup 1
```

Static per-session scheduling

This is the default option. The workers are assigned when the user starts a session with TProof::Open and are released only after the session is finished.

Current Developments

If you have any questions, feedback or requests for scheduling in different use cases, don't hesitate to contact Jan Iwaszkiewicz. There are new features being developed including preemption and fair-share scheduling (based on usage history).

PROOF Installation

Please note that **we highly recommend** Proof-On-Demand based installations for PROOF.

The purpose of this section is to describe how to enable PROOF on a machine or a cluster of machines. Enabling PROOF on a cluster of machines means to configure and start a dedicated daemon on each machine running as master / worker, hereafter referred to as the *servers*.

Nothing special has to be done (in addition to install and enable ROOT) on the local machine (the client) to be able to start a PROOF-enabled clusters: the relevant plug-in libraries are included in ROOT and will be automatically loaded.

N.B. Starting with ROOT 5.32.00, XRootD is not distributed with ROOT anymore. To build the 'xproofd' executable and the PROOF plugins libProofx, libXrdProofd it is required to enable XRootD as external package. See the dedicated page for detailed instructions.

Recommended way to install PROOF. Virtual Analysis Facility.

The recommended way to install PROOF is to use Proof-On-Demand(PoD). For Cloud-managed resources, an HTCondor resource management system on which to run PoD can be created via the *Virtual Analysis Facility*. Detailed instructions can be found in the PoD and VAF documentation.

PROOF on a multicore desktop / laptop: PROOF-Lite

PROOF-Lite (available starting from ROOT version 5.22/00) is a version of PROOF optimized for multicore desktops and laptops.

Note that PROOF-Lite is not yet available for Windows OS; the porting is under way.

PROOF-Lite does not require any additional configuration. To start a PROOF-Lite session, open a ROOT session and type TProof::Open(""):

```
$ root -l
root [0] TProof::Open("")
+++ Starting PROOF-Lite with 4 workers +++
Opening connections to workers: OK (4 workers)
Setting up worker servers: OK (4 workers)
PROOF set to parallel mode (4 workers)
(class TProof*)0xa84f50
root [1]
```

The default number of workers is the number of cores of the machine. One can force a different number of cores with the parameter workers:

```
$ root -l
root [0] TProof::Open("workers=2")
 +++ Starting PROOF-Lite with 2 workers +++
Opening connections to workers: OK (2 workers)
Setting up worker servers: OK (2 workers)
PROOF set to parallel mode (2 workers)
(class TProof*)0xa85250
root [1]
```

For testing purposes (or if using a ROOT version older than 5.22/00) it is of course still possible to start a standard local PROOF installation.

Standard PROOF installation on a desktop / laptop

In this section we describe how to get a standard (via daemon) PROOF session on the local machine.

The advised way to run PROOF on a desktop is using PROOF-Lite. However, for testing purposes (or if the ROOT version id older than 5.22/00) one may still want to fully enable PROOF on the local machine.

To do that, the only thing needed is a very simple configuration file for xrootd (downloadable from here):

```
### Load the XrdProofd protocol
xrd.protocol xproofd libXrdProofd.so
```

Assuming the this file is located at $MYCONFS/xpd.cf, you just need to start xrootd from the command line

```
xrootd -c $MYCONFS/xpd.cf -b -l /tmp/xpd.log
```

and a PROOF session from your ROOT session:

```
root[0] proof = TProof::Open("localhost")
```

A session with a number of workers equal to the number of CPUs in your machine will be started. The number of workers can be changed using the xpd.localwrks in $MYCONFS/xpd.cf; for example, repeating the exercise after having added

```
xpd.localwrks 4
```

to $MYCONFS/xpd.cf, the PROOF session will have 4 workers, regardless of the number of processors in the machine.

Configuring XPROOFD (or the PROOF plugin for XROOTD)

XrdProofdProtocol is an implementation of the generic XrdProtocol XROOTD protocol class. The class is compiled in a dedicated shared library named *libXrdProofd* and located under $ROOTSYS/lib.

Starting with ROOT version 5.24/00 an executable named *xproofd* is also available under $ROOTSYS/bin. The *xproofd* executable is build out of the same main program used to build the *xrootd* executable. However, it loads by default the *XrdProofdProtocol* instead of the data-serving protocol *XrdXrootdProtocol*. The purpose of this daemon is to simplify the configuration when data-serving is not needed or it is handled differently.

For small standalone tests *xproofd* can be started from the command line in *foreground* mode:

```
$ xproofd -c xpd.cf
```

or in *daemon* mode:

```
$ xproofd -c xpd.cf -b -l /tmp/xpd.log
```

Here xpd.cf is the containing the configuration directives described below (for a very quick test the file is not even needed).

To have *xrootd* loading the protocol the standard SCALLA directive xrd.protocol must be used:

```
### Load the XrdProofdProtocol to serve PROOF sessions
if exec xrootd
xrd.protocol xproofd:1093 libXrdProofd.so
fi
```

The condition on 'exec' allows to use the same configuration file for both xproofd and xrootd. From now on we will refer interchangeably to *xproofd* or *xrootd*, assuming that when *xrootd* is used the xrd.protocol is present.

The text file xpd,cf mentioned above contains the directives governing the behavior of *XrdProofdProtocol*; these are described in detail in the reference guide. See the *quick-setup page* and the more advanced examples (*cluster with pool disks*, *CAF*) for samples of configuration files.

Starting *xproofd* on the command line as shown above may be sufficient from a PROOF system serving a single user. To serve PROOF to multiple users, ensuring the privacy of their sandboxes, the daemon must be started with super-user privileges (needed to be able to log the user into its own sandbox). For this purpose *xproofd* provides the command line option -R followed by the username of an unprivileged user:

It is also possible to force an unprivileged daemon to serve multiple users using the xpd.multiuser directive:

```
### Force the daemon to serve multiple user
### no matter the process privileges
xpd.multiuser 1
```

In such a case the sandboxes and PROOF applications will be owned by the effective user of the daemon; this means, in particular, that the privacy of the sandboxes is not ensured.

In production *xrootd* should be started with a script. Being a multi-port/protocol handler itself it can not be integrated in the (x)inetd schema. Standard service startup scripts work fine. We describe and dissect here the ones used at the CERN CAF under linux SLC4.

The daemon is started by the script /etc/init.d/xrootd . The xrootd related parts here are:

- Definition of the executable to be run and of the path with the *xrootd* plug-in libraries

```
XROOTD=/opt/root/bin/xrootd
XRDLIBS=/opt/root/lib
```

- The start() function sets the environment and starts the daemon in the background:

```
start() {
echo -n $"Starting $prog:"
# Options are specified in /etc/sysconfig/xrootd .
# See $ROOTSYS/etc/daemons/xrootd.sysconfig for an example.
# $XRDUSER *must* be the name of an existing non-privileged user.
export LD_LIBRARY_PATH=$XRDLIBS:$LD_LIBRARY_PATH
daemon $XROOTD -b -l $XRDLOG -R $XRDUSER -c $XRDCF $XRDDEBUG
RETVAL=$?
echo
[ $RETVAL -eq 0 ] touch /var/lock/subsys/xrootd
return $RETVAL
}
```

The variables used by /etc/init.d/xrootd are defined in /etc/sysconfig/xrootd (the following examples are taken from the CAF):

- Location of the *xrootd* config file

```
#
```

```
# Specify here the full path to the configuration file to be used
XRDCF="/afs/cern.ch/user/a/alicecaf/public/conf/xrd-lxb.cf"
```

- Location of the log file location

```
#
# Give here the log file location
# (nb: user XRDUSER must be allowed to write in this directory)
XRDLOG="/var/log/xrootd/xrootd.log"
```

- Definition of the user name under which the daemon as to run

```
#
# Specifiy here the normal assumed as effective owner of the daemon.
XRDUSER="alicecaf"
```

- Switching on debugging

```
#
# Debug switch: leave empty for no debug
# XRDDEBUG="-d"
XRDDEBUG=""
```

- Define the path of an additional environment setting configuration file

```
#
# Additional environment settings; this file is sourced before
# starting the daemon
XRDENVCONFIG="/afs/cern.ch/user/a/alicecaf/public/conf/xrd-userconfig.sh"
```

Defining groups of users

Groups of users can be defined in PROOF via the group file, an example of which can be found at $ROOTSYS/etc/proof/xpd.group.sample. The group file is loaded via the groupfile directive

```
xpd.groupfile /etc/proof/groupfile
```

The group file allows to define the groups and several properties of the groups. This file is currently parsed by the re-nicing mechanism implemented in XrdProofd and by the dataset manager. These applications use different directives which are explained in the dedicated sessions. Here we describe the generic directive for defining groups and assigning members to them, and to define a group property. A group named 'default' is always created and users are assigned by default to it. New groups and their members are defined using the 'group' directive. The syntax is

```
group name user1,user2,user3,...
```

A user can be member of many groups; the first occurrence sets the default group. The directive 'property' defines properties for groups. The syntax is

```
property groupname propertyname nominal_value [effective_value] .
```

If missing, the effective value is set to the nominal value. Starting with ROOT version 5.27/04 (SVN rev 33458) it is possible to include recursively files in the group file via the directive 'include':

```
include /etc/proof/subgroupfile
```

This allows to reduce repetitions when configuring common and ad-hoc parts for different sites.

Defining the set of workers or submasters

The list of available workers or submasters for a PROOF cluster has traditionally been given via a dedicated file typically called proof.conf .

Since version 5.22/00 it is possible to define the workers in the XROOTD configuration file via the xpd.worker directive.

The PROOF configuration file (proof.conf)

The PROOF configuration file is a plane text file containing information about the available nodes and their role.

The file is typically named proof.conf but this is not mandatory. The location of the file is communicated to XrdProof plug-in via the directive xpd.resource, for example

```
xpd.resource static /etc/proof/proof.conf
```

This directive is only parsed by nodes that can be masters (see xpd.role). The parser of the file ignores empty lines or lines starting with a '#'; the latters can be used to enter comments. Valid lines have the format:

```
role host options
```

where:

1. **role** can be one among '*master*', '*submaster*' or '*worker*';
2. **host** is the host name of the node where the worker or the submaster have to be started (for the master line host indicates the entry point of the cluster);
3. **options** are one or more additional *key=value* pairs allowing to set an alternative port (e.g. port=2093) or to assign a mass storage domain to the worker (msd=se12) which is a string used for dataset assignment in multimaster mode (other key=value pairs were originally available but are now obsolete).

The way to start more workers on the same machine is to repeat a valid line. The order the workers are started is the one found in the file, so if more workers have to be started on a set of machines, it is good practice to speedup startup to alternate the lines.

For example, to setup a cluster with 'proofmst' as master and 4 workers each on machines 'proofwrk1' and 'proofwrk2', listening on port 2093, one has to enter

```
# The entry point, i.e. the one to be given to TProof::Open(...)
master proofmst

# The workers, listening on port 2093
worker proofwrk1 port=2093
worker proofwrk2 port=2093
worker proofwrk1 port=2093
worker proofwrk2 port=2093
worker proofwrk1 port=2093
worker proofwrk2 port=2093
worker proofwrk1 port=2093
worker proofwrk2 port=2093
```

To login with a different username on the workers one can enter the wanted login username in front of the hostname, with the usual URL syntax:

```
username@hostname
```

If using authentication, this may require special massaging (case dependent!) to make sure that the relevant credentials are available to authenticate 'username' to the workers.

The xpd.worker directive

Since version 5.22/00 it is possible to defined directly the workers in the XROOTD configuration file using the xpd.worker directive.

The format of this directive is very similar to the one for the valid lines in proof.conf:

```
xpd.worker role host options
```

where {role,host,options} have the same meaning as in proof.conf . The xpd.worker directive supports multiple host specification, allowing to write in condensed way the directives for hosts with similar names; for example, if the machine names are pw12.dom.ain, pw15.dom.ain and pw16.dom.ain, one can write a single line

```
xpd.worker worker pw[12,15-16].dom.ain port=12343
```

instead of three lines. The proof.conf example in proof.conf is equivalent to

```
# The workers, listening on port 2093
xpd.worker worker proofwrk[1,2] port=2093
xpd.worker worker proofwrk[1,2] port=2093
xpd.worker worker proofwrk[1,2] port=2093
xpd.worker worker proofwrk[1,2] port=2093
```

Starting from the development version 5.23/02 (trunk #27276) an additional *key=value* pair, '*repeat=N*', is supported, allowing to further condense the specification of available workers on multi-core machines; this option allows to specify how many time the line or the multi-line has to be repeated. Using 'repeat' the above example becomes just:

```
# The workers, listening on port 2093
xpd.worker worker proofwrk[1,2] port=2093 repeat=4
```

Some useful configuration use-cases

We collect configuration examples for some useful use-cases.

Standard PROOF installation on a cluster of machines

The purpose of this section is to describe how to enable a cluster of machines to run PROOF. Enabling PROOF means to configure and start a dedicated daemon on each machine running as master / worker, hereafter referred to as the *servers*.

The dedicated daemon - which is implemented as a plug-in for the multi-purpose xrootd daemon - processes PROOF-related requests on server nodes, which may come from the client or from a master on behalf of a client. The daemon is running on the PROOF server machines accepting connections on port 1093 (assigned by IAAA). It performs two tasks:

- Authenticates the requests: it checks that the request makes sense and comes from an authorized entity; the strength of the checks depends on the configuration settings;
- Starts a ROOT session and puts the client in connection with it; technically this is obtained by forking a child and execv'ing a proofserv application within it; proofserv instantiates a TProofServ which inherits from TApplication and runs the ROOT event loop.

There is still the possibility to run PROOF using the standalone proofd daemon; however, the development of proofd has been frozen in favour of the xrootd-based solution which provides additional coordination functionality and also the possibility to serve PROOF sessions and files with the same daemon.

Standard PROOF installation on a cluster of machines

The purpose of this section is to describe how to enable a cluster of machines to run PROOF. Enabling PROOF means to configure and start a dedicated daemon on each machine running as master / worker, hereafter referred to as the *servers*.

The dedicated daemon - which is implemented as a plug-in for the multi-purpose xrootd daemon - processes PROOF-related requests on server nodes, which may come from the client or from a master on behalf of a client. The daemon is running on the PROOF server machines accepting connections on port 1093 (assigned by IAAA). It performs two tasks:

- Authenticates the requests: it checks that the request makes sense and comes from an authorized entity; the strength of the checks depends on the configuration settings;
- Starts a ROOT session and puts the client in connection with it; technically this is obtained by forking a child and execv'ing a proofserv application within it; proofserv instantiates a TProofServ which inherits from TApplication and runs the ROOT event loop.

There is still the possibility to run PROOF using the standalone proofd daemon; however, the development of proofd has been frozen in favour of the xrootd-based solution which provides additional coordination functionality and also the possibility to serve PROOF sessions and files with the same daemon.

Configuring a cluster of workers with pool disk on each worker

In this section we describe the configuration of a more advanced and realistic cluster composed by a certain number of worker nodes, each with a certain, non-negligible, amount of disk space to store locally data files. In this case we use xrootd not only to serve PROOF but also to manage the disk pool resulting from the cluster of local disks. For this reason we need to run an *additional daemon* , called cmsd , which interconnects the disk pools and allows to perceive the whole system as a unique disk.

In principle, as the configuration directives for the various component protocols (cmsd, xrootd, xproofd) are different, we would need separated configuration files. However, using the conditional directives we can fit all in one file. The file example2.cf is dissected in the next sections.

Global view of the system

Figure 1 shows a schematic view of the cluster and of the relevant components. The entry points for data and PROOF serving may in general be different; in our example we keep them on the same machine, which will be called the *master* in PROOF terminology, or the *redirector* in data-serving terminology. Separating *master* and *redirector* may be necessary in the case the PROOF load on the master node is large, which is typically the case when the number of concurrent PROOF users is large. In such a case it may be even necessary to reserve more machines to the role of master. It is planned to make his more transparent with automatic load balancing between the masters.

Figure 1. *Schematic view of a cluster with local storage*
proof-cluster.jpg

In the following we will refer to *node00* as to the master/redirector node, and to *node01 ... node0n* as the worker/data-server nodes. In this example the disk space on each machine will be used to store data and to host the sandboxes for PROOF users; for that purpose we need to structure it creating three subdirectories:

- /pool/proofbox, for the sandboxes;
- /pool/data, for the data files.

Finally, we assume that ROOT is installed on /opt/root .

Configuring the data-serving part

We dissect in this section the directives configuring the data-serving part.

- Define the port on which the xrootd service on the redirector will be listening. Here we use the default (1094). This is the place where to change the port if already in use or to configure a different setup, for example for test purposes.

```
#
# XRD port
xrd.port 1094
```

- Define the admin paths for the daemons; the default is under /tmp but this may leads to problems for long running daemons, as the automatic cleanup of temporary space may delete important files.

```
#
# Admin paths
all.adminpath /pool/admin
```

- Open File System section; here we tell the system to use the built-in version of the file-system abstraction and tell the *redirector* that it should redirect any request on files to the *leaves*.

```
xrootd.fslib /opt/root/lib/libXrdOfs.so
if node00
  ofs.forward all
fi
```

- Define the paths exported to clients; by default only /tmp is exported, so any additional path should be declared here.

```
#
```

```
# Export /pool/data
all.export /pool/data
```

- Clustering section: here we tell the system that the manager is node00, listening on port 3121 and that it accept requests only from hosts with name 'node*'.

```
#
# Clustering section
if node00
  all.role manager
else
  all.role server
fi
# Manager location (ignored by managers)
all.manager node00 3121
cms.allow host node*
```

Configuring PROOF

We dissect in this section the directives configuring the PROOF part.

- Load the Xproofd protocol processing requests incoming on the default port 1093 (the specification of the default port is not mandatory; in example2.cf we have indeed omitted it); we use the conditional on the executable name to avoid the olbd from loading the protocol.

```
### Load the XrdProofd protocol:
### using absolute paths (; with the path to the ROOT distribution)
if exec xrootd
xrd.protocol xproofd:1093 /opt/root/lib/libXrdProofd.so
fi
```

- Define the ROOT distribution to use, for example, to load the proofserv executable.

```
### ROOTSYS
xpd.rootsys /opt/root
```

- User sandboxes under /pool/proofbox

```
###
### Working directory for sessions [/proof]
xpd.workdir /pool/proofbox
```

- This defines the resource finder. For the time being the only possibility is a static file with the list of nodes to use, located in this case in /opt/root/etc/proof.conf . Alternative finders are foreseen for the future.

```
###
### Resource finder
### NB: 'if ' not supported for this directive.
# xpd.resource static [] [ucfg: [wmx:] [selopt:]
# "static", i.e. using a config file
#             path alternative config file
#                       [$ROOTSYS/proof/etc/proof.conf]
#         if "yes": enable user private config files at
#                       $HOME/.proof.conf or $HOME/, where
#                        is the second argument to
#                       TProof::Open("","") ["no"]
#          Maximum number of workers to be assigned to user
#                       session [-1, i.e. all]
#        If  != -1, specify the way workers
#                       are chosen:
#                       "roundrobin"  round-robin selection in bunches
#                                     of n(=) workers.
#                                     Example:
#                                     N = 10 (available workers), n = 4:
#                                     1st (session): 1-4, 2nd: 5-8,
#                                     3rd: 9,10,1,2, 4th: 3-6, ...
#                         "random"    random choice (a worker is not
#                                     assigned twice)
xpd.resource static /opt/root/etc/proof.conf
```

- Define the role of the different componets: node00 is the master, all the others workers. Order matters: the last 'xpd.role' superseeds any previous setting.

```
###
### Server role (master, submaster, worker) [default: any]
### Allows to control the cluster structure.
### The following example will set node00 as master, and all
### the others node* as workers
xpd.role worker
if node00
  xpd.role master
fi
```

- Control who is allowed to be master on the worker nodes

```
###
### Master(s) allowed to connect. Directive active only for Worker or
### Submaster session requests. Multiple 'allow' directives can
### be specified. By default all connections are allowed.
xpd.allow node00
```

- Entry point for the data storage: this is communicated back to clients and used as default URL for storage

```
###
### URL and namespace for the local storage if different from defaults.
### By the default it is assumed that the pool space on the cluster is
### accessed via a redirector running at the top master under the common
### namespace /proofpool.
### Any relevant protocol specification should be included here.
xpd.poolurl root://node00
xpd.namespace /pool/proofpool
```

CAF configuration at CERN

PROOF at the CAF

The CAF (*Central or CERN Analisys Facility*) is foreseen to provide the computing power for data analysis for LHC at CERN. Experiments plan to use the facility in different ways. ALICE is strongly willing to exploit it via PROOF.

Setup

The 15 machines have dual-quad-core processors, each with 16 GB RAM and about 2.5 TB of pool disk. They all run the CERN official SLC4 linux. The set of machines has been partitioned into two parts:

dev
: 1 machine reserverd for testing bug fixes and new features;

pro
: 14 machines for *production-style* usage, concurrently used by about 20 ALICE users

One machine (alicecaf.cern.ch or lxfsrd0506.cern.ch) is running in *master* mode, acting also as redirectors, and the remaining in *worker* mode. All the machines are called *lxfsrd**.

Configuration and script files

The relevant files (scripts, configuration) can be downloaded from here:

- the main configuration file;
- the /etc/init.d scripts for xrootd and cmsd;
- the /etc/sysconfig files for xrootd and cmsd;
- the xrd-userconfig.sh requested by xrootd.

Configuring a federated cluster (aka multi-level master)

Configuring a federated cluster (aka multi-level master)

WARNING : page under construction

Contents:

Introduction

In this section we describe how to set up a federate PROOF cluster, i.e. a cluster which results from grouping together machines which may reside on different geographical domains. The example we going to discuss is schematically displayed in Figure 1.

Figure 1. *Schematic view of a multi-level master setup federating two simple PROOF clusters*

The two clusters at *.domain.one* and *.domain.two* are two simple PROOF clusters which can be accessed directly at *master.domain.one* and *master.domain.two* . Both cluster export the */data* directory for data access, and have the user PROOF sandboxes under */data/proofbox*. The goal of the exercise is to setup a third machine, *proofgate.domain* , to act as federator in such a way that the client starting a PROOF session at *proofgate.domain* will have a 6 worker machines in the session.

In this example we federate only the PROOF part. Data serving will be kept separated, with two entry points at *master.domain.one* and *master.domain.two* .

In order to achieve out goal we need two configuration files in addition to the ones needed to setup the two simple PROOF cluster. We start, however, by reviewing the configuration of the simple clusters and then we discuss how to join them together.

Submaster configuration

PROOF configuration files

The PROOF configuration is very simple; see proof.domain.one.conf and proof.domain.two.conf. The only addition is the specification of the 'msd' option ('msd' stands for Mass Storage Domain), a string that is used to instruct the packetizer to partition the files to be processed (more later about this).

XROOTD configuration files

Also the XROOTD configuration files are rather standard: xrd.domain.one.cf and xrd.domain.two.cf . We have made use of the `all.export, all.role, all.manager` new directives to simplify the data serving part. The file assumes that the PROOF configuration files are located under the path defined by the `PROOFCFG` environment variable.

Testing the setup

Starting xrootd and oldb on the two sub-clusters should setup two working PROOF clusters accessible at *master.domain.one* and *master.domain.two* .

Supermaster configuration

PROOF configuration files

The PROOF configuration file is again very simple; we use the keyword 'submaster' to specify the subclusters that we want to federate.

XROOTD configuration files

In the XROOTD configuration file the data serving part is not really configured, as it is not used; however, we change the port used to avoid conflicts with other xrootd related activities. In the PROOF part we just need to specify that the role is 'supermaster' and to give the path to the PROOF config file .

Federating the data serving part

To federate also the data serving part, so that the main entry point for data is root://proofgate.domain, one needs to modify the xrootd configuration files and to start an 'olbd' daemon also on 'proofgate.domain'. The configuration file for proofgate.domain needs to be added a real data serving file as shown in xrd.supermaster-redir.cf. The configuration files for the sub-clusters need to define the *master.domain.one* and *master.domain.two* role as 'supervisor' and to *define as manager olbd the top olbd* , i.e. the one running on proofgate.domain : see xrd-spv.domain.one.cf and xrd-spv.domain.two.cf .

Controlling access

The directives xpd.allowedusers and xpd.allowedgroups can be used to control which users and groups (UNIX or PROOF) are allowed to start PROOF sessions on the cluster. The policy is described in this section.

First the general directive for groups, xpd.allowedgroups, is checked; a user of a specific group (both UNIX or PROOF groups) can be rejected by prefixing a '-'. The group check fails if active (the xpd.allowedgroups directive has entries) and at least one of the two groups (UNIX or PROOF) are explicitly denied with the other not explicitly allowed. The result of the group check is superseeded by any explicit specification in the allowedusers, either positive or negative.

In the following examples, we assume that user 'katy' has UNIX group 'alfa' and PROOF group 'student', and users 'jack' and 'john' have UNIX group 'alfa' and PROOF group 'postdoc'.

1. Users 'katy', 'jack' and 'john' are allowed because part of UNIX group 'alfa' (no 'allowedusers' directive)

```
xpd.allowedgroups alfa
```

2. User 'katy' is allowed because part of PROOF group 'student'; users 'jack' and 'john' are denied because not part of PROOF group 'student' (no 'allowedusers' directive)

```
xpd.allowedgroups student
```

3. User 'katy' is denied because part of PROOF group 'student' which is explicitely denied; users 'jack' and 'john' are allowed becasue part of UNIX group 'alfa' (no 'allowedusers' directive)

```
xpd.allowedgroups alfa,-student
```

4. User 'katy' is allowed because explicitely allowed by the 'allowedusers' directive; user 'jack' is denied because explicitely denied by the 'allowedusers' directive; user 'john' is allowed because part of 'alfa' and not explicitely denied by the 'allowedusers' directive (the allowedgroups directive is in this case ignored for users 'katy' and 'jack').

```
xpd.allowedgroups alfa,-student
xpd.allowedusers katy,-jack
```

NB: The behavior of these directives has been reviewed for ROOT 5.32/00 and the reviewed behavior described ported back into the last patched versions of previous ROOT production versions, starting from 5.28/00f and 5.30/00c .

Enabling query monitoring

---

1. Syslog-based monitoring

Since ROOT version 5.27/04 (SVN revision #33369; the functionality is also available in the special branch 5.26/00-proof) it is possible to configure PROOF in such a way that some information about the activity on the cluster is logged via the *syslog* machinery. A tool to parse and visualize this information is under development.

1.1 Configuration

Enabling monitoring via *syslog* requires two steps: i) instructing the syslog system to redirect the 'local5' logs to a log file dedicated to PROOF monitoring; ii) instructing PROOF to send information to syslog.

1.1.1 Defining the log file

The *syslog* facility used by PROOF is '***local5***' . In order to connect this facility to a local file, first choose a location for the file, e.g. */var/log/xpdmon.log* (in the following we will use this path in the examples), and create the empty file. We have then to instruct the syslog daemon that logs to local5 should go to tye chosen file; to achieve this we have to edit the syslog config file, usually /etc/syslog.conf to add the following line

```
#  /etc/syslog.conf     Configuration file for syslogd.
...
#
# PROOF logging
local5.debug                     /var/log/xpdmon.log
```

We have also to tell syslog not to send *local5* logs to the default file, usually /var/log/messages. For that we need to find in /etc/syslog.conf the line similar to this

```
*.info;mail.none;news.none;authpriv.none  /var/log/messages
```

and change it to look as this

```
*.info;local5,mail.none;news.none;authpriv.none  /var/log/messages
```

To make the changes affective we need to restart the syslog daemon

```
/etc/init.d/syslog restart
```

If a full restart is not desired, forcing a re-initialization should be sufficient: this can be done by sending a SIGHUP signal to the daemon

```
kill -SIGHUP `cat /var/run/syslogd.pid`
```

1.1.2 The ProofServ.LogToSysLog directive

By default, the proofserv application does not log anything to syslog. To enable sending the information one needs to set a ProofServ.LogToSysLog directive . This is usually done via the xpd.putrc directive of the XrdProofd daemon.

```
### Log to syslog
xpd.putrc ProofServ.LogToSysLog
```

The directive defines a 2 character string: the first character is a letter which can take the following values:

a	log all
m	log only master
w	log only workers

The second character is a number indicating the log level:

0	log disabled
1	log start-of-session, main actions (exec, process, dataset handling, cache/package handling, query handling) end-of-session
2	As 1 but log also all the remaining activity on the input socket (this includes handling of some auxilliary messages)
3	Log all proofserv logs (i.e. those usually going to the log file) in addition to those obtained with 2

The typical setting to monitor the cluster usage is 'm1', i.e. log level 1 on the master.

Example of output obtained from running the "eventproc" tutorial example:

```
May  4 10:43:55 pcphsft64 proofm-0[14889]: ganis:proofteam 0 0.000 0.000
May  4 10:43:57 pcphsft64 proofm-0[14889]: ganis:proofteam 1012 0.346 0.000 +event.par
May  4 10:43:58 pcphsft64 proofm-0[14889]: ganis:proofteam 1018 0.949 0.020 6 event 0
May  4 10:43:59 pcphsft64 proofm-0[14889]: ganis:proofteam 1018 0.069 0.000 7 event 0
May  4 10:43:59 pcphsft64 proofm-0[14889]: ganis:proofteam 1018 0.001 0.000 8 0
May  4 10:43:59 pcphsft64 proofm-0[14889]: ganis:proofteam 1012 0.055 0.020 ProofEventProc.C
May  4 10:43:59 pcphsft64 proofm-0[14889]: ganis:proofteam 1012 0.009 0.000 ProofEventProc.h
May  4 10:44:58 pcphsft64 proofm-0[14889]: ganis:proofteam 1015 59.365 0.670 0 1000000 596961398 83.530
May  4 10:45:04 pcphsft64 proofm-0[14889]: ganis:proofteam -1 60.819 0.720
```

1.2 Information posted

We describe in this section the format of the information obtained in the case of level 1. The record hs the following format

```
date nodename proofid[pid]: user:group what realtime cputime auxinfo
```

where

- *date* is the date in the form 'day month time', e.g. 'May 4 11:37:54'
- *nodename* is the name of the node where the information is recorded
- *proofid* is a tag indicating the role of the node, e.g. *proofm-0* for a top master, or *proofw-0.16* for a worker (ordinal 0.16)
- *pid* is the process ID of the proofserv logging the information
- *user* is the user name of the session producing the info
- *group* is the PROOF group of the user producing the info
- *what* is the message type of the requested action
- *realtime* is the real time spent executing the action
- *cputime* is the CPU time spent executing the action
- *auxinfo* is some auxilliary information which may be present depending on what (see below).

A record with what = 0 indicates the beginning of the session (realtime and cputime are 0.000 in this case). A record with what = -1 indocates the end of the session; realtime and cputime are, in this case, the total real and CPU time spent in execution during the whole session, respectively.

Auxilliary information is present in the following cases:

The monitoring information is sent by the master at the end of the query. The quantities posted are shown in the table:

what (name)	what (id)	Format	Description
kMESS_CINT	5	cmd	Command being executed

kPROOF_PROCESS	1015	status entries bytes cputime	*status*: query status *entries*: number of entries processed (or cycles performed) *bytes*: number of bytes read if processing data *cputime*: total CPU time used by the workers
kPROOF_RETRIEVE kPROOF_REMOVE	1032 1034	queryref	Query reference ID
kPROOF_ARCHIVE	1033	queryref path	Query reference ID and archiving path
kPROOF_CHECKFILE	1012	filename	Name of the file being checked
kPROOF_CACHE	1018	type aux	Cache handling sub-action type; the auxilliary info is described in the table below
kPROOF_DATASETS	1042	type aux	Dataset handling sub-action type; the auxilliary info is described in the table below

2. The ProofServ.Monitoring directive

It is possible to configure PROOF in such a way that some information about the queries processed on the cluster are sent to a MonALISA monitoring system or to a SQL database. Starting from 5.29/02 (and 5.28/00c) it is possible to send the information to multiple collectors entering multiple 'ProofServ.Monitoring' directives, separated by a ',' or a '|' or a '\'; for example

```
ProofServ.Monitoring SQL mysql://localhost:3306  proof.proofquerylog,
+ProofServ.Monitoring: MonaLisa 192.168.2.22 ::::
```

To enable posting the monitoring information one has to give instruction about which monitoring system has to be used. This is done via the rootrc directive 'ProofServ.Monitoring'. This directive takes a name and up to 9 'const char *' additional configuration arguments for the relevant TProofMonSender plug-in. The TProofMonSender plug-ins available currently are TProofMonSenderML (for a MonALISA backend) and TProofMonSenderSQL (for SQL backends).

Note that these variables can be defined in the XROOTD configuration file via the 'xpd.putrc' directive .

Note that, for ROOT version up to 5.28/00f and 5.30/01, the directives were used to intialize directly the TMonaLisaWriter and TSQLMonitoringWriter plug-ins.

2.1 Configuration

2.1.1 MonALISA

The directive to load the MonALISA writer has the form:

```
 ProofServ.Monitoring MonaLisa
```

The parameter ***server*** specifies the server to whom to send the monitoring UDP packets. If not specified, the hostname is specified in the environment variable APMON_CONFIG, with the default port.

The parameter ***tag*** is equivalent to the MonaLisa farm name; for the case of process monitoring it can be a process identifier e.g. a PROOF session ID.

The parameter ***id*** is equivalent to the MonaLisa node name; for the case of process monitoring it can be just an identifier to classify the type of jobs e.g. PROOF_PROCESSING. If *id* is not specified, TMonaLisaWriter tries to set it from environment variables in this order: PROOF_JOB_ID, GRID_JOB_ID, LCG_JOB_ID, ALIEN_MASTERJOB_ID, ALIEN_PROC_ID.

The parameter ***subid*** can be used to have finer identifier granularity.

The parameter ***option*** is currently only used as local/global switch: if set as "global" the global gMonitoringWriter is set to the instantiated instance of TMonaLisaWriter, so that a unique monitoring writer is used in the session. In PROOF, use "global" to send fine grained processing information (see below).

The ***sendopts*** switch is described below.

2.1.2 SQL

The directive to load the SQL writer is:

```
 ProofServ.Monitoring SQL  [. [.] [.]] [sendopts:]
```

The parameter ***DBMS*** is the URL used to identify the DB server, in the form

```
://[:][/]
```

The parameter ***user*** is the user-name to be used to connect to the DBMS.

The parameter ***passwd*** is the password associated with the user-name above mentioned.

The fourth parameter indicates the database and the table where to insert the monitoring information. Specifying the tables databases and names is not mandatory. The default for the database is 'proof' and for the tables 'proofquerylog', 'proofquerydsets' and 'proofqueryfiles'. If the *summarytable* and its database *dbs* are specified and the other tables are not, then the database *dbs* is used for all the tables, with default table names.

The ***sendopts*** switch is described below.

2.1.3 The sendopts switch

The *sendopts* string allows to configure the records to be sent and their version (see below); the string contains ':' separated information for each record in the form **[{+,-}]*RecordTag*[*RecordVersion*]**; the '-'('+') prefix disables(enables) the sending of a record (if absent '+' is assumed). Recognized values for the *RecordTag* are given in the table:

RecordTag	Description
S	Query summary
D	Information about processed datasets
F	Information about processed files

For *example*, if sendopts is set to 'S1:D0:-F', the process will post the query summary record, version 1, and the dataset record, version 0, while detailed information about the processed files will not be sent.

The optional *RecordVersion* is an integer number specifying the version of the record. For dataset and files information records, the version constrols the amount of information in the record as described below in the relevant sections. For the query summary record, it is a way to provide the possibility to post information as in previous ROOT versions. The following table gives the correspondence:

Summary Record Version	ROOT Versions
0	
1	5.28/00c -> 5.28/00f, 5.30/00, 5.30/01
2	> 5.28/00f, > 5.30/01

2.2 Information posted

The monitoring information is sent by the master at the end of the query. In this section we give a detailed scription of the records sent under the different configuration options.

2.2.1 MonALISA

2.2.1.1 Query Summary Record

The quantities posted in the ***query summary record*** are shown in the table:

Description	Name	Type	ver 0	ver 1	ver 2
User name	user	XDR_STRING	o	o	o
Group name	group	XDR_STRING	o		
Group name	proofgroup	XDR_STRING		o	o
Starting processing time	begin	XDR_STRING	o	o	o
End processing time	end	XDR_STRING	o	o	o
Total wall time	walltime	XDR_REAL64	o	o	o
Total CPU time	cputime	XDR_REAL64	o	o	o
Number of bytes read	bytesread	XDR_REAL64	o	o	o
Number of events processed	events	XDR_REAL64	o	o	o
Number of workers assigned	workers	XDR_REAL64	o	o	o
Max virtual memory used by workers (kB)	vmemmxw	XDR_REAL64		o	o
Max resident memory used by workers (kB)	rmemmxw	XDR_REAL64		o	o
Max virtual memory used during merging (kB)	vmemmxm	XDR_REAL64		o	o
Max resident memory used during merging (kB)	rmemmxm	XDR_REAL64		o	o
Dataset name	dataset	XDR_STRING		o	
Number of files in the dataset	numfiles	XDR_REAL64		o	o
Number of missing files	missfiles	XDR_REAL64			o
Query exit status	status	XDR_REAL64			o
ROOT versions	rootver	XDR_STRING			o

The entry is tagged with the unique tag "-", e.g. "pcphsft64-1237223578-30248-3" .

The 'vmemmxw' and 'rmemmxw' represent the maximum virtual and resident memory (in kB) used by worker during processing; the 'vmemmxm' and 'rmemmxm' are the maximum values during merging on the master.

For version 1, the user and proof-group names are left in the dataset string only if different from the 'user' and 'proofgroup' of the query; multiple dataset are entered as a comma- separated list; for non-dataset based queries, the name of the class used to define the set of files is entered, e.g. TChain, TDSet or TFileCollection. The string '+++none+++' is used for non-data drive analysis.

The 'status' field contains the exit status of the query (0 = OK; 1 = stopped, 2 = aborted).

2.2.1.2 Dataset Record

The quantities posted in the ***dataset record*** are shown in the table:

Description	Name	Type	ver 0	ver 1
Dataset name	dsn	XDR_STRING	o	o
Query tag	querytag	XDR_STRING	o	o
Starting processing time	querybegin	XDR_STRING		o
Number of files in the dataset	numfiles	XDR_REAL64	o	o
Number of missing files	missfiles	XDR_REAL64	o	o

The entry is tagged with "dataset_", e.g. "dataset_d056756f0" . The 'querytag' gives the link with the query summary record to lookup other quantities, e.g. the user requiring the dataset. The number of files is the number of files registered to this dataset; the number of missing files is the number of the required files not available or that could not be opened at the moment the query was run.

2.2.1.3 Files Record

The quantities posted in the ***files record*** are shown in the table:

Description	Name	Type	ver 0	ver 1
File basename	lfn	XDR_STRING	o	o
Full path (including Url)	path	XDR_STRING	o	o
Starting processing time	querybegin	XDR_STRING		o
Availability status	status	XDR_REAL64	o	o

The entry is tagged with "file_", e.g. "file_e256736a1" . The 'querytag' gives the link with the query summary record to lookup other quantities, e.g. the user requiring the file. The 'status' tells if the file was successfully analysed (value 1) or not (value 0) by the query.

2.2.2 SQL

2.2.2.1 Query Summary Record

The quantities posted in the ***query summary record*** are shown in the table:

Description	Name	Type	ver 0	ver 1	ver 2
User name	user	VARCHAR(32)	o	o	
User name	proofuser	VARCHAR(32)			o
Group name	group	VARCHAR(32)	o		
Group name	proofgroup	VARCHAR(32)		o	o
Starting processing time	begin	DATETIME	o	o	
Starting processing time	querybegin	DATETIME			o
End processing time	end	DATETIME	o	o	
End processing time	queryend	DATETIME			o
Total wall time	walltime	INT	o	o	o
Total CPU time	cputime	FLOAT	o	o	o
Number of bytes read	bytesread	BIGINT	o	o	o
Number of events processed	events	BIGINT	o	o	o
Number of workers assigned	workers	INT	o	o	o
Query unique tag	query	VARCHAR(64)		o	o
Max virtual memory used by workers (kB)	vmemmxw	BIGINT		o	o
Max resident memory used by workers (kB)	rmemmxw	BIGINT		o	o
Max virtual memory used during merging (kB)	vmemmxm	BIGINT		o	o
Max resident memory used during merging (kB)	rmemmxm	BIGINT		o	o
Dataset name	dataset	VARCHAR(512)		o	
# of files requested	numfiles	INT		o	o
# of missing files	missfiles	INT			o
Query exit status	status	INT			o
ROOT version	rootver	VARCHAR(32)			o

The 'vmemmxw' and 'rmemmxw' represent the maximum virtual and resident memory (in kB) used by worker during processing; the 'vmemmxm' and 'rmemmxm' are the maximum values during merging on the master.

For version 1, the user and proof-group names are left in the dataset string only if different from the 'user' and 'proofgroup' of the query. Multiple dataset are entered as a comma- separated list. The dataset string is in any case truncated at 512 max characters. For non-dataset based queries, the name of the class used to define the set of files is entered, e.g. TChain, TDSet or TFileCollection. The string '+++none+++' is used for non-data drive analysis.

The 'status' field contains the exit status of the query (0 = OK; 1 = stopped, 2 = aborted).

The table can created in the following way (any name can be used for the table, provided that the correct value is used in the relevant ProofServ.Monitoring directive) :

Version 2:

```
CREATE TABLE proofquerylog (
  id int(11) NOT NULL auto_increment,
  proofuser varchar(32) NOT NULL,
  proofgroup varchar(32) default NULL,
  querybegin datetime default NULL,
  queryend datetime default NULL,
  walltime int(11) default NULL,
  cputime float default NULL,
  bytesread bigint(20) default NULL,
  events bigint(20) default NULL,
  totevents bigint(20) default NULL,
  workers int(11) default NULL,
  querytag varchar(64) NOT NULL,
  vmemmxw bigint(20) default NULL,
  rmemmxw bigint(20) default NULL,
  vmemmxm bigint(20) default NULL,
  rmemmxm bigint(20) default NULL,
  numfiles int(11) default NULL,
  missfiles int(11) default NULL,
  status int(11) default NULL,
  rootver varchar(32) NOT NULL,
  PRIMARY KEY  (id)
)
```

Version 1:

```
CREATE TABLE proofquerylog (
  id int(11) NOT NULL auto_increment,
  proofuser varchar(32) NOT NULL,
  proofgroup varchar(32) default NULL,
  querybegin datetime default NULL,
  queryend datetime default NULL,
  walltime int(11) default NULL,
  cputime float default NULL,
  bytesread bigint(20) default NULL,
  events bigint(20) default NULL,
  totevents bigint(20) default NULL,
  workers int(11) default NULL,
  querytag varchar(64) NOT NULL,
  vmemmxw bigint(20) default NULL,
  rmemmxw bigint(20) default NULL,
  vmemmxm bigint(20) default NULL,
  rmemmxm bigint(20) default NULL,
  numfiles int(11) default NULL,
  dataset varchar(512) NOT NULL,
  PRIMARY KEY  (id)
)
```

Version 0:

```
CREATE TABLE proofquerylog (
  id int(11) NOT NULL auto_increment,
  proofuser varchar(32) NOT NULL,
  proofgroup varchar(32) default NULL,
  querybegin datetime default NULL,
  queryend datetime default NULL,
  walltime int(11) default NULL,
  cputime float default NULL,
  bytesread bigint(20) default NULL,
  events bigint(20) default NULL,
  totevents bigint(20) default NULL,
  workers int(11) default NULL,
  PRIMARY KEY  (id)
)
```

2.2.2.2 Dataset Record

The quantities posted in the ***dataset record*** are shown in the table:

Description	Name	Type	ver 0	ver 1
Dataset name	dsn	VARCHAR(512)	o	o
Query unique tag	querytag	VARCHAR(64)	o	o
Starting processing time	querybegin	DATETIME		o
# of files requested	numfiles	INT	o	o
# of missing files	missfiles	INT	o	o

The 'querytag' gives the link with the query summary record to lookup other quantities, e.g. the user requiring the dataset. The number of files is the number of files registered to this dataset; the number of missing files is the number of the required files not available or that could not be opened at the moment the query was run.

The table can created in the following way (any name can be used for the table, provided that the correct value is used in the relevant ProofServ.Monitoring directive) :

Version 1:

```
CREATE TABLE proofquerydsets (
   id int(11) NOT NULL auto_increment,
   dsn varchar(512) NOT NULL,
   querytag varchar(64) NOT NULL,
   querybegin datetime default NULL,
   numfiles int(11) default NULL,
   missfiles int(11) default NULL,
   PRIMARY KEY (id),
   KEY ix_querytag (querytag)
)
```

Version 0:

```
CREATE TABLE proofquerydsets (
   id int(11) NOT NULL auto_increment,
   dsn varchar(512) NOT NULL,
   querytag varchar(64) NOT NULL,
   numfiles int(11) default NULL,
   missfiles int(11) default NULL,
   PRIMARY KEY (id),
   KEY ix_querytag (querytag)
)
```

2.2.2.3 Files Record

The quantities posted in the ***files record*** are shown in the table:

Description	Name	Type	ver 0	ver 1
File basename	lfn	VARCHAR(255)	o	o
Full path (including Url)	path	VARCHAR(2048)	o	o
Query unique tag	querytag	VARCHAR(64)	o	o
Starting processing time	querybegin	DATETIME		o
File availability status	status	enum('Ok','Failed')	o	o

The 'querytag' gives the link with the query summary record to lookup other quantities, e.g. the user requiring the file. The 'status' tells if the file was successfully analysed (value 1) or not (value 0) by the query.

The table can created in the following way (any name can be used for the table, provided that the correct value is used in the relevant ProofServ.Monitoring directive):

Version 1:

```
CREATE TABLE proofqueryfiles (
   id int(11) NOT NULL auto_increment,
   lfn varchar(255) NOT NULL,
   path varchar(2048) NOT NULL,
   querytag varchar(64) NOT NULL,
   querybegin datetime default NULL,
   status enum('Ok','Failed') NOT NULL default 'Ok',
   PRIMARY KEY (id),
   KEY ix_querytag (querytag)
)
```

Version 0:

```
CREATE TABLE proofquerydsets (
   id int(11) NOT NULL auto_increment,
   lfn varchar(255) NOT NULL,
   path varchar(2048) NOT NULL,
   querytag varchar(64) NOT NULL,
   status enum('Ok','Failed') NOT NULL default 'Ok',
   PRIMARY KEY (id),
   KEY ix_querytag (querytag)
)
```

2.2.2.4 Configuring the bulk INSERT

The dataset and files records are posted with a bulk INSERT. The maximal size for such an operation is controlled by the *max_allowed_packet* configuration parameter of 'mysqld', which should default to 16 MB . The TSQLMonitoring plug-in honours this by sending off a bulk chunk as soon as it reaches 90% of the maximal value, set by default to 16 MB. The maximal value can be changed with the directive ***SQLMonitoringWriter.MaxBulkSize*** which understands bytes, kilobytes (suffix 'k' or 'K'), megabytes (suffix 'm' or 'M') and gigabytes (suffix 'g' or 'G'). For example, the following

```
xpd.putrc SQLMonitoringWriter.MaxBulkSize 1048576
```

```
xpd.putrc SQLMonitoringWriter.MaxBulkSize 1024k
```

```
xpd.putrc SQLMonitoringWriter.MaxBulkSize 1M
```

are equivalent ways to set the maximal value to 1 MB.

3 Additional monitoring (MonALISA only)

If working with MonALISA, it is possible to send processing progress information to the monitoring server. The following sequence is sent on per worker base:

1) Just before requesting the first packet, an entry with:

Name	Type	Description	Value
status	XDR_STRING	Status flag	STARTED
hostname	XDR_STRING	Worker host name	
subid	XDR_STRING	MonaLisa instance subid	

2) After each call to TSelector::Process, an entry with:

Name	Type	Description	Value
subid	XDR_STRING	MonaLisa instance subid	
events	XDR_REAL64	Number of events processed	
processedbytes	XDR_REAL64	Number of bytes read	
realtime	XDR_REAL64	Real (wall) time from the start	
cputime	XDR_REAL64	CPU time used	
totmem	XDR_REAL64	Total memory used	
rssmem	XDR_REAL64	Resident memory used	
shdmem	XDR_REAL64	Shared memory used	0
events_str	XDR_STRING	String version of 'events'	
processedbytes_str	XDR_STRING	String version of 'processedbytes'	
realtime_str	XDR_STRING	String version of 'realtime'	
cputime_str	XDR_STRING	String version of 'cputime'	
totmem_str	XDR_STRING	String version of 'totmem'	
rssmem_str	XDR_STRING	String version of 'rssmem'	
shdmem_str	XDR_STRING	String version of 'shdmem'	"0"
hostname	XDR_STRING	Worker host name	

3) Just after the last call to TSelector::Process, an entry with:

Name	Type	Description	Value
status	XDR_STRING	Status flag	DONE
hostname	XDR_STRING	Worker host name	
subid	XDR_STRING	MonaLisa instance subid	

To enable the sending of this information the option parameter must be set to "global" in the MonALISA plug-in configuration directive.

Limiting memory usage on master and workers

This section describes how to set a limit on the memory usage on the master and workers.
Two ways are available:

- using the system call *setrlimit* to set soft and/or hard limits handled by the system;
- apply a soft check on the process memory consumption while looping on the entries to process (workers only).

The two sort of limits can be enabled at the same time.

Setting system limits

System limits are set when starting the proofserv application. They are controlled by the following environment variables:

- ROOTPROOFASSOFT
  Soft limit (in *setrlimit* terms) in MBytes on the virtual memory (address space); automatic stack expansion will fail (and generate a SIGSEGV that kills the process; since the value is a long, on machines with a 32-bit long either this limit is at most 2 GiB, or this resource is unlimited.
- ROOTPROOFASHARD
  Hard limit for the the soft limit above; this should be used to avoid users to increase their soft limit to infinite. See documentation of *setrlimit*.

These variables must be set via the xproofd configuration file using the xpd.putenv directive. For examples, the following sets limits to 2GBytes:

```
xpd.putenv ROOTPROOFASSOFT=2047
xpd.putenv ROOTPROOFASHARD=2047
```

Soft limits per event

It is also possible to set soft limits checked by the application (proofserv) itself on by-event bases. These limits can address the virtual and resident memory and are also controlled by the following environment variables:

- PROOF_RESMEMMAX
  Upper limit in kBytes on the resident memory used by the process; the value against which the limit is checked is the one returned by TSystem::GetProcInfo .
- PROOF_VIRTMEMMAX
  Upper limit in kBytes on the virtual memory used by the process; the value against which the limit is checked is the one returned by TSystem::GetProcInfo .

Once the limits are reached processing is terminated gracefully on the worker. Note that this limits will allow to detect regular memory leaks: huge allocations during entry processing (for example due to corrupted number of bytes) will not be protected against by these upper limits, and can only be handled by the system.

Example: the following sets limits to 800 MBytes and 1600 MBytes on resident and virtual memory, respectively:

```
xpd.putenv PROOF_RESMEMMAX=800000
xpd.putenv PROOF_VIRTMEMMAX=1600000
```

Multi-user mode and VO, user mapping

NB: the functionality described in this page is not not yet in the SVN trunk. There is no release date yet, as the addressed use-case has changed in the meanwhile, and a working solution has been found in some cases.

Contents:

Introduction

The multi-user option is provided to allow creating sessions under usernames not recongnized by the system, i.e. not in the password file. Initially this option was mainly meant for testing purposes, facilitating the creation of sessions for different users without having to fully create them. However, a new use-case emerged from the experience of groups running on grids, which can be brought back to the multi-user case: this is the case where a real user exists per VO (Virtual Organization) while all members of such a VO have working areas under the {UiD,GiD} of their own VO. Support for this use-case is available in PROOF starting from version 5.25/0x. In this pages we describe how to use this functionality.

The xpd.multiuser directive

The multi-user mode is OFF by default and the xpd.multiuser directive is used to switch it ON. This directive allows also to define a template for the user working areas. For usernames not known to the system, these working areas will be owned by the effective user of the daemon.

An example of this directive is:

```
### Switch on multi-user.
### Create the user working areas named after the username under
### /users
xpd.multiuser 1 /users/
```

Advanced user mapping

By default, when the multi-user option is enabled, usernames not recognized by the system are mapped to the effective user of the daemon. PROOF provides the possibility to modify this default mapping via the directive xpd.umap; this directive allows to map a target username or VO name (see below) to a different effective user.

An example of this directive is:

```
### Map test user proof34 and VO proofvo to existing user 'alitest'
xpd.umap proof34 alitest
xpd.umap proofvo alitest
```

VO definition

Users are assigned to Virtual Organizations (VO) by the appropriate authorities. The information is typically included in the authentication credentials (e.g. in the X509 certificates). In the case this information is missing, PROOF provides the possibility to define VOs aout of group of users or groups or groups. This can be seen also as a way to simplify the global mapping to an (or to a few) effective user(s).

An example of this directive is:

```
### Define VO 'proofvo' with test users proof01, proof34 and proof87
### and the group 'proofteam'
xpd.vo proofvo proof34,proof87,proof01,g:proofteam
```

Note that any VO information found in the credentials has priority.

Scripts to administrate a cluster

Scripts for cluster administration

Contents:

In this section we describe how to administrate a small cluster with the help of a few scripts which can be found under $ROOTSYS/etc/proof/utils/xpd , hereafter referred to as $XPDUTILS .

Basic assumptions

The nodes in the cluster have to be specified in the file $XPDUTILS/nodes , which may look like this:

```
#
# Nodes in the cluster; to comment out use '# ' (with the space)
master00
worker00 worker01
# ... on more lines, space separated
worker02 worker04 worker05 worker06
```

The circle script: using ssh for distributed actions

If the machines in the cluster are not reachable via a distributed shell or similar utilities, one can use the $XPDUTILS/circle script to automatize the serial execution of a command on the cluster. The usage of the script is the following:

```
Usage:
./circle ssh "[command(s)]"
or
./circle scp "[what]" "[to where]"
or
./circle key
```

Running 'circle key' uploads the user DSA public - taken from ~/.ssh/id_dsa.pub - to the node machines, so that by properly adding your key to the SSH agent with 'ssh-add', any subsequent action on the nodes can be performed without being prompted for the password. Of course, during 'circle key' the password will be asked for each node.

Enabling authentication

Enabling authentication

WARNING : page under construction

Contents:

Introduction

In this section we describe how to enable authentication for a PROOF cluster. By default authentication is switched off and needs to be explicitely enabled in the xrootd configuration file.

Authentication is performed using the security modules delivered with xrootd. A set of dedicated directives (sec.protocol, sec.protbind, ...) are used to configure security services. By default these directives apply to both services, *data-serving* and *PROOF*. However, it is possible to configure differently PROOF authentication by prefixing the 'sec.' directives by 'xpd.' . This is done in the following of this section, so that the sample directives below apply to PROOF only.

Authentication at the different tiers can be configured differently. For example, a system administrator may decide that a client needs to authenticate only to the master, implicity assuming that all workers trust their master. This may be a good assumption for clusters closed from outside, where only the machines but the master are protected by a firewall. However, this may not work if the workers need credentials to access data files: in such a case the client credentials need to be forwarded to workers in a secure way. While it is possible to envisage secure channels to perform this task, usually the easiest way to achieve credential forwarding is to perform authentication at all steps. When all the handshakes are successful the forwarded credentials will be automatically available on all the machines. Credential renewal is another issue that may become relevant at some point. This is not currently implemented.

In the next sections we dissect the settings that allow to achieve credential forwarding in PROOF for the three authentication protocols currently available: password-based, Kerberos, GSI.

All the protocol plug-ins need the general security framework to be loaded, so all the directives below need to be preceded by the following generic directive

```
xpd.seclib libXrdSec.so
```

Password authentication

The password-based authentication module can either use a dedicated password file, a user private file (like $HOME/.rootdpass) or the system password file. The way to do this is detailed elsewhere, and will be only recalled here. Credential forwarding in PROOF requires the server to save the credentials somewhere. Xrootd provides the possibility to save them in memory and to retrieve them when needed. For the password-based protocol this is triggered by passing the parameter '-keepcreds' to the sec.protocol line:

```
xpd.sec.protocol pwd -keepcreds
```

The credentials are saved in memory and passed to proofserv in hexadecimal form as an environment variable. The above directive expects that the dedicated password file $HOME/.xrd/pwdadmin exists. To use a different file - for example /etc/xrd/pwdadmin - the option '-dir' must be used (the file name must always be 'pwdadmin')

```
xpd.sec.protocol pwd -keepcreds -dir:/etc/xrd
```

To enable user-private files for users recognized by the system (i.e. with an entry in /etc/passwd) the option '-upwd' must be used: for example, the following settings to enable the well-known $HOME/.rootdpass files:

```
xpd.sec.protocol pwd -keepcreds -dir:/etc/xrd -upwd:2 -cryptfile:.rootdpass
```

Finally, to enable checking against /etc/passwd (or /etc/shadow for systems using shadowed passwords)

```
xpd.sec.protocol pwd -keepcreds -syspwd
```

The daemon must be able to read those files, which is typically achieved by starting is as superuser.

AFS

For an AFS-enabled cluster there is the possibility to use the AFS API to authenticate a client. ROOT (and xrootd) must be built enabling

AFS, which is done by passing '--enable-afs' to ROOT/configure (the setting is automatically transmitted to xrootd/configure; it requires the *dlevel AFS package* on RH systems which is not installed by default).

The directive to be used is the same as the one used to enable system password files above. In this case, however, there is no need to start the daemon as superuser. The AFS token will be also instantiated inside the TProofServ instances , so that the master and worker sessions will all be able to access files over AFS, if needed.

Kerberos authentication

Full credential forwarding for a Kerberos-enabled system requires the system to run Kerberos V. This is because, while xrootd provides also a kerberos IV plug-in, it is only with kerberos V that support for forwardable tickets has been provided. The client ticket must be forwardable: this is checked by the plug-in on the client side and the client is asked to re-initialize the token if the forwardable bit is not set.

On the server side, the option '-exptkn[:cache_location]' must be used to request the forwarding of the token and to indicate where to cache it so that it can be used by *proofserv*. By default the name of the file is /tmp/krb5cc_. The cache location can contain keywords and/or to be expanded on per-user base. The following is a typical setting:

```
xpd.sec.protocol krb5 -exptkn:/tmp/proof//.creds/krb5cc_  xrd/pcepsft43.cern.ch
```

the cache with the exported credentials of user 'pippo' with uid 3456 will be created at

```
/tmp/proof/pippo/.creds/krb5cc_3456
```

The server automatically exports this path in the variable KRB5CCNAME .

GSI authentication

GSI (Globus Security Infrastructure) authentication uses X509 certificates to perform single sign-on authentication. To enable the creation of a proxy certificate representing the client on the server machine we need to pass the option '-dlgpxy:1'

```
xpd.sec.protocol gsi -dlgpxy:1
```

The dlgpxy option requires clients to sign a proxy certificate to be used by the server on their behalf. The proxy certificate is kept in memory and delivered to proofserv via an environment variable.

It may be more appropriate in some cases to export the proxy into a file. This is done using '-dlgpxy:2' and the switch '-exppxy' to control the location of the file; to be fully effective, the environment variable X509_USER_PROXY needs to point to new location; this can be done using the xpd.putenv directive. For example:

```
xpd.sec.protocol gsi -dlgpxy:2 -exppxy:/tmp/proof//.creds/x509up_u
xpd.putenv X509_USER_PROXY=/tmp/proof//.creds/x509up_u
```

Scheduling configuration options

As mentioned above, PROOF provides several ways to control how resources are used. Load-balancing during a query is provided by the packetizer and it is discussed elsewhere. In this section we focus on the available options to control the number of assigned workers, the number and the priority of running sessions.

Please note that the functionality discussed in this section has been gradually introduced in PROOF, so some of the options are only available in the latest version or in the trunk.

Controlling the number of running sessions

Since the development version 5.23/02 (trunk #23270) it is possible to control the number of sessions running concurrently in a smooth, way by queueing the ones that cannot be started into a FIFO queue.

This functionality is controlled by the directive xpd.schedparam. The queue is enabled by the option 'queue=fifo'. Session running control works better in conjunction with per-query worker assignment. This mode is enabled by the directive

```
xpd.putrc Proof.DynamicStartup 1
```

For load-based worker assignment this mode is mandatory (for the time being, the directive has to be explicitly put also for load-based assignment; this will be automatized in the future). For standard worker assignment (full set or sub-set), the maximum number of sessions running concurrently is defined by the parameter 'mxrun'; for example, the following setting runs only 5 sessions concurrently:

```
xpd.schedparam queue:fifo mxrun:5
```

For load-based worker assignment the number of sessions running is determined by the algorithm which determines the worker assignment. Processing requests which cannot be satisfied are queued and the running mode switched to asynchronous.

Controlling the number of workers assigned to a session

By default all workers defined via proof.conf or xpd.worker are assigned to each session. It is possible to assign only a sub-set of workers to the session by using the option 'wmx' of the xpd.schedparam directive. The way the sub-set is chosen out of the whole set of avalable workers is either round-robin or random.

Controlling the priority of running sessions

The priority of a running session can be controlled at group level either *centrally*, i.e. with priorities propagated by the master to the workers, or *locally*, i.e. with priorities defined on each worker. Groups of users can be defined in PROOF via the group file. The priority of a running session is determined by its nice value; XrdProofd implements a mechanism to adapt the nice value according to the priority of the associated group. The priority of the group is defined by the 'priority' directive in the form of a positive, non-normalized number, e.g.

```
### Priority of group 'tpc' is twice the one of group 'ecal'
property tpc priority 200
property ecal priority 100
```

The effective priorities are then normalized taking into account the active sessions only, so that a session with low priority will always get all the resources when it is the only one active.

The re-nicing mechanism is switched off by default and needs to be explicitly enabled using the schedopt directive. This directive allows to choose between *central* and *local* control and to change the re-nicing range.

The system also allows to read the effective priorities for the groups defined in the group file from a dedicated text file the location of whihc is defined using the 'priorityfile' directive, e.g.

```
### File from where updated effective priorities are read
priorityfile /pool/cpupriorities.txt
```

The syntax of this file is 'group=value', e.g.

```
tpc=10.5
ecal=7.3
```

In central mode this can be used together with a monitoring system - e.g. based on MonALISA - to update dynamically the effective group priorities taking into account, for example, the usage history.

Miscellanea

Miscellaneous issues

Options to proofd

Options to proofd

The daemon proofd accepts options on the command line; the generic syntax is

proofd *args [rootsys-dir]*

Recognized arguments are:

-A *[rootauthrc]*
: tells proofserv to read user's $HOME/.rootauthrc, if any; by default such private file is ignored to allow complete control on the authentication directives to the cluster administrator, via the system.rootauthrc file; if the optional argument *rootauthrc* is given, this file takes the highest priority (private user's file being still read with next-to-highest priority) providing a mean to use non-standard file names for authentication directives.

-b *tcpwindowsize*
: specifies the tcp window size in bytes (e.g. see http://www.psc.edu/networking/perf_tune.html). Default is 65535. Only change default for pipes with a high bandwidth*delay product.

-C *hostcertfile*
: defines a file where to find information for the local host Globus information (see GLOBUS.README for details).

-d *level*
: level of debug info written to syslog

0	=	no debug (default)
1	=	minimum
2	=	medium
3	=	maximum

-D *rootdaemonrc*
: read access rules from file . By default /system.rootdaemonrc is used for access rules; for privately started daemons $HOME/.rootdaemonrc (if present) takes highest priority.

-E
: obsolete; up to v4.00.08 this option was used to force exclusivity of the authentication tokens; with the new approach for authentication tab files this option is dummy.

-f
: do not run in daemon mode, but in the foreground.

-G *gridmapfile*
: defines the location of the gridmap file to be used for globus authentication if different from globus default (/etc/grid-security/gridmap); (re)defines the GRIDMAP environment variable.

-h
: print usage message.

-i
: says we were started by (x)inetd.

-noauth
: do not require client authentication.

-p *port*
: specifies the port on which to listen for incoming connections.

-s *sshd-port*
: specifies the port number for the sshd daemon (deafult is 22).

-S *keytabfile*
: use this keytab file, instead of the default (option only supported when compiled with Kerberos5 support).

-T *tmpdir*
: specifies the directory path to be used for temporary files; default is /usr/tmp.

-w
: do not check /etc/hosts.equiv, $HOME/.rhosts for UsrPwd authentications; by default these files are checked by calling ruserok(...); if this option is specified a password is always required.

rootsys-dir
: directory which must contain bin/proofserv and etc/proof.conf. If not specified, ROOTSYS or built-in path (as specified to ./configure) is tried. (*MUST* be the last argument).

Running PROOF using proofd

Configuring and starting proofd

The proofd daemon can be started on the command line or via the standard service scripts. Several options can be used to fine-tune the behaviour (see the reference guide for the full list)

By default the daemon sends itself in the background (daemon mode) sending the log messages to *syslog*. However, for testing, it may be more practical to have it running in the foreground; and option is provided for that.

Some of the typical use-cases are:
Command line in the background answering on port 5151 (instead of the default 1093)

```
$> proofd -p 5151
```

- Command line in the foreground, answering on the default port, authentication turned off, and a lot of debugging printouts

```
$> proofd -f -noauth -d 3
```

- As init.d service; the script /etc/init.d/proofd must be created (check that the paths for the executable are correct for your installation); the variable PROOFDOPTS must be defined in /etc/sysconfig/proofd.
- Handled by xinetd; the script /etc/xinetd.d/proofd must be created; option -i is MANDATORY in this case.

Configuration Reference Guide

XrdProofdProtocol directives

To load the XrdProofd plugin the following directive must be included in the xrootd config file:

xrd.protocol xproofd *[:port] [path]* libXrdProofd.so

Here *port* is the port chosen for PROOF (default 1093) and *path* is the path containing the libXrdProofd.so plug-in .

The directives governing the behaviour of xproofd all begin with "xpd." . These directives are processed via the dedicated tools available for this purpose in the SCALA package. Therefore the standard SCALLA if-else-fi constructs and other features (e.g variable replacement) are automatically supported. Support for the old-style `'if '` constructs, e.g.

```
  xpd.role worker if lxb*.cern.ch
  xpd.role master if lxp*.cern.ch
```

is deprecated but still kept for backward compatibility.

Unless differently specified, patterns may contain any number of wild cards; the best match is retained (max number of matching chars; if two are equal, the last specified wins).

The following directives are recognized:

- adminreqto
- allow
- allowedusers
- allowedgroups
- bonjour
- clientmgr
- cpcmd
- datadir
- datasetsrc
- exportpath
- filterlibpaths
- groupfile
- image
- intwait
- localwrks
- maxoldlogs
- maxsessions
- multiuser
- namespace
- poolurl
- port
- priority
- proofplugin
- proofservmgr
- putenv
- putrc
- reconnto
- resource
- role
- rootd
- rootdallow
- rootsys
- schedopt
- schedparam
- seclib
- sec.protocol
- shutdown
- sockpathdir
- superusers
- tmp
- trace
- umap
- vo

---

adminreqto

xpd.adminreqto *sec*

Function
: Defines the timeout on requests broadcast to workers; as there are 4 attempts, so the actual timeout is 4 times this number

Default
: 30 secs

Example

```
xpd.adminreqto 15
```

---

allow

xpd.allow *node*

Function
: Define the master node(s) allowed to connect. This directive is ignored by masters. Multiple 'allow' directives can be specified.

Default
: all type of connection allowed.

Example

```
xpd.allow lxb6041.cern.ch
```

---

allowedusers

xpd.allowedusers *[-]usr1,[-]usr2,...*

Function
: Defines a list of users allowed to connect to the cluster. Users can be denied to connect by prefixing the name by '-'. Users explicitely authorized via this directive are allowed to connect regardless of the xpd.allowedgroups settings for their UNIX or PROOF group.

Default
: All users (with an entry in the password file, if not in multiuser mode) are allowed to connect.

Example

```
xpd.allowedusers usr1,usr2,usr3
```

See also the *Controlling Access* section.

---

allowedgroups

xpd.allowedgroups *grp1,grp2,...*

Function

Defines the list of UNIX or PROOF groups allowed to connect to the cluster. Groups can be denied to connect by prefixing the name by '-'. Either group must be explicitly allowed with the other not explicitly denied. Starting from version 5.34/19 UNIX groups can be either primary or secondary.

Default

If this directive is missing all users (with an entry in the password file, if not in multiuser mode) are allowed to connect.

Example

```
xpd.allowedgroups grp1,grp2,grp3
```

See also the *Controlling Access* section.

---

bonjour

xpd.bonjour *mode [servicetype=type] [name=name] [domain=dns] [cores=number]*

Function

Enables and defines the options for the Bonjour/Zeroconf node discovery and publish service

Parameters

mode

Mandatory parameter. Its value can be one of the following:

- `register | publish`: the PROOF node will register on the local multicast responder, publishing itself on the network. This is intended for worker nodes.
- `discover | browse`: the node will discover other PROOF nodes from the multicast resolver and will keep an updated list of them. This works like a subscription and the list will be updated dynamically. Is also possible to establish also a list of static workers using the worker directive at the same time. This is intended for master nodes.
- `both | all`: enable both the publishing and the registering features at the same time.

servicetype

Defines the name of the service type to search for or to publish. It must conform to the Zeroconf guidelines defined in RFC 2782 (there is also a list of common service names). This name is intended to identify the class of the service and all the machines in the cluster must use the same. Also, a categorization of machines can be done using this parameter.

name

The name of the instance. This name identifies in a unique way the name of the instance, it is like a DNS hostname (but it is not, and it will translated to a hostname and IP address by the system).

domain

Custom domain used to publish the services. By default, the local domain is used, but one can use a DNS domain like cern.ch (note that proper DNS server configuration is needed).

cores

In the case of a worker machine, sets the number of workers that will be started in that machine when a PROOF session is opened. This is useful for multicore machines.

Default

servicetype = _proof._tcp. , name = hostname of the machine, domain = local. , cores = 1

Notes

To be correctly honored, any port specification via xpd.port must appear in the configuration file before the xpd.bonjour directive.

Example

```
xpd.bonjour register servicetype=_xproofd._tcp name=worker1 cores=8
```

---

clientmgr

xpd.clientmgr *checkfq:[seconds] activityto:[seconds]*

Function

Defines the parameters controlling the client manager

Parameters

checkfq

defines the frequency in secs of the regular checks performed on client activities

activityto

defines the timeout in secs to consider inactive a client connection

Default

checkfq = 60 secs, activityto = 1200 secs

Example

```
xpd.clientmgr checkfq:120
```

---

cpcmd

xpd.cpcmd *protocol cmd put:{1|0} fmt:[format string]*

Function

Enable or disable a file transfer command to move files in or our of the sandbox.

Parameters

protocol
The protocol string on the base of which this command is chosen; if this string is prefixed with '-' this protocol is disabled
cmd
The actual command to be run for the movement
put

0: this command can only be used to retrieve files (e.g. wget)
1: this command can be used in both directions

fmt
The format to be used to build the command; it must contain two '%s' converters or, respectively, the source and destination paths, and the relevant options
Default

the standard cp is used for internal moves; xrdcp to handle 'root' or 'xrd' protocols; 'wget' ('curl' on MacOsX) to handle 'http' or 'https' protocols (retrieve only)

Examples

```
xpd.cpcmd -http
xpd.cpcmd alien aliencp put:1 fmt:%s %s
```

---

datadir

xpd.datadir *datadir [options]*

Function

Define the root for the user data directories and their write permissions.

Parameters

datadir
Directory under which the user data directories are created, in the form *datadir/group/user/ordinal/session-tag* .
opt
Options controlling where to create the directories and the write permissions. Options are entered as single letters with the following meaning:

g	extend write permissions to the group members;
o *or* a	extend write permissions to all users;
W	assert directories only on worker nodes;
M	assert directories only on non-worker nodes;

If the option string is absent no action or check is done on the directories (they will be created by proofserv).

Default

No action.

Example

```
xpd.datadir /data/proof aW
```

---

datasetsrc

xpd.datasetsrc *type url:URL opt:[options string] rw:{1|0}*

Function

Define the source for dataset meta information. Currently only file-based repositories are suppo

Parameters

type
String defining the type of the repository; currently only file-based repositories are supported; the tag for these is "file".
url
The entry point for the repository; for file-based repositories this is the full path to the directory with the dataset information; for read-only repositories the directory can also be remote; files are handled via the protocol-driven TFile and TSystem ROOT interfaces.
opt
Options to be passed to the dataset manager instance. See the dataset manager description.
rw

0: the repository is read-only [default]
1: the repository can be modified

Examples

```
xpd.datasetsrc file url:root://alicecaf.cern.ch:11094//dataset-xpd opt:-Ar:-Av: rw:0
xpd.datasetsrc file url:/data2/dataset opt:Av:As:
```

---

exportpath

xpd.exportpath *path1[,path2] [path3[,path4[,...]]]*

Function
Defines a list of comma- or black- separated paths which can be browsed by all users using the admin functionality (ls, more, ...); many of these directives can be specified.
Default
none
Example

```
xpd.exportpath /tmp,/data2 /pool/proofbox
```

---

filterlibpaths

xpd.filterlibpaths 1|0 *[path1[,path2] [path3[,path4[,...]]]]*

Function
: Allows to remove paths from the library path (LD_LIBRARY_PATH or equivalent) passed to proofserv. The first switch refers to the ROOT library path, i.e. $ROOTSYS/lib; the rest is the list of additional paths to be removed.

Default
: As of ROOT 5.30/03, ROOT 5.28/00h, 5.32/00, the library path passed to proofserv is exactly the same as the one seen by xproofd. In previous versions the ROOT library path was removed.

Example

```
xpd.filterlibpaths 1
```

---

groupfile

xpd.groupfile *path*

Function
: Defines the path to the file containing the definition of the groups. For an example of the file see $ROOTSYS/etc/proof/xpd.groups.sample.

Default
: none

Example

```
xpd.groupfile $ROOTSYS/etc/proof/xpd.groups
```

---

image

xpd.image *image*

Function
: Define the image name for this server

Default
: node fully-qualified domain name

Example

```
xpd.image nfs
```

---

intwait

xpd.intwait *sec*

Function
: Defines the timeout for waits internal to xproofd (in seconds). Increasing this to O(min) is needed for debugging operations.

Default
: 5 seconds

Example

```
xpd.intwait 500
```

---

localwrks

xpd.localwrks *nwrks*

Function
: Defines the number of workers for local sessions.

Default
: the number of CPUs in the local machine

Example

```
xpd.localwrks 2
```

---

maxoldlogs

xpd.maxoldlogs *max*

Function
: Define maximum number of old PROOF sessions for which the working directory is kept with all the relevant files in (logs, env, ...); non-positive values mean no limit

Default
: 10

Example

```
xpd.maxoldlogs 50
```

---

maxsessions

xpd.maxsessions *max*

Function
: Defines the maximum number of concurrent PROOF sessions that a client can have on the cluster; -1 indicates no limit.

Default
: -1

Example

```
xpd.maxsessions 1
```

---

multiuser

xpd.multiuser *opt [subWORKdir]*

Function
Enable/disable multi-user mode.
Parameters

opt
0 OFF: only the effective user of the daemon can start a PROOF session;
1 ON: all connecting users are served (if allowed according to xpd.allowedusers - see below);

subWORKdir
Template for the user working areas; supported keywords:
the working directory (see xpd.workdir)
the effective owner of the daemon
the target username
the initial of the target username
Default
The default for this switch is OFF; when switched on, the default template for the sub-directories is //user//
Example
```
xpd.multiuser 1
```
Notes
The user areas (sandboxes) and the *proofserv* processes will be owned by the reference effective user (the owner of the daemon or the effective user to which the username has been mapped - see xpd.umap); in particular, the privacy of the user areas is not ensured.

-

-

namespace

xpd.namespace *name*

-

Function
namespace for the local storage if different from defaults.
Default
by the default it is assumed that the pool space on the cluster is accessed under the common namespace /proofpool.
Example
```
xpd.namespace /store
```

-

-

poolurl

xpd.poolurl *pool*

-

Function
URL for the local storage if different from defaults.
Default
by the default it is assumed that the pool space on the cluster is accessed via a redirector running at the top master.
Example
```
xpd.poolurl lxb0105.cern.ch
```

-

-

-

port

xpd.port *port*

Function
: Define the port where the daemon is listening.

Default
: The default port for PROOF is 1093.

Example
: `xpd.port 11093`

---

priority

xpd.priority *delta-priority [if auser]*

Function
: Modify the priority of sessions belonging to *auser* by *delta-priority* ; if *auser* is missing, apply the change to all sessions. This directive requires special privileges, so it may be ineffective if the latters are missing

Default
: No change in priority

Example
: `xpd.priority 4`

---

proofplugin

xpd.proofplugin *prefix*

Function
: Specify the prefix for the TProof implementation plug-in to be used on the master. The plug-in is loaded by the ROOT plugin manager. This directive is needed, for example, when working with Condor.

Default
: none

Example
: `xpd.proofplugin condor:`

---

proofservmgr

xpd.proofservmgr *[checkfq:sec] [termto:sec] [verifyto:sec] [recoverto:sec] [checklost:0|1] [usefork:0|1]*

Function
: Defines the parameters controlling the PROOF sessions manager

Parameters

checkfq
: defines the frequency in secs of the regular checks performed on sessions

termto
: defines the time in secs given to sessions to terminate gently

verifyto

defines the time in secs to proof that they are still alive by touching their admin path

recoverto

defines the time in secs given to a session to recover its connection after being asked to do so

checklost

toggles on/off the check for orpheline proofserv processes; the serach is based on the process name; default is on, but when running with applications like valgrind which may modify the process string, it may be useful to have it off.

usefork

toggles the usage of 'fork' to start a *proofserv* process; if set to 0, *proofserv* is launched using 'system', as in PROOF-Lite.

Default

checkfq = 60 secs, termto = checkfq - 10 secs, verifyto = 3 * checkfq , recoverto = 10 secs, checklost:1, usefork:1

Example

```
xpd.proofservmgr checkfq:120 verifyto:120 usefork:0
```

---

putenv

xpd.putenv *[u:usr1,usr2,...] [g:gr1,gr2,...] [s:[svnmi][-][svnmx]] [v:[versmi][-][versmx]] variable*

Function

Set additional environment variable for 'proofserv'. This is useful, for instance, to set client-side security options.

Parameters

usr1,usr2,...

User(s) to which this setting applies; can contain the wild card '*'

grp1,grp2,...

Group(s) to which this setting applies; can contain the wild card '*'

svnmi (svnmx)

minimum (maximum) SVN revision number for which this setting applies

versmi (versmx)

minimum (maximum) ROOT version numbers for which this setting applies

variable

Variable to be set; it can contain the following place holders which will be replaced before launching 'proofserv':

	replaced with *WORKdir* (see above)
	replaced with the machine host name
	replaced with the user $HOME dir (as returned by *getpwuid*)
	replaced with the user's username
	replaced with the users's PROOF group
	replaced with the users's UNIX user id
	replaced with the users's UNIX group id

Notes

Support for selective setting has been introduced in ROOT version 5.28/00 .

Default

none

Example

```
xpd.putenv  MYENV=//.creds
xpd.putenv  g:z2 PROOF_VIRTMEMMAX=1000
```

---

putrc

xpd.putrc *[u:usr1,usr2,...] [g:gr1,gr2,...] [s:[svnmi][-][svnmx]] [v:[versmi][-][versmx]] variable*

_

Function
Set an additional rootrc-like variable for 'proofserv'. This allows to have full control on the parameters needed by 'proofserv' from one configuration file.
Parameters
usr1,usr2,...
User(s) to which this setting applies; can contain the wild card '*'
grp1,grp2,...
Group(s) to which this setting applies; can contain the wild card '*'
svnmi (svnmx)
minimum (maximum) SVN revision number for which this setting applies
versmi (versmx)
minimum (maximum) ROOT version numbers for which this setting applies
variable
variable to be set in the form: . :
(there can be a space between the colon after the name and the value).
Notes
Support for selective setting has been introduced in ROOT version 5.28/00 .
Default
none
Example

```
xpd.putrc Packetizer.MaxWorkersPerNode: 8
xpd.putrc v:52706- Proof.SchemaEvolution: 0
```

_

_

reconnto

xpd.reconnto *sec*

_

Function
Defines the time in secs given to clients to reconnect after a xrootd restart. This directive is used both by the client and session managers.
Default
300 secs
Example

```
xpd.reconnto 120
```

_

_

resource

xpd.resource *source [cfg-file] [ucfg:ucfg-opt] [wmx:max-wrks] [selopt:mode]*

_

Function
Define the way the composition of the PROOF sessions for a given user is determined
Parameters

source
source of information; only *static* (from a static configuration file) enabled for the time being
cfg-file
location of the static configuration file; on-the-fly changes to this file are automatically detected (no need to restart the daemon). Default if $ROOTSYS/etc/proof.conf .
ucfg-opt

enables ("yes") or disables ("no") user private configuration files. If enabled, are looked for at *$HOME/.proof.conf* or *$HOME/.usr_cfg* , where *usr_cfg* is the second argument to `TProof::Open("master","usr_cfg"` ; by default user private files are disabled.

mode

specify the way workers are chosen (for positive *max-workers*); options are:
"roundrobin" : round-robin selection in bunches of n (= *max-workers*) workers. Example: N = 10 (available workers), n = 4: 1st (session): 1-4, 2nd: 5-8, 3rd: 9,10,1,2, 4th: 3-6, ...
"random" : random choice (a worker is not assigned twice)

Default

```
xpd.resource static $ROOTSYS/etc/proof.conf
```

Example

```
xpd.resource static /opt/proof/proof.test.conf wmx:2 selopt:random
```

-

-

role

xpd.role *role*

-

Function

Define the role of a given server (master, submaster, worker, any). Allows to control the cluster structure.

Default

the server can assume any role.

Example

# lxb*cern.ch machines are all workers ...

```
if lxb*.cern.ch
    xpd.role worker
fi
```

# ... except lxb6041.cern.ch which can also be anything.

```
if lxb6041.cern.ch
    xpd.role any
fi
```

-

-

rootd

xpd.rootd deny *[rootsys:rootsystag] [path:abs-path/] [mode:ro|rw] [auth:none|full] [other_rootd_args]*

-

WARNING

This directive is now deprecated if you are using XRootD v >= 4 : use 'xpd.xrootd' instead

Function

Controls file-serving via the 'rootd' on the same port. This is useful to access the files produced by the workers or residing on the workers when there is no distributed file system and an independent file-serving system can not be added. This works by identifying file-access requests and forking a 'rootd' daemon to serve them. The port is therefore the same used for PROOF. The protocol to be used to trigger this service is 'rootd://'. When this service is enabled, the env variable LOCALDATASERVER is by default automatically set to 'rootd://:' so that files handled by TProofOutputFile have the correct URL. The service is enabled by default.

Parameters

deny

disables the service; alternative names: *off*, dis*able*.

rootsys

defines the tag of the valid ROOT distribution from which 'rootd' should be taken. tags are those defined via ' xpd.rootsys'. Default is the distribution of the running 'xproofd'.

path

defines the absolute path to the 'rootd' daemon to be used. Default is the same of the running 'xproofd'.

mode

defines access mode for 'rootd'; possible values are 'ro' for 'read-only' (default) and 'rw' for 'read-write'; the latter should be used only if authentication is used.

auth
: controls the authentication for 'rootd'; possible values are 'none' (default) and 'full' to use the settings defined by the default access control file (etc/system.rootdaemonrc).

other_rootd_args
: any other argument to be passed to 'rootd'.

Example

```
xpd.rootd allow
xpd.rootd allow mode:rw auth:full
```

---

rootdallow

xpd.rootdallow *host1,host2 host3 ...*

WARNING
: This directive is now deprecated if you are using XRootD v >= 4 : use 'xpd.xrootd' instead

Function
: Defines the lists of hosts allowed to access files via the 'rootd://' protocol (see the directive xpd.rootd). Host names are separated by a comma or by a space. Names can contains the wild card '*'. Multiple lines can be specified.

Example

```
xpd.rootdallow lxbuild*.cern.ch
xpd.rootdallow lxplus102.cern.ch,lxplus103 lxbatch118
```

---

rootsys

xpd.rootsys *ROOTdir [tag]*

Function
: Defines a ROOT distribution that can be used for PROOF sessions. The binary for the proofserv application is searched for under *ROOTdir/bin* and ROOTSYS is set to *ROOTdir* for proofserv. The administrator should define one of these directives per version/distribution supported. The first directive defines the ROOT version to be used by default. Specifying a tag is optional: if missing, the ROOT version tag - e.g. 5.15/03 - is taken (however, the tag must be unique, the first occurence is retained). If none of these directives are specified, $ROOTSYS is used as default ROOT version; if $ROOTSYS is not defined, xrootd will fail initialization.

Example

```
xpd.rootsys /opt/root_v5-13-06  pro
xpd.rootsys /opt/root_v5-13-06_dbg  pro_dbg
```

---

schedopt

xpd.schedopt *opt [min:val] [max:val]*

Function
: control the renicing of sessions according to the priorities defined in the group file

Parameters

opt

controlling option; can have two values:

central
: group priorities are read by the master and propagated to the workers

local
: each worker reads priorities from a local file

min
: defines the minimum nice value for sessions

max
: defines the maximum nice value for sessions

Default
: renicing is off by default; when on, the default values for min is 1 and for max 20

Example
: `xpd.schedopt central min:-10`

---

schedparam

xpd.schedparam *[scheduler] [key1:par1] [key2:par2] ...*

Function

Define the *scheduler* to be used and its configuration parameters; a *scheduler* is an implementation of the XrdProofSched interface; each scheduler has its own {key:parameter} pairs; those governing the behaviour of the *default* scheduler are described below. $ *{key:parameter}* pairs valid for the *default* scheduler:

[wmx:mxw]
: defines the maximum number of workers to be assigned to user session; the default is -1, i.e. all possible workers as defined in the configuration file.

[mxsess:mxs]
: defines the maximum number of sessions which can be started on each node; the default is -1, i.e. no limitation. It can be also used with the load-based scheduling. For load-based scheduling, setting this to a positive value just sets the minforquery parameter to 0, so that there is no guarantee to get a set of workers.

[mxrun:mxr]
: defines the maximum number of sessions which can be run concurrently. This is active only for non-load-based assignment and requires 'queue=fifo' to be set. For load-based scheduling the number of concurrently running sessions is determined dynamically by the algorithm used for the worker assignment.

[queue:fifo]
: enables the FIFO queue when controlling the number of sessions which can be run concurrently.

[selopt:mode]
: specify the way workers are chosen (for positive *max-workers*); options are:
"roundrobin" : round-robin selection in bunches of n (= *max-workers*) workers. Example: N = 10 (available workers), n = 4: 1st (session): 1-4, 2nd: 5-8, 3rd: 9,10,1,2, 4th: 3-6, ...
"random" : random choice (a worker is not assigned twice)
"load" : choice taking into account the load, defined as the number of sessions per node; the behaviour is steered by the parameters described below.

[fraction:f]
: defines the fraction of free processors assigned to a session; default is 0.5; *applies only to mode 'load'* .

[optnwrks:optw]
: defines the optimal number of workers per node; this is used to estimate the number of free processors; default is 2; *applies only to mode 'load'* .

[minforquery:minq]
: defines the minimal number of workers for a session; default is 2; *applies only to mode 'load'* . Note that this is ignored if mxs is set positive (see above).

Example
: `xpd.schedparam selopt:load optnwrks:2`

---

seclib

xpd.seclib *protocol lib*

Function
: Define xproofd specific security directives, allowing for independent rules from the ones applying to data serving. If this directive is missing xproofd falls back to the security setup defined for data serving.

Notes
: conditional structures are not supported for these directives (xpd.protbind can used for similar purposes).

Default
: security checks are disabled by default

Example
: `xpd.seclib libXrdSec.so`

---

sec.protocol

xpd.sec.protocol *protocol params*

Function
: Configure a security protocol for xproofd; same rules as for xrootd.

Default
: none

Example
: `xpd.sec.protocol gsi -crl:0 -gmapopt:1 -dlgpxy:1`

---

shutdown

xpd.shutdown *opt delay*

Function
: Defines what to do when no client sessions are attached to a client area.

Parameters

opt
: is the type of action to be taken when a client completly disconnets; possible values are:
0 remain connected
1 terminate when idle
2 terminate no matter the processing state

delay
: is the delay after which the action for option 1 or 2 is performed; in seconds; to indicate minutes or hours use the suffix 'm' or 'h', respectively; e.g. 5m for 5 minutes.

Default
: `xpd.shutdown 1 0`

Example
: `xpd.shutdown 2 1m`

---

sockpathdir

xpd.sockpathdir *dir*

_

Function

Defines the directory to be used to create the UNIX sockets used by the proofserv applications to communicate with the controlling daemon

Default

The default is /.xproofd./socks , *e.g. /pool/admin/xproofd.1093/socks .*

Example

```
xpd.sockpathdir /tmp/xpd-sock
```

_

_

superusers

xpd.superusers *usr1,usr2,...*

_

Function

Define list of users with special privileges.

Default

the effective user under which the daemon runs (-R option to xrootd on the command line) is always privileged.

Example

```
xpd.superusers thisusr1,thatusr
```

_

_

tmp

xpd.tmp *TMPdir*

_

Function

Defines the directory to be used to create temporary files

Example

```
xpd.tmp /pool/tmp
```

_

_

trace

xpd.trace *[-]keyword1 [-]keyword2 ...*

_

Function

Specifies tracing options. This works by levels and domains. Each option may be optionally prefixed by a minus sign to turn off the setting.

Keywords

err :	trace errors	[on]
req :	trace protocol requests	[on]*
dbg :	trace details about actions	[off]
hdbg :	trace more details about actions	[off]
login :	trace details about login requests	[on]*
fork :	trace proofserv forks	[on]*
mem :	trace mem buffer manager	[off]

Domains

rsp :	server replies	[on]
aux :	aux functions	[on]
cmgr :	client manager	[on]
smgr :	session manager	[on]
nmgr :	network manager	[on]
pmgr :	priority manager	[on]
gmgr :	group manager	[on]
sched :	details about scheduling	[on]

Global switches:

all (or dump) :	full tracing of everything

Notes
Order matters: 'all' in last position enables everything; in first position is corrected by subsequent settings

Default
Defaults are shown in brackets; '*' shows the default when the '-d' option is passed on the command line. Each option may be optionally prefixed by a minus sign to turn off the setting.

Example

```
xpd.trace fork -err rsp
```

---

umap

xpd.umap *name effuser*

Function
Map *name* to effective user *effuser*; *effuser* must be known by the system, i.e. it must be in the password file; *name* can be either a target user name, or the name of a VO (which can be the result of the authentication handshake or defined via xpd.vo).

Default
By default, the VO *default* (always created including all users) is mapped to the effective owner of the daemon.

Example

```
xpd.umap alienvo alipro
```

---

vo

xpd.vo *voname usr_1 [,usr_2 [,...]] [,g:grp_1,...]*

voname
name of the VO

usr_1 [,usr_2 [,...]]
comma-separated list of users to be associated to VO *voname.*

g:grp_1,...
comma-separated list of PROOF groups (see xpd.groupfile) to be associated to VO *voname.*

Function
Define VO, i.e. group of names or of groups. This mapping can be used in conjunction with xpd.umap to map groups or VO to effective users.

Parameters :

Example

```
xpd.vo votest proof21,g:proofteam
```

---

workdir

xpd.workdir *WORKdir*

Function
: Defines the root path for user sandboxes (working directories); user sandbox for *username* will be asserted/created as *WORKdir/username*

Default
: if this directive is missing the user sandbox will be *$HOME/proof* (i.e. without any additional username specification)

Example
: `xpd.workdir /tmp/proof`

---

worker

xpd.worker *type [user@]host[:port] [workdir=path] [port=val] [repeat=val] \ [image=string] [msd=string]*

type
: type of worker: master, submaster or worker

host
: the host name of the worker; a username different from the local one can be specified in the usual URL format. The host name can contain ranges in square brackets, eg lxb60[45-60] to indicate all hosts lxb6045, lxb6046, ..., lxb6060 . A non-standard port can be appended in URL format or via the dedicated keyword (see below).

workdir
: remote path for working directories (sandboxes)

port
: port to contact fpr the connection

repeat
: number of times this line or multi-line has to be repeated; this allows to avoid duplicating the same line many times when defining multiple workers on the same machine

image
: a string used to identifying host sharing a file system

msd
: string used to identify machines sharing the mass storage

Function
: Defines a PROOF node(s). This directive is supposed to replace the proof.conf file giving more flexibility in the definition of the host name.

Parameters :

Example
: `xpd.worker worker lxb60[47-80] workdir=/tmp/proof1`

---

xrootd

xpd.xrootd *path-to-plugin-with-XrdXrootdProtocol*

Function
: Enables file serving via XRootD. If a request for a file is identified on the open connection, the request is served by XRootD. This works internally by creating a XrdXrootdProcotol object and attaching it to the open link.

Notes
: This directive should be used instead of 'xpd.rootd' . The behaviour of the XrdXrootdProtocol plug-in can be controlled with standard XRootD directives in this same configuration file.

Default
: Disabled.

Examples

```
          xpd.xrootd libXrdXrootd-4.so
          xpd.xrootd /opt/xrootd/lib/libXrdXrootd-4.so
```

-

PEAC

PEAC is a set of tools to facilitate setting up a full featured analysis facility on set of machines or a multi-core machine.

Current PEAC documentation can be found at http://mon1.saske.sk/peac/doc/peac-main/ .

Configuration FAQ

Frequently Asked Questions about PROOF installation

Where to find the PROOF distribution?

PROOF is included in the ROOT distribution. To use PROOF, one has to download ROOT (source or binaries) and then install and configure PROOF.

Xrootd failing to find or to load the plug-in libraries

This is typically caused by the LD_LIBRARY_PATH not being correctly defined. The solution is to make sure that the paths with the shared libraries needed are seen by the loader, either via LD_LIBRARY_PATH or configuration files (e.g. /etc/ld.so.conf). Make also sure that the `xrootd` config file does not contain wrong absolute paths which may create version inconsistencies.

Undefined symbols at xrootd startup

This is usually caused by the absence of the ROOT map file. xproofd makes a test run of the proofserv application with the purpose to check that it starts correctly and to get the PROOF version. However, if the map is missing, `proofserv` will fail to run. In such a case running 'gmake map' within $ROOTSYS usually fixes the problem.

What are the sites running PROOF?

For instance ALICE CERN Analysis Facility (CAF) and U.S. ATLAS are running PROOF. If you have a website describing your experience or configuration of PROOF let us know.

What do I need to get Bonjour/Zeroconf support?

To get support of Bonjour/Zeroconf on your PROOF installation you must install the appropiate libraries for you platform and compile enabling the Bonjour support.

Libraries needed by platform

- Mac OS X: no library install is needed, since everything is embedded in libSystem since Mac OS X 10.4 and newer.
- GNU/Linux: Bonjour support on Linux is based on the Avahi framework that came by default in many distributions and is in the repositories of many others in RPM and APT formats. The pacakges needed are the following in a version equal or newer than 0.6.16. For SLC5:
  - avahi-compat-libdns_sd and avahi-compat-libdns_sd-dev (This provides compatibility for common functions)
  - avahi and avahi-devel (This is the daemon and the resolver, plus the native headers).
  - avahi-tools (Command line tools, useful for debug some configuration problems).

  In the case of Ubuntu/Debian:
  - libavahi-compat-libdnssd-dev and libavahi-compat-libdnssd1 (This provides compatibility for common functions)
  - avahi-autoipd, avahi-daemon, avahi-discover, avahi-dnsconfd, libavahi-client-dev, libavahi-common-dev, libavahi-core-dev and libavahi-core6 (This is the daemon and the resolver, plus the native headers).
  - avahi-utils and avahi-dbg (Command line tools, useful for debug some configuration problems).

  Please note that name of the packages may change slightly between distributions (these are from SLC5 and Ubuntu).
- Microsoft Windows: currently, there is not support for Windows.

Libraries needed by platform

In order to enable the Bonjour support (that is disabled by default), you must compile running the configure command in the next way. After that, run the standard make and make install commands:

```
./configure [arch] --enable-bonjour
```

Using PROOF

The purpose of this page is to describe how to run your analysis using PROOF.

Connecting to PROOF

In this section we discuss the different ways to open a PROOF session.

Starting and running PROOF from the command line

The client interaction with a PROOF session goes via an instance of the API class TProof . The interaction with the server daemon is internally managed by a dedicated manager class, TProofMgr , which is invoked by TProof when creating a session.

To create a session from the ROOT prompt just type

```
root[0] TProof *p = TProof::Open("")
```

or

```
root[1] TProof *p = TProof::Open("@:2093")
```

Here "" represents the name of the master machine on the PROOF enabled cluster; if the user name to be used for identification is different from the local one and/or if the service is run on a port different from the standard one (1093) then the username and/or the port must be specified using an URL syntax as shown in the second example.

The *static* TProof::Open(...) takes care of creating the correct manager instance for the required cluster; the manager instance is available via TProof::GetManager().

If the user has already a session running on the cluster, by default TProof::Open(...) attaches to the existing session. To force creation of a new session the option N must be used:

```
root[0] TProof *p = TProof::Open("/?N")
```

Changing the default ROOT version

The version of the ROOT distribution installed on the PROOF cluster is the ROOT version run by PROOF by default. It is possible to choose a different version for the 'proofserv' applications run on the master and worker machines. The only requirement is that the versions are compatible with the version run by the xrootd/xproofd daemon applications, which is typically the case for ROOT later than 5.22/00. This functionality also provides a convenient way to easy have debug and optimized version available to users.

The versions available on the cluster are defined by the PROOF administrator in the configuration file using the ' xpd.rootsys' directive, e.g.

```
# Default version optimized
xpd.rootsys /opt/root/v5-27-06/root v5-27-06
# Debug version
xpd.rootsys /opt/root/v5-27-06_dbg/root v5-27-06_dbg
```

The first 'xpd.rootsys' defines the default version. The user can browse the available versions using the ShowROOTVersions method of the PROOF manager:

```
root [1] TProof::Mgr("cernvm24.cern.ch")->ShowROOTVersions()
----------------------------------------------------------

Available versions (tag ROOT-vers remote-path PROOF-version):

* v5-27-06 5.27/06 /opt/root/v5-27-06/root 29
v5-27-06_dbg 5.27/06 /opt/root/v5-27-06_dbg/root 29

----------------------------------------------------------
```

The current version if the one marked by an asterisk (v5-27-06 in the example above); here 'tag' is the unique tag identifying the version which is used as argument to SetROOTVersion to change version. Example:

```
root [2] TProof::Mgr("cernvm24.cern.ch")->SetROOTVersion("v5-27-06_dbg")
root [3] TProof::Mgr("cernvm24.cern.ch")->ShowROOTVersions()
----------------------------------------------------------

Available versions (tag ROOT-vers remote-path PROOF-version):

v5-27-06 5.27/06 /opt/root/v5-27-06/root 29
* v5-27-06_dbg 5.27/06 /opt/root/v5-27-06_dbg/root 29

----------------------------------------------------------
```

The change must be done before opening a new PROOF session. If the change is active the user is notified at startup that the ROOT version in use is not the default one:

```
root [0] p = TProof::Open("cernvm24.cern.ch")
Starting master: opening connection ...

| Message from server:
| ++++ Using NON-default ROOT version: v5-27-06_dbg 5.27/06 /pool/sw/root/v5-27-06_dbg/root 29 ++++

Starting master: OK
Opening connections to workers: OK (50 workers)
Setting up worker servers: OK (50 workers)
PROOF set to parallel mode (50 workers)
(class TProof*)0x17c2330
root [1]
```

Connecting to PROOF via the tunnel

Introduction

A potential problem in connecting to a PROOF cluster may come from the fact that the master stays behind a firewall, so that direct connections to it are not allowed. For example, the CAF machines at CERN are reachable only within the CERN domain; direct connections - e.g. from home - are not possible.

SSH tunnels can be used to circumvent this problem. This technology allows to create an underlying connection between a port on a local machine and the wished entry point on the target machine (the master, in our case) using a third party machine to which the client can connect.

Setting up the SSH tunnel

To setup the channel we need to know:

1. the name of master machine and the port on which the PROOF related daemons accept connections; in our example 'proofmaster.domain.org' and '1093', respectively;
2. the name of a third party machine open to the outside world and from which direct connections to the master are possible; hereafter we name it 'open.domain.org'
3. a local port number available fo outside connections; e.g. '3000'.

To setup the SSH channel between the local port 3000 and port 1093 on proofmaster.domain.org just execute

```
ssh -N -f -4 -L 3000:proofmaster.domain.org:1093 open.domain.org
```

This is a brief (incomplete) explanation of the ssh options used:

-N
: do not expect to execute a remote command (we are just forwarding ports);

-f
: run into the background (only needed if we want to continue using the current window);

-4
: use IPv4 addresses only (to prevent problems with machines non supporting IPv6; may not be needed; it depends on the setup);

-L
: define the end-points of the tunnel

.

The user credentials on open.domain.org are of course required to execute successfully this command.

Connecting to PROOF via the tunnel

Once the tunnel is created the remote target entity is mapped on the local port, so the connection to the PROOF cluster goes via the local port:

```
root[0] TProof *p = TProof::Open("localhost:3000")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (3 workers)
Setting up worker servers: OK (3 workers)
PROOF set to parallel mode (3 workers)
(class TProof*)0x82e9670
```

Using the GUI

Since ROOT v5.04 a dedicated GUI is available to control PROOF sessions. The GUI allows starting a session, to handle packages, to define and run queries and to handle the results of queries. To usage of the GUI is described in here.

Client-side authentication issues

WARNING : page under construction

Contents:

Authentication is performed using the security modules delivered with xrootd. In this section we address some setup issues which could occur on the client-side.

Password authentication

GSI authentication

libXrdSecgsi.so

The plug-in needed for GSI authentication is build automatically during a ROOT build if OpenSSL is available on the machine. However, for ROOT versions older than 5.21/06 the configuration option '--enable-globus' needs to be specified. Note that, despite what ./configure --help outputs, you do not need the Globus Tool Kit for libXrdSecgsi.so.

CA certificates

In addition to a valid a {certificate, key} pair, GSI authentication requires the certificates of the CA issuing the certificate of the server machine (the master). By default these are located under /etc/grid-security/certificates. If a certificate is missing there (or has expired) it can be obtained from the related CA web site.If the certificate cannot be copied into the standard directory, for example because of lack of rights, it can be saved to any directory, provide that the chosen directory is communicated to the client application by means of the environment variable X509_CERT_DIR .

Host names

The GSI plug-in coming with ROOT versions newer than 5.21/04 contains a security fix which does not accept any longer mismatches between the real server name and the name found in the server certificate: the latter should be in the form '*/' or ''. If a message like this appears

```
root [0] p = TProof::Open("alicecaf")
081027 13:37:31 001 Proofx-E: Conn::Authenticate: cannot obtain credentials
081027 13:37:31 001 Proofx-E: Conn::GetAccessToSrv: client could not login at [lxfsrd0506.cern.ch:1093]
081027 13:37:31 001 Proofx-E: Conn::Connect: failure: cannot obtain credentials: Secgsi: ErrParseBuffer: \
server certificate CN 'lxb6041.cern.ch' does not match the expected format(s): \
'*/lxfsrd0506.cern.ch' (default); exceptions are controlled by the env XrdSecGSISRVNAMES
081027 13:37:31 001 Proofx-E: XrdProofConn: XrdProofConn: severe error occurred while opening a connection to server \
[lxfsrd0506.cern.ch:1093]
(class TProof*)0x0
root [1]
```

and the server is trusted, exceptions to the rule can be set using the environment variable XrdSecGSISRVNAMES; in the example above the following fixes problem:

```
export XrdSecGSISRVNAMES="lxb6041.cern.ch"
```

It is possible to specify multiple exceptions using the separator '|'; the wild card '*' is also accepted.

How to use PROOF from the Python shell

- Enable the ROOT Python bindings
- Starting a PROOF session within Python

Enable the ROOT Python bindings

First all make sure the the ROOT Python bindings have been built: under $ROOTSYS/lib there should be modules ROOT.py and libPyROOT.so (under windows the related modules appear under$ROOTSYS/bin). If the modules are absent you should re-configure ROOT with the option '--enable-python' and re-build. Do not forget to make the new modules visible to Python: for that, you should add $ROOTSYS/lib to $PYTHONPATH or $PYTHONDIR .

Starting a PROOF session within Python

In Python shell first import TProof from ROOT

```
>>> from ROOT import TProof
```

then just open a session, remembering that double quotes must be replaced by single quotes and '::' by a simple '.' :

```
>>> p = TProof.Open('lxfsrd0706.cern.ch')
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (8 workers)
Setting up worker servers: OK (8 workers)
PROOF set to parallel mode (8 workers)
>>>
```

At this point the session is open; in using the 'p' object remember to replace '->' by '.', e.g.

```
>>> p.Print()
Connected to:             lxfsrd0706.cern.ch (valid)
Port number:              1093
User:                     ganis
ROOT version|rev:         5.21/05|r26159
Architecture-Compiler:    linuxx8664gcc-gcc412
Proofd protocol version:  17
Client protocol version:  19
Remote protocol version:  17
Log level:                0
Session unique tag:       lxfsrd0706-1226574772-21889
Default data pool:        root://lxfsrd0506.cern.ch//proofpool
*** Master server 0 (parallel mode, 8 workers):
Master host name:           lxfsrd0706.cern.ch
Port number:                1093
User/Group:                 ganis/group03
ROOT version|rev|tag:       5.21/01|r24746|v5-21-01-alice_dbg
Architecture-Compiler:      linuxx8664gcc-gcc346
Protocol version:           17
Image name:                 lxfsrd0706.cern.ch:/pool/proofbox/ganis
Working directory:          /pool/proofbox/ganis/session-lxfsrd0706-1226576324-24040/master-0-lxfsrd0706-1226576324-24040
Config directory:
Config file:                proof.conf
Log level:                  0
Number of workers:          8
Number of active workers:   8
Number of unique workers:   2
Number of inactive workers: 0
Number of bad workers:      0
Total MBs processed:        0.00
Total real time used (s):   0.011
Total CPU time used (s):    0.000
>>>
```

You can test processing using the ProofSimple example in $ROOTSYS/tutorials/proof

```
>>> p.Process('$ROOTSYS/tutorials/proof/ProofSimple.C+',100000)
Mst-0: grand total: sent 101 objects, size: 94354 bytes
0L
>>>>
```

which should produce a canvas with 100 1-D histograms with random gaussian distributions

The progress dialog

This page will describe the PROOF progress dialog and its functionality. The progress dialog pop-ups automatically on the client screen at query start-up. The dialog allow to monitoring the progress of the query, instantaneous and average rates.

If for some reason (e.g. slowness of the X connection) the graphic dialog is non desired, it is possible to disable it by setting the TProof::kUsingSessionGui on the TProof object before any call to Process, e.g.

```
root [0] TProof *p = TProof::Open(...)
root [1] p->SetBit(TProof::kUsingSessionGui)
root [2] p->Process(...)
```

NB: page under preparation

Accessing the sandbox

Since version 5.25/02 (more precisely, SVN trunk #29580) the PROOF manager class TProofMgr provides some functionality to access the sandbox. This page describes the functionality available in the trunk.

Available functionality:

Functionality implying modifications is limited to the sandbox. Browsing functionality can be extended by the administrator to a list of paths. Superusers can browse all the paths tey are allowed by their credentials.

---

1. Functions to access all the sandboxes

The functions described in this sub-section take three arguments: *what*, specifying the path(s), *how*, specifying options, and *where*, specifying the nodes where to run. The meaning of the first two, *what* and *how*, depend on the command and are detailed in each sub-section. The string *where*, instead, has the same meaning for all the functions, with some restrictions underlined where relevant: *where* indicates where to run the command, defaulting to master only; *where*="all" runs the command on all workers (the output, when relevant - e.g. ls, is prefixed by the node name); *where*="" runs the command on the specified worker node; wild cards are not supported.

1.1 Listing the content of the sandbox

```
TProofMgr::Ls(const char *what, const char *how, const char *where)
```

This command runs the unix command 'ls' on the master or on the worker nodes. The string *what* specifies the path to list, absolute or referred to the sandbox, e.g. "" or "~/" list the sandbox content; listing of absolute paths is restricted by default to superusers. The string *how* specifies the options to be passed to 'ls', e.g. '-l' to show full information about the file status (see here for *where*).
Example:

```
root [0] mgr = TProofMgr::Create("alicecaf.cern.ch:31093")
root [1] mgr->Ls()
Node: lxfsrd0506.cern.ch:31093
-----
cache                                 session-lxfsrd0506-1238687519-26321
datasets                              session-lxfsrd0506-1238687684-26814
last-master-session                   session-lxfsrd0506-1238688131-28013
packages                              session-lxfsrd0506-1245399462-18523
queries                               session-lxfsrd0506-1247917607-18236
session-lxfsrd0506-1238687383-26012  session-lxfsrd0506-1247918526-22124
session-lxfsrd0506-1238687447-26173
root [2] mgr->Ls("packages","-lt")
Node: lxfsrd0506.cern.ch:31093
-----
total 344
-rw-r--r--  1 ganis sf  91245 Jul 23 12:00 STEERBase.par
drwxr-xr-x  3 ganis sf    844 Jul 18 13:25 ANALYSIS
-rw-r--r--  1 ganis sf  36575 Jul 18 13:25 ANALYSIS.par
drwxr-xr-x  3 ganis sf  16384 Jul 18 13:25 ESD
-rw-r--r--  1 ganis sf 186850 Jul 18 13:24 ESD.par
drwxr-xr-x  3 ganis sf  16384 Jul 18 13:24 STEERBase
```

1.2 Removing files from the sandbox

```
TProofMgr::Rm(const char *what, const char *how, const char *where)
```

This command runs the unix command 'rm' on the path specified by *what*. The meaning of the arguments is the same as for TProofMgr::Ls . The following limitations apply:

1. Wild cards are only allowed for files in the sandbox
2. It is not possible to delete the sandbox basic directories.

Example:

```
root [9] mgr->Ls("~/packages")
Node: lxfsrd0506.cern.ch:31093
-----
ANALYSIS  ANALYSIS.par  ESD  ESD.par  STEERBase  STEERBase.par
root [10] mgr->Rm("~/packages/ANALYSIS.par")
Do you really want to remove '~/packages/ANALYSIS.par'? [N/y]Y
Node: lxfsrd0506.cern.ch:31093
-----
root [11] mgr->Ls("~/packages")
Node: lxfsrd0506.cern.ch:31093
-----
ANALYSIS  ESD  ESD.par  STEERBase  STEERBase.par
root [12] mgr->Rm("~/packages","-fr")
Do you really want to remove '~/packages'? [N/y]y
Node: lxfsrd0506.cern.ch:31093
-----
lxfsrd0506.cern.ch: removing a basic sandbox directory is not allowed: /pool/proofbox-xpd/ganis/packages
```

1.3 Accessing the content of a file

```
TProofMgr::More(const char *what, const char *how, const char *where)
TProofMgr::Tail(const char *what, const char *how, const char *where)
TProofMgr::Grep(const char *what, const char *how, const char *where)
```

These commands run the related unix commands ('more', 'tail', 'grep') on the path specified by *what*. The meaning of the arguments is the same as for TProofMgr::Ls .

Grep example:

```
root [1] mgr->Grep("~/cache/AliAnalysisTaskPt.cxx", "-n ::")
Node: lxfsrd0506.cern.ch:31093
-----
41:AliAnalysisTaskPt::AliAnalysisTaskPt(const char *name)
48:  DefineInput(0, TChain::Class());
50:  DefineOutput(0, TH1F::Class());
54:void AliAnalysisTaskPt::ConnectInputData(Option_t *)
68:    AliESDInputHandler *esdH = dynamic_cast (AliAnalysisManager::GetAnalysisManager()->GetInputEventHandler());
78:void AliAnalysisTaskPt::CreateOutputObjects()
109:  TGrid* alien = TGrid::Connect("alien://");
110:  TGridResult* result = alien->Command("ls", 0,TAlien::kOUTPUT);
115:  //TFile::Open("alien:///alice/data/2008/LHC08d/000062564/ESDs/pass1/08000062564000.10/AliESDs.root");
129:void AliAnalysisTaskPt::Exec(Option_t *)
157:void AliAnalysisTaskPt::Terminate(Option_t *)
```

More example:

```
root [2] mgr->More("~/cache/AliAnalysisTaskPt.h")
Node: lxfsrd0506.cern.ch:31093
-----
::::::::::::::
/pool/proofbox-xpd/ganis/cache/AliAnalysisTaskPt.h
::::::::::::::
#ifndef AliAnalysisTaskPt_cxx
#define AliAnalysisTaskPt_cxx

// example of an analysis task creating a p_t spectrum
// Authors: Panos Cristakoglou, Jan Fiete Grosse-Oetringhaus, Christian Klein-Boesing

class TH1F;
class AliESDEvent;

#include "AliAnalysisTask.h"

...
```

Tail example:

```
root [3] mgr->Tail("~/cache/AliAnalysisTaskPt.cxx")
Node: lxfsrd0506.cern.ch:31093
-----
  fHistPt = dynamic_cast (GetOutputData(0));
  if (!fHistPt) {
    Printf("ERROR: fHistPt not available");
    return;
  }

  TCanvas *c1 = new TCanvas("AliAnalysisTaskPt","Pt",10,10,510,510);
  c1->cd(1)->SetLogy();
  fHistPt->DrawCopy("E");
}
```

1.4 Accessing the status of a file

```
Int_t TProofMgr::Stat(const char *what, FileStat_t &status, const char *where)
```

This function allows to get information about the file *what* on the master. The meaning of *what* and of *where* are the same as for TProofMgr::Ls, except that *where*="all" is not supported;the output is the same of the TSystem::GetPathInfo functions.

1.5 Calculating the MD5 check sum of a file

```
Int_t TProofMgr::Md5sum(const char *what, TString &sum, const char *where)
```

This function allows to get the MD5 check sum the file *what* on the master. The meaning of *what* and of *where* are the same as for TProofMgr::Ls, except that *where*="all" is not supported ; the output is the string version of the check sum.
Example:

```
root [4] TString sum
root [5] mgr->Md5sum("~/cache/AliAnalysisTaskPt.cxx", sum)
(Int_t)0
root [6] sum
(class TString)"b24e6f4ca256014b959a6ad7af0f45b4"
```

2. Functions to access files on the master

2.1 Downloading a file from the master

```
Int_t TProofMgr::GetFile(const char *remote, const char *local, const char *options)
```

This function allows to download the file *remote* (same conventions as for *what* in TProofMgr::Ls) to the local file specified by *local*. If *local* is missing or specifies a directory, the file name of *remote* is added to 'local'. If the 'local' file exists, a check on the MD5 sums is done; if the files are not equal the caller is asked whether the she/he wants to overwrite the file. If *option*="force" the file is always overwritten without asking.
A text base progress bar is displayed during downloading. The MD5 check sums are checked for consistency at the end of downloading.
Example:

```
root [1] mgr->GetFile("~/packages/STEERBase.par", "~/")
[GetFile] Total 0.09 MB |====================| 100.00 % [7.4 MB/s]
(Int_t)0
```

2.2 Uploading a file to the master

```
Int_t TProofMgr::PutFile(const char *local, const char *remote, const char *options)
```

This function allows to upload the file *local* to the file *remote* (same conventions as for *what* in TProofMgr::Ls). If *remote* is missing or specifies a directory, the file name of *local* is added to *remote*. If the *remote* file exists, a check on the MD5 sums is done; if the files are not equal the caller is asked whether the she/he wants to overwrite the file. If *option*="force" the file is always overwritten without asking.
A text base progress bar is displayed during uploading.The MD5 check sums are checked for consistency at the end of uploading.
Example:

```
root [2] mgr->PutFile("~/STEERBase.par", "~/packages/", "force")
[PutFile] Total 0.09 MB |====================| 100.00 % [10.1 MB/s]
(Int_t)0
```

2.3 Copying a file from/to the master sandbox

```
Int_t TProofMgr::Cp(const char *src, const char *dst, const char *fmt)
```

This command allows to remotely copy a file from the sandbox to another destination, possibly remote, or to copy a file from a possibly remote source to the sandbox. Either the source *src* or the destination *dst* must be in the sandbox. The file moving is done with some predefined commands specified via the card xpd.cpcmd; the command to use is chosen based on the file protocol. Protocols supported by default are

1. **root://** or **xrd://** using '**xrdcp**';
2. **http://** or **https://** using '**wget**' ; this is supported only as source.

If both source and destination are local the standard 'cp' is used. The *fmt* can be used to modify the format of the command, for example to add options: note that it should always contain two string converters '%s' for the source and destination files.

Example:

```
root [8] mgr->Cp("http://root.cern.ch/files/h1/dstarmb.root","~/cache/")
--14:52:53--  http://root.cern.ch/files/h1/dstarmb.root
           => `/pool/proofbox-xpd/ganis/cache/dstarmb.root'
```

```
Resolving root.cern.ch... 128.142.141.37
Connecting to root.cern.ch|128.142.141.37|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 21,330,730 (20M) [text/plain]
    0K .......... .......... .......... .......... ..........  0%   44.84 MB/s
   50K .......... .......... .......... .......... ..........  0%   76.53 MB/s
...
20650K .......... .......... .......... .......... .......... 99%  108.99 MB/s
20700K .......... .......... .......... .......... .......... 99%   96.50 MB/s
20750K .......... .......... .......... .......... .......... 99%  111.48 MB/s
20800K .......... .......... ..........                      100%  153.42 MB/s
14:52:53 (100.61 MB/s) - `/pool/proofbox-xpd/ganis/cache/dstarmb.root' saved [21330730/21330730]
(Int_t)0
```

9. Directives modifying the behavior of these commands

Two directive exists to modify the behavior of these commands: xpd.exportpath to add browsable paths to the default list (the sandbox) and xpd.cpcmd to specify additional commands to be used to with TProofMgr::Cp .

Basic processing

---

1. Introduction

Basic processing in PROOF is steered by the **TProof::Process** procedure. In PROOF terminology a call to any of the *TProof::Process* represents a query. A query is defined by an algorithm, the number of time a basic task is repeated and possibly a dataset. In the latter case the number of times the task is repeated is typically the number of entries in the dataset files. The algorithm must be defined in the TSelector framework (see also Developing a TSelector). The TSelector is passed either by implementation file, from which a TSelector object is instantiated on each working process, or by selector name, if the TSelector inheriting class is already known to the system (for example because included in a loaded package).

Starting with ROOT version 5.34 is possible also to pass the selector by object, that is, to create a TSelector object in the client session and give it to *TProof::Process*; the selector will be then streamed to the relevant processes.

2. Dataset definition

2.1 Using a TFileCollection

The TFileCollection class is a named collection of files (each described by a TFileInfo) and therefore represents the recommended way to describe a dataset in ROOT. TFileCollection allows to store meta-information about the content of the files and therefore allows to describe at once all the trees included in a file. In addition, some of the experiment catalogues (e.g. the ALICE one) output a TFileCollection object from their query engine. PROOF provides a way to register TFileCollection objects on the master which can be referred to by-name. See ...

2.2. Using a TDSet

The TDSet class is the way PROOF internally handles the dataset. Starting from a TDSet will minimize the number of internal transformations, but may be less convenient if the original collection is not in TDSet format.

2.3. Using a TChain

TChain is a class that allows to describe a dataset - i.e. a set of files - with a TTree interface. By construction a TChain can describe only one TTree, so if more trees are contained in the files one needs to define a TChain per each TTree. PROOF internally must transform the TChain into a TDSet, so for large datasets starting from a TChain is quite inefficient.

3. TProof::Process API

In this section we describe the TProof methods devoted for processing.

In all cases the *option* field is available inside the selector via TSelector::GetOption() (some special options, e.g. "ASYN", "feedback=...", are filtered out before starting processing).

The methods taking a TSelector object are only available starting from ROOT 5.34 . In this case the client can create and configure the TSelector object before start processing. Note that the TSelector object must be streamable for this to work, i.e. a positive class version must be set via the **ClassDef** macro, increased each time the selector class members are modified. Also, the TSelector class must be known on the master and workers nodes. Typical logical workflow:

```
root [] TProof *proof = TProof::Open("")
root [] proof->Load("MySelector.C+") // Load locally and on the cluster
root [] MySelector *mysel = new MySelector(arg1, arg2)
root [] mysel->setMyParms(par1, par2, par3)
root [] proof->Process(mysel, 1000000)
```

3.1 Cycle-driven processing

In this case we only need to specify a TSelector and the number of times, *ncycles*, its Process method is called.

```
Long64_t Process(const char *selector, Long64_t ncycles, Option_t *option = "")
Long64_t Process(TSelector *selector, Long64_t ncycles, Option_t *option = "")
```

3.2 Data-driven processing

Data-driven processing is steered by a dataset, i.e. by a main TTree stored in a set of files. The Process methods take, therefore, a dataset - in one of the forms described above - and a TSelector; the number of entries, *nentries*, and the first entry to process, *firstentry*, are optional and only needed to restrict processing to a sub-sample of events.

3.2.1 Processing a TFileCollection by object

The dataset is in this case defined by a TFileCollection object created by the client. This is object is passed directly to Process and then transferred by PROOF to the master.

```
Long64_t Process(TFileCollection *fc, const char *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0)
Long64_t Process(TFileCollection *fc, TSelector *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0)
```

3.2.2 Processing a TFileCollection by name

Processing a TFileCollection by name is possible when the TFileCollection has been registered on the master using the PROOF dataset manager interface (see Working with data sets). The available datasets can be displayed using TProof::ShowDataSets().

```
Long64_t Process(const char *dsetname, const char *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0, TObject *enl = 0)
Long64_t Process(const char *dsetname, TSelector *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0, TObject *enl = 0)
```

3.2.3 Processing a TDSet

The dataset is in this case defined by a TDSet object created by the client. This is object is passed directly to Process and then transferred by PROOF to the master.

```
Long64_t Process(TDSet *dset, const char *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0)
Long64_t Process(TDSet *dset, TSelector *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0)
```

3.2.4 Processing a TChain

The Process(...) methods used in this case are those of TChain:

```
Long64_t Process(const char *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0)
Long64_t Process(TSelector *selector, Option_t *option = "", Long64_t nentries = -1, Long64_t firstentry = 0)
```

The TChain::SetProof(TProof *) method is used to tell the chain to use PROOF for processing instead of the local ROOT session. This is the typical flow:

```
root [] TChain chain("MyTree")
root [] TProof *proof = TProof::Open("")
root [] chain.SetProof(proof)
root [] chain.Process("myselector.C+")
root [] chain.SetProof(0) // Detach from the PROOF session
```

The TChain is internally transformed in a TDSet object which is sent to the master.

4. Information about the last query

Detailed information about the last query can be obtained from the TQueryResult object returned by TProof::GetQueryResult() method. For example

```
root [3] gProof->GetQueryResult()->Print()
+++ #:1 ref:"pcphsft64-1444127173-2526:q1" sel:h1analysis finalized evts:0-283812
root [4] gProof->GetQueryResult()->Print("F")
+++
+++ #:1 ref:"pcphsft64-1444127173-2526:q1" sel:h1analysis finalized
+++         started:   Tue Oct  6 12:26:13 2015
+++         init:      2.932 sec
+++         process:   11.957 sec (CPU time: 5.2 sec)
+++         merge:     0.313 sec
+++         processed: 283813 events (size: 36.352 MBs)
+++         rate:      23736.8 evts/sec
+++         # workers: 8
+++         results:   sent to client
+++         outlist:   19 objects
```

Controlling the PROOF behavior

The behavior of PROOF can be controlled via environment variables, ROOT directives (.rootrc file family) and INPUT parameters. In this section the different options to control the way a PROOF session behaves are described.

The environment

In this section we describe how to control the environment for the PROOF session. The PROOF session consists of a set of processes (master and workers); these processes take the environment of the shells from which they are started up. Therefore, to be effective, any environment change must be done before the process startup.

In the case of PROOF-Lite, the workers inherit the environment of the client session. However, the functionality described in this session applies also for PROOF-Lite.

---

1. The TProof interface

To control the environment, the interface class TProof owns a static list of environment variables to be forwarded to the daemons before starting the session processes. The internal list is controlled by a few static methods:

- AddEnvVar: add an environment variable to the list

  ```
  void TProof::AddEnvVar(const char *name, const char *value)
  ```

  This method adds an environment variable called 'name' with value 'value' to the internal list.
- DelEnvVar: remove form the list an environment variable

  ```
  void TProof::DelEnvVar(const char *name)
  ```

  This method removes form the internal list the first occurrence of an environment variable called 'name'.
- GetEnvVars: get the list of environment variables

  ```
  const TList *TProof::GetEnvVars()
  ```

  This method returns a pointer to the internal list of environment variables.
- ResetEnvVars: clear the list of environment variables

  ```
  void TProof::ResetEnvVars()
  ```

  This method clears the list of environment variables.

The *name* and *values* of the specified environment variables can contain the place-holders and which are substituted with the *username* and the `ROOTSYS` path, respectively.

Since ROOT 5.27/06 (SVN revision 34718) it is possible to define - via the special variable PROOF_ENVVARS - a list of environment variables to be transmitted to nodes. This allows to change the settings without changing the application or the macro running TProof::Open . For example:

```
$ export MYENV1=myenviron1 ; export MYENV2=myenviron2 ; export MYENV3=myenviron3
$ export PROOF_ENVVARS="MYENV1,MYENV2,MYENV3"
$ root -l
root [0] TProof *p = TProof::Open("")
Info in <:init>: the following env variables have added to the list: 'MYENV1,MYENV2,MYENV3'
Starting master: opening connection ...
...
```

PROOF notifies the variables added. A variable must be defined in the shell where ROOT is run; otherwise it is not added (a warning is issued).

The variables defined via PROOF_ENVVARS are added to the internal list when constructing the TProof instance; they can be removed from the list with TProof::DelEnvVar at any moment prior to the call to TProof::Open.

2. The proofserv wrapper script

The PROOF sessions on the server side (master and workers) are instances of the TProofServ class instantiated via the $ROOTSYS/bin/proofserv.exe application. The application is not launched directly but via the wrapper script $ROOTSYS/bin/proofserv . The wrapper script contains basically two parts. The first were an initialization command can be executed to setup the environment; this is controlled by the variable PROOF_INITCMD. The second in which the proofserv.exe is launched, possibly via another wrapper application; this is controlled by the PROOF_WRAPPERCMD variable.

2.1 The PROOF_INITCMD

The PROOF_INITCMD variable defines a procedure which outputs a command to be executed before starting '*proofserv.exe*'. This can be a simple 'echo' command or a complicated script.
So, for example, the following are equivalent:

```
root [] TProof::AddEnvVar("MYVAR", "all")
```

is equivalent to

```
root [] TProof::AddEnvVar("PROOF_INITCMD", "echo export MYVAR=all")
```

If the setup is defined by a script, *e.g. /some/path/setup-env.sh*, then the script should be displayed, e.g. with 'cat':

```
root [] TProof::AddEnvVar("PROOF_INITCMD","cat /some/path/setup-env.sh")
```

If the script outputs the command to be executed, *e.g. /some/path/getscram.sh*, then just put the script path:

```
root [] TProof::AddEnvVar("PROOF_INITCMD","/some/path/getscram.sh")
```

2.2 The PROOF_WRAPPERCMD

The variable PROOF_WRAPPERCMD describes an application or a script to be used to launch proofserv.exe . The natural example is valgrind (PROOF valgrinding is implemented using this technique).

The ROOT directives

In this page the ROOT directives affecting a PROOF session are described. These directives can affect either the local session (including PROOF-lite) or the server side sessions.

NB: page under construction

ROOT directives active in PROOF
(PL=PROOF-Lite, PC=PROOF client, M=master, W=workers)

Name	Scope	Default	Description
Proof.Sandbox	PL,PC	~/.proof	Location of the working area
ProofLite.Sandbox	PL	~/.proof	Location of the PROOF-Lite working area
ProofServ.Sandbox	M,W	~/.proof	Location of the proofserv working area
ProofLite.CacheDir	PL	/cache	Path to area to cache files (other than packages)
ProofLite.PackageDir	PL	/packages	Path to area to install PAR packages
ProofLite.QueryDir	PL	/queries	Path to area to save query summaries (TQueryResult)
ProofLite.DataSetDir	PL	/datasets	Path to file-based dataset repository
ProofServ.DataDir	M,W	/data	Path to are for data files created by workers
ProofServ.CacheDir	M,W	/cache	Location of the proofserv cachedir

The INPUT list

The *input list* is a list of TObject which is made available at running time on each participating node and in the TSelector object (fInput member). This list is meant to provide a way to provide relevant information for the query and can therefore be used to control the job.

---

1. Adding / Removing objects to / from the input list

PROOF provides the following API to handle generic input objects:

void TProof::AddInput(TObject *obj)

Purpose	Provide a hook to add an object to the input list
obj	Object to be added; though not mandatory, it is recommended that this object is named, so that it can be easily searched for by name.

void TProof::ClearInput()

Purpose	Clear the input list

TList *TProof::GetInputList()

Purpose	Get a pointer to the input list

2. Working with parameters

One of the main usage of the input list is to set parameters for the selector. PROOF provides and interface to automatically add or retrieve TParameter objects to / from the input list:

void TProof::SetParameter(const char *name, const char *par)

void TProof::SetParameter(const char *name, Int_t par)

void TProof::SetParameter(const char *name, Long_t par)

void TProof::SetParameter(const char *name, Long64_t par)

void TProof::SetParameter(const char *name, Double_t par)

Purpose	Provide a way to add named parameters of the specified type
name	Name of the parameter
par	Value of the parameter

TObject *TProof::GetParameter(const char *name)

Purpose	Provide a way to get a named parameter from the input list; the returned object must be cast to the relevant TParameter<...> implementation
name	Name of the parameter

Int_t TProof::GetParameter(TCollection *c, const char *name, TString &par)

Int_t TProof::GetParameter(TCollection *c, const char *name, Int_t &par)

Int_t TProof::GetParameter(TCollection *c, const char *name, Long_t &par)

Int_t TProof::GetParameter(TCollection *c, const char *name, Long64_t &par)

Int_t TProof::GetParameter(TCollection *c, const char *name, Double_t &par)

Purpose	Provide a way to retrieve named parameters of the specified type from a generic collection; these methods are **static**.
name	Name of the parameter
par	Value of the parameter
Return	0 on success; -1 on error (parameter not found)

3. Working with large input objects

When the input objects are large - e.g. TH3 calibration maps - the default way of distributing - i.e. the input list by streaming in the main message - can be very inefficient. PROOF provides an optimized way to distribute these *input data* constisting in saving these objects into a ROOT file and distributing the file in an optimized way, i.e. compressed, only if any of these objects has changed and only to the unique workers. These objects are available in the nodes via the input list just as the others. The following API is available for theis purpose:

void TProof::AddInputData(TObject *obj, Bool_t push)

Purpose	Add an object to the input data list
obj	Object to be added; though not mandatory, it is recommended that this object is named, so that it can be easily searched for by name.
push	If TRUE the input data are sent over even if no apparent change occured to the input data list

void TProof::ClearInputData(TObject *obj)

Purpose	Remove object from the input data list
obj	Object to be removed; if null, all input data objects are removed

void TProof::ClearInputData(const char *name)

Purpose	Remove named object from the input data list
name	Name of the object to be removed

void TProof::SetInputDataFile(const char *datafile)

Purpose	Set the file to be used to optimally distribute the input data objects. If the file exists the object in the file are added to those in the input data list
datafile	Path to the file

List of INPUT parameters used by PROOF

Contents:

---

1. Summary

Parameters used by PROOF; when missing the type is ignored (setting the parameter enables the corresponding feature)

Name	Type	Output	Description	Availability
Packetizer				
PROOF_PacketizerStrategy	Int_t		Packetizer strategy: 0, 1	
PROOF_Packetizer	const char *		Name of the packetizer to use	
PROOF_ForceLocal	Int_t		Force local processing only	
PROOF_MaxSlavesPerNode	Int_t		Limit workers accessing a server	
PROOF_PacketAsAFraction	Int_t		See text	
PROOF_MinPacketTime	Double_t		See text	
PROOF_MaxPacketTime	Double_t		See text	
PROOF_ValidateByFile	Int_t		See text	>= 5.27/04
PROOF_PacketizerCalibNum	Long64_t		Size of the calibration packet; see text	>= 5.27/04
PROOF_PacketizerFixedNum	Int_t		See text	>= 5.27/04
PROOF_PacketizerTimeLimit	Double_t		See text	>= 5.27/04
Tree Cache				
PROOF_UseTreeCache	Int_t		Enable/disable the tree cache	>= 5.25/04
PROOF_CacheSize	Long64_t		Tree cache size in bytes	>= 5.25/04
Monitoring				
PROOF_StatsHist			Enable monitoring histograms	

PROOF_StatsTrace			Enable recording of trace information on the master	
PROOF_SlaveStatsTrace			Enable recording of trace information on the workers	
Performance				
PROOF_PacketsHist		TH1D	Processed packets per worker	
PROOF_EventsHist		TH1D	Processed events per worker	
PROOF_NodeHist		TH1D	Workers per file serving node	
PROOF_LatencyHist		TH2D	Packet receive latency per worker	
PROOF_ProcTimeHist		TH2D	Packet process time per worker	
PROOF_CpuTimeHist		TH2D	Packet CPU time per worker	
Miscellanea				
PROOF_DataSetSrvMaps	const char *		Defines mapping(s) for data servers	>= 5.27/02
PROOF_LookupOpt	const char *		Controls the dataset lookup	>= 5.27/04
PROOF_UseMergers	Int_t		Controls the use of submergers	>= 5.26/00

2. Description

1. Packetizer Parameters

Packetizer generates packets to be processed on PROOF workers. A packet is an event range (begin entry and number of entries) or object range (first object and number of objects) in a TTree (entries) or a directory (objects) in a file. Packets are generated taking into account the performance of the remote machine, the time it took to process a previous packet on the remote machine, the locality of the data files, etc.
Below we describe way of using different types of packetizers as well as parameters for changing the behavior of the packetizers.

`PROOF_PacketizerStrategy (Int_t)`

`Define t`he strategy to be used in the packetizer. Default is 1 (adaptive strategy). The only available strategy alternative is the old strategy (0).
Example: proof->SetParameter("PROOF_PacketizerStrategy", (Int_t)0) .

`PROOF_Packetizer (const char *)`

Define the class name of the packetizer to use. The class can be uploaded as a PAR file. Default is TPacketizerAdaptive. The old packetizer is called TPacketizer.
Example: proof->SetParameter("PROOF_Packetizer", "TPacketizer") .

`PROOF_ForceLocal (Int_t)`

Force local processing to minimize the network traffic at the expense of longer processing times. If this parameter is set to 1, each worker processes only the files which are located on it own node. In makes sense to set the parameter, only when the dataset is entirely located on the cluster and the session has worker processes on all the cluster nodes with data. Default is 0.
Example: proof->SetParameter("PROOF_ForceLocal", (Int_t)1) .

`PROOF_MaxSlavesPerNode (Int_t)`

Set the maximum number of workers accessing data on any single file server node. Default is -1 (no limit).
Example: proof->SetParameter("PROOF_MaxSlavePerNode", 8) .

`PROOF_PacketAsAFraction (Int_t)`

Defines how big is the packet size as a fraction of the data set size. Setting it to k means that packets will not be bigger than N / (n * k), where N is the number of events in the dataset and n is the number of workers. Default is 4 for TPacketizerAdaptive, 20 for TPacketizer.
Example: proof->SetParameter("PROOF_PacketAsAFraction", 8) .

`PROOF_MinPacketTime (Double_t)`

In TPacketizerAdaptive, defines the minimal size of a packet in seconds. It can be useful to make bigger packets, when 8 or more workers from one session run on the same machine. Default is 3
Example: proof->SetParameter("PROOF_MinPacketTime", 8.) .

`PROOF_MaxPacketTime (Double_t)`

In TPacketizerAdaptive, defines the maximum size of a packet in seconds. It can be useful, for example, to avoid that the whole file is assigned to the same worker, when the number of workers is larger than the number of files. Default is 20
Example: proof->SetParameter("PROOF_MaxPacketTime", 5.) .

`PROOF_ValidateByFile (Int_t)`

When processing a subsample of N entries if this parameter is set to 1 then only the minimum number of files containing enough entries to satisfy the request is validated. By default a limited number of files is validated but the number can exceed the minimum by (N workers-1).
Example: proof->SetParameter("PROOF_ValidateByFile", 1) .

`PROOF_PacketizerCalibNum (Long64_t)`

NB: TPacketizerUnit only

Size of the first packet used to calibrate the worker performance. Default is 5.
Example: proof->SetParameter("PROOF_PacketizerCalibNum", 100) .

`PROOF_PacketizerFixedNum (Int_t)`

NB: TPacketizerUnit only

If this parameter is set to 1 then the packetizer will distribute packets in such a way that all th workers will process the same number of cycles, independently of the single worker performance. Default is off.
Example: proof->SetParameter("PROOF_PacketizerFixedNum", (Int_t)1) .

`PROOF_PacketizerTimeLimit (Double_t)`

NB: TPacketizerUnit only

Max packet size in seconds. This can be used to control the frequency of progress reporting. Default is 1 second.
Example: proof->SetParameter("PROOF_PacketizerTimeLimit", 10.) .

2. Tree cache parameters

PROOF_UseTreeCache (Int_t)

Switch to enable/disable the use of the tree cache in processing data. Default is 1 (enabled).
Example: proof->SetParameter("PROOF_UseTreeCache", 0) .

PROOF_CacheSize (Long64_t)

Set the size of the tree cache (value in bytes). Default is 30000000.
Example: proof->SetParameter("PROOF_CacheSize", 20000000) .

3. Monitoring Parameters

These are the parameters governing the PROOF monitoring (see also `$ROOTSYS/test/ProofBench/README`).

`PROOF_StatsHist`

Enable use of monitoring histograms (the value is ignored).
Example: proof->SetParameter("PROOF_StatsHist", "");

`PROOF_StatsTrace`

Enable the recording of a trace information on the master (the value is ignored). The result is returned in the output list in the performance tree PROOF_PerfStats .

Example: proof->SetParameter("PROOF_StatsTrace", "");

`PROOF_SlaveStatsTrace`

Enable the recording of a trace information on the slaves (the value is ignored). The result is returned in the output list in the performance tree PROOF_PerfStats .
Example: proof->SetParameter("PROOF_SlaveStatsTrace", "");

4. Performance Parameters

These are the parameters governing the PROOF benchmark utilities.

`PROOF_PacketsHist`

Fill a histogram with the number of packets assigned to each worker. The histogram is dynamically updated for usage in the feedback list.

`PROOF_EventsHist`

Fill a histogram with the number of events processed by each worker. The histogram is dynamically updated for usage in the feedback list.

`PROOF_NodeHist`

Fill a histogram with the number of workers accessing a file serving node. The histogram is dynamically updated for usage in the feedback list.

`PROOF_LatencyHist`

Fill a 2D histogram with the latency in receiving the next packet sampled by worker.

`PROOF_ProcTimeHist`

Fill a 2D histogram with the packet processing time sampled by worker.

`PROOF_CpuTimeHist`

Fill a 2D histogram with the packet CPU time sampled by worker.

5. Miscellanea

`PROOF_DataSetSrvMaps`

Define {match,target} pairs to change the servers from which the files in a dataset have to be taken. The parameter is a string in the form "srv1|map1 srv2|map2 ..." where srv1 are the {*protocol,host,port*} of the server to be matched, and map1 are the {*protocol,host,port*} to be used for those files mapping srv1, and so on.
The matching server string can contain the wildcard '*' in the host field or it can be empty, in which case the replacement is done for all the dataset files.
If the '|' character is missing the whole string is assigned to the map part; i.e. 'map1' alone is equivalent to '*|map1' or '|map1'.
Example:
proof->SetParameter("PROOF_DataSetSrvMaps", "root://dserv-*.dom.ain/|root://redir.dom.ain/")

`PROOF_LookupOpt`

Define which dataset files to lookup. Use 'none' to skip the lookup step. Use "all" to lookup all the files, also those flagged as corrupted or offline (non-staged). Use 'stagedOnly' to lookup only the files flagged as online and non-corrupted.
Example:
proof->SetParameter("PROOF_LookupOpt", "none")

`PROOF_UseMergers`

Toggles the use of sub-mergers for merging the results. The integer parameter defines the number of mergers; setting it to 0 makes the master to choose the optimal number based on the number of available workers.
Example:
proof->SetParameter("PROOF_UseMergers", 0)

Developing a TSelector

Developing a TSelector

Overview of the TSelector class

The TSelector is a framework for analyzing event like data. The user derives from the TSelector class and implements the member functions with specific analysis algorithms. When running the analysis, ROOT calls the member functions in a well defined sequence and with well defined arguments. By following the model this analysis class can be used to process data sequentialy on a local workstation and in batch or in parallel using PROOF.

ROOT provides the TTree::MakeSelector function to generate a skeleton class for a given TTree.

For example, if the file "treefile.root" contains the tree "T", a skeleton called "MySelector" can be generated in following way:

```
root [0] TFile *f = TFile::Open("treefile.root")
root [1] TTree *t = (TTree *) f->Get("T")
root [2] t->MakeSelector("MySelector")
root [3] .!ls MySelector*
MySelector.C  MySelector.h
root [4]
```

This skeleton class is a good a starting point for the analysis class. It is recommended that users follow this method.

When running with PROOF a number of worker processes are used to analyze the events. The user creates a PROOF session from the client workstation which allocates a number of workers. The workers instantiate an object of the users analysis class. Each worker processes a fraction of the events as determined by the relative performance of the servers on which the workers are running. The PROOF system takes care of distributing the work. It calls the TSelector functions in each worker. It also distributes the input list to the workers. This is a TList with streamable objects provided in the client. After processing the events PROOF combines the partial results of the workers and returns the consolidated objects (e.g. histograms) to the client session.

Calling sequences

The two sequences below show the order in which the TSelector member functions are called when either processing a tree or chain on a single workstation or when using PROOF to process trees or collections of keyed objects on a distributed system. When running on a sequential query the user calls TTree::Process() and TChain::Process(), when using PROOF the user calls TDSet::Process() (a few other entry points are available as well). Each of the member functions is described in detail after the call sequences.

Local, sequential query

```
Begin()
SlaveBegin()
Init()
   Notify()
      Process()
      ...
      Process()
   ...
   Notify()
      Process()
      ...
      Process()
SlaveTerminate()
```

```
Terminate()
```

Distributed, parallel query, using PROOF

```
+++ CLIENT Session +++       +++ (n) WORKERS +++
Begin()
                             SlaveBegin()
                             Init()
                                 Notify()
                                 Process()
                                 ...
                                 Process()
                                 ...
                             Init()
                                 Notify()
                                 Process()
                                 ...
                                 Process()
                                 ...
                             SlaveTerminate()
Terminate()
```

Main Framework Functions

Begin(), SlaveBegin()

The Begin() function is called at the start of the query. It always runs in the client ROOT session. The SlaveBegin() function is either called in the client or when running with PROOF, on each of the workers. All initialization that is needed for Process() (see below) must therefore be put in SlaveBegin(). Code which needs to access the local client environment, e.g. graphics or the filesystem must be put in Begin(). When running with PROOF the input list (fInput) is distributed to the workers after Begin() returns and before SlaveBegin() is called. This way objects on the client can be made available to the TSelector instances in the workers.

The tree argument is deprecated. (In the case of PROOF the tree is not available on the client and 0 will be passed. The Init() function should be used to implement operations depending on the tree)

Signature:

```
virtual void Begin(TTree *tree);
virtual void SlaveBegin(TTree *tree);
```

Init()

The Init() function is called when the selector needs to initialize a new tree or chain. Typically here the branch addresses of the tree will be set. It is normaly not necessary to make changes to the generated code, but the routine can be extended by the user if needed. Init() will be called many times when running with PROOF.

Signature:

```
virtual void Init(TTree *tree);
```

Notify()

The Notify() function is called when a new file is opened. This can be either for a new TTree in a TChain or when a new TTree is started when using PROOF. Typically here the branch pointers will be retrieved. It is normaly not necessary to make changes to the generated code, but the routine can be extended by the user if needed.

Signature:

```
virtual Bool_t Notify();
```

Process()

The Process() function is called for each entry in the tree (or possibly keyed object in the case of PROOF) to be processed. The entry argument specifies which entry in the currently loaded tree is to be processed. It can be passed to either TTree::GetEntry() or TBranch::GetEntry() to read either all or the required parts of the data. When processing keyed objects with PROOF, the object is already loaded and is available via the fObject pointer.

This function should contain the "body" of the analysis. It can contain simple or elaborate selection criteria, run algorithms on the data of the event and typically fill histograms.

Signature:

```
virtual Bool_t Process(Int_t entry);
```

SlaveTerminate(), Terminate()

The SlaveTerminate() function is called after all entries or objects have been processed. When running with PROOF it is executed by each of the workers. It can be used to do post processing before the partial results of the workers are merged. After SlaveTerminate() the objects in the fOutput lists in the workers are combined by the PROOF system and returned to the client ROOT session. The Terminate() function is the last function to be called during a query. It always runs on the client, it can be used to present the results graphically or save the results to file.

Signature:

```
virtual void SlaveTerminate();
virtual void Terminate();
```

Utility Functions

Version()

The Version() function is introduced to manage API changes in the TSelector class. Version zero, uses the ProcessCut() and ProcessFill() pair rather then Process(). Version one uses Process(), introduces the SlaveBegin() / SlaveTerminate() functions and clarifies the role of Init() and Notify().

Signature:

```
virtual int Version();
```

SetInputList()

Setter for the input list of objects to be transfered to the remote PROOF servers. The input list is transfered after the execution of the Begin() function, so objects can still be added in Begin() to this list. These objects are then available during the selection process (e.g. predefined histograms, etc.). Does not transfer ownership.

Signature:

```
virtual void SetInputList(TList *input);
```

GetOutputList()

Getter for the output list of objects to be transfered back to the client. The output list on each worker is transfered back to the client session after the execution of the SlaveTerminate() function. The PROOF master server merges the objects from the worker output lists in a single output list (merging partial objects into a single one). Ownership remains with the selector. Each query will clear this list.

Signature:

```
virtual TList *GetOutputList();
```

SetObject()

Setter for the object to be processed (internal use only).

Signature:

```
virtual void SetObject(TObject *obj);
```

SetOption()

Setter for the query option string (internal use only).

Signature:

```
virtual void SetOption(const char *option);
```

GetOption()

Getter for the option string that was passed by the user to Tree::Process() or TDSet::Process().

Signature:

```
virtual const char *GetOption();
```

Merging customized classes

Merging customized classes

The PROOF merging phase takes place on the master when the lists of results are received from the workers. PROOF looks for the Merge method of the object class and applies it to the list of objects in hands.

The Merge(TCollection *) interface

The signature of the Merge() method is:

```
void myClass::Merge(TCollection *)
```

The idea is that an object is passed a list of similar objects and it should take the necessary steps to incorporate the information contained in these objects into itself. For common objects, like histograms and trees, the Merge() method is provided by ROOT.

The following is an example of the way TH1::Merge could look like in his basic form:

```
void TH1::Merge(TCollection *hlist)
{
   // Possible implementation for TH1::Merge; the real one add checks on the ranges ...

   if (hlist) {
      TH1 *xh = 0;
      TIter nxh(hlist);
      while (xh = (TH1 *) nxh()) {
         // Add this histogram to me
         Add(xh);
      }
   }

}
```

The real implementation is more complicated because of the checks on the bin ranges.

Note that currently the return value from Merge() is ignored. It may be used in the future.

Finally, for full ROOT functionality (hadd, etc...), if the new class is provided via a new dedicated library, e.g. libmyClass.so, a rootmap file has to be created and placed together with the the library. The rootmap file is a text file with information about the content of the library in terms of class; the name of the file must be the same of the library, with extension '.rootmap', for example libmyClass.rootmap . See examples under $ROOTSYS/lib .

Using the feedback mechanism

In this section we describe how to enable and use the realtime feedback mechanism provided by PROOF.

---

1. Introduction: the PROOF feedback mechanism

The feedback mechanism provided by PROOF consists in giving the user the possibility to get back, at a chosen frequency, the current status of a set of output objects. These objects must be previously registered by the user in the feedback list. When the internal feedback timer expires, the objects in the feedback list are sent by the workers to the master, merged and sent to the user, and made available via the ROOT signal technology.

Needless to say that the all process may be quite heavy and is intended for light monitoring objects.

2. TProof interface

2.1. Enabling basic feedback on per query base (ROOT version >= 5.34)

Starting with ROOT version 5.34 is it possible to enable basic feedback using the option field of the selector. This change is client-side only, so it is enough to upgrade the client to have it working with cluster running older versions.

The keywords enabling automatic setup of feedback are 'fb=comma-separated-list-of-names ' or 'feedback=comma-separated-list-of-names '; the comma-separated list of names is ended by a space, to allow identification of the next option. When these key-pair values are detected feedback is enable for the specified object names, and a TDrawFeedback object created to display them. For example:

```
root [] proof->Process("tutorials/proof/ProofSimple.C+", 10000000, "fb=h10")
```

enables feedback for the histogram 'h10' of the tutorial selector ProofSimple.

The special name 'stats' is reserved for the set of internal statistics histograms {processed events, processed packets, being processed packets}-per-worker (see Creating and Saving the Performance Tree):

```
root [] proof->Process("tutorials/proof/ProofSimple.C+", 10000000, "fb=stats")
```

In this case the feedback handling is done with TStatsFeedback instead of TDrawFeedback. The histograms are plot one in one canvas in three vertical pads.

2.2. Modifying and displaying the feedback list

To enable the feedback mechanism it is sufficient to register at least one object. The way to register the objects is straightforward: TProof has a feedback list member and provides the following methods to modify it:

- *void* TProof::AddFeedback(const char **name*)
  Add the output object named *name* in the feedback list.
- *void* TProof::RemoveFeedback(const char **name*)
  Remove the object named *name* from the feedback list.
- *void* TProof::ClearFeedback()
  Remove all objects from the feedback list.
- *void* TProof::ShowFeedback()
  Display the content of the feedback list.
- *TList* *TProof::GetFeedbackList()
  Get a pointer to the feedback list.

2.3. Setting the frequency

Once enabled the feedback mechanism by adding at least one object to the list, PROOF start sending back the feedback objects at a default frequency of 0.5 Hz, i.e. every 2000 milliseconds. The default feedback period can be modified using the parameter PROOF_FeedbackPeriod which takes the period in milliseconds (a Long_t integer). For example, to set the period to 5 seconds, just set

```
// Set the feedback period to 5 seconds
proof->SetParameter("PROOF_FeedbackPeriod", (Long_t) 5000);
```

before starting the query.

2.4. Getting the feedback objects: the TProof::Feedback(TList *) signal

Once reaching the client, the list of feedback objects is made available via a ROOT signal (see TQObject technology) with signature TProof::Feedback(TList *). To use these objects the client must connect to the signal and provide a method or a function to process the list (i.e. displaying the monitoring histograms). An example of class using this can be found in TDrawFeedback under $ROOTSYS/proof/proofplayer ; this utility class is described to some detail in the next section.

3. **The TDrawFeedback example**

In the section we dissect the utility class TDrawFeedback . Its purpose is to show how to display in separate canvases a set of 1D histograms registered to the list.

The definition of the class is the following:

```
class TDrawFeedback : public TObject, public TQObject {
private:
   Bool_t         fAll;     //draw all or selected objects
   THashList     *fNames;   //selected objects

protected:
   Option_t      *fOption;  //draw option
   TProof        *fProof;   //handle to PROOF session

public:
   TDrawFeedback(TProof *proof = 0, TSeqCollection *names = 0);
   ~TDrawFeedback();

   void Feedback(TList *objs);
   void SetOption(Option_t *option) { fOption = option; }

   ClassDef(TDrawFeedback,0)  // Present PROOF query feedback
};
```

The class derives from TObject and TQObject: this is needed to be enable to usage of the ROOT signal mechanism. The class has four members. The first two (fAll, fNames) allow to choose a subset of feedback histograms to draw. The third one is the drawing option. Finally the class save the pointer to the PROOF session: this is needed to connect and disconnect from the feedback signal.

A part from constructor and destructor, the class has one main method, Feedback(TList *), used to process the signal, and one setter to set a specific drawing option.

If now we look at the constructor implementation we see that the main part is the connection to the signal:

```
//______________________________________________________________________________
TDrawFeedback::TDrawFeedback(TProof *proof, TSeqCollection *names)
  : fAll(kFALSE)
{
   // Constructor

   ...

   Bool_t ok = proof->Connect("Feedback(TList *objs)", "TDrawFeedback",
                  this, "Feedback(TList *objs)");
   ...
}
```

here we request that the method Feedback(TList *) of this object of type TDrawFeedback is called when the PROOF session of interest (e.g. 'proof') emits the Feedback(TList *). In the destructor we just break this link:

```
//______________________________________________________________________________
TDrawFeedback::~TDrawFeedback()
{
   // Destructor

   delete fNames;
   fProof->Disconnect("Feedback(TList*)", this, "Feedback(TList*");
}
```

If we now look at the main method TDrawFeedback(TList *), we see that we just go through the list of objects received from TProof, check whether it is a TH1 and if we are asked to draw it, create a canvas and draw a copy of the histogram:

```
//______________________________________________________________________________
void TDrawFeedback::Feedback(TList *objs)
{
   // Display feedback

   ...

   TIter next(objs);
   TObject *o;
   while( (o = next()) )
   {
      TString name = o->GetName();
      if (fAll || fNames->FindObject(name.Data())) {

         name += "_canvas";
         TVirtualPad *p = (TVirtualPad*) canvases->FindObject(name.Data());

         if ( p == 0 ) {
            gROOT->MakeDefCanvas();
            gPad->SetName(name);
            PDB(kFeedback,2) Info("Feedback","Created canvas %s", name.Data());
         } else {
            p->cd();
            PDB(kFeedback,2) Info("Feedback","Used canvas %s", name.Data());
         }

         if (TH1 *h = dynamic_cast(o)) {
            h->DrawCopy(fOption);
         }

         gPad->Update();
      }
   }
   ...
}
```

This is of course the place where to play wit graphics or to do other manipulations if required by the specific task.

Uploading data files to a PROOF cluster

Access to the data to be processed should be provided to the workers using the appropriate means, i.e. by opening access to the required mass storages and/or instrumenting the storage available on the nodes with appropriate systems, e.g. xrootd, and exploiting the related tools, e.g. xrdcp.

Assuming that the chosen mass storage is accessible by the client machine via the standard ROOT tools (TFile::Open, TSystem, ...), there are a couple of simple functions which could be used to upload on this mass storage a set of files provide via a TList or a text file.

These static functions are hosted by TProofMgr and will be described in this section. These functiona are available starting from ROOT development version 5.33/02 .

The TProofMgr::UploadFiles functions

The signatures of the functions are the following:

```
static TFileCollection *UploadFiles(TList *src, const char *mss, const char *dest = 0);
static TFileCollection *UploadFiles(const char *srcfiles, const char *mss, const char *dest = 0);
```

They differ in the first argument.
In the first case the argument is a TList either of TFileInfo or of TObjString. The function will interpret the first URL in TFileInfo or the string in TObjString as the URL of the file to be uploaded.
In the second case the argument is either the path to a text file where the paths to the files to be uploaded are given or a the path of a directory containing the files to be uploaded. In the case of the text file, the must be specified one per line, with lines beginning by '#' being ignored:

```
# The H1 files under "/data/h1" (this line is ignored)
/data/h1/dstarmb.root
/data/h1/dstarp1a.root
/data/h1/dstarp1b.root
/data/h1/dstarp2.root
```

The second and third arguments are the same for the two functions.
The second argument specifies the URL of the destination. This includes the protocol, host, port and possibly the first components of the path. This argument is mandatory.

The third argument specifies the remaining part of the path at destination and it accepts place-holders allowing to modify the final path. The supported place-holders are given in the table:

UploadFiles: supported place-holders to define the destination

,,,...	n-th componet of the sore sub-path
	basename in the source path
	sequential number of the file in the list or text file
	path file in the source path
	local user name
	local group name

This argument is not mandatory and default to '.
As an example, if the source file is

protosrc://host//d0/d1/d2/d3/d4/d5/myfile

dest is

/pool/user////#

mss is

protodst://hostdst//nm/

then the corresponding destination path is

protodst://hostdst//nm/pool/user/d3/d4/d5/99/myfile

Example

This example shows how to upload the files from directory "/data/files" to the MSSS associated with the PROOF cluster at 'master', adding the sequential number to the destination path; the files are then registered as dataset 'mydataset':

```
root [] TProof *proof = TProof::Open("master")
root [] TFileCollection *fc = TProofMgr::UploadFiles("/data/files", proof->Mgr()->GetMssUrl(), ".")
root [] proof->RegisterDataSet("mydataset", fc)
```

Working with data sets

NB: A new dataset management model, designed to solve scalability issues experienced with large number of datasets, has been introduced in 5-34-05 with as concrete example implementation the case of ALICE. The interface seen by the user is basically unchanged; however, some important additions have been done; ALICE users are invited to read the dedicated documentation.

This section describes how to the concept of dataset is translated in ROOT and how to work with datasets in PROOF. When performing repeated analysis of large amounts of data, the dataset PROOF interface allows to save a lot of time during the validation phase by caching the information needed by PROOF.

A dataset is basically a named list of file information. The basic ROOT class to work with is TFileCollection. ROOT provides a dedicated manager class TDataSetManager to handle TFileCollection's, and PROOF provides a full interface to the functionality offered by TDataSetManager.

To be used in PROOF a dataset needs first to be registered; a registered dataset can be verified and its information retrieved. The list of existing datasets can be browsed. More detailed information about the file distribution can also be retrieved. Finally a dataset can be removed.

Registered datasets can be referred *by name* in TProof::Process ; multiple datasets can be processed at once.

Content of this page:

Naming conventions

If, in principle, one can refer to a dataset by a simple string, in reality it is convenient to use the string to uniquely identify a dataset and to pass more information about the way the dataset will be used. A naming convention has therefore been developed. The general form is the following:

Dataset names have the following form:

[[/group/]user/]**dsname**[#[*subdir/*]*objname*][*?enl=entrylist*][*/*]

The first part allows to classify datasets at user and/or group level.

The part after the '#' is only relevant when using the dataset information; '*subdir*' is an optional directory inside the file and '*objname*' the name of the object to be used; '*entrylist*' is either the name of an existing TEntryList object or the path to a file containing the TEntryList to be used.
A few examples:

"mydset"	Analysis of the first TTree in the top directory of the dataset named 'mydset'
"mydset#T"	Analysis of TTree 'T' in the top directory of the dataset named 'mydset'
"mydset#adir/T"	Analysis of TTree 'T' in the 'adir' subdirectory of the dataset named 'mydset'
"mydset#adir/"	Analysis of the first TTree in the 'adir' subdirectory of the dataset named 'mydset'
"mydset	Analysis of the first TTree in the top directory of the dataset named 'mydset' filtered with the TEntryList 'mylist'
"mydset#adir/?enl=mylist.root"	Analysis of the first TTree in the 'adir' subdirectory of the dataset named 'mydset' filtered with the TEntryList from file 'mylist.root'

Dataset handling interface in TProof

The basic API functionality will be described below. For the examples a local PROOF cluster has been used wit the files of the H1 example available form the ROOT HTTP server. The TFileCollection objects used in the examples can be generated with the macro getCollection.C .

Browsing the existing datasets

The first thing done when working with datasets is to browse the existing information: the method TProof::ShowDataSets does that. If we never used datastes before the list will be empty and we get somethign like this:

```
 $ root -l
root [0] TProof *p = TProof::Open("localhost")
...
root [1] p->ShowDataSets()
Dataset repository: /home/ganis/.proof/datasets
Dataset URI                                | # Files | Default tree | # Events |   Disk   | Staged
root [2]
```

The directory used for the repository is shown (default: 'datasets' in the master sandbox) together with the header of the listing. See examples below for less empty outputs.

Registering a dataset

The method TProof::RegisterDataSet is provided to register a dataset on the PROOF master. The method arguments are the name of the dataset, a pointer to the TFileCollection object describing the dataset and an option field. The option field is a string which can contain (a combination of, where meaningful) the following characters:

O	Overwrite existing dataset withe same name, if any
U	Update existing dataset with the information in the TFileCollection object; the new files are added at the end and duplicates are ignored
V	Verify the dataset (in addition to registering)
T	Trust the information contained in the dataset elements
S	Verify the dataset serially (for ROOT >= 5.33/02)

Example: register the H1 files with name 'h1set'

```
root [1] .L getCollection.C+
root [2] TFileCollection *fch1 = getCollection("h1")
root [3] p->RegisterDataSet("h1set", fch1)
(Bool_t)1
root [4] p->ShowDataSets()
Dataset repository: /home/ganis/.proof/datasets
Dataset URI                                | # Files | Default tree | # Events |   Disk   | Staged
/default/ganis/h1set                       |       4 |        N/A             |   190 MB |    0 %
```

The dataset hase been registered under the default group 'default'; the number of files is correct, but remaining information missing or estimated. This will be filled after verification.

Verifying a dataset

Verification is the process of asserting the files of the dataset, finding out their exact location and their content (number of trees, entries for each tree, ...). For a verified dataset PROOF skips the validation step, which represents a significant gain if the number of files is large. For ROOT >= 5.33/02 verification is run in parallel by the workers with a dedicated TSelector (TSelVerifyDataSet). For older versions, verification was run on the master serially.

The method TProof::VerifyDataSet is provided to verify a dataset. The only meaningful arguments is the dataset name; the string option field is ignored up to ROOT 5.32. For newer version it can contain the character 'S' to force serial verification.

Example: verify the dataset 'h1set'

```
root [6] p->VerifyDataSet("h1set")
Mst-0: 13:34:59 25816 Mst-0 | Info in <:scandataset>: opening 4 files that appear to be newly staged
Mst-0: 13:34:59 25816 Mst-0 | Info in <:scandataset>: processing 0.'new' file: http://root.cern.ch/files/h1/dstarmb.root
Mst-0: 13:34:59 25816 Mst-0 | Info in <:scandataset>: processing 1.'new' file: http://root.cern.ch/files/h1/dstarp1a.root
Mst-0: 13:34:59 25816 Mst-0 | Info in <:scandataset>: processing 2.'new' file: http://root.cern.ch/files/h1/dstarp1b.root
Mst-0: 13:34:59 25816 Mst-0 | Info in <:scandataset>: processing 3.'new' file: http://root.cern.ch/files/h1/dstarp2.root
Mst-0: 13:34:59 25816 Mst-0 | Info in <:scandataset>: 4 files 'new'; 0 files touched; 0 files disappeared
(Int_t)0
root [7] p->ShowDataSets()
Dataset repository: /home/ganis/.proof/datasets
Dataset URI                                | # Files | Default tree | # Events |   Disk   | Staged
/default/ganis/h1set                       |       4 | /h42         |   283813 |   264 MB |   100 %
```

Now the name of the TTree, the number of entries and the size of the files are correct.

Showing detailed information about a dataset

The details about a dataset can be viewed with the method TProof::ShowDataSet . The arguments are the dataset name and a string option field. The option field is a string which can contain a combination of the following characters:

M	Display also meta data entries (default)

F	Show details about all the files in the collection

Example: show all details about 'h1set'

```
root [11] p->ShowDataSet("h1set","MF")
TFileCollection h1set -  contains: 4 files with a size of 277298426 bytes, 100.0 % staged - default tree name: '/h42'
The files contain the following trees:
Tree /h42: 283813 events
The collection contains the following files:
Collection name='THashList', class='THashList', size=4
 UUID: 604b445a-0ff7-11df-9717-0101007fbeef
MD5:  d41d8cd98f00b204e9800998ecf8427e
Size: 21330730
 === URLs ===
 URL:  http://root.cern.ch/files/h1/dstarmb.root
 === Meta Data Object ===
 Name:    /h42
 Class:   TTree
 Entries: 21920
 First:   0
 Last:    -1
 UUID: 6064b52a-0ff7-11df-9717-0101007fbeef
MD5:  d41d8cd98f00b204e9800998ecf8427e
Size: 71464503
 === URLs ===
 URL:  http://root.cern.ch/files/h1/dstarp1a.root
 === Meta Data Object ===
 Name:    /h42
 Class:   TTree
 Entries: 73243
 First:   0
 Last:    -1
 UUID: 608005c8-0ff7-11df-9717-0101007fbeef
MD5:  d41d8cd98f00b204e9800998ecf8427e
Size: 83827959
 === URLs ===
 URL:  http://root.cern.ch/files/h1/dstarp1b.root
 === Meta Data Object ===
 Name:    /h42
 Class:   TTree
 Entries: 85597
 First:   0
 Last:    -1
 UUID: 6096e838-0ff7-11df-9717-0101007fbeef
MD5:  d41d8cd98f00b204e9800998ecf8427e
Size: 100675234
 === URLs ===
 URL:  http://root.cern.ch/files/h1/dstarp2.root
 === Meta Data Object ===
 Name:    /h42
 Class:   TTree
 Entries: 103053
 First:   0
 Last:    -1
root [12]
root [12] p->ShowDataSet("h1set","")
TFileCollection h1set -  contains: 4 files with a size of 277298426 bytes, 100.0 % staged - default tree name: '/h42'
```

Retrieving a copy of a dataset

The dataset information, i.e. the corresponding TFileCollection object, can be retrieved with the method TProof::GetDataSet . The arguments are the dataset name and a string option field. The option can be used to select the subset of files served by a given or a list of servers: specify the server(s) (comma-separated) in the option filed.

Removing a dataset

A dataset can be removed with the method TProof::RemoveDataSet . Option filed is currently ignored.

Example: removing 'h1set'

```
root [13] p->RemoveDataSet("h1set")
(Int_t)0
root [14] p->ShowDataSets()
Dataset repository: /home/ganis/.proof/datasets
Dataset URI                               | # Files | Default tree | # Events |   Disk   | Staged
```

Processing datasets by name

Verified datasets can be referred-to by name in TProof::Process . The following example shows how to run the H1 example on the above introduced dataset 'h1set':

```
root [0] TProof *p = TProof::Open("localhost")
```

```
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (4 workers)
Setting up worker servers: OK (4 workers)
PROOF set to parallel mode (4 workers)
root [1] p->ShowDataSets()
Dataset repository: /home/ganis/.proof/datasets
Dataset URI                               | # Files | Default tree | # Events |   Disk   | Staged
/default/ganis/h1set                      |       4 | /h42         |   283813 |   264 MB |  100 %
root [2] p->Process("h1set", "tutorials/tree/h1analysis.C+")
Info in <:begin>: starting h1analysis with process option:
Looking up for exact location of files: OK (4 files)
Looking up for exact location of files: OK (4 files)
Validating files: OK (4 files)
Mst-0: merging output objects ... done
Mst-0: grand total: sent 4 objects, size: 5491 bytes
 FCN=-23769.9 FROM MIGRAD    STATUS=CONVERGED     214 CALLS         215 TOTAL
                     EDM=5.32355e-08    STRATEGY= 1  ERROR MATRIX UNCERTAINTY   1.7 per cent
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           9.60009e+05   9.09409e+04   0.00000e+00  -1.03870e-08
   2  p1           3.51137e-01   2.33454e-02   0.00000e+00   2.83204e-02
   3  p2           1.18504e+03   5.74357e+01   0.00000e+00   2.75574e-06
   4  p3           1.45569e-01   5.50738e-05   0.00000e+00  -5.42172e-01
   5  p4           1.24391e-03   6.38933e-05   0.00000e+00  -1.56632e+00
                               ERR DEF= 0.5
(Long64_t)0
```

To process datasets which are registered but not verified one needs to force validation by setting the parameter **PROOF_LookupOpt** to '**all**'

```
root [] p->SetParameter("PROOF_LookupOpt", "all")
```

This is because, by default, PROOF assumes that the information in the TFileCollection object is valid, and an unverified dataset has all files marked as unstaged, so that no valid files are found in the collection.

Processing many datasets at once

Since the ROOT development version 5.27/02 it is possible to process more than one dataset in one go. There are two options: treat all the datasets as a unique dataset or process them sequentially giving the possibility to the user to keep the results separated. It is also possible to specify a text file from where the names of the datasets to be processed are to be read: the dataset names are specified on one or multiple lines; the lines found are joined as in *grand-dataset* case, unless the file path is followed by a ',' (e.g. p->Process("datasets.txt,",...)) in which case they are treated as in the *keep-separated* case; the file is open in raw mode with TFile::Open(...) and therefore it cane be remote, e.g. on a Web server. The table summarizes options and syntax.

"dset1|dset2|..."	Grand dataset	One set of results
"dset1 dset2 ..." or "dset1,dset2,..."	Keep datasets separated	User can check dedicated bits in the current processed element in the selector to find out when processing of a new dataset starts
"datasets.txt"	Grand dataset (read from file)	The datasets to be processed are read from the text file *datasets.txt*; the lines are joined as in the *grand-dataset* case
"datasets.txt,"	Keep-separated (read from file)	The datasets to be processed are read from the text file *datasets.txt*; the lines are joined as in the *keep-separated* case

To show this at work we use the getCollection.C macro to generate two datasets, 'h1seta' and 'h1setb', which together make the 'h1set' dataset.

```
root [2] TFileCollection *fch1a = getCollection("h1",1,2)
root [3] TFileCollection *fch1b = getCollection("h1",3,2)
root [4] p->RegisterDataSet("h1seta", fch1a, "V")
18:12:15 31667 Mst-0 | Info in <:scandataset>: opening 2 files that appear to be newly staged
18:12:15 31667 Mst-0 | Info in <:scandataset>: processing 0.'new' file: http://root.cern.ch/files/h1/dstarmb.root
18:12:15 31667 Mst-0 | Info in <:scandataset>: processing 1.'new' file: http://root.cern.ch/files/h1/dstarp1a.root
18:12:15 31667 Mst-0 | Info in <:scandataset>: 2 files 'new'; 0 files touched; 0 files disappeared
(Bool_t)1
root [5] p->RegisterDataSet("h1setb", fch1b, "V")
18:12:23 31667 Mst-0 | Info in <:scandataset>: opening 2 files that appear to be newly staged
18:12:23 31667 Mst-0 | Info in <:scandataset>: processing 0.'new' file: http://root.cern.ch/files/h1/dstarp1b.root
18:12:23 31667 Mst-0 | Info in <:scandataset>: processing 1.'new' file: http://root.cern.ch/files/h1/dstarp2.root
18:12:23 31667 Mst-0 | Info in <:scandataset>: 2 files 'new'; 0 files touched; 0 files disappeared
(Bool_t)1
root [6] p->ShowDataSets()
Dataset repository: /home/ganis/.proof/datasets
Dataset URI                               | # Files | Default tree | # Events |   Disk   | Staged
/default/ganis/h1set                      |       4 | /h42         |   283813 |   264 MB |  100 %
/default/ganis/h1seta                     |       2 | /h42         |    95163 |    88 MB |  100 %
/default/ganis/h1setb                     |       2 | /h42         |   188650 |   175 MB |  100 %
```

In the '*grand dataset*' mode processing is fully equivalent to the case seen above:

```
root [7] p->Process("h1seta|h1setb", "tutorials/tree/h1analysis.C+")
Info in <:begin>: starting h1analysis with process option:
Looking up for exact location of files: OK (4 files)
Looking up for exact location of files: OK (4 files)
Validating files: OK (4 files)
Mst-0: merging output objects ... done
Mst-0: grand total: sent 4 objects, size: 5491 bytes
 FCN=-23769.9 FROM MIGRAD    STATUS=CONVERGED     214 CALLS          215 TOTAL
                     EDM=5.32355e-08    STRATEGY= 1  ERROR MATRIX UNCERTAINTY   1.7 per cent
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           9.60009e+05   9.09409e+04   0.00000e+00  -1.03870e-08
   2  p1           3.51137e-01   2.33454e-02   0.00000e+00   2.83204e-02
   3  p2           1.18504e+03   5.74357e+01   0.00000e+00   2.75574e-06
   4  p3           1.45569e-01   5.50738e-05   0.00000e+00  -5.42172e-01
   5  p4           1.24391e-03   6.38933e-05   0.00000e+00  -1.56632e+00
                               ERR DEF= 0.5
(Long64_t)0
```

In the '*keep separated*' mode the result is the same but we see notification for the two runs: each dataset is processed separately with its own packetizer:

```
root [8] p->Process("h1seta h1setb", "tutorials/tree/h1analysis.C+")
Info in : unmodified script has already been compiled and loaded
Info in <:begin>: starting h1analysis with process option:
Looking up for exact location of files: OK (2 files)
Looking up for exact location of files: OK (2 files)
Validating files: OK (2 files)
Looking up for exact location of files: OK (2 files)
Looking up for exact location of files: OK (2 files)
Validating files: OK (2 files)
Mst-0: merging output objects ... done
Mst-0: grand total: sent 4 objects, size: 5491 bytes
Warning in <:constructor>: Deleting canvas with same name: c1
 FCN=-23769.9 FROM MIGRAD    STATUS=CONVERGED     214 CALLS          215 TOTAL
                     EDM=5.32355e-08    STRATEGY= 1  ERROR MATRIX UNCERTAINTY   1.7 per cent
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           9.60009e+05   9.09409e+04   0.00000e+00  -1.03870e-08
   2  p1           3.51137e-01   2.33454e-02   0.00000e+00   2.83204e-02
   3  p2           1.18504e+03   5.74357e+01   0.00000e+00   2.75574e-06
   4  p3           1.45569e-01   5.50738e-05   0.00000e+00  -5.42172e-01
   5  p4           1.24391e-03   6.38933e-05   0.00000e+00  -1.56632e+00
                               ERR DEF= 0.5
(Long64_t)0
```

Entry lists can be applied to each dataset following the syntax described above. In particular, the same dataset can be processed several time in a row with different entry lists.

Accessing the information about the current element

Since ROOT version 5.26/00 the packet being currently processed can be accessed in the selector via the input list. The packet is described by the class *TDSetElement*. In version 5.27/02 new information has been added to this class: bits *TDSetElement::kNewRun* and *TDSetElement::kNewPacket* to flag new runs (new dataset) and new packets, respectively; the name of the dataset being currently processed; a list of files possibly associated to the file being processed (for future use). The *TDSetElement* object if the value of a *TPair* named **PROOF_CurrentElement** available from the input list. The following is an example of how to use this information:

```
//______________________________________________________________________
Bool_t mySelector::Process(Long64_t entry)
{
   // entry is the entry number in the current Tree

   // ...

   // Link to current element, if any
   TPair *elemPair = 0;
   if (fInput && (elemPair = dynamic_cast(fInput->FindObject("PROOF_CurrentElement")))) {
      TDSetElement *fCurrent = dynamic_cast(elemPair->Value());
      if (fCurrent) {
         if (fCurrent->TestBit(TDSetElement::kNewRun)) {
            Info("Process", "entry %lld: starting new run for dataset '%s'",
                             entry, fCurrent->GetDataSet());
         }
         if (fCurrent->TestBit(TDSetElement::kNewPacket)) {
            Info("Process", "entry %lld: new packet from: %s, first: %lld, last: %lld",
                             entry, fCurrent->GetName(), fCurrent->GetFirst(),
                             fCurrent->GetFirst()+fCurrent->GetNum()-1);
         }
      }
   }

   // ...
}
```

Datasets in ROOT: TFileCollection and TFileInfo

The generic definition of a dataset is that of a '*named list of files optionally including some meta information about what they represent*'.

TFileCollection

In ROOT, datasets are described by the class TFileCollection, which derives from TNamed and contains essentially a list of TFileInfo objects, each describing a file, and some meta information about the collection.

TFileCollection objects are typically the result of a query to the experiment catalog. They can also be built out of a plain text file in while URLs for each file are given separated by '\n'.

TFileInfo

TFileInfo is a class providing the most general description of a file in the Grid world: it owns a list of possible locations of the file and a list of meta-information about its content (e.g. name of the tree, number of entries in the tree, etc. etc.).

TFileInfo provides a main constructor taking the main URL and optionally information about the size, the UUID, the MD5 and a first mets-information object. Information can be added/updated at any time.

TMetaInfoData

The meta-information is generically described by lists of TMetaInfoData objects; TMetaInfoData derives from TNamed and currently contains fields for the number of entries, the first and last valid entries, the compressed and uncompressed sizes.

TMetaInfoData provides a main constructor allowing to pass all information in one go, and a special version fo it for TTree objects.

Managing datasets: TDataSetManager

(Page under preparation)

This section describes the use of a class, TDataSetManager, intended to facilitate the handling and management of datasets. Although not abstract, the class describes the interface layer to the concept of dataset. A full implementation using a file system and ROOT files as backend is provided by the derived class TDataSetManagerFile.

Content of this page:

Initializing a dataset manager

The TDataSetManagerFile constructor takes the 'group' and 'user' names and an option field in which the URL for the directory where to store the information can be specified. The directory can also be a remote one: in such a case, depending on the remote permissions, only some functionality (browsing, retrieving) may be available.
On the other end this opens the possibility for centralized dataset information (databases) accessed via HTTP or remote ROOT access protocols.

Dataset manager API

Browsing existing datasets

The first thing one typically does is to check which datasets are available. The ShowDataSets method is available for this purpose.

Getting a dataset

Information about a dataset can be retrieved in the form of a TFileCollection object.

Registering a dataset

To register a new dataset the method RegisterDataSet is provided ...

This methods require writing privileges on the backend.

Scanning a dataset

To scan a dataset means to verify its content, which in turn means staging and opening the files, and to read their headers. During scan the information relevant for processing in PROOF is cached, so that while processing a verified datasets the validation phase is significantly faster.

Removing a dataset

To remove a dataset the method RemoveDataSet is provided ...

This methods require writing privileges on the backend.

Additional functionality

Other available functionality will be described here.

The PQ2 tools

NB: The tools described in this page are not maintained any-longer and the plan is to remove them starting with version 6-16/00. In the unlikely circumstance in which this removal turns out to be disruptive for your workflow, please contact the ROOT team via the Forum, the roottalk digets or the JIRA bug/task/request tracker.

Since ROOT version 5-24/00 a set of scripts to facilitate the interaction with a dataset manager are available under $ROOTSYS/etc/proof/utils/pq2. PQ2 stands for PROOF Quick Query, and echoes the ATLAS tool dq2.

In version 5-27/02 (and 5-26-00-proof) a new version of the scripts (and some new scripts) has been introduced. The scripts are now using a new ROOT executable - pq2 - available under $ROOTSYS/bin. The new (version of the) scripts is described in detail in man pages distributed with ROOT which will be shortly available in this page.

NB: The rest of this page is not yet updated. It gives a feeling of the functionality provided by the scripts but the synatx is not always exact. This will be corrected soon.

There are currently seven scripts, four for browsing the available information and three for modifying the repository. As mentioned in the TDataSetManager page, the repository is a directory, structured per-group and pr-user, with a ROOT file per dataset, containing a TFileCollection object.

The scripts access the repository via a PROOF master. This is not strictly needed (a TDataSetManager object can instantiatied from everywhere on any accessible directory) and future developments may allow more flexible setups.

The scripts requires the URL of the PROOF master which is taken from the command line or from the environment variable PROOFURL.

---

1. Browsing scripts

These scripts allow to browse the content of the repository

1.2. pq2-ls

pq2-ls *[what] [masterurl]*

what
    Selection string: use "*" (with quotes!) for all datasets; use "/*mygroup*/*" to show all datasets belonging to group '*mygroup*'
masterurl
    Entry point for the PROOF master to be used to query the dataset repository

Function
    Display header information about the selected datasets
Arguments
Default
    Display the datasets owned by the client and the ones under COMMON.

Example:

```
$~> echo $PROOFURL
alicecaf.cern.ch:31093
$~> etc/proof/utils/pq2/pq2-ls
Dataset URI                               | # Files | Default tree | # Events |   Disk   | Staged
/COMMON/COMMON/LHC08c11_10TeV_0.5T        |    1850 | /esdTree     |   277500 |  129 GB |  100 %
/COMMON/COMMON/LHC08c12_0.9TeV_0.5T       |    2126 | /esdTree     |   318900 |  101 GB |  100 %
/COMMON/COMMON/LHC08c13_10TeV_0T          |    2312 | /esdTree     |   346500 |  213 GB |   99 %
/COMMON/COMMON/LHC08c14_0.9TeV_0T         |    1646 | /esdTree     |   246900 |   94 GB |  100 %
/COMMON/COMMON/LHC08c15_10TeV_0.5T_Phojet |    1526 | /esdTree     |   222300 |   98 GB |   97 %
/COMMON/COMMON/LHC08c16_0.9TeV_0.5T_Phojet|    1415 | /esdTree     |   201750 |   71 GB |   95 %
/COMMON/COMMON/LHC08c17_10TeV_0T_Phojet   |    1314 | /esdTree     |   180750 |  109 GB |   91 %
/COMMON/COMMON/LHC08c18_0.9TeV_0T_Phojet  |    1554 | /esdTree     |   222750 |   96 GB |   95 %
/COMMON/COMMON/LHC09a4_10TeV              |    6831 | /HLTesdTree  |  1342200 |  486 GB |   98 %
/COMMON/COMMON/LHC09a4_10TeV_200k         |     989 | /HLTesdTree  |   193800 |   70 GB |   97 %
/COMMON/COMMON/LHC09a4_run8010X           |    3042 | /HLTesdTree  |   602200 |  250 GB |   98 %
/COMMON/COMMON/LHC09a4_run8100X           |    2947 | /HLTesdTree  |   586800 |  154 GB |   99 %
/COMMON/COMMON/LHC09a4_run81011           |     989 | /HLTesdTree  |   197600 |   52 GB |   99 %
/COMMON/COMMON/LHC09a4_run8101X           |    7744 | /esdTree     |  1547000 |  408 GB |   99 %
/COMMON/COMMON/LHC09a4_run8158X           |    9683 | /HLTesdTree  |  1932000 |  511 GB |   99 %
/COMMON/COMMON/LHC09a4_run8159X           |    5745 | /HLTesdTree  |  1146800 |  303 GB |   99 %
/COMMON/COMMON/LHC09a5_run9000X           |    8125 | /HLTesdTree  |  1611400 |  450 GB |   99 %
/COMMON/COMMON/LHC09a6_run9200X           |    5791 | /HLTesdTree  |  1141200 |  486 GB |   98 %
/COMMON/COMMON/tutorial_small             |     200 | /esdTree     |    20000 |    8 GB |  100 %
/proofteam/ganis/LHC09a4_10TeV-dev        |    6831 | /esdTree     |  1341000 |  486 GB |   98 %
/proofteam/ganis/LHC09a4_10TeV_a          |    3126 | /esdTree     |   614800 |  222 GB |   98 %
/proofteam/ganis/LHC09a4_run8010X_a       |    1387 | /esdTree     |   276200 |  114 GB |   99 %
/proofteam/ganis/LHC09a4_run8100X_a       |    1321 | /esdTree     |   263600 |   69 GB |   99 %
/proofteam/ganis/LHC09a4_run81011_a       |     451 | /esdTree     |    90200 |   24 GB |  100 %
/proofteam/ganis/LHC09a4_run8101X_a       |    3526 | /esdTree     |   705200 |  185 GB |  100 %
/proofteam/ganis/LHC09a4_run8158X_a       |    4005 | /esdTree     |   801000 |  211 GB |  100 %
/proofteam/ganis/LHC09a4_run8159X_a       |    2350 | /esdTree     |   470000 |  124 GB |  100 %
/proofteam/ganis/LHC09a5_run9000X_a       |    3360 | /esdTree     |   670400 |  186 GB |   99 %
/proofteam/ganis/LHC09a6_run9200X_a       |    2404 | /esdTree     |   477600 |  201 GB |   99 %
/proofteam/ganis/ds-test                  |       4 | /esdTree     |      400 |  157 MB |  100 %
/proofteam/ganis/ds-test-loc              |       4 | /esdTree     |      400 |  171 MB |  100 %
$~>
```

1.3. pq2-ls-files

pq2-ls-files *datasetname [masterurl]*

datasetname
    Name of the dataset whose files have to be listed
masterurl
    Entry point for the PROOF master to be used to query the dataset repository

Function
    Display the file content of a dataset
Arguments
Default
    None

Example:

```
$~> etc/proof/utils/pq2/pq2-ls-files ds-test
pq2-ls-files: dataset 'ds-test' has 4 files
pq2-ls-files:      #   File                                                                                Size   #Objs Obj|Type|Entries, ...
pq2-ls-files:      1   root://lxfsrd0507.cern.ch//alien/alice/sim/PDC_08/LHC08b1/300000/1086/root_archive.zip#AliESDs.root       39 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
```

```
pq2-ls-files:      2   root://lxfsrd0507.cern.ch//alien/alice/sim/PDC_08/LHC08b1/300000/1088/root_archive.zip#AliESDs.root       40 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files:      3   root://lxfsrd0508.cern.ch//alien/alice/sim/PDC_08/LHC08b1/300000/1092/root_archive.zip#AliESDs.root       38 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files:      4   root://lxfsrd0509.cern.ch//alien/alice/sim/PDC_08/LHC08b1/300000/1094/root_archive.zip#AliESDs.root       38 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
$~>
```

1.4. pq2-ls-files-server

pq2-ls-files-server *datasetname server [masterurl]*

datasetname
: Name of the dataset whose files have to be listed

server
: Name of the server for which the information is wanted; can be in URL form

masterurl
: Entry point for the PROOF master to be used to query the dataset repository

Function
: Display information about the files of dataset available on the specified server

Arguments

Default
: None

Example:

```
$~> etc/proof/utils/pq2/pq2-ls-files-server /COMMON/COMMON/tutorial_small lxfsrd0509
pq2-ls-files-server: dataset '/COMMON/COMMON/tutorial_small' has 15 files on server lxfsrd0509
pq2-ls-files-server:      #   File                                                                   Size   #Objs Obj|Type|Entries, ...
pq2-ls-files-server:      1   /alien/alice/sim/PDC_08/LHC08b1/300000/022/root_archive.zip#AliESDs.root              36 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      2   /alien/alice/sim/PDC_08/LHC08b1/300000/023/root_archive.zip#AliESDs.root              42 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      3   /alien/alice/sim/PDC_08/LHC08b1/300000/024/root_archive.zip#AliESDs.root              41 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      4   /alien/alice/sim/PDC_08/LHC08b1/300000/035/root_archive.zip#AliESDs.root              44 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      5   /alien/alice/sim/PDC_08/LHC08b1/300000/056/root_archive.zip#AliESDs.root              45 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      6   /alien/alice/sim/PDC_08/LHC08b1/300000/1001/root_archive.zip#AliESDs.root             35 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      7   /alien/alice/sim/PDC_08/LHC08b1/300000/1011/root_archive.zip#AliESDs.root             46 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      8   /alien/alice/sim/PDC_08/LHC08b1/300000/1020/root_archive.zip#AliESDs.root             47 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:      9   /alien/alice/sim/PDC_08/LHC08b1/300000/1027/root_archive.zip#AliESDs.root             41 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:     10   /alien/alice/sim/PDC_08/LHC08b1/300000/1081/root_archive.zip#AliESDs.root             47 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:     11   /alien/alice/sim/PDC_08/LHC08b1/300000/1087/root_archive.zip#AliESDs.root             45 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:     12   /alien/alice/sim/PDC_08/LHC08b1/300000/109/root_archive.zip#AliESDs.root              41 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:     13   /alien/alice/sim/PDC_08/LHC08b1/300000/1090/root_archive.zip#AliESDs.root             48 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:     14   /alien/alice/sim/PDC_08/LHC08b1/300000/1091/root_archive.zip#AliESDs.root             40 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
pq2-ls-files-server:     15   /alien/alice/sim/PDC_08/LHC08b1/300000/1094/root_archive.zip#AliESDs.root             38 MB      2  esdTree|TTree|100,HLTesdTree|TTree|100
$~>
```

1.5. pq2-info-server

pq2-info-server *server [masterurl]*

server
: Name of the server for which the information is wanted

masterurl
: Entry point for the PROOF master to be used to query the dataset repository

Function
: Display information about the datasets on a given server

Arguments

Default
: None

Example:

```
$~> etc/proof/utils/pq2/pq2-info-server lxfsrd0509
+++
+++ Dumping: subset of 'LHC08c12_0.9TeV_0.5T' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 175 files, 8 GB, staged 100 %, tree: /esdTree, 26250 entries, 8.1 % of total
+++
+++
+++ Dumping: subset of 'LHC08c13_10TeV_0T' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 212 files, 19 GB, staged 100 %, tree: /esdTree, 31800 entries, 9.1 % of total
+++
+++
+++ Dumping: subset of 'LHC08c11_10TeV_0.5T' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 151 files, 10 GB, staged 100 %, tree: /esdTree, 22650 entries, 8.1 % of total
+++
+++
+++ Dumping: LHC09a4 10TeV Pythia produced with v4-16-Rev-08 (subset on server root://lxfsrd0509.cern.ch:1094):
+++ Summary: 221 files, 11 GB, staged 100 %, tree: /HLTesdTree, 44200 entries, 7.5 % of total
+++
+++
+++ Dumping: subset of 'LHC09a4_81011' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 75 files, 4 GB, staged 100 %, tree: /HLTesdTree, 15000 entries, 7.6 % of total
+++
+++
+++ Dumping: LHC09a4 10TeV Pythia produced with v4-16-Rev-07 (subset on server root://lxfsrd0509.cern.ch:1094):
+++ Summary: 231 files, 19 GB, staged 99 %, tree: /HLTesdTree, 45800 entries, 7.6 % of total
+++
+++
+++ Dumping: subset of 'LHC09a4_run8101X' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 584 files, 31 GB, staged 100 %, tree: /esdTree, 116800 entries, 7.5 % of total
+++
+++
+++ Dumping: subset of 'LHC09a4_run8158X' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 792 files, 41 GB, staged 99 %, tree: /HLTesdTree, 158000 entries, 8.1 % of total
+++
+++
+++ Dumping: subset of 'LHC09a4_run8159X' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 459 files, 23 GB, staged 99 %, tree: /HLTesdTree, 91600 entries, 7.9 % of total
+++
+++
+++ Dumping: subset of 'LHC09a6_run9200X' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 477 files, 40 GB, staged 98 %, tree: /HLTesdTree, 94200 entries, 8.2 % of total
+++
+++
+++ Dumping: subset of 'LHC08c14_0.9TeV_0T' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 136 files, 7 GB, staged 100 %, tree: /esdTree, 20400 entries, 8.2 % of total
+++
+++
+++ Dumping: subset of 'LHC08c15_10TeV_0.5T_Phojet' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 121 files, 7 GB, staged 100 %, tree: /esdTree, 18150 entries, 7.9 % of total
+++
+++
+++ Dumping: subset of 'tutorial_small' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 15 files, 644 MB, staged 100 %, tree: /esdTree, 1500 entries, 7.5 % of total
+++
+++
+++ Dumping: subset of 'LHC08c17_10TeV_0T_Phojet' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 90 files, 7 GB, staged 100 %, tree: /esdTree, 13500 entries, 6.8 % of total
+++
+++
+++ Dumping: subset of 'LHC09a4_10TeV' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 521 files, 37 GB, staged 98 %, tree: /HLTesdTree, 102600 entries, 7.6 % of total
```

```
+++
+++
+++ Dumping: subset of 'LHC09a4_10TeV_200k' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 75 files, 5 GB, staged 97 %, tree: /HLTesdTree, 14600 entries, 7.7 % of total
+++
+++
+++ Dumping: subset of 'LHC08c16_0.9TeV_0.5T_Phojet' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 103 files, 5 GB, staged 100 %, tree: /esdTree, 15450 entries, 7.2 % of total
+++
+++
+++ Dumping: subset of 'LHC08c18_0.9TeV_0T_Phojet' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 116 files, 7 GB, staged 100 %, tree: /esdTree, 17400 entries, 7.4 % of total
+++
+++
+++ Dumping: subset of 'LHC09a5_run9000X' on server root://lxfsrd0509.cern.ch:1094:
+++ Summary: 657 files, 36 GB, staged 99 %, tree: /HLTesdTree, 131000 entries, 8.1 % of total
+++
$~>
```

2. Modifiers scripts

These scripts allow to modify the content of the repository; they may fail if the remote settings do not allow modification.

2.1. pq2-put

pq2-put *datasetfile [masterurl]*

datasetfile
: Path to the file with the list of files in the dataset or directory with the files containing the file lists of the datasets to be registered; in the first case wildcards '*' can be specified in the file name, i.e. "

    /fil*" is ok but "

    /*/file" is not. In all cases the name of the dataset is the name of the file used, e.g. "

    /mydset" will register a dataset called "mydset".

masterurl
: Entry point for the PROOF master to be used to query the dataset repository

Function
: Register one or more datasets

Arguments

Default
: None

Example:

```
$~> etc/proof/utils/pq2/pq2-put ~/local/root/test/proof/examples/h1/h1-http.txt
pq2-put: 1 dataset(s) registered
$~> etc/proof/utils/pq2/pq2-ls
Dataset URI                                | # Files | Default tree | # Events |   Disk   | Staged
...
/proofteam/ganis/h1-http.txt               |       4 |       N/A              |    190 MB |    0 %
$~>
```

2.2. pq2-verify

pq2-verify *datasetname [masterurl]*

datasetname
: Name of the dataset to be removed; it accepts the '*' wild card: in such a case the full path - as shown by pq2-ls - should be given in quotes, e.g. "/default/ganis/h1-set5*

masterurl
: Entry point for the PROOF master to be used to query the dataset repository

Function
: Verify the content of one or more datasets

Arguments

Default
: None

Example:

```
$~> etc/proof/utils/pq2/pq2-verify /proofteam/ganis/h1-http.txt
13:31:40 15348 Mst-0 | Info in <:scandataset>: processing 1.'new' file: http://root.cern.ch/files/h1/dstarmb.root
pq2-verify: 1 dataset(s) verified
$~> etc/proof/utils/pq2/pq2-ls
Dataset URI                                | # Files | Default tree | # Events |   Disk   | Staged
...
/proofteam/ganis/h1-http.txt               |       4 | /h42         |   283813 |   264 MB |  100 %
$~> etc/proof/utils/pq2/pq2-ls-files /proofteam/ganis/h1-http.txt
pq2-ls-files: dataset '/proofteam/ganis/h1-http.txt' has 4 files
pq2-ls-files:      #   File                                                                                      Size   #Objs Obj|Type|Entries, ...
pq2-ls-files:      1   http://root.cern.ch/files/h1/dstarmb.root                                                20 MB      1  h42|TTree|21920
pq2-ls-files:      2   http://root.cern.ch/files/h1/dstarp1a.root                                               68 MB      1  h42|TTree|73243
pq2-ls-files:      3   http://root.cern.ch/files/h1/dstarp1b.root                                               79 MB      1  h42|TTree|85597
pq2-ls-files:      4   http://root.cern.ch/files/h1/dstarp2.root                                                96 MB      1  h42|TTree|103053
$~>
```

2.3. pq2-rm

pq2-rm *datasetname [masterurl]*

datasetname
: Name of the dataset to be removed; it accepts the '*' wild card: in such a case the full path - as shown by pq2-ls - should be given in quotes, e.g. "/default/ganis/h1-set5*

masterurl
: Entry point for the PROOF master to be used to query the dataset repository

Function
: Remove one or more datasets

Arguments

Default
: None

Example:

```
$~>  etc/proof/utils/pq2/pq2-rm /proofteam/ganis/h1-http.txt
pq2-rm: 1 dataset(s) removed
$~>
```

Preparing, uploading and enabling additional software

The additional software required by an analysis can be made available to the PROOF sessions either in the form of a macro to be loaded via the interpreter or via ACLiC or in the form of a gzipped tar file. All the relevant information is given in the following subsections.

Loading a macro or a class

Loading a macro or a class

A functionality similar to '.L' on the ROOT shell is available in PROOF (since v5.15.04) via

```
TProof::Load(const char *macro, Bool_t notOnClient = kFALSE)
```

The first argument contains the path to the file implementing the macro. This file may have any extension. If existing, a file with the same name and extension .h or .hh is also uploaded to the system. The second argument can be used to avoid a local enabling of the macro.

As an example, we show how to enable the Event class using this technology:

```
root [0] TProof *p = TProof::Open("alicecaf.cern.ch:41093")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (13 workers)
Setting up worker servers: OK (13 workers)
PROOF set to parallel mode (13 workers)
(class TProof*)0x22dab40
root [1] p->Load("test/Event.cxx+")
08:41:41 30591 Mst-0 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 30591 Mst-0 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 31872 Wrk-0.8 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 31872 Wrk-0.8 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41  1799 Wrk-0.0 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41  1799 Wrk-0.0 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 24025 Wrk-0.9 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 24025 Wrk-0.9 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 20838 Wrk-0.11 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 20838 Wrk-0.11 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 12673 Wrk-0.1 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 12673 Wrk-0.1 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 17015 Wrk-0.5 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 17015 Wrk-0.5 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 32426 Wrk-0.4 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 32426 Wrk-0.4 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41   966 Wrk-0.3 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41   966 Wrk-0.3 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 29938 Wrk-0.7 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 29938 Wrk-0.7 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 30450 Wrk-0.12 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 30450 Wrk-0.12 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 21695 Wrk-0.2 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 21695 Wrk-0.2 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41 11688 Wrk-0.10 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41 11688 Wrk-0.10 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
08:41:41  9690 Wrk-0.6 | Info in <:handlecache>: loading macro Event.cxx+ ...
08:41:41  9690 Wrk-0.6 | Info in <:aclic>: creating shared library <...> ./Event_cxx.so
(Int_t)0
root [2]
```

Working with packages (PAR files)

Working with packages (PAR files)

The PAR file technology addresses the problem of uploading and enabling a *package* , a set of files which, after proper enabling, provide additional, usually well defined, functionality. In this section we describe what is a PAR file and the TProof interface to work with these files. The interface has been somewhat simplified starting with ROOT 5.27/02, where the automatism to download the needed packages has been introduced. The situation up to 5.26/00 (and related patches) is described in the last part.

Content of this section:

To be used the packages must have been built in dedicated directories on the client and on the cluster nodes. Each user has its own package directory on the client (default $HOME/.proof/packages) and on the cluster (default /packages). Both clients and cluster administrators can defined global package directories, where stable packages of public interest could go.

The ':' separated list of global package directories are defined via the rootrc environment variable Proof.GlobalPackageDirs:

```
Proof.GlobalPackageDirs  /pool/globalpack-PH1:/pool/globalpack-PH2
```

Each global package directory gets automatically labeled with 'Gn' where 'n' is the order of appearance in the list (e.g. G0 will refer to '/pool/globalpack-PH1' in the example above). This label is used to address a specific package when working with the PROOF package interface described below. Of course, any operation which modifies the directory content requires the appropriate privileges; so, typically, only a subset of people can modify the global package directories.

Before continuing we need to have a closer look to what a PAR file is.

1. What is a PAR file?

A PAR file is an archive containing all the files needed to build and setup a package. A package is uniquely identified by its name *packname* . The PAR file *packname.par* of a package is a gzipped tarball of a well defined directory structure:

- *packname*
  Source/binary files, header files, makefile
- *packname* /PROOF-INF
  a. BUILD.sh
     Script to build the package; typically if executes 'make' in the upstream source directory; may be missing.
  b. SETUP.C
     Macro to enable the package; this is the place where to handle the dependencies .

A simple example of PAR file can generate following the instructions in $ROOTSYS/test/ProofBench/README . The resulting event.par has the following content

```
pcepsft43:~$ tar tzvf $ROOTSYS/test/ProofBench/event.par
drwxr-xr-x ganis/sf          0 2007-02-21 19:38:16 event/
-rw-r--r-- ganis/sf      14712 2007-02-21 19:38:16 event/Event.cxx
-rw-r--r-- ganis/sf       7873 2007-02-21 19:38:16 event/Event.h
-rw-r--r-- ganis/sf        259 2007-02-21 19:38:16 event/EventLinkDef.h
-rw-r--r-- ganis/sf      13440 2007-02-21 19:38:16 event/Makefile
-rw-r--r-- ganis/sf      13390 2007-02-21 19:38:16 event/Makefile.arch
drwxr-xr-x ganis/sf          0 2007-02-21 19:38:16 event/PROOF-INF/
-rwxr-xr-x ganis/sf         29 2007-02-21 19:38:16 event/PROOF-INF/BUILD.sh
-rw-r--r-- ganis/sf         47 2007-02-21 19:38:16 event/PROOF-INF/SETUP.C
```

2. The BUILD.sh script

If present, the BUILD.sh script is executed (via TSystem->Exec(...)) to build the package. The return code of the script is checked for failures with 0 indicating success and != 0 indicating failure. The example shows the BUILD.sh script in the 'tutorials/proof/event.par' example package.

```
#! /bin/sh
# Build libEvent library.

if test ! "x$ROOTPROOFLITE" = "x"; then
   echo "event-BUILD: PROOF-Lite node (session has $ROOTPROOFLITE workers)"
elif test ! "x$ROOTPROOFCLIENT" = "x"; then
   echo "event-BUILD: PROOF client"
else
   echo "event-BUILD: standard PROOF node"
fi

if [ "" = "clean" ]; then
   make distclean
   exit 0
fi

make
rc=$?
echo "rc=$?"
if [ $? != "0" ] ; then
   exit 1
fi
exit 0
```

The BUILD.sh must of course be executable. The environment variables ROOTPROOFLITE and ROOTPROOFCLIENT allow to find out whether the BUILD.sh is executed in PROOF-Lite or a client session, respectively, so that dedicated actions may be taken if needed. They are available starting from ROOT 5.28/00a .

3. The SETUP.C function

If present, the SETUP.C function is executed to setup the package; this includes all the settings needed to properly use the package (e.g. dependency library loading). The basic signature of the SETUP() function is

```
Int_t SETUP()
```

To properly stop the execution flow the function should return -1 on failure and 0 on success.
Starting with ROOT version 5.27/04 to additional signatures are supported for SETUP():

```
Int_t SETUP(const char *opt)
Int_t SETUP(TList *optls)
```

The 'opt' string or the 'optls' object list can be passed via TProof::EnablePackage (see below).
The example shows the SETUP.C function in the 'tutorials/proof/event.par' example package.

```
Int_t SETUP()
{
   if (gSystem->Getenv("ROOTPROOFLITE")) {
      Printf("event-SETUP: PROOF-Lite node (session has %s workers)",
                                 gSystem->Getenv("ROOTPROOFLITE"));
   } else if (gSystem->Getenv("ROOTPROOFCLIENT")) {
      Printf("event-SETUP: PROOF client");
   } else {
      Printf("event-SETUP: standard PROOF node");
   }
   if (gSystem->Load("libEvent") == -1)
      return -1;
   return 0;
}
```

Again, the environment variables ROOTPROOFLITE and ROOTPROOFCLIENT (available starting from ROOT 5.28/00a) allow to find out whether the SETUP.C is executed in PROOF-Lite or a client session, respectively, so that dedicated actions may be taken if needed.

4. Listing available packages

Once a PROOF session has been started, one of the first things to do is to see which packages are already avalaible on the cluster. This is achieved by issuing a call to

```
TProof::ShowPackages(Bool_t all = kFALSE)
```

for example:

```
root [0] p = TProof::Open("alicecaf.cern.ch:41093")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (13 workers)
Setting up worker servers: OK (13 workers)
PROOF set to parallel mode (13 workers)
(class TProof*)0x2230da0
root [1] p->ShowPackages()
*** Package cache client:/home/ganis/.proof/packages ***
total 32
drwxr-xr-x 3 ganis ganis  4096 2010-03-29 14:29 ANALYSIS
lrwxrwxrwx 1 ganis ganis    50 2010-03-29 14:29 ANALYSIS.par -> /home/ganis/.proof/packages/downloaded/ANALYSIS.par
drwxr-xr-x 3 ganis ganis 12288 2010-03-29 14:28 ESD
lrwxrwxrwx 1 ganis ganis    45 2010-03-29 14:28 ESD.par -> /home/ganis/.proof/packages/downloaded/ESD.par
drwxr-xr-x 3 ganis ganis  4096 2010-03-29 14:30 event
lrwxrwxrwx 1 ganis ganis    59 2010-03-29 14:29 event.par -> /home/ganis/local/root/trunk/root/tutorials/proof/event.par
drwxr-xr-x 3 ganis ganis 12288 2010-03-29 12:37 STEERBase
lrwxrwxrwx 1 ganis ganis    51 2010-03-29 14:27 STEERBase.par -> /home/ganis/.proof/packages/downloaded/STEERBase.par

*** Global Package cache G0 lxfsrd0506.cern.ch:/pool/proofbox-xpd3/alicecaf/packages ***
total 428
drwxr-xr-x  3 alicecaf z2    690 Mar 29 13:54 ANALYSIS
-rw-r--r--  1 alicecaf z2  40517 Mar 29 13:52 ANALYSIS.par
drwxr-xr-x  3 alicecaf z2  16384 Mar 29 13:54 ESD
-rw-r--r--  1 alicecaf z2 193055 Mar 29 13:52 ESD.par
drwxr-xr-x  3 alicecaf z2  16384 Mar 29 13:53 STEERBase
-rw-r--r--  1 alicecaf z2 165116 Mar 29 13:52 STEERBase.par

*** Package cache lxfsrd0506.cern.ch:/pool/proofbox-xpd3/ganis/packages ***
total 12
drwxr-xr-x  3 ganis sf  196 Mar 29 14:54 event
-rw-r--r--  1 ganis sf 9505 Mar 29 14:54 event.par
root [2]
```

The argument controls whether the command has to be executed also on the workers or only on the client and master; the workers should have the same packages as the master, so the default is 'false'. In the above example we see that locally there is no global directory on the client, that there is a global directory 'G0' at /pool/proofbox-xpd3/alicecaf/packages on the cluster with three packages, and that there is only one private package 'event' available on the cluster.

3. The client *downloaded* cache

Starting with ROOT 5.27/02, clients may download the packages needed by the query from the master. The PAR files for the downloaded packages are cached under the *'downloaded'* sub-directory of the package directory (default $HOME/.proof/packages/downloaded'). The downloaded packages are automatically updated whenever the copy on the master is updated. If the client uploads a private package with same name (for example a privately modified version of an existing package), the download cache is automatically cleaned of the related package so that the automatic cross-check for updates in the master is disabled.

In the example above, the packages *STEERBase*, *ESD* and *ANALYSIS* have been downloaded from the master, while the package *event* has been uploaded by the client.

4. Enabling a package

A package that ShowPackages() lists as available on the cluster can be enabled (i.e. loaded - and built, if needed) by calling one of the following three methods:

```
TProof::EnablePackage(const char *packname, Bool_t notOnClient = kFALSE)
TProof::EnablePackage(const char *packname, TList *optls = 0, Bool_t notOnClient = kFALSE)
TProof::EnablePackage(const char *packname, const char *opt, Bool_t notOnClient = kFALSE)
```

The first argument contains the name of the package (if present, the extension '.par' is automatically stripped off). The 'notOnClient' argument can be used to avoid enabling the package locally (i.e. on the client). The 'opt' or 'optls' arguments allow to specify, respectively, a string or a list of objects to be passed to the SETUP function (see above).

A package not available yet on the client is downloaded automatically in the client local .download cache. A package that was previously downloaded (i.e. whose PAR file is stored locally in the .download cache) is checked for updates and eventually downloaded again. When a package is downloaded from the master repository, any local reference to the package is removed and a full build started again.

On the cluster, enabling of a package is done in parallel: the master first forwards the enabling request to the worker nodes and then starts its own build. The logs from the building operations on the master are feedback to the client in real-time; those from the worker nodes come after.

The list of enabled packages can be displayed with

```
TProof::ShowEnabledPackages(Bool_t all = kFALSE)
```

Again, only the packages enabled on the master are shown by default. If 'all' is KTRUE the requests are run also on the worker nodes.

For example:

```
root [0] p = TProof::Open("alicecaf.cern.ch:41093")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (13 workers)
Setting up worker servers: OK (13 workers)
PROOF set to parallel mode (13 workers)
(class TProof*)0x20fb140
root [1] p->ShowEnabledPackages()
*** Enabled packages on client on pcphsft64
*** Enabled packages ***
root [2] p->EnablePackage("STEERBase")
Info in <:downloadpackage>: 'STEERBase' cross-checked against master repository (local path: /home/ganis/.proof/packages/.download/STEERBase.par)
make: `libSTEERBase.so' is up to date.
Mst-0: building STEERBase ...
Mst-0: make: `libSTEERBase.so' is up to date.
Wrk-0.0: building STEERBase ...
Wrk-0.0: make: `libSTEERBase.so' is up to date.
...
(Int_t)0
root [3] p->EnablePackage("ESD")
Info in <:downloadpackage>: 'ESD' cross-checked against master repository (local path: /home/ganis/.proof/packages/.download/ESD.par)
make: `libESD.so' is up to date.
Mst-0: building ESD ...
Mst-0: make: `libESD.so' is up to date.
Wrk-0.6: building ESD ...
Wrk-0.6: make: `libESD.so' is up to date.
...
(Int_t)0
root [4] p->EnablePackage("ANALYSIS")
Info in <:downloadpackage>: 'ANALYSIS' cross-checked against master repository (local path: /home/ganis/.proof/packages/.download/ANALYSIS.par)
make: Nothing to be done for `default-target'.
Mst-0: building ANALYSIS ...
Mst-0: make: Nothing to be done for `default-target'.
Wrk-0.11: building ANALYSIS ...
Wrk-0.11: make: Nothing to be done for `default-target'.
...
(Int_t)0
root [5] p->ShowEnabledPackages()
*** Enabled packages on client on pcphsft64
STEERBase
ESD
ANALYSIS
*** Enabled packages ***
STEERBase
ESD
ANALYSIS
root [6]
```

5. Downloading a package

As mentioned in the discussion of TProof::EnablePackage, packages are automatically downloaded when needed. However, one can force download of a package issuing directly the dedicated method:

```
TProof::DownloadPackage(const char *par, const char *dstdir = 0)
```

The default destination directory is the *downloaded* sub-directory in the packages directory (default $HOME/.proof/packages/downloaded). This functionality allows to download packages from the master repository at user will, for example it allows to create a local repository of packages for further distribution.

6. Uploading a package

A new package (or a new version of a package) can be uploaded to the session via

```
TProof::UploadPackage(const char *parfile, EUploadPackageOpt opt = kUntar)
```

The first argument contains the path to the PAR file (with or without '.par' extension). The first time a PAR file is uploaded, it is cached by master and worker sessions in the user's sandbox. The next time the user requests the upload, the MD5 of the PAR file on the farm nodes is checked and the file is uploaded only if changed.

The second argument indicates the way the new version of and existing PAR file has to be treated. By default PROOF just untars the PAR file, so that binaries of unchanged source files can be re-used during build. if 'opt' is set to 'kRemoveOld' then the exiting package directory is removed from the sandbox.

The 'upload' functionality is useful for private packages or for modified version of common packages.

7. Removing packages

A package can be removed with

```
TProof::ClearPackage(const char *packname)
```

This operation deletes both the building directory and the PAR file, both on the client and the cluster. Non backup is provided for cleared packages, so this method has to be used with care. After TProof::ClearPackage an TProof::UploadPackage is needed before re-enabling the package.

The following removes all installed packages

```
TProof::ClearPackages()
```

8. Differences in ROOT version 5.26/00 and older

The functionality in the previous sub-sections is fully available only starting from ROOT version 5.27/02. The DownloadPackage and related functionality (automatic download; client *downloaded* cache) are missing from previous versions. DownloadPackage uses the sandbox access interface introduced in ROOT 5.25/02, which therefore can be used in ROOT 5.26/00 to retrieve the relevant files. Furthermore, in ROOT 5.27/02 a few bugs affecting the usage of global package directories have been fixed. These fixes have been included in the patch version 5.26/00c .

Creating a PAR package to read a given ROOT file

Starting with version 5.32/00 it is possible to easily create a PAR file to read the content of data stored into a ROOT file. This is done via TFile::MakeProject, exploiting the fact that ROOT files are self-describing and contain enough information to read what is stored. The method MakeProject of TFile has been augmented with the possibility to store the relevant information in form of a PAR file. To get the PAR file just open the file in READ mode

```
root [] TFile *f = TFile::Open("/path/to/myfile.root")
```

and run MakeProject in this way:

```
root [] f->MakeProject("/path/with/packages/mypack", "*", "par")
```

This will create *mypack.par* under */path/with/packages*. The created PAR file is ready to be used with PROOF.

The following creates the equivalent of *tutorials/proof/event.par* :

```
$ root -l
root [0] TFile * f = TFile::Open("http://root.cern.ch/files/data/event_1.root")
Warning in <:tclass>: no dictionary for class Event is available
Warning in <:tclass>: no dictionary for class EventHeader is available
Warning in <:tclass>: no dictionary for class Track is available
root [1] f->MakeProject("packages/myevent", "*", "par")
MakeProject has generated 3 classes in packages/myevent
Files Makefile, Makefile.arch, PROOF-INF/BUILD.sh and PROOF-INF/SETUP.C have been generated under 'packages/myevent'
myevent/
myevent/EventHeader.h
myevent/Event.h
myevent/myeventProjectInstances.h
myevent/myeventProjectHeaders.h
myevent/PROOF-INF/
myevent/PROOF-INF/BUILD.sh
myevent/PROOF-INF/SETUP.C
myevent/myeventLinkDef.h
myevent/Makefile.arch
myevent/Track.h
myevent/Makefile
myevent/myeventProjectSource.cxx
Info in <:makeproject>: PAR file packages/myevent.par generated
root [2] .q
$  ls packages/
myevent  myevent.par
$ ls packages/myevent
Event.h        Makefile       myeventLinkDef.h         myeventProjectInstances.h  PROOF-INF
EventHeader.h  Makefile.arch  myeventProjectHeaders.h  myeventProjectSource.cxx   Track.h
$ ls packages/myevent/PROOF-INF/
BUILD.sh  SETUP.C
```

Handling outputs

Content:

---

1. **Introduction**

By default PROOF, the output of a PROOF query is kept in memory and available via the output list. It is known that for large outputs this can be problematic. The solution proposed by PROOF to this problem is to use files to swap objects from memory; the files can be either merged at the end or accessed via a global common view via a dataset.This technology provides indeed a good solution to the problem; however, it turned out to be difficult to setup for the average user.

To simplify access to this technology in particular - and to output handling more in general - a new set of options have been added to TProof::Process. These new options are the subject of these pages. They are available in the trunk (PROOF-Lite support starting from r45632) and in the 5.32 and 5.34 patch branches, starting from tags 5.34/02 and 5.32/05 .

2. **Addressed use-cases**

One of the more frequent PROOF user questions is how to save to file the results of a run. This is not strictly connected to the technology used to handle the output but more with the fact the Terminate() method is not much used, if not to save the results to a file. Therefore a quick way to define a file where to save the results without having to re-implement the same code in each TSelector would certainly be welcome by may users. Another missing functionality is the possibility to save the partial results while processing , so that in case of a crash, not all is lost and can be partially recovered.

The options described in this page allow to control via the option field of TProof::Process the following cases:

1. Define an output file where to save all the objects which are not already saved in other output files;
2. Decide if the merging process to create the output file happens in memory or via file;
3. Decide if partial results have to saved after each packet
4. Give the possibility to the cluster administrator to control file-saving by setting a memory threshold above which object swapping to file is done whatever the user's setting will be;
5. In alternative to file merging, give the possibility to create a dataset with the files created on the nodes; the user can then decide what to do with the dataset.

3. **Client-side options**

3.1. *Enable save-to-file technology*

The keyword 'stf' or 'savetofile' can be used in the option field to force merging via file. By default the final file is kept in the user data directory on the master. For example, this is how the 'ProofSimple' tutorial looks like when this option is passed:

```
root [1] p->SetParameter("ProofSimple_NHist", (Long_t) 16) // NB: ProofSimple_NHist is needed by the ProofSimple tutorial, not by the safe-to-file functionality
root [2] p->Process("tutorials/proof/ProofSimple.C+", 40000000, "stf")
Mst-0: merging output objects ... done
Output file: rootd://proofadm@cernvm24.cern.ch:1093//home/proofadm/PEAC/proof/proofbox/ganis/data/0/cernvm24-1343918756-15791/output-cernvm24-1343918756-15791.q1.root
Mst-0: grand total: sent 8 objects, size: 1405 bytes
ntuple opts: 0  0
(Long64_t)0
```

Internally, PROOF creates a TProofOutputFile object and adds it to the output list:

```
root [3] p->GetOutputList()->Print()
Collection name='TList', class='TList', size=1
Info in <:print>: -------------- output-cernvm24-1343918756-15791.q1.root : start (cernvm28.cern.ch:1093) ------------
Info in <:print>:  dir:              rootd://proofadm@cernvm24.cern.ch:1093//home/proofadm/PEAC/proof/proofbox/ganis/data/0/cernvm24-1343918756-15791/
Info in <:print>:  raw dir:          /home/proofadm/PEAC/proof/proofbox/ganis/data/0/cernvm24-1343918756-15791/
Info in <:print>:  file name:        output-cernvm24-1343918756-15791.q1.root
Info in <:print>:  run type:         create a merged file
Info in <:print>:  merging option:   keep remote
Info in <:print>:  output file name: rootd://proofadm@cernvm24.cern.ch:1093//home/proofadm/PEAC/proof/proofbox/ganis/data/0/cernvm24-1343918756-15791/output-cernvm24-1343918756-15791.q1.root
Info in <:print>:  ordinal:          0
Info in <:print>: -------------- output-cernvm24-1343918756-15791.q1.root : done -------------
```

The method TProof::GetOutput() can be used to access transparently the output objects: if the searched object is not found in the output list and the output list contains TProofOutputFile objects, these files are opened and searched for the object; this is what ProofSimple::Terminate does to make the tutorial behaviour unchanged.

3.2. *Saving merged objects to an output file*

It is possible to specify a file where to save the output object with the keywords 'of' or 'outfile'; for example

```
root [1] p->SetParameter("ProofSimple_NHist", (Long_t) 16) // NB: ProofSimple_NHist is needed by the ProofSimple tutorial, not by the safe-to-file functionality
root [2] p->Process("tutorials/proof/ProofSimple.C+", 40000000, "of=test.root")
Mst-0: merging output objects ... done
Mst-0: grand total: sent 23 objects, size: 15506 bytes
ntuple opts: 0  0
 Output saved to test.root
(Long64_t)0
```

The specified file path is interpreted fro the client machine and can be also a full URL. If 'master:' is prefixed then the path is interpreted from the master machine:

```
root [3] p->Process("tutorials/proof/ProofSimple.C+", 40000000, "of=master:test.root")
Info in : unmodified script has already been compiled and loaded
Mst-0: merging output objects ... done
Output file: rootd://proofadm@cernvm24.cern.ch:1093//home/proofadm/PEAC/proof/proofbox/ganis/data/0/cernvm24-1344002021-23918/test.root
Mst-0: grand total: sent 8 objects, size: 1005 bytes
ntuple opts: 0  0
(Long64_t)0
```

By default file are created in the user data directory on the master; a TProofOutputFile is always sent back to the user with the location of the file and the URL to open it remotely. If the option 'stf' is specified (or if the server side settings enforce it) then merging goes via file and the location of the intermediate file is notified via TProofOutputFile:

```
root [5] p->Process("tutorials/proof/ProofSimple.C+", 40000000, "of=test.root;stf")
Info in : unmodified script has already been compiled and loaded
Mst-0: merging output objects ... done
Output file: rootd://proofadm@cernvm24.cern.ch:1093//home/proofadm/PEAC/proof/proofbox/ganis/data/0/cernvm24-1344002021-23918/test.root
Mst-0: grand total: sent 8 objects, size: 1281 bytes
ntuple opts: 0  0
[TFile::Cp] Total 0.02 MB       |====================| 100.00 % [1.9 MB/s]
 Output successfully copied to test.root
(Long64_t)0
```

3.3. *Creating a dataset*

To create a dataset use the keyword 'ds':

```
root [6] p->Process("tutorials/proof/ProofSimple.C+", 40000000, "ds=testds")
Info in : unmodified script has already been compiled and loaded
Registering dataset 'testds' ... OK (1 workers still sending)
Mst-0: merging output objects ... done
Mst-0: grand total: sent 8 objects, size: 26539 bytes
ntuple opts: 0  0
(Long64_t)0
```

Option 'safe-to-file' is enforced in this case. By default the dataset is only registered; to force verification add '|V':

```
root [7] p->Process("tutorials/proof/ProofSimple.C+", 40000000, "ds|V")
Info in : unmodified script has already been compiled and loaded
Registering dataset 'dataset_cernvm24-1344002021-23918_q6' ... OK
Mst-0: merging output objects ... done
Mst-0: grand total: sent 8 objects, size: 26569 bytes
ntuple opts: 0  0
Collection name='TList', class='TList', size=12
 Collection name='FeedbackList', class='TList', size=0
  TParameter      ProofSimple_NHist = 16
 OBJ: TNamed    PROOF_DefaultOutputOption       ds:dataset_
  TParameter       PROOF_SavePartialResults = 1
 OBJ: TNamed    PROOF_QueryTag  session-cernvm24-1344002021-23918:q6
 OBJ: TNamed    PROOF_FilesToProcess    dataset:dataset_cernvm24-1344002021-23918_q6
 OBJ: TNamed    PROOF_Packetizer        TPacketizerFile
 OBJ: TNamed    PROOF_VerifyDataSet     dataset_cernvm24-1344002021-23918_q6
 OBJ: TNamed    PROOF_VerifyDataSetOption
  TParameter       PROOF_IncludeFileInfoInPacket = 1
 OBJ: TNamed    PROOF_MSS
 OBJ: TNamed    PROOF_StageOption
Registering dataset 'dataset_cernvm24-1344002021-23918_q6' ... OK
Mst-0: merging output objects ... done
Mst-0: grand total: sent 102 objects, size: 29567 bytes
Info in <:verifydataset>: dataset_cernvm24-1344002021-23918_q6: changed? 1 (# files opened = 24, # files touched = 0, # missing files = 0)
(Long64_t)0
```

In this example we see that is is not necessary to specify a name for the dataset: if missing, the name will be in the form 'dataset__q'.

3.4 *Summary of option keywords*

The client-side keywords described in this section are summarized in Table 1.

Table 1. *Client-side keywords*

Long name	Short	Description	Subsection
safetofile[=*opt*]	stf[=*opt*]	Control saving of partial results to file; the optional *opt* field is in the form *o1**10 + *o0* with *o0* = 0 save if required by admin 1 force saving *o1* = 1 save after each packet 0 save at query end Default is opt = 1 when the keyword is specified (0 if not specified).	3.1
outfile=*fileout*	of=*fileout*	Enables saving the query output to file *fileout*. The path (which could be a full URL) is interpreted from the client session unless it starts with 'master', in which case it is created from the master session. Using 'of=master' saves the results in the master data directory with name .q.root If not specified, this option is internally set to 'master' in the case the administrator forces saving to file.	3.2

Long name	Short	Description	Subsection
dataset[=*name*]	ds[=*name*]	Enables creation of a dataset out of the saved files. The list of files is also returned in the output list in the form of a TFileCollection. The dataset is registered under *name* if registration is allowed by the administrator. The dataset is also verified if the *name* field contains '|V'; the sequence '|V' is in such a case removed from the final dataset name. The name is set to 'dataset__q if not specified.	3.3

4.**Server-side options**

5. **Summary of parameters and environment variables**

The parameters and RC environment variables affecting the options defined in this section are summarized in Table 2.

Table 2. *Parameters and RC variables*

Name	Type	Description	Remarks
PROOF_DefaultOutputOption	*Param*	String parameter used internally to pass the output file or the dataset name; in the first case it is in the form 'of:*fileout*', while for datasets it has the form 'ds:*name*'	
PROOF_SavePartialResults	*Param*	Int_t parameter containing the 'safetofile' option	
ProofPlayer.SavePartialResults	*RC var*	As PROOF_SavePartialResults	Server side only
ProofPlayer.SaveMemThreshold	*RC var*	Float_t parameter defining the threshold to force file saving; it is expressed as fraction of physical memory per core	Server side only

Handling large outputs via ROOT files

Introduction

One on the ideas behind PROOF is that outputs (a few histograms) are typically much smaller than input data (large trees). However, the PROOF technology can be used for a wider range of analysis than those producing a few 1D histograms and, if the output is a tree or many 3D large histograms, memory problems may come up.

As a first solution to this problem support for merging output objects via files has been introduced in version 5.18.00. The basic idea is that the output object expected to be large are saved locally in the worker instead to be added to the output list to be sent back to the master. What is sent back to the master are the coordinates of the file containing the objects. At the end of processing, the master will either merge directly the files into the final output file or create (and register) the file collection. A new class TProofOutputFile (TProofFile for ROOT < 5.20) was introduced to steer the automatic merging of the files produced on each worker with the output objects.

In 5.25/02 the functionality of TProofOutputFile has been extended to also give the possibility to automatically derive a dataset (TFileCollection) out of the files produced on the workers. The dataset can optionally registered (and verified) in the dataset database and can be directly and straightly used for further analysis.

TProofOutputFile

The TProofOutputFile class has been introduced to help handling output files produced on the workers. The class has one main constructor taking up to four arguments:

```
TProofOutputFile(const char *path, ERunType type, UInt_t opt = kRemote, const char *dsname = 0)
```

1. The file name (const char *); the user is allowed to create files either in the assigned directory (available via TProofServ::GetDataDir(), or in the working area of the sandbox; if an absolute path to a different area is given, the path is treated as relative to the assigned directories; to guarantee unicity and avoid concurrency problems, a string in the form "*<ord>/<session-tag>/<query-#>*" is prepended to the filename. In merging mode the files are always created in the working area;
2. A type argument (TProofOutputFile::ERunType) which can take two values:

   ```
   enum ERunType {  kMerge = 1, kDataset = 2};
   ```

    1. *kMerge*: merge the produced files
    2. *kDataset*: create a dataset instead of merging the files; the related TFileCollection object is returned in the output list;
3. An option argument (unsigned int) containing an 'ORed' combination of ProofOutputFile::ETypeOpt:

   ```
   enum ETypeOpt { kRemote = 1, kLocal = 2, kCreate = 4, kRegister = 8, kOverwrite = 16, kVerify = 32};
   ```

    1. *kRemote*: in merging mode, do not make local copies of the files to be merged;
    2. *kLocal*: in merging mode, make local copies of the files to be merged before mergin (see the TFileMerger constructor);
    3. *kCreate*: in dataset mode, just create the TFileCollection and add it to the output list (i.e. do not register the dataset);
    4. *kRegister*: in dataset mode, register the created dataset;
    5. *kOverwrite*: in dataset register mode, force replacement of the dataset if another one with the same name exists already;
    6. *kVerify*: in dataset registering mode, create, verify the created dataset
4. A name for the dataset to be created (const char *); this argument is optional; by default the base-name of argument 1. is taken as dataset name in case of need.

A second constructor taking up to three arguments is also available (mainly to preserve the initial signature):

```
TProofOutputFile(const char *path, const char *option = "M", const char *dsname = 0)
```

1. The file name (const char *); see above;
2. An option argument (const char *) which can be a combination of the following:
    1. 'M': merge the produced files
    2. 'D': create a dataset instead of merging the files; the related TFileCollection object is returned in the output list;
    3. 'L': in merging mode, make local copies of the files to be merged before mergin (see the TFileMerger constructor);
    4. 'H': in merging mode, merge histograms in one go; a bit faster but potentially much more memory consuming.
    5. 'R': in dataset mode, register the created dataset;
    6. 'O': in dataset register mode, force replacement of the dataset if another one with the same name exists already;
    7. 'V': in dataset registering mode, create, verify the created dataset

   Passing a null string or "LOCAL" in the option field is equivalent to, respectively, "M" and "ML";

3. A name for the dataset to be created (const char *); see above.

In merging mode the master must be able to read out the files from the worker machines. This means that the working directories, where the files are created, must be exported to the master. If there is an data-serving xrootd system running on the cluster, this can be used for this purpose. The administrator must make sure that the sandboxes are exported in the xrootd configuration file.

If the created file needs to be served by a local server the URL of the server can be passed using the environment variable LOCALDATASERVER. Also, any 'localroot' path definition via the RC-env 'Path.Localroot' are automatically trimmed out; this kind of envs can be set via the xrootd directives 'xpd.putenv' and 'xpd.putrc'. It can also be changed by the user in the way explained in the setting the environment section. When the files are on a shared partition LOCALDATASERVER must be set to 'file://' .

The URL for the output file is controlled by the RC-env '**Proof.OutputFile**'; the defaut is the input file name created in the master sandbox.

Both local and final filenames can contain placeholders which are resolved in the constructor of TProofOutputFile. The following place-holders are recognized:

1. <user>, the user name
2. <u>, the user name initial
3. <group>, the PROOF group name
4. <stag>, the session tag
5. <ord>, the worker ordinal number
6. <file>, the base-name of the path used to initialize the TProofOutputFile

Note that <ord> is dropped from the file name in the output file.

In the following of this page we describe how all this works with the help of a selector generating the tutorials demo ntuple and displaying its content.The selector code can be found at ProofNtuple.C under the $ROOTSYS/tutorials/proof directory. An example of steering macro is found in runProof.C under the same directory.

Dissection of ProofNtuple

Creating the TProofOutputFile object, opening the file and creating the ntuple

The instance of the TProofOutputFile class is created in SlaveBegin:

```
   // We may be creating a dataset or a merge file: check it
   TNamed *nm = dynamic_cast<TNamed *>(fInput->FindObject("SimpleNtuple.root"));
   if (nm) {
      // Just create the object
      UInt_t opt = TProofOutputFile::kRegister | TProofOutputFile::kOverwrite | TProofOutputFile::kVerify;
      fProofFile = new TProofOutputFile("SimpleNtuple.root",
                                        TProofOutputFile::kDataset, opt, nm->GetTitle());
   } else {
      // For the ntuple, we use the automatic file merging facility
      // Check if an output URL has been given
      TNamed *out = (TNamed *) fInput->FindObject("PROOF_OUTPUTFILE_LOCATION");
      Info("SlaveBegin", "PROOF_OUTPUTFILE_LOCATION: %s", (out ? out->GetTitle() : "undef"));
      fProofFile = new TProofOutputFile("SimpleNtuple.root", (out ? out->GetTitle() : "M"));
      out = (TNamed *) fInput->FindObject("PROOF_OUTPUTFILE");
      if (out) fProofFile->SetOutputFileName(out->GetTitle());
   }
```

The way the object is created depends on the type of run and reflects the two ways this TSelector implementation is used in the tutorials.

The first conditional scope is used when creating a dataset ("dataset" tutorial in runProof.C). A named object called "SimpleNtuple.root" triggers the creation of a dataset which name is the title of the named object. The dataset will be registered, forcing replacement and verified.

The second part of the conditional scope is used when merging the files ("ntuple" tutorial in runProof.C). In this case we use the named object "PROOF_OUTPUTFILE_LOCATION" to determine whether the files are copie on the master before merging or not.

The file must be open using TProofOutputFile::OpenFile:

```
// Open the file
TDirectory *savedir = gDirectory;
if (!(fFile = fProofFile->OpenFile("RECREATE"))) {
   Warning("SlaveBegin", "problems opening file: %s/%s",
   fProofFile->GetDir(), fProofFile->GetFileName());
}
```

The TNtuple is created and attache to the file as usual in ROOT:

```
// Now we create the ntuple
fNtp = new TNtuple("ntuple","Demo ntuple","px:py:pz:random:i");
// File resident
fNtp->SetDirectory(fFile);
fNtp->AutoSave();
```

Filling the ntuple

The TNtuple is filled in Process :

```
Bool_t ProofNtuple::Process(Long64_t entry) {


   // Fill ntuple
   Float_t px, py;
   fRandom->Rannor(px,py);
   Float_t pz = px*px + py*py;
   Float_t random = fRandom-&gt;Rndm(1);
   Int_t i = (Int_t) entry;
   fNtp->Fill(px,py,pz,random,i);

   return kTRUE;
}
```

Finalizing the file

The file must be finalized and closed in SlaveTerminate :

```
   // Write the ntuple to the file
   if (fFile) {
      Bool_t cleanup = kFALSE;
      TDirectory *savedir = gDirectory;
      if (fNtp->GetEntries() > 0) {
         fFile->cd();
         fNtp->Write();
         fProofFile->Print();
         fOutput->Add(fProofFile);
      } else {
         cleanup = kTRUE;
      }
      fNtp->SetDirectory(0);
      gDirectory = savedir;
      fFile->Close();
      // Cleanup, if needed
      if (cleanup) {
         TUrl uf(*(fFile->GetEndpointUrl()));
         SafeDelete(fFile);
         gSystem->Unlink(uf.GetFile());
         SafeDelete(fProofFile);
      }
   }
```

We add the TProofOutputFile to the output list only if the ntuple is not empty, otherwise we cleanup the file.

Accessing the output objects

For "ntuple" runs the output objects in Terminate are read from the output file and used for the final plot:

```
   // Do nothing is not requested (dataset creation run)
   if (!fPlotNtuple) return;

   // Get the ntuple form the file
   if ((fProofFile =
           dynamic_cast<TProofOutputFile*>(fOutput->FindObject("SimpleNtuple.root")))) {

      TString outputFile(fProofFile->GetOutputFileName());
      TString outputName(fProofFile->GetName());
      outputName += ".root";
      Printf("outputFile: %s", outputFile.Data());

      // Read the ntuple from the file
      fFile = TFile::Open(outputFile);
      if (fFile) {
         Printf("Managed to open file: %s", outputFile.Data());
         fNtp = (TNtuple *) fFile->Get("ntuple");
      } else {
         Error("Terminate", "could not open file: %s", outputFile.Data());
      }
      if (!fFile) return;

   } else {
      Error("Terminate", "TProofOutputFile not found");
      return;
   }
```

For "dataset" runs the graphical finalization is done outside the selector in runProof.C, providing an example of drawing functionality via PROOF.

```
proof->AddInput(new TNamed("PROOF_OUTPUTFILE", "root://arthux.do.main:5151//data/out/MyOutput.root"));
```

The PROOF benchmark framework: TProofBench

Introduction

This page describes the usage of the new benchmark module developed by Sangsu Ryu (KISTI) and available starting with ROOT 5.29/02. The code is located in the dedicated sub-directory 'proof/proofbench'. The old benchmark utilities under $ROOTSYS/test/ProofBench, still provided for legacy only, are described in *here*.

The steering class is called TProofBench. This class has one constructor whose purpose is to connect to the cluster to be bechmarked. TProofBench steers the running of two type of benchmarks: *cycle-driven* (aka CPU-intensive) and *data-driven* (aka IO-intensive). For IO-intensive benchmarks TProofBench provides a framework to generate the relevant data files and create the related datasets. By default, the selectors provided by benchsuite are used. However it is possible to use alternative selectors and data files.

Creating the TProofBench object

The TProofBench constructor takes three arguments:

```
TProofBench(const char *url, const char *outfile = "", const char *proofopt = 0)
```

1. The URL of the PROOF master or "lite" for a PROOF-Lite session.
2. The full path to a file where to save the results of the benchmark. By default the results are saved to a file created in the current directory with a name in the form *proofbench-master-Nw-yyyymmdd-hhmm.root* .
3. Additional options to be passed to TProof::Open when opening the PROOF session for the benchmark.
   Example:

```
root [0] TProofBench pb("cernvm24.cern.ch")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (40 workers)
Setting up worker servers: OK (40 workers)


  ### Welcome to the PROOF test cluster of the SFT group ###


PROOF set to parallel mode (40 workers)
Info in <:setoutfile>: using default output file: 'proofbench-cernvm24.cern.ch-40w-20110208-1217.root'
root [1]
```

The CPU benchmark

Default benchmark

The default CPU benchmark consists in creating 16 1d histos filled with 30000*Nworkers random numbers.

Using an alternative selector

The name of the alternative selector must be set using TProofBench::SetCPUSel(const char *sel); the slector must be known to the system before running; for example, it can be loaded with TProof::Load. Alternatively, a comma-separated list of PAR files to be enabled before the run can be passed via TProofBench::SetCPUPar(const char *parlist).

Running the benchmark

RunCPU: stepping on the number of workers

```
Int_t RunCPU(Long64_t ncycles=-1, Int_t start=-1, Int_t stop=-1, Int_t step=-1);
```

1. ncycles: number of cycles. Default 1000000;
2. start: number of workers to start with. Default 1;
3. stop: maximum number of workers for the scan. Default: number of active workers;
4. step: increase in number of workers between two points. Default 1;

RunCPUx: stepping on the number of workers per worker node

```
Int_t RunCPUx(Long64_t ncycles=-1, Int_t start=-1, Int_t stop=-1);
```

1. ncycles: number of cycles. Default 1000000;
2. start: number of workers per node to start with. Default 1;
3. stop: maximum number of workers per node for the scan. Default: number of active workers per node;

When the benchmark is run a dedicated TCanvas pops-up showing the results in real time. The canvas is divided vertically in two regions. On the left zone the absolute scaling plot is shown (cycles/s); on the right zone the same plot normalized to the number of workers is displayed; this second plot allows to pot easily deviations from ideal scalability. An example fo the realtime canvas is shown below.

The I/O benchmark

Default benchmark

The default benchmark is based on reading trees of Event structures (see $ROOTSYS/test/Event.h, .cxx).

Creating the default dataset

The default dataset can be created using the MakeDataSet method taking three arguments:

```
Int_t MakeDataSet(const char *dset = 0, Long64_t nevt = -1, const char *fnroot = "event")
```

1. the name of the created dataset. Default is 'BenchDataSet';
2. the number of events in the files. Default is 30000;
3. the root for the file names. Defaut is 'event', so the files are named 'event- -1.root', 'event--2.root', etc etc.
   Example:

```
root [1] pb.MakeDataSet()
Info in <:makedataset>: uploading 'proof/proofbench/src/ProofBenchDataSel.par' ...
Info in <:makedataset>: enabling 'ProofBenchDataSel' ...
Collection name='TMap', class='TMap', size=5
 Key:   TObjString = cernvm30.cern.ch
 Value:  Collection name='THashList', class='THashList', size=32
 Key:   TObjString = cernvm34.cern.ch
```

```
  Value:  Collection name='THashList', class='THashList', size=32
  Key:   TObjString = cernvm28.cern.ch
  Value:  Collection name='THashList', class='THashList', size=32
  Key:   TObjString = cernvm32.cern.ch
  Value:  Collection name='THashList', class='THashList', size=32
  Key:   TObjString = cernvm26.cern.ch
  Value:  Collection name='THashList', class='THashList', size=32
 Mst-0: merging output objects ... done
 Mst-0: grand total: sent 45 objects, size: 49008 bytes
 Collection name='TList', class='TList', size=45
  Collection name='MissingFiles', class='TList', size=0
  OBJ: TStatus   PROOF_Status    OK
  OBJ: TOutputListSelectorDataMap        PROOF_TOutputListSelectorDataMap_object Converter from output list to TSelector data members
  Collection name='PROOF_FilesGenerated_cernvm30.cern.ch_0.27', class='TList', size=5
  Collection name='PROOF_FilesGenerated_cernvm30.cern.ch_0.17', class='TList', size=6

  ...

  Collection name='PROOF_FilesGenerated_cernvm26.cern.ch_0.5', class='TList', size=4
  Collection name='PROOF_FilesGenerated_cernvm26.cern.ch_0.30', class='TList', size=4
  OBJ: TParameter        PROOF_MinPacketTime     Named templated parameter type
  OBJ: TParameter        PROOF_MaxPacketTime     Named templated parameter type
 TFileCollection dum - dum contains: 0 files with a size of 0 bytes, 0.0 % staged - default tree name: '(null)'
 The collection contains the following files:
 Collection name='THashList', class='THashList', size=160
  root://cernvm30.cern.ch//pool/proofbox/data/default/ganis/event-cernvm30.cern.ch-1.root -|-|- d41d8cd98f00b204e9800998ecf8427e
  root://cernvm30.cern.ch//pool/proofbox/data/default/ganis/event-cernvm30.cern.ch-10.root -|-|- d41d8cd98f00b204e9800998ecf8427e

  ...

  root://cernvm26.cern.ch//pool/proofbox/data/default/ganis/event-cernvm26.cern.ch-152.root -|-|- d41d8cd98f00b204e9800998ecf8427e
  root://cernvm26.cern.ch//pool/proofbox/data/default/ganis/event-cernvm26.cern.ch-160.root -|-|- d41d8cd98f00b204e9800998ecf8427e
 16:12:39 25785 Mst-0 | Info in <:scandataset>: opening 160 files that appear to be newly staged
 16:12:39 25785 Mst-0 | Info in <:scandataset>: processing 0.'new' file: root://cernvm26.cern.ch//pool/proofbox/data/default/ganis/event-cernvm26.cern.ch-129.root
 16:12:39 25785 Mst-0 | Info in <:scandataset>: processing 16.'new' file: root://cernvm26.cern.ch//pool/proofbox/data/default/ganis/event-cernvm26.cern.ch-145.root
 16:12:39 25785 Mst-0 | Info in <:scandataset>: processing 32.'new' file: root://cernvm28.cern.ch//pool/proofbox/data/default/ganis/event-cernvm28.cern.ch-65.root
 16:12:40 25785 Mst-0 | Info in <:scandataset>: processing 48.'new' file: root://cernvm28.cern.ch//pool/proofbox/data/default/ganis/event-cernvm28.cern.ch-81.root
 16:12:40 25785 Mst-0 | Info in <:scandataset>: processing 64.'new' file: root://cernvm30.cern.ch//pool/proofbox/data/default/ganis/event-cernvm30.cern.ch-1.root
 16:12:40 25785 Mst-0 | Info in <:scandataset>: processing 80.'new' file: root://cernvm30.cern.ch//pool/proofbox/data/default/ganis/event-cernvm30.cern.ch-24.root
 16:12:40 25785 Mst-0 | Info in <:scandataset>: processing 96.'new' file: root://cernvm32.cern.ch//pool/proofbox/data/default/ganis/event-cernvm32.cern.ch-100.root
 16:12:40 25785 Mst-0 | Info in <:scandataset>: processing 112.'new' file: root://cernvm32.cern.ch//pool/proofbox/data/default/ganis/event-cernvm32.cern.ch-116.root
 16:12:41 25785 Mst-0 | Info in <:scandataset>: processing 128.'new' file: root://cernvm34.cern.ch//pool/proofbox/data/default/ganis/event-cernvm34.cern.ch-33.root
 16:12:41 25785 Mst-0 | Info in <:scandataset>: processing 144.'new' file: root://cernvm34.cern.ch//pool/proofbox/data/default/ganis/event-cernvm34.cern.ch-49.root
 16:12:41 25785 Mst-0 | Info in <:scandataset>: 160 files 'new'; 0 files touched; 0 files disappeared
 (Int_t)0
 root [2]
```

Running the benchmark

RunDataSet: stepping on the number of workers

```
Int_t RunDataSet(const char *dset = "BenchDataSet",
                    Int_t start = 1, Int_t stop = -1, Int_t step = 1);
```

1. dset: dataset name. Default 'BenchDataSet';
2. start: number of workers to start with. Default 1;
3. stop: maximum number of workers for the scan. Default: number of active workers;
4. step: increase in number of workers between two points. Default 1;

RunDataSetx: stepping on the number of workers per worker node

```
Int_t RunDataSetx(const char *dset = "BenchDataSet", Int_t start = 1, Int_t stop = -1)
```

1. dset: dataset name. Default 'BenchDataSet';
2. start: number of workers per node to start with. Default 1;
3. stop: maximum number of workers per node for the scan. Default: number of active workers per node;

When the benchmark is run, a dedicated TCanvas pops-up showing the results in real time. The canvas is divided in tfour regions. On the left zone the absolute scaling plots are shown in terms of events/s (top plot) and MBytes/s (bottom); on the right zone the same quantities normalized to the number of workers are displayed. An example of the realtime canvas is shown below.

The output file

The results are saved in the output file at the path passed by the caller (or at the default path created automatically). The results of the CPU benchmark are saved under the directories 'RunCPU' or 'RunCPUx', those of the IO intensive benchmark under 'RunDataSet' or 'RunDataReadx'.

Saving the performance tree

The proofbench tool allows to save in the output file the performance trees from the various runs done during the benchmark; these can be useful for detailed studies of the results. To save the performance trees one has to set the debug variable to true calling - *before the run* - the method TProofBench::SetDebug :

```
p.SetDebug(kTRUE);
```

The trees are saved under the relevant subdirectory - i.e., RunCPU, RunCPUx, etc - and are called PROOF_PerfStats_***Type***_***N***wrks_***T***thtry, where: *Type* is the type of benchmark, CPU or DataRead; *N* is the number of workers during the related run and *T* is the ordinality of the runt.

Example: *PROOF_PerfStats_CPU_8wrks_0thtry* will be the name of the tree corresponding to the first run (T=0) of the CPU benchmark (Type=CPU) with 8 workers (N=8).

Displaying the results

Two methods are provided to show the results: DrawCPU and DrawDataSet .

TProofBench::DrawCPU: drawing the results of the CPU benchmark

```
static void DrawCPU(const char *filewithresults, const char *opt = "std:", Bool_t verbose = kFALSE, Int_t dofit = 0);
```

1. filewithresults: the file with the results to be displayed; this is filled during RunCPU or RunCPUx;
2. opt: what to plot:
    1. 'std:' : the standard plot rate vs number of workers (default);
    2. 'stdx:' : the standard plot rate vs number of workers per node;
    3. 'norm:' : the normalized rate plot vs number of workers;
    4. 'normx:' : the normalized rate plot vs number of workers per node;
3. verbose: if kTRUE print details about the graph points;
4. dofit: control extraction of performance specs (see below);

TProofBench::DrawDataSet: drawing the results of the I/O benchmark

```
static void DrawDataSet(const char *filewithresults, const char *opt = "std:", const char *type = "mbs", Bool_t verbose = kFALSE);
```

1. filewithresults: the file with the results to be displayed; this is filled during RunCPU or RunCPUx;
2. opt: what to plot:
    1. 'std:' : the standard plot rate vs number of workers (default);
    2. 'stdx:' : the standard plot rate vs number of workers per node;
    3. 'norm:' : the normalized rate plot vs number of workers;
    4. 'normx:' : the normalized rate plot vs number of workers per node;
3. type: which rate:
    1. 'mbs:' : I/O rate (default);
    2. 'evts:' : event rate;
4. verbose: if kTRUE print details about the graph points;

Getting the performance specs

Starting from the development versions 5.33/02 (and tagged patched versions 5.32/01 and 5.30/06) TProofBench provides a way to extract some performance specs from the output of the CPU benchmark. This is controlled by the last argument in TProofBench::DrawCPU. This integer can take 3 values: 0 do nothing; 1 extract the number using a 1st degree parametrization; 2, as 1 but use 2nd degree parametrization.

For the scalability plot the parametrization are just 1st or 2nd degree polynomials. For the normalized plot, the fitting function is the rational expression obtained from the ration of the same polynomials and the simple 1st degree with parameters {0,1.}.

When the argument is 1 or 2 a table with the relevant measurements is printed on the screen.

The method TProofBench::GetPerfSpecs provides a simplified wrapper around TProofBench::DrawCPU, adding the possibility to scan a directory for proofbench outputs with the possibility to chose which one to use for measurement.

A collection of performnace specs from a few setup can be found here.

Importing *proofbench* into an older ROOT version

It is possible to import and build the *proofbench* module in previous ROOT versions. The module is client side-only, so once built it can be used to benchmark any PROOF installation.

The following instructions assume that we are in $ROOTSYS; they have been tested on ROOT 5.28/00 but should work with other reasonable recent ROOT versions.

1. *Check-out the module from the trunk*

```
$ cd proof
$ svn co http://root.cern.ch/svn/root/trunk/proof/proofbench
A    proofbench/src
A    proofbench/src/TProofBenchRun.cxx
A    proofbench/src/TProofBenchDataSet.cxx
A    proofbench/src/TSelHist.cxx
A    proofbench/src/TProofNodes.cxx
A    proofbench/src/TSelEventGen.cxx
A    proofbench/src/TProofBenchRunDataRead.cxx
A    proofbench/src/TProofBenchRunCPU.cxx
A    proofbench/src/TProofBench.cxx
A    proofbench/src/TSelEvent.cxx
A    proofbench/src/TSelHandleDataSet.cxx
A    proofbench/inc
A    proofbench/inc/TProofBench.h
A    proofbench/inc/TSelEvent.h
A    proofbench/inc/TSelHandleDataSet.h
A    proofbench/inc/LinkDef.h
A    proofbench/inc/TProofBenchRun.h
A    proofbench/inc/TProofBenchTypes.h
A    proofbench/inc/TProofBenchDataSet.h
A    proofbench/inc/TSelHist.h
A    proofbench/inc/TProofNodes.h
A    proofbench/inc/TSelEventGen.h
A    proofbench/inc/TProofBenchRunDataRead.h
A    proofbench/inc/TProofBenchRunCPU.h
A    proofbench/Module.mk
Checked out revision 40758.
$
```

2. Build the module

To build the module we need to make the main ROOT Makefile aware of the new module; for that we modify the MODULES variable on the command line; we define a minimal set of modules, the core ones, to be able to build proofbench. However, **before running make we must create by hand the directory where to create the default PAR files needed by the benchmark**:

```
$ cd ..
$ make etc/proof/proofbench
$ make MODULES="build cint/cint core/metautils core/pcre core/clib core/utils core/base core/cont core/meta core/zip core/thread core/newdelete proof/proofbench" all-proofbench
cp /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofBenchDataSet.h include/TProofBenchDataSet.h
cp /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofBench.h include/TProofBench.h
cp /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofBenchRunCPU.h include/TProofBenchRunCPU.h
cp /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofBenchRunDataRead.h include/TProofBenchRunDataRead.h
cp /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofBenchRun.h include/TProofBenchRun.h
cp /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofBenchTypes.h include/TProofBenchTypes.h
cp /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofNodes.h include/TProofNodes.h
bin/rmkdepend -R -fproof/proofbench/src/TProofBench.d -Y -w 1000 -- -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -D__cplusplus -- /home/ganis/local/root
g++ -O2 -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -o proof/proofbench/src/TProofBench.o -c /home/ganis/local/root/5-28-00-patches/root/proof/proofben
bin/rmkdepend -R -fproof/proofbench/src/TProofBenchDataSet.d -Y -w 1000 -- -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -D__cplusplus -- /home/ganis/loc
g++ -O2 -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -o proof/proofbench/src/TProofBenchDataSet.o -c /home/ganis/local/root/5-28-00-patches/root/proof/p
bin/rmkdepend -R -fproof/proofbench/src/TProofBenchRunCPU.d -Y -w 1000 -- -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -D__cplusplus -- /home/ganis/loca
g++ -O2 -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -o proof/proofbench/src/TProofBenchRunCPU.o -c /home/ganis/local/root/5-28-00-patches/root/proof/pr
bin/rmkdepend -R -fproof/proofbench/src/TProofBenchRun.d -Y -w 1000 -- -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -D__cplusplus -- /home/ganis/local/r
g++ -O2 -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -o proof/proofbench/src/TProofBenchRun.o -c /home/ganis/local/root/5-28-00-patches/root/proof/proof
bin/rmkdepend -R -fproof/proofbench/src/TProofBenchRunDataRead.d -Y -w 1000 -- -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -D__cplusplus -- /home/ganis
g++ -O2 -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -o proof/proofbench/src/TProofBenchRunDataRead.o -c /home/ganis/local/root/5-28-00-patches/root/pro
bin/rmkdepend -R -fproof/proofbench/src/TProofNodes.d -Y -w 1000 -- -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -D__cplusplus -- /home/ganis/local/root
g++ -O2 -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread -o proof/proofbench/src/TProofNodes.o -c /home/ganis/local/root/5-28-00-patches/root/proof/proofben
Generating dictionary proof/proofbench/src/G__ProofBench.cxx...
core/utils/src/rootcint_tmp -cint -f proof/proofbench/src/G__ProofBench.cxx -c /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/TProofBenchDataSet.h /home/ganis/local/
bin/rmkdepend -R -fproof/proofbench/src/G__ProofBench.d -Y -w 1000 -- \
           -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread  -D__cplusplus -Icint/cint/lib/prec_stl \
           -Icint/cint/stl -I/home/ganis/local/root/5-28-00-patches/root/cint/cint/inc -- proof/proofbench/src/G__ProofBench.cxx
g++  -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fPIC -Iinclude  -pthread  -I. -I/home/ganis/local/root/5-28-00-patches/root/cint/cint/inc  -o proof/proofbench/src/G__ProofBenc
g++ -shared -Wl,-soname,libProofBench.so -m64 -O2 -o lib/libProofBench.so proof/proofbench/src/TProofBench.o proof/proofbench/src/TProofBenchDataSet.o proof/proofbench/src/TProofBench
==> lib/libProofBench.so done
bin/rlibmap -o lib/libProofBench.rootmap -l lib/libProofBench.so \
                  -d  -c /home/ganis/local/root/5-28-00-patches/root/proof/proofbench/inc/LinkDef.h
Generating PAR file etc/proof/proofbench/ProofBenchDataSel.par...
Generating PAR file etc/proof/proofbench/ProofBenchCPUSel.par...
```

3. Run the module

At this point we should be able to run the module; however, since the dependencies file cannot be modified on the fly **we need to load the library need by *libProofBench* by hand, in the ROOT session and in the PROOF session**, once started, i.e. after the TProofBench object has been created:

```
$ root -l
root [0] gSystem->Load("libProofPlayer.so")
(int)0
root [1] TProofBench pb("proofadm@cernvm24.cern.ch")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (40 workers)
Setting up worker servers: OK (40 workers)
PROOF set to parallel mode (40 workers)
Info in <:setoutfile>: using default output file: 'proofbench-cernvm24.cern.ch-40w-20110830-1727.root'
root [2] gProof->Exec("gSystem->Load(\"libProofPlayer.so\")")
(int)0
(int)0
...
(Int_t)0
root [3] pb.RunCPU()
```

Here we go!

The old PROOF benchmark utilities (obsolete)

NB: this page describes the benchmark scripts under $ROOTSYS/test/ProofBench . These macros are provided for legacy reasons and ***are not supported*** *any longer. The new benchmark suite steered by TProofBench is described in here.*

Introduction

A set of utilities to test a PROOF cluster is available under the ProofBench sub-directory of $ROOTSYS/test . The tests use files containing trees of the Event structure defined in $ROOTSYS/test/Event.h and $ROOTSYS/test/Event.C . A shared library with the 'Event' class can be created in the following way:

```
$ cd $ROOTSYS; source bin/thisroot.sh
$ cd test; gmake Event
g++ -g -Wall -fPIC -pthread -m32 -I/home/ganis/local/root/cvs/root/include -c Event.cxx
Generating dictionary EventDict.cxx...
g++ -g -Wall -fPIC -pthread -m32 -I/home/ganis/local/root/cvs/root/include -c EventDict.cxx
g++ -shared -g -m32 Event.o EventDict.o -o  libEvent.so
libEvent.so done
g++ -g -Wall -fPIC -pthread -m32 -I/home/ganis/local/root/cvs/root/include -c MainEvent.cxx
g++ -g -m32 MainEvent.o /home/ganis/local/root/cvs/root/test/libEvent.so -L/home/ganis/local/root/cvs/root/lib
-lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -lPostscript -lMatrix -lPhysics -pthread
-lm -ldl -rdynamic  -o Event
Event done
```

It is always advised to take the latest version of the utilities from the SVN trunk:

```
$ svn co https://root.cern.ch/svn/root/trunk/test/ProofBench ProofBench.
```

Benchmarking PROOF

Setup for a benchmark

See the installation page for instructions to install and configure PROOF.

After creating libEvent.so as explained in the introduction, create the 'event' PAR file by executing

```
$ ./make_event_par.sh
```

Create a PROOF session from a ROOT shell and enable the 'event' package:

```
root[] = TProof *p = TProof::Open("master")
root[] = p->UploadPackage("event")
root[] = p->EnablePackage("event")
```

The files on the nodes in the cluster can be created by running the 'make_event_trees.C' macro. The first argument is the directory the files will reside in, and the second argument is the number of events in each file. The last argument is the number of files on each node (not the number of files per worker!):

```
root [] .x make_event_trees.C("/data1/tmp", 100000, 4)
```

To create the TDSet for the files created by make_event_trees.C, the macro make_tdset.C is provided:

```
root [] .L make_tdset.C
root [] TDSet *d = make_tdset("/data1/tmp",4)
root [] d->Print("a")
OBJ: TDSet      type TTree      EventTree       in /    elements 2
TDSetElement file='root://gluon.local//data1/tmp/event_tree_gluon.local_1.root' dir='' obj='' first=0 num=-1
TDSetElement file="root://gluon.local//data1/tmp/event_tree_gluon.local_3.root' dir='' obj='' first=0 num=-1
```

To test the system with a simple command

```
root [] d->Draw("fTemperature")
```

You are now ready to run the benchmark!

Performance Monitoring

A set of performance histograms and a tree can be generated for detailed studies of a query. The way to generated such an information is described in a dedicated page.

Run the Benchmark

The benchmark provides 3 selectors, each reading a different amount of data:

EventTree_NoProc.C	Reads no data
EventTree_ProcOpt.C	Reads 25% of the data
EventTree_Proc.C	Reads all the data

First make sure the PAR file is up to date and enabled

```
root[] p->UploadPackage("event")
root[] p->EnablePackage("event")
```

Request dynamic feedback of some of the monitoring histograms

```
root[] p->AddFeedback("PROOF_ProcTimeHist")
root[] p->AddFeedback("PROOF_LatencyHist")
root[] p->AddFeedback("PROOF_EventsHist")
```

Create a TDrawFeedback object to automatically draw these histograms

```
root[] TDrawFeedback fb(p)
```

And request the timing of each command

```
root[] gROOT->Time()
```

Running one of the provided selectors is straight forward, using the TDSet that was created earlier:

```
root[] p->Load("EventTree_Proc.C+");
root[] d->Process("EventTree_Proc","")
```

The monitoring histograms should appear shortly after the processing starts. The resulting histogram from the selector will also be drawn at the end. Loading the selector before processing is not strictly needed but it allows to circumvent a problem with file distribution that was present in some versions, including 5.22/00 and 5.22/00a.

The above set of commands are included in the script Run_Simple_Test.C .

Extra Scripts Included

The script Draw_Time_Hists.C can be used to create the timing histograms from the trace tree and draw them on screen.

The script Run_Node_Tests.C can be used to run a full sequence of tests. The results can be presented graphically using Draw_PerfProfiles.C .

The script Draw_Slave_Access.C will draw a graph depicting the number of workers accessing a file serving node as a function of time.

Adapted from M. Ballintijn's $ROOTSYS/test/ProofBench/README

Creating and saving the performance tree

As explained in the benchmark utilities page, it possible to generate some performance histograms and a tree with information which can be used to monitor the performance of a cluster. This can be used to trace possible bottlenecks and therefore it is a useful debugging tool.

The histograms contain the following information:

Name	Type	Description
PROOF_PacketsHist	TH1D	"Packets processed per Worker"
PROOF_EventsHist	TH1D	"Events processed per Worker"
PROOF_ProcPcktHist	TH1I	"Packets being processed per Worker"
PROOF_NodeHist	TH1D	"Workers per Fileserving Node"
PROOF_LatencyHist	TH2D	"GetPacket Latency per Worker"
PROOF_ProcTimeHist	TH2D	"Packet Processing Time per Worker"
PROOF_CpuTimeHist	TH2D	"Packet CPU Time per Worker"

All histograms are dynamically updated, so they can be used in feedback to monitor the worker activity. Starting with ROOT v5.34, the option "fb=stats" in TProof::Process enables feedback for the first three histograms displayed in one canvas.

The tree contains detailed information about 'packet' processing and file opening; the interpretation of the tree requires some expertise, though.

The purpose of this section is to explain how to create and save the relevant information.

1. Creating the performance tree with ROOT >= 5.33

For ROOT >= 5.33 (SVN rev 42382) TProof provides two methods to facilitate the generation and saving of the performance tree:

```
void   TProof::SetPerfTree(const char *pf = "perftree.root", Bool_t withWrks = kFALSE);
```

A call to SetPerfTree before the query set ups the relevant parameters to create the performance tree; when the query is over the tree is saved to the file specified by the first argument (default 'perftree.root'). The second argument controls whether the start and stop information from the workers should also be recorded (default is not); *generation of the performance tree must be re-enabled before each run*.

```
Int_t  TProof::SavePerfTree(const char *pf = 0, const char *qref = 0);
```

This method saves the performance tree for query whose unique reference is 'qref' (by default last query) to be saved to the file indicated by the first argument (default is the path previously set via a call to SetPerfTree or 'perftree.root'). This method is called internally by TProof::Process when the creation and saving of the performance tree is enabled.

2. Creating the performance tree with ROOT

2.1 Setting up the parameters

To create the performance tree and histograms, the following should be done before running the query:

```
root[] proof->SetParameter("PROOF_StatsHist", "")
root[] proof->SetParameter("PROOF_StatsTrace", "")
```

or

```
root[] gEnv->SetValue("Proof.StatsHist",1)
root[] gEnv->SetValue("Proof.StatsTrace",1)
```

(to save detailed information per worker the following setting needs also to be added

```
root[] proof->SetParameter("PROOF_SlaveStatsTrace", "")
```

or

```
root[] gEnv->SetValue("Proof.SlaveStatsTrace",1)
```

but this is typically not required).

After processing the query, the output list should contain, in addition to the user output objects, the following objects:

```
root[] proof->GetOutputList()->ls()
 ...
 OBJ: TTree     PROOF_PerfStats PROOF Statistics : 0 at: 0x163ca50
 OBJ: TH1D      PROOF_PacketsHist       Packets processed per Worker : 0 at: 0x163a180
 OBJ: TH1I      PROOF_ProcPcktHist      Packets being processed per Worker : 0 at: 0x163a180
 OBJ: TH1D      PROOF_EventsHist        Events processed per Worker : 0 at: 0x1a93b70
 OBJ: TH1D      PROOF_NodeHist  Slaves per Fileserving Node : 0 at: 0x1639b20
 OBJ: TH2D      PROOF_LatencyHist       GetPacket Latency per Worker : 0 at: 0x1a90c40
 OBJ: TH2D      PROOF_ProcTimeHist      Packet Processing Time per Worker : 0 at: 0x1b07c40
 OBJ: TH2D      PROOF_CpuTimeHist       Packet CPU Time per Worker : 0 at: 0x1b40860
 ...
root []
```

The tree "PROOF_PerfStats" contains detailed information about the query.

2.2 Saving the tree

To save the performance tree and histograms into a file one can execute the macro $ROOTSYS/test/ProofBench/SavePerfInfo.C with, as argument, the full path to the file where to save the information:Crea

```
root[] .x $ROOTSYS/test/ProofBench/SavePerfInfo.C("/tmp/perf.root")
```

Analysing the performance tree

The performance tree is a TTree containing detailed information about the workflow of a PROOF query. The tree is not created by default: detailed instructions to create and save it can be found in the dedicated section.
Content of this page:

---

1. TProofPerfAnalysis: set of tools to analyse the performance tree

The container class TProofPerfAnalysis is an interface to a set of tools to analyse the performance tree of a given query. The class is located under $ROOTSYS/proof/proofbench and therefore appears in the $ROOTSYS/lib/libProofBench.so plug-in. The class has been introduced during the development cycle 5.33/01 and back-ported to the patch branches of production versions 5.30 and 5.32; it therefore ships with the ROOT development release 5.33/02 and with the patch releases 5.30/07 and 5.32/01.

1.1 Constructor

The class constructor has the following signature:

```
TProofPerfAnalysis(const char *perffile, const char *title = "", const char *treename = "PROOF_PerfStats");
```

with one mandatory and two optional arguments:

- *perffile*, mandatory, is the file to be analysed; can also be located remotely, for example on the web.
- *title*, optional, is the title to be used on the canvas to characterize the results of the analyzed queries; by default is set to ""
- *treename*, optional, the name of the tree to be analyzed; default is "PROOF_PerfStats". It is also possible to speficy a subdirectory or to add the string passed her to 'PROOF_PerfStats', so that the final serached name is 'PROOF_PerfStats'. If the name specified by treename, or derived by treename, is not found, the constructor will look into the file directory structure for the first TTree whose name contains treename. This is typically enough tolocate the performance tree in the file.

If the initialization is successful, the constructor will run some analysis on the worker performance and print a summary of the result.

Here is an example:

```
root [0] TProofPerfAnalysis ppa1("H4muEOS-001.root")
 +++ TTree 'PROOF_PerfStats' has 612 entries
 +++ 16 workers were active during this query
 +++ Total query time: 713.634155 secs (init: 0.133229 secs, merge: 432.601013 secs)
 +++ Avg processing rates: 153.8833 evts/s, 29.3369 MB/s
 +++ Max processing rates: 594.7538 evts/s, 80.6259 MB/s
 +++ 11 files were processed during this query
```

The summary information can be also re-printed with the method

```
   void  Summary(Option_t *opt = "", const char *out = "") .
```

Currently only option 'S' is supported to get a compact format:

```
root [1] ppa1.Summary("S")
16 713.634155 0.133229 432.601013 153.883303 594.753847 29.336907 80.625946
```

The second argument specifies a text file where to save the information; the file is (re)created at each call, unless a '+' is pre-pended to the file name, in which case the information is appended to en existing file.

1.2 Worker query information

The performance tree contains detailed information about packets. This can be used to the performance of a given workers during the query. A summary of the worker activity can be displayed using the method PrintWrkInfo, available with two signatures:

```
   void  PrintWrkInfo(Int_t showlast = 10);
   void  PrintWrkInfo(const char *wrk);
```

The first form can be used to printout the information about the showlast slowest workers. The second to display the the information of a given worker or a set of workers whose ordinal number or FQDN matches the string passed argument. Here are two ways to display the same information using the two methods:

```
root [1] pf.PrintWrkInfo(2)
 +++ TWrkInfo +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Worker:             0.100, lxfssl3310.cern.ch
 +++ Activity interval:  3.842385 -> 433.544617
 +++ Amounts processed:  32 packets (32 remote), 263633 evts, 2516378014 bytes
 +++ Processing time:    429.675842 s (CPU: 48.580000 s)
 +++ Averages:           613.562538 evts/s, 5.585152 MB/s
 +++ Total latency:      0.025222
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ TWrkInfo +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Worker:             0.112, lxfssl3404.cern.ch
 +++ Activity interval:  3.500722 -> 433.617889
 +++ Amounts processed:  18 packets (18 remote), 234250 evts, 2084701916 bytes
 +++ Processing time:    430.099195 s (CPU: 49.230000 s)
 +++ Averages:           544.641801 evts/s, 4.622484 MB/s
 +++ Total latency:      0.017882
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

and

```
root [1] pf.PrintWrkInfo("0.100,lxfssl3404.cern.ch")
 +++ TWrkInfo +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Worker:             0.100, lxfssl3310.cern.ch
 +++ Activity interval:  3.842385 -> 433.544617
 +++ Amounts processed:  32 packets (32 remote), 263633 evts, 2516378014 bytes
 +++ Processing time:    429.675842 s (CPU: 48.580000 s)
 +++ Averages:           613.562538 evts/s, 5.585152 MB/s
 +++ Total latency:      0.025222
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ TWrkInfo +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Worker:             0.112, lxfssl3404.cern.ch
 +++ Activity interval:  3.500722 -> 433.617889
 +++ Amounts processed:  18 packets (18 remote), 234250 evts, 2084701916 bytes
 +++ Processing time:    430.099195 s (CPU: 49.230000 s)
 +++ Averages:           544.641801 evts/s, 4.622484 MB/s
 +++ Total latency:      0.017882
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Graphically is interesting to see how the processing activity developed during the query time. This can be displayed using the WorkerActivity method

```
void WorkerActivity();
```

An example of output produced by this method is shown in the figure.

The upper plots show the query start-up and processing end. For a good query - like the one shown in here - the starting and finishing time is very close for all the workers. Bad performing workers will show up as long tails in here. The bottom plot quantifies the query finishing time and measures the spread across the workers.

1.3 Event and packet distributions

The packet and event distribution across workers can be displayed using the method EventDist():

```
void  EventDist();
```

A call to this creates a canvas with in the top plot the events per worker and in the bottom the packets per worker. An example is shown in the figure:

1.4 Processing rate vs query time

The method RatePlot() displays the event and MB processing rates as a function of the query time per each worker. The signature is the following:

```
 void  RatePlot(const char *wrks = 0);
```

The argument allows to select one or more workers by their ordinal number (comma-separated list). The MB rate makes sense only for data-driven queries. In such a case results from remote packets are displayed in red. By pointing on a curve and pressing the right button of the mouse is possible to find out which curve related to which worker, and have a second run for the given worker(s).

An example of the plot is shown in the figure.

1.5 Packet retrieval latency vs query time

The latency in getting a packet from the master measured by the worker can displayed as a function of the query time using the method LatencyPlot():

```
void  LatencyPlot(const char *wrks = 0);
```

As in the case of rates, the argument allows to select one or more workers by their ordinal number (comma-separated list). By pointing on a curve and pressing the right button of the mouse is possible to find out which curve related to which worker, and have a second run for the given worker(s).

An example of the plot is shown in the figure.

1.6 File distribution

For data-driven queries reading files from several servers it may be interesting to look at the distribution of packets across the servers and workers. This is the purpose of the FileDist() method:

```
void  FileDist(Bool_t writedetails = kFALSE);
```

which produces two canvases, one with the distribution of packets and MB across the data servers, which gives an idea of how even are data distributed, and a second canvas with a 3D plots showing the correlations between workers and servers for data serving. Optionally the details of this information can be saved into a text file.

Examples of the plots are shown in the following figures.

1.7 Detailed processing information per file

Two methods are provided to display detailed information about the processing of each file:

```
   void  PrintFileInfo(Int_t showslowest = 10, const char *opt = "", const char *out = 0);
   void  PrintFileInfo(const char *fn, const char *opt = "P", const char *out = 0);
```

The first method displays information about the files that took longer to process (by default the last ten):

```
root [6] ppa1.PrintFileInfo(3)
 +++ TFileInfo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Server:         root://eosatlas/
 +++ File:           /eos/atlas/user/g/gganis/data1/mc12_8TeV.160155.PowhegPythia8_AU2CT10_ggH125_ZZ4lep.merge.NTUP_HSG2.e1191_s1469_s1470_r3542_r3549_p1344_tid01133894_00/NTUP_HSG2.0
 +++ Processing interval:  0.090499 -> 229.785583
 +++ Packets:         45 (45 remote)
 +++ Processing wrks: 7 (7 remote)
 +++ MB rates:       5.388210 MB/s (avg), 2.640950 MB/s (min), 10.326082 MB/s (max)
 +++ Sizes:          222  (avg), 19 (min), 1000 (max)
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ TFileInfo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Server:         root://eosatlas/
 +++ File:           /eos/atlas/user/g/gganis/data1/mc12_8TeV.160155.PowhegPythia8_AU2CT10_ggH125_ZZ4lep.merge.NTUP_HSG2.e1191_s1469_s1470_r3542_r3549_p1344_tid01133894_00/NTUP_HSG2.0
 +++ Processing interval:  0.062248 -> 612.267883
 +++ Packets:         11 (11 remote)
 +++ Processing wrks: 2 (2 remote)
 +++ MB rates:       7.362400 MB/s (avg), 0.239816 MB/s (min), 11.984926 MB/s (max)
 +++ Sizes:          909  (avg), 686 (min), 1180 (max)
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ TFileInfo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Server:         root://eosatlas/
 +++ File:           /eos/atlas/user/g/gganis/data1/mc12_8TeV.160155.PowhegPythia8_AU2CT10_ggH125_ZZ4lep.merge.NTUP_HSG2.e1191_s1469_s1470_r3542_r3549_p1344_tid01133894_00/NTUP_HSG2.0
 +++ Processing interval:  151.918732 -> 713.634155
 +++ Packets:         159 (159 remote)
 +++ Processing wrks: 15 (15 remote)
 +++ MB rates:       7.150163 MB/s (avg), 0.129910 MB/s (min), 19.217761 MB/s (max)
 +++ Sizes:          62  (avg), 8 (min), 1000 (max)
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The second prototype allows to address a given file. Using the option filed, detailed information about the packets can be obtained:

```
root [10] ppa1.PrintFileInfo("/eos/atlas/user/g/gganis/data1/mc12_8TeV.160155.PowhegPythia8_AU2CT10_ggH125_ZZ4lep.merge.NTUP_HSG2.e1191_s1469_s1470_r3542_r3549_p1344_tid01133894_00/NT
 +++ TFileInfo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 +++ Server:         root://eosatlas/
 +++ File:           /eos/atlas/user/g/gganis/data1/mc12_8TeV.160155.PowhegPythia8_AU2CT10_ggH125_ZZ4lep.merge.NTUP_HSG2.e1191_s1469_s1470_r3542_r3549_p1344_tid01133894_00/NTUP_HSG2.0
 +++ Processing interval:  0.062248 -> 612.267883
 +++ Packets:         11 (11 remote)
 +++ Processing wrks: 2 (2 remote)
Collection name='TList', class='TList', size=2
  Worker: 0.3,  packet(s): 10
              1000 evts,          4.92 MB/s,         0.062 ->       22.280 s
               900 evts,          5.63 MB/s,        22.283 ->       53.295 s
               713 evts,         11.98 MB/s,        53.296 ->       63.765 s
               820 evts,         10.39 MB/s,        63.766 ->       71.557 s
               960 evts,          9.76 MB/s,        71.557 ->       81.750 s
              1075 evts,          9.92 MB/s,        81.751 ->       92.715 s
              1180 evts,          5.75 MB/s,        92.715 ->      117.761 s
               892 evts,          8.44 MB/s,       117.761 ->      127.196 s
               774 evts,          5.77 MB/s,       127.196 ->      144.127 s
               686 evts,          8.17 MB/s,       144.128 ->      151.290 s
  Worker: 0.8,  packet(s): 1
              1000 evts,          0.24 MB/s,         0.111 ->      612.268 s
 +++ MB rates:       7.362400 MB/s (avg), 0.239816 MB/s (min), 11.984926 MB/s (max)
 +++ Sizes:          909  (avg), 686 (min), 1180 (max)
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The last argument allows to specify a text file where the information displayed on the screen can be saved.

1.8 Saving drawn objects to file

Starting with version 5.34/32 (and 6.04/00) the temporary objects created during the various calls described above can be save to a file for further processing. This functionality is controlled by the TProofPerfAnalysis method SetSaveResult:

```
Int_t SetSaveResult(const char *file = "results.root", Option_t *mode = "RECREATE")
```

The fuctionality remains enabled after a first call to SetSaveResult and can be disabled by a call with 'file' null . By default a new file is created each time SetSaveResult is calle with a non-null 'file'; the second argument allows to update files instead of recreate them.

Example:

```
root [] TProofPerfAnalysis paf("perf_analysis_output.root")
root [] paf.SetSaveResult()
Drawn objects will be saved in file 'results.root'
root [] paf.RatePlot()
root [] paf.SetSaveResult("results-global.root", "UPDATE")
Drawn objects will be saved in file 'results-global.root'
root [] paf.RatePlot()
root [] paf.SetSaveResult(0)
Drawn objects saving disabled
```

2. Description of the performance tree

The name of the tree is *PROOF_PerfStats* and its entries are instances of the class TPerfEvent. The tree is filled via the TPerfStats interface class, an implementation of TVirtualPerfStats. The class TPerfEvent contains the following:

```
class TPerfEvent : public TObject {

public:
   TString                       fEvtNode;   // node on which the event was generated
   TTimeStamp                    fTimeStamp; // time offset from start of run
   TVirtualPerfStats::EEventType fType;
   TString                       fSlaveName;
   TString                       fNodeName;
   TString                       fFileName;
   TString                       fFileClass;
   TString                       fSlave;
   Long64_t                      fEventsProcessed;
   Long64_t                      fBytesRead;
   Long64_t                      fLen;
   Double_t                      fLatency;
   Double_t                      fProcTime;
   Double_t                      fCpuTime;
   Bool_t                        fIsStart;
   Bool_t                        fIsOk;

   TPerfEvent(TTimeStamp *offset = 0);
   virtual ~TPerfEvent() {}

   Bool_t   IsSortable() const { return kTRUE; }
   Int_t    Compare(const TObject *obj) const;
   void     Print(Option_t *option="") const;

   ClassDef(TPerfEvent,3) // Class holding TProof Event Info
};
```

The meaning of the different members depend of the event type. The types are described in TVirtualPerfStats.h :

```
   enum EEventType {
      kUnDefined,
      kPacket,       //info of single packet processing
      kStart,        //begin of run
      kStop,         //end of run
      kFile,         //file started/finished in packetizer
      kFileOpen,     //opening data file statistics
      kFileRead,     //reading data file event
      kRate,         //processing {evt, MB} rates
      kNumEventType  //number of entries, must be last
   };
```

Not all event types are filled during a PROOF query, but only the following:

1. *kStart*, marking the begin of query processing;
2. *kPacket*, after each packet has been processed;
3. *kFile*, when a file is assigned to a worker and when there is no more packets from the file to be given to the worker;
4. *kRate*, when a message with processing progress information is sent to the client;
5. *kStop*, marking the end of query processing.

Each entry in the tree has a timestamp, so that time correlation between the different events are possible.

Running queries in asynchronous (batch) mode

Running queries in asynchronous (batch) mode

Support for asynchronous running in PROOF has been added in ROOT version 5.04.00 . Disconnection / reconnection functionality requires a XROOTD backend and is available since ROOT version 5.08.00 .

This mode is automatically used when queries are queued by the scheduler.

Submitting a query in asynchronous mode

To have a query processed asynchronously you need to specify the option "ASYN" in the option string to TProof::Process(...)

```
root [] p = TProof::Open("lxb6043")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (3 workers)
Setting up worker servers: OK (3 workers)
PROOF set to parallel mode (3 workers)
(class TProof*)0x82862e8
root [] .x preph1.C("http")  // defines the chain 'ch1'
root [] ch1->Process("h1analysis.C+", "ASYN")
(Long64_t)1
root [] ch1->Process("h1analysis.C+", "ASYN")
(Long64_t)2
root [] ch1->Process("h1analysis.C+", "ASYN")
(Long64_t)3
```

The queries are queued on the master and processed sequentially.

Checking the status of processing

The status of the submitted queries can be checked with TProof::ShowQueries() :

```
root [] p->ShowQueries()
+++
+++ Queries processed during this session: selector: 3, draw: 0
+++ #:1 ref:"session-lxb6043-1205935539-667:q2" sel:h1analysis running
+++ #:2 ref:"session-lxb6043-1205935539-667:q3" sel:h1analysis submitted
+++ #:3 ref:"session-lxb6043-1205935539-667:q1" sel:h1analysis completed evts:0-283812
+++
```

In this example we see that the first query has been completed and the second is being processed.

Unique query reference

Each query is uniquely identified by a reference string composed by the session tag and a submission sequential number; in the above example, 'lxb6043-1205935539-667' is the session tag (composed by the master hostname, the starting time of the master and the master process ID) and the submission number is the number following the ':q'.

Getting the output of a completed query

For each submitted query a TQueryResult object is created. This object is saved into a root file in the master sandbox and automatically sent to the attached client sessions. A pointer to the relevant result object which can be obtaibed with TProof::GetQueryResult(const char *qref):

```
root [] TQueryResult *qr
root [] qr = p->GetQueryResult("session-lxb6043-1205935539-667:q1")
```

The TQueryResult object contains all the information about the query (inputs, selector, ...) and the output list, which can be directly accessed:

```
root [] qr->GetOutputList()
(class TList*)0x8516610
root [] qr->GetOutputList()->ls()
OBJ: TStatus    PROOF_Status     : 0 at: 0x86fb7a0
OBJ: TH1F       hdmd    dm_d : 0 at: 0x86fb7e0
OBJ: TH2F       h2      ptD0 vs dm_d : 0 at: 0x86fbc10
```

Archiving the result object to a root file

The TQueryResult object associated with a query can be archived to a file using TProof::Archive(const char *qref, const char *path):

```
root [] p->Archive("session-lxb6043-1205935539-667:q1", "castor:/castor/cern.ch/user/g/ganis/proofresults/")
```

the output file will be in the form 'path/session-tag-num.root', e.g. 'castor:/castor/cern.ch/user/g/ganis/proofresults/session-lxb6043-1205935539-667-1.root' in the above example. The output file is open via TFile::Open(...), so any protocol available from the root installation can be used.

The TQueryResult object can alternatively archived via TProof::Retrieve(const char *qref, const char *path); in this case the file is first received in the client memory and then saved to 'path' (the string is unmodified in this case). Again, 'path' is open via TFile::Open(...), so any protocol available from the root installation can be used.

Disconnecting and reconnecting

After having submitted the queries in asynchronous mode the client session can quit without affecting the PROOF session. If the client re-open a session of the same PROOF cluster, it will be automatically re-attached to the processing session and notified of the progress (the progress dialog pops-up as soon as the re-connection is established). At this point the client can browse the status of the submitted queries as explained above.

The TQueryResult objects for the queries completed while disconnected are **not** sent back to reconnecting client automatically: they have to explicitly retrieved using TProof::Retrieve(const char *qref) before running TProof::GetQueryResult(const char *qref):

```
root [] p->Retrieve("session-lxb6043-1205935539-667:q2")
root [] TQueryResult *qr
root [] qr = p->GetQueryResult("session-lxb6043-1205935539-667:q2")
```

The full list of results available on the master get be browsed with:

```
root [] p->ShowQueries("A")
+++
+++ Queries processed during other sessions: 25
+++ #:1 ref:"session-lxb6043-1205777525-22784:q1" sel:AliAnalysisSelector completed evts:0-1999
+++ #:2 ref:"session-lxb6043-1205775923-16609:q2" sel:AliAnalysisSelector completed evts:0-19999
+++ #:3 ref:"session-lxb6043-1205775923-16609:q4" sel:AliAnalysisSelector completed evts:0-19999
+++ #:4 ref:"session-lxb6043-1205775923-16609:q3" sel:AliAnalysisSelector completed evts:0-1999
+++ #:5 ref:"session-lxb6043-1205775923-16609:q5" sel:AliAnalysisSelector completed evts:0-19999
+++ #:6 ref:"session-lxb6043-1205775923-16609:q1" sel:AliAnalysisSelector completed evts:0-19999
+++ #:7 ref:"session-lxb6043-1205144837-29021:q2" sel:AliAnalysisSelector completed evts:0-1999
+++ #:8 ref:"session-lxb6043-1205144837-29021:q4" sel:AliAnalysisSelector completed evts:0-199999
+++ #:9 ref:"session-lxb6043-1205144837-29021:q3" sel:AliAnalysisSelector completed evts:0-1999
+++ #:10 ref:"session-lxb6043-1205144837-29021:q5" sel:AliAnalysisSelector completed evts:0-199999
+++ #:11 ref:"session-lxb6043-1205144837-29021:q1" sel:AliAnalysisSelector completed evts:0-1999
+++ #:12 ref:"session-lxb6043-1205777661-23192:q1" sel:AliAnalysisSelector completed evts:0-1999
+++ #:13 ref:"session-lxb6043-1205777397-22180:q1" sel:AliAnalysisSelector completed evts:0-1999
+++ #:14 ref:"session-lxb6043-1205794348-22616:q1" sel:AliAnalysisSelector completed evts:0-1999
+++ #:15 ref:"session-lxb6043-1205792836-14191:q1" sel:AliAnalysisSelector completed evts:0-1999
+++ #:16 ref:"session-lxb6043-1205792318-12380:q1" sel:AliAnalysisSelector completed evts:0-1999
+++ #:17 ref:"session-lxb6043-1205935468-534:q1" sel:h1analysis completed evts:0-283812
+++ #:18 ref:"session-lxb6043-1205774860-12032:q1" sel:h1analysis completed evts:0-283812
+++ #:19 ref:"session-lxb6043-1205825635-7383:q1" sel:AliAnalysisSelector completed evts:0-19999
+++ #:20 ref:"session-lxb6043-1205935168-31354:q2" sel:h1analysis completed evts:0-283812
+++ #:22 ref:"session-lxb6043-1205935168-31354:q3" sel:h1analysis completed evts:0-283812
+++ #:23 ref:"session-lxb6043-1205935168-31354:q1" sel:h1analysis completed evts:0-283812
+++
+++ Queries processed during this session: selector: 3, draw: 0
+++ #:24 ref:"session-lxb6043-1205935539-667:q2" sel:h1analysis running
+++ #:25 ref:"session-lxb6043-1205935539-667:q3" sel:h1analysis submitted
+++ #:26 ref:"session-lxb6043-1205935539-667:q1" sel:h1analysis completed evts:0-283812
+++
```

and retrieved with TList *TProof::GetListOfQueries("A") . Be aware that, depending on the number and size of the result objects, this operation may be very resource-consuming .

Removing queries on the cluster

The method TProof::Remove(const char *qref) is provided to remove any reference to a query on the cluster:

```
root [] p->Remove("session-lxb6043-1205935539-667:q2")
```

If a query is not yet running it is removed from the queue of submitted queries.

This method can also be used to clean-up the queue of queries waiting to be processed:

```
root [] p->Remove("cleanupqueue")
```

Remarks

- The default behavior is that the session will continue to run on the cluster until processing is over, if no client session has re-connected in the meantime, the session is shutdown
- The results of the completed queries are kept in the sandbox on the master
- Currently it is not possible to move to asynchronous a query started synchronously (Ctrl-Z functionality), except if the scheduler decides so;
- There is currently no way to quickly refer to the queries processed during the very previous session; this will be added soon.

What to do if things go wrong

A parallel interactive application running on a large distributed cluster is inherently more complex than a single application running on a single machine. So problems can and will happen. Here we describe ways that might help you diagnose what went wrong.

Resetting the user account

In the case a session hangs because of a problem somewhere, it may help to reset the user's account on the cluster. There are two possibilities: soft reset (default) and hard reset. One typically starts with a soft request and then, if it does not work, try a hard request. It may be needed to Ctrl-C (and exit) the ROOT session between attempts.

Soft reset

A request for a soft reset is issued by the following command:

```
root [] TProof::Reset("master")
```

where "master" is the hostname of the PROOF master, the entry point in the system. A soft request fully cleans the users instances on the master. This should trigger automatic cleaning of the users session also on the workers. Note that, while the sessions are invalidate as soon as the command is issued, the real processes can be cleaned only after a few minutes, depending on the frequency of the session checks done on the cluster machines.

Hard reset

A request for a hard reset is issued by the following command:

```
root [] TProof::Reset("master", kTRUE)
```

A hard request fully cleans the users instances on the master and forwards the same request also to the worker machines via an independent channel. This should solve all the cases where for some reason a soft request does not help.

Getting the logs of the sessions: TProof::LogViewer

In the case a session hangs because of a problem somewhere, one should interrupt PROOF on the client (possibly exiting and re-starting the ROOT session), reset the PROOF session and have a look at the session logs.

PROOF provides the class TProofLog to facilitate collecting and browsing the logs. Instances of this class are created via the method TProofLog *TProofMgr::GetSessionLogs(...).

Starting from ROOT v5.24/00, a graphic interface to this method is available via the static TProof::LogViewer . This static method opens a frame where you can enter the URL of the cluster and the relative session number, and to choose which logs to retrieve and display. The log frame open in this way is the same that, starting with version 5.20/00, is obtained by pushing the buttong "Show Logs" in the PROOF dialog box during or at the end of running.

As an example, entering

```
root [] TProof::LogViewer("ganis@alicecaf.cern.ch")
```

the following window should pop up:

On the top left you can change the master URL (you need to push the 'Get logs info' for making that effective) and choose the session from which you want the logs: '0' means the last one (default), negative numbers indicate the relative position wrt the last one (e.g. use '-1' to get the next to last one).

On the bottom left of the frame you can choose from which node to get the logs. Once you have made your choice you have to push 'Display' to get the last 100 lines of each chose log displayed in the box. You can set the number of lines to display (check 'all' to get all what available) on the bottom control bar.

Note the PROOF generates some service messages (e.g. for memory monitoring) which by default are always filtered-out from the logs (even if you have checked 'all'). You should also check 'svcmsg' to get really everything.

Note also that for large logs retrieval and displaying may be quite slow.

Browsing logs from the command line

In the case graphics is not available or too slow, the logs can be browsed directly in the session window. The PROOF session logs for the last session can be retrieved from the master and workers via:

```
root [] TProofLog *pl = TProof::Mgr("master")->GetSessionLogs()
```

The TProofLog class contains methods to browse and search the logs. To display the last 10 lines of each log file just type

```
root [] pl->Display()
```

this typically allows to identify the node(s) where the problem is. The full log for one node, e.g. "0.3", can be displayed specifying the node ordinal and the starting line

```
root [] pl->Display("0.3",0)
```

If it is not clear which node(s) had problems, one can try by searching for a specific text or keywords, e.g.

```
root [] pl->Grep("violation")
```

the list of occurencies of "violation" will be displayed, with details about the node ordinal numer (e.g. "0.5") and the line(s) where it occured (e.g. 23); one can then browse directly the related portion of the log with

```
root [] pl->Display("0.5", 20, 100)
```

which will display 100 lines from the log of node "0.5", starting from line 20.

The logs can be saved into a file with

```
root [] pl->Save("0.7", "file_with_logs_of_worker_0.7.txt")
root [] pl->Save("*", "file_with_all_logs.txt")
```

Running the PROOF query in valgrind

For linux-based systems, the possibility to run PROOF queries in valgrind is available since ROOT version 5.14/00. The several manual settings needed to achieve that where automatized in ROOT 5.24/00, for standard PROOF, and 5.26/00 for PROOF-Lite. For versions earlier than these (and >= 5.14/00) the manual setup described below should still work.

The idea behind the valgrind runs is to help tracing memory leaks, corruptions, and anything else that can be caught with valgrind. ***Note, however, that these kind of runs are meaningful only if running a ROOT version built with debug symbols (configured with '--build-debug').*** Note, also, that valgrind must be available on the cluster: no check is currently done for this, and the absence may lead to unresponsiveness of the system.

Automatic setup

Master session

To run the master session under valgrind you should start PROOF with the option 'valgrind' or 'valgrind=master'

```
root [] p = TProof::Open("ganis@alicecaf.cern.ch", "valgrind")

 ---> Starting a debug run with valgrind (master:YES, workers:NO)
 ---> Please be patient: startup may be VERY slow ...
 ---> Logs will be available as special tags in the log window (from the progress dialog or TProof::LogViewer())
 ---> (Reminder: this debug run makes sense only if you are running a debug version of ROOT)

Starting master: opening connection ...
Starting master: connection open: setting up server ...
```

The session may take a few minutes to start. Once the session is started, you should run your query as in a normal run. The output from valgrind is saved in a separated log file which is also available via the TProofLog object, with a special '-valgrind' tag:

Note that master valgrinding does not make sense in PROOF-Lite but in 5.26/00 a protection against this is missing leading to weird error messages.

Workers sessions

To valgrind the worker sessions one has to start PROOF with the option 'valgrind=workers'

```
root [] p = TProof::Open("ganis@alicecaf.cern.ch", "valgrind=workers")

 ---> Starting a debug run with valgrind (master:NO, workers:2)
 ---> Please be patient: startup may be VERY slow ...
 ---> Logs will be available as special tags in the log window (from the progress dialog or TProof::LogViewer())
 ---> (Reminder: this debug run makes sense only if you are running a debug version of ROOT)

Starting master: opening connection ...
Starting master: connection open: setting up server ...
Opening connections to workers:
```

By default, to reduce the overhead on the machines, only two worker sessions are started for this run. This is typically enough to study a problem on the workers. The number of workers started under valgrind can be passed as an option using the '#' character: for example, to start 5 workers enter 'valgrind=workers#5' .

As for the master run, the logs are available in the log viewer frame with special '-valgrind' tags.

To run both the master and workers under valgrind, enter the option 'valgrind=master+workers' (the of workers is controlled by the '#' also in this case).

Options to valgrind

By default valgrind is started with the following options:

-v verbose mode
--suppressions=$ROOTSYS/etc/valgrind-root.supp
  apply the known suppressions from the ROOT distribution being used
--log-file=.valgrind.log'
  save logs so that the standard log retrieval mechanisms can find them

It is possible to add options using the TProof::AddEnvVar mechanism, the variable 'PROOF_WRAPPERCMD' and the 'valgrind_opts:' prefix. For example, to run a full memory check on the workers, one has to enter enter the following:

```
root [] TProof::AddEnvVar("PROOF_WRAPPERCMD", "valgrind_opts:--leak-check=full")
root [] p = TProof::Open("ganis@alicecaf.cern.ch", "valgrind=workers")
Info in <:parseconfigfield>: valgrind run: resetting 'PROOF_WRAPPERCMD': must be set again for next run , if any
 ---> Starting a debug run with valgrind (master:NO, workers:2)
...
```

Valgrinding the tutorials (runProof, stressProof)

It is possible to trigger the automatic valgrind setup by defining the env GETPROOF_VALGRIND. For example, to run the master in valgrind do

```
$ export GETPROOF_VALGRIND="valgrind=master"
```

(or

```
$ export GETPROOF_VALGRIND="valgrind=master valgrind_opts:--leak-check=full"
```

to also set some options) before running runProof. Note that this acts at the level of *getProof* which is also called internally by runProof; therefore this is also the way to valgrind stressProof, because 'stressProof' also uses getProof
to start the PROOF session.

Caveats

Note that valgrind logs many useful information only when quitting the application; therefore, to get the information that you are looking for, you may need to quit the session, restart and browse the logs with the log viewer.

PROOF-Lite

In PROOF-Lite workers valgrinding should be the default; however, in 5.26/00 you still need to enter the full 'valgrind=workers' option string to enable it. Also, in 5.26/00 you cannot control the number of workers in the PROOF-Lite session via the '#' character; you have to do it via the standard PROOF-Lite way, e.g. TProof::Open("workers=2","valgrind=workers") will start a valgrind PROOF-Lite session with 2 workers, no matter how many cores the machine has.

Manual setup

We describe here how to manually setup a valgrind run. This functionality is available starting with ROOT version 5.14/00 . The all idea is to run the PROOF sessions within valgrind. To achieve this one needs to define 'valgrind' as wrapper command via the proper environment variables and to modify the relevant timeouts to account for the longer startup time of the sessions within valgrind.

Note that with the manual setup is not possible to retrieve automatically the valgrind log files via the PROOF log viewer: you must be able to access those files directly in a alternative way.

Standard PROOF

The environment variables that control the wrapper to be used to start a PROOF session are given in the table:

PROOF_MASTER_WRAPPERCMD	Wrapper command to be used to start the master session; workers are started normally

PROOF_SLAVE_WRAPPERCMD	Wrapper command to be used to start the worker sessions; master is started normally
PROOF_WRAPPERCMD	Wrapper command to be used to start all the sessions

In order to make these effective for the sessions one has to add them to the proper list using the static TProof::AddEnvVar. For example:

```
root [] TProof::AddEnvVar("PROOF_WRAPPERCMD","valgrind -v --log-file=/tmp/vg/proof-%p.log")
```

starts all the sessions (master and workers) via valgrind, in verbose mode, with the log files created under /tmp/vg and named 'proof-%p .log' (the placeholder '%p' is available in recent valgrind versions: check your valgrind installation for the exact options available).

The timeout which is relevant in this case is the the one defined via xpd.intwait to be set in the xproofd (xrootd) configuration file: one should set it to some high value (>600):

```
root [] TProof::AddEnvVar("PROOF_NWORKERS","2")
```

When checking workers it is also a good idea to reduce the number of workers to a small number to speedup a bit the run: most of the problems that can be spotted by valgrind are already visible with 2 workers (this is the default in the automatic setup). For ROOT versions >= 5.24/00 the number of workers to be started is controlled by the environment variable PROOF_NWORKERS, e.g.

```
xpd.intwait 600
```

For previous versions the modification of the proof.conf file is required.

PROOF-Lite

In the case of PROOF-Lite the wrapperin principle the wraper can be set manually in exactly the same way as for the standard case. However, this was not working properly before 5.26/00 . Fortunately, the way PROOF-Lite works provide an easy workaround. In fact, the PROOF-Lite workers inherit automatically the environment of the client starting the sessions. Therefore the TProof::AddEnvVar calls can be just replaced by the equivalent setting to be done before starting ROOT. For example, to valgrind 2 PROOF-Lite workers the following should be done:

```
$ export PROOF_WRAPPERCMD="valgrind -v --log-file=/tmp/vg/lite-22-%p"
$ root -l
root [0] gEnv->SetValue("ProofLite.StartupTimeOut", 500)
root [1] TProof *p = TProof::Open("workers=2")
```

The example also shows how to control the relevant timeout and the number of workers to be valgrinded.

PROOF Hands-on tutorial

The purpose of these pages is to illustrate the basic concepts of PROOF by guiding the reader hands-on through the main concepts.

Starting PROOF

In this section we describe how to start a PROOF session in a few relevant cases:

---

1. Start a session in a PROOF-enabled facility

To start a PROOF session we have to know the network coordinates of the master, which represents the entry point to the PROOF-enabled facility. Assuming that the URL of the master is 'master.url.dom' we will get some thing like this:

```
root [0] TProof *proof = TProof::Open("master.url.com")
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (114 workers)
Setting up worker servers: OK (114 workers)
PROOF set to parallel mode (114 workers)
root [1]
```

1.1 Controlling the number of workers

For initial tests on large facilities it may be wise to start a limited number of workers. This can be done by using the option 'workers=n' as second argument:

```
root [0] TProof *proof = TProof::Open("master.url.com", "workers=10")
Starting master: opening connection ...
Starting master: OK
+++ Starting max 10 workers following the setting of PROOF_NWORKERS
Opening connections to workers: OK (10 workers)
Setting up worker servers: OK (10 workers)
PROOF set to parallel mode (10 workers)
root [1]
```

2. Start a PROOF-Lite session

PROOF-Lite sessions are started passing the special URL "lite://":

```
root [1] TProof *plite = TProof::Open("lite://")
 +++ Starting PROOF-Lite with 2 workers +++
Opening connections to workers: OK (2 workers)
Setting up worker servers: OK (2 workers)
PROOF set to parallel mode (2 workers)
root [2]
```

The empty string "" defaults to "lite://" and therefore could also be used; however the meaning of the empty string can be customized differently, so, depending on the configuration, it can lead to something else.

3. gProof

Note, in the above examples, that many PROOF sessions - i.e. instances on TProof - can be started in the same client session. The global gProof is set by default to the latest instance of TProof created. It can be modified using the TProof::cd method:

```
root [2] gProof
(class TProof*)0x1018dc800
root [3] plite
(class TProof*)0x1018dc800
root [4] proof
(class TProof*)0x101860600
root [5] proof->cd()
root [6] gProof
(class TProof*)0x101860600
root [7]
```

So gProof is not a singleton - like gROOT or gSystem - but a convenient global pointer to the PROOF session to use; gProof is available via the TProof.h header file.

TProof: the PROOF shell

TProof is the interface class to the PROOF session. Everything done in the PROOF session is done using the methods of the TProof class.
TProof has many methods which we will encountered while trying to use PROOF. We start with two of them which allow to examine the session:

---

1. Print(Option_t opt)

This method allows to print a summary status of the session. By default it shows information about the client and the master:

```
root [7] proof->Print()
Connected to:             alice-caf.cern.ch (valid)
Port number:              1093
User:                     ganis
ROOT version|rev:         5.33/03|r43741
Architecture-Compiler:    macosx64-gcc421
Proofd protocol version:  34
Client protocol version:  35
Remote protocol version:  34
Log level:                0
Session unique tag:       lxbsq1409-1334421396-5704
os: 'root://alice-caf.cern.ch///pool/data/01/xrdnamespace'
Default data pool:        root://alice-caf.cern.ch///pool/data/01/xrdnamespace
*** Master server 0 (parallel mode, 10 workers):
Master host name:           lxbsq1409.cern.ch
Port number:                1093
User/Group:                 ganis/default
ROOT version|rev|tag:       5.33/03|current
Architecture-Compiler:      linuxx8664gcc-gcc412
Protocol version:           34
Image name:                 lxbsq1409.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
Working directory:          /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/master-0-lxbsq1409-1334421396-5704
Config directory:
Config file:
Log level:                  0
Number of workers:          10
Number of active workers:   10
Number of unique workers:   9
Number of inactive workers: 0
Number of bad workers:      0
Total MBs processed:        0.00
Total real time used (s):   0.006
Total CPU time used (s):    0.000
root [8]
```

In the case of PROOF-Lite, the master and client collapsed in one entity:

```
root [9] plite->Print()
*** PROOF-Lite cluster (parallel mode, 2 workers):
Host name:                  macphsft12.local
User:                       ganis
ROOT version|rev|tag:       5.33/03|r43741
Architecture-Compiler:      macosx64-gcc421
Protocol version:           35
Working directory:          /Users/ganis/local/root/opt/root
Communication path:         /var/folders/uC/uC0RGjQUFlmzR689bg+JJU++0gQ/-Tmp-/plite-93716
Log level:                  0
Number of workers:          2
Number of active workers:   2
Number of unique workers:   1
Number of inactive workers: 0
Number of bad workers:      0
Total MBs processed:        0.00
Total real time used (s):   0.000
Total CPU time used (s):    0.000
```

Adding the argument "a" (for 'all') information about the workers is also displayed:

```
root [8] proof->Print("a")
Connected to:             alice-caf.cern.ch (valid)
Port number:              1093
User:                     ganis
ROOT version|rev:         5.33/03|r43741
Architecture-Compiler:    macosx64-gcc421
Proofd protocol version:  34
Client protocol version:  35
Remote protocol version:  34
Log level:                0
Session unique tag:       lxbsq1409-1334421396-5704
Default data pool:        root://alice-caf.cern.ch///pool/data/01/xrdnamespace
*** Master server 0 (parallel mode, 10 workers):
Master host name:           lxbsq1409.cern.ch
Port number:                1093
User/Group:                 ganis/default
ROOT version|rev|tag:       5.33/03|current
Architecture-Compiler:      linuxx8664gcc-gcc412
Protocol version:           34
Image name:                 lxbsq1409.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
Working directory:          /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/master-0-lxbsq1409-1334421396-5704
Config directory:
Config file:
Log level:                  0
Number of workers:          10
Number of active workers:   10
Number of unique workers:   9
Number of inactive workers: 0
Number of bad workers:      0
```

```
Total MBs processed:        0.00
Total real time used (s):   0.010
Total CPU time used (s):    0.000
List of workers:
*** Worker 0.13  (valid)
    Host name:                lxbsq1240.cern.ch
    Port number:              1093
    Worker session tag:
    ROOT version|rev|tag:     5.33/03|r43580|current
    Architecture-Compiler:    linuxx8664gcc-gcc412
    User/Group:               ganis/default
    Proofd protocol version: 34
    Image name:               lxbsq1240.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:        /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.13-lxbsq1240-1334421397-10317
    Performance index:        100
    MBs processed:            0.00
    MBs sent:                 0.00
    MBs received:             0.00
    Real time used (s):       0.001
    CPU time used (s):        0.000
*** Worker 0.23  (valid)
    Host name:                lxbsq1419.cern.ch
    Port number:              1093
    Worker session tag:
    ROOT version|rev|tag:     5.33/03|r43580|current
    Architecture-Compiler:    linuxx8664gcc-gcc412
    User/Group:               ganis/default
    Proofd protocol version: 34
    Image name:               lxbsq1419.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:        /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.23-lxbsq1419-1334421397-13511
    Performance index:        100
    MBs processed:            0.00
    MBs sent:                 0.00
    MBs received:             0.00
    Real time used (s):       0.001
    CPU time used (s):        0.000
*** Worker 0.33  (valid)
    Host name:                lxfssi3307.cern.ch
    Port number:              1093
    Worker session tag:
    ROOT version|rev|tag:     5.33/03|r43580|current
    Architecture-Compiler:    linuxx8664gcc-gcc412
    User/Group:               ganis/default
    Proofd protocol version: 34
    Image name:               lxfssi3307.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:        /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.33-lxfssi3307-1334421397-17702
    Performance index:        100
    MBs processed:            0.00
    MBs sent:                 0.00
    MBs received:             0.00
    Real time used (s):       0.001
    CPU time used (s):        0.000
*** Worker 0.43  (valid)
    Host name:                lxfssl3310.cern.ch
    Port number:              1093
    Worker session tag:
    ROOT version|rev|tag:     5.33/03|r43580|current
    Architecture-Compiler:    linuxx8664gcc-gcc412
    User/Group:               ganis/default
    Proofd protocol version: 34
    Image name:               lxfssl3310.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:        /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.43-lxfssl3310-1334421397-21869
    Performance index:        100
    MBs processed:            0.00
    MBs sent:                 0.00
    MBs received:             0.00
    Real time used (s):       0.001
    CPU time used (s):        0.000
*** Worker 0.53  (valid)
    Host name:                lxfssl3402.cern.ch
    Port number:              1093
    Worker session tag:
    ROOT version|rev|tag:     5.33/03|r43580|current
    Architecture-Compiler:    linuxx8664gcc-gcc412
    User/Group:               ganis/default
    Proofd protocol version: 34
    Image name:               lxfssl3402.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:        /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.53-lxfssl3402-1334421397-16133
    Performance index:        100
    MBs processed:            0.00
    MBs sent:                 0.00
    MBs received:             0.00
    Real time used (s):       0.001
    CPU time used (s):        0.000
*** Worker 0.63  (valid)
    Host name:                lxbsq1226.cern.ch
    Port number:              1093
    Worker session tag:
    ROOT version|rev|tag:     5.33/03|r43580|current
    Architecture-Compiler:    linuxx8664gcc-gcc412
    User/Group:               ganis/default
    Proofd protocol version: 34
    Image name:               lxbsq1226.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:        /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.63-lxbsq1226-1334421397-13757
    Performance index:        100
    MBs processed:            0.00
    MBs sent:                 0.00
    MBs received:             0.00
    Real time used (s):       0.001
    CPU time used (s):        0.000
*** Worker 0.73  (valid)
    Host name:                lxbsq1412.cern.ch
    Port number:              1093
    Worker session tag:
    ROOT version|rev|tag:     5.33/03|r43580|current
    Architecture-Compiler:    linuxx8664gcc-gcc412
    User/Group:               ganis/default
    Proofd protocol version: 34
    Image name:              lxbsq1412.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:       /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.73-lxbsq1412-1334421397-18986
    Performance index:       100
    MBs processed:           0.00
    MBs sent:                0.00
    MBs received:            0.00
    Real time used (s):      0.001
    CPU time used (s):       0.000
*** Worker 0.100  (valid)
    Host name:               lxfssl3310.cern.ch
    Port number:             1093
    Worker session tag:
    ROOT version|rev|tag:    5.33/03|r43580|current
    Architecture-Compiler:   linuxx8664gcc-gcc412
    User/Group:              ganis/default
    Proofd protocol version: 34
    Image name:              lxfssl3310.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:       /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.100-lxfssl3310-1334421396-21858
    Performance index:       100
    MBs processed:           0.00
    MBs sent:                0.00
    MBs received:            0.00
    Real time used (s):      0.001
    CPU time used (s):       0.000
*** Worker 0.109  (valid)
    Host name:               lxfssl3401.cern.ch
    Port number:             1093
    Worker session tag:
    ROOT version|rev|tag:    5.33/03|r43580|current
    Architecture-Compiler:   linuxx8664gcc-gcc412
    User/Group:              ganis/default
    Proofd protocol version: 34
    Image name:              lxfssl3401.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:       /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.109-lxfssl3401-1334421396-21599
    Performance index:       100
    MBs processed:           0.00
    MBs sent:                0.00
    MBs received:            0.00
    Real time used (s):      0.001
    CPU time used (s):       0.000
*** Worker 0.112  (valid)
    Host name:               lxfssl3404.cern.ch
    Port number:             1093
    Worker session tag:
    ROOT version|rev|tag:    5.33/03|r43580|current
    Architecture-Compiler:   linuxx8664gcc-gcc412
    User/Group:              ganis/default
    Proofd protocol version: 34
    Image name:              lxfssl3404.cern.ch:/pool/PROOF-AAF/proof//proofbox/ganis
    Working directory:       /pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334421396-5704/worker-0.112-lxfssl3404-1334421396-24623
    Performance index:       100
    MBs processed:           0.00
    MBs sent:                0.00
    MBs received:            0.00
    Real time used (s):      0.001
    CPU time used (s):       0.000
root [9]
```

You may notice that each worker gets assigned a unique ordinal number in the form '0.n' .

Print() allows to find information about the ROOT versions, platforms, location of sandboxes, global statistics about CPU time and I/O used by the session actors, etc. etc. The group shown is the PROOF gorup, which defaults to 'default' if not explicitly set by the cluster administrator.

2. Exec(const char *cmd)

This allows to execute *ROOT commands* on workers or on the master. A *ROOT command* is anything that can be executed on the ROOT prompt, for example 'gSystem->Getenv("ROOTSYS")' or '.x mymacro.C'. Two questions may arise right away by looking at these two examples: 'how do we handle the character " delimiting strings?' and 'the mymacro.C is local, i.e. on the client machine: how does it get on the worker nodes?'. The answer to the first question is that the string delimiter " must be escaped using '\', so we will type something like this:

```
root [1] proof->Exec("gSystem->Getenv(\"ROOTSYS\")")
(const char* 0x14300a38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0xeb61a38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0x1c036a38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0xc41aa38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0x298ca38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0xfc11a38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0xfb37a38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0x6e44a38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0xda15a38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(const char* 0x76ba38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(Int_t)0
```

In the second case, PROOF detects CINT commands requiring a file (actually only '.L', '.x' and '.X') and make sure that an updated version of the file exists on the nodes (this means that PROOF checks if the file exists already on the nodes and compare the md5 checksums, so the file is uploaded only if needed). For example, with the following unnamed macro (which we call getROOTSYS.C):

```
{
   Printf(" ROOTSYS: %s", gSystem->Getenv("ROOTSYS"));
}
```

we get a result similar to the above:

```
root [2] proof->Exec(".x getROOTSYS.C")
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
(Int_t)0
```

The first time you run this you will notice some latency due to the distribution of the macro to the workers. If you do not change the macro, the second time it goes much faster.

By default the command is executed only on the workers, i.e. not on the master. To execute it on the master one can do the following trick:

```
root [3] proof->SetParallel(0)
PROOF set to sequential mode
(Int_t)0
root [4] proof->Exec("gSystem->Getenv(\"ROOTSYS\")")
(const char* 0x706fa38)"/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a"
(Int_t)0
root [5] proof->Exec(".x getROOTSYS.C")
 ROOTSYS: /pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
(Int_t)0
root [6] proof->SetParallel(99999)
PROOF set to parallel mode (10 workers)
(Int_t)10
```

The call SetParallel(0) make PROF think that it has no workers, i.e. that is in sequential mode; the command is therefore executed on the unique node available, the master. Do not forget to set it in parallel mode after this by calling SetParallel again with a very large number, e.g. SetParallel(99999).

2.1 Accessing the unix shell

We know that ROOT allows to escape to the underlying shell (which on PROOF master and workers is always Unix) with the sequence '.!'. This make Exec() a powerful way to get information about the machines on which the workers are started. For example, this is an alternative way to access ROOTSYS on the master:

```
root [7] proof->SetParallel(0)
PROOF set to sequential mode
(Int_t)0
root [8] proof->Exec(".!echo $ROOTSYS")
/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
(Int_t)0
root [9] proof->SetParallel(99999)
PROOF set to parallel mode (10 workers)
(Int_t)10
```

2.2 Using Exec output

The output of Exec is displayed on the screen. This may be OK in some cases. Sometimes, however, we may need to get the output in a more usable form, i.e. a string or a number. There is no generic way to do so, but with the help of output redirection we can get something locally from which we can extract the required information. The following example shows how to get the value of any environment variable as a TString on the client session, with the help of two macros, a steering macro getEnv.C:

```
#include "TMacro.h"
#include "TObjString.h"
#include "TProof.h"
#include "TString.h"
#include "TSystem.h"

TString getEnv(const char *env, const char *where= "0")
{
   // Return TString with the value of the environment variable 'env' on node with ordinal number 'where'.
   // The ordinal numbers are '0' for the master, '0.n' for the workers.
   // If the specified ordinal numebr is not found the full output is displayed so that it can be rerun with
   // the correct value.

   // This will be our output string
   TString sout;

   // We need a valid PROOF session up and running
   if (!gProof || (gProof && !gProof->IsValid())) {
      Printf("A valid PROOF session must be up and running!");
      return sout;
   }

   // Master only or workers?
   Bool_t onmaster = (strcmp(where, "0") == 0) ? kTRUE : kFALSE;

   // If on master only, set PROOF on parallel mode
   if (onmaster) gProof->SetParallel(0);

   // Temporary file for the output, making sure it does not exist already
   TString ftmp = TString::Format("%s/.getEnv", gSystem->TempDirectory());
   gSystem->Unlink(ftmp);

   // Redirect the output
   gSystem->RedirectOutput(ftmp, "w");

   // Run the macro to extract the information
   TString cmd = TString::Format(".x getEnvOrd.C(\"%s\")", env);
   gProof->Exec(cmd);

   // Restore the output
   gSystem->RedirectOutput(0);

   // If on master only, restore in parallel mode
   if (onmaster) gProof->SetParallel(99999);

   // Parse the output now using TMacro
   TMacro mm;
   mm.ReadFile(ftmp);
   TString tag = TString::Format("o:%s ", where);
   TObjString *os = mm.GetLineWith(tag);
   if (os) {
      TString info;
      Ssiz_t from = 0;
      os->GetString().Tokenize(info, from, " ");
      os->GetString().Tokenize(info, from, " ");
      os->GetString().Tokenize(sout, from, " ");
   } else {
      // Print the file
      Printf("Ordinal number %s not found! This is what we have:", where);
      mm.Print();
   }

   // Done
   return sout;
}
```

and a macro to fill the output with the wanted information:

```
#include "TProofServ.h"
#include "TSystem.h"

void getEnvOrd(const char *env)
{
   // Macro to be used with getEnv.C to format the output

   if (gSystem && gProofServ) {
      Printf("o:%s h:%s %s", gProofServ->GetOrdinal(), gSystem->HostName(), gSystem->Getenv(env));
   } else if (gSystem) {
      Printf("o: h:%s %s", gSystem->HostName(), gSystem->Getenv(env));
   } else {
      Printf("o: h:");
   }
}
```

Note the use of TSystem::RedirectOutput and of TMacro in the steering macro. Note also that a valid PROOF session must be available before running the steering macro. The result would be something like this:

```
root [10] .L getEnv.C+
Info in <:aclic>: creating shared library /Users/ganis/local/root/opt/root/./getEnv_C.so
root [11] TString rs = getEnv("ROOTSYS", "0.13")
root [12] Printf("%s", rs.Data());
/pool/PROOF-AAF/alien_packages/VO_ALICE/ROOT/v5-33-02a/v5-33-02a
```

The PROOF sandbox

Each user gets a working area on each node of the cluster. This area is called sandbox. The default location on a local machine is $HOME/.proof (on PROOF-Lite $HOME/.proof/path-from-where-we-started). However, on eal clusters the administrator decides where the user sandboxes will be created. You can find out the location of the sandbox with Print().

1. The sandbox

The sandbox has several sub-directories:

- `cache`
  Area where tarballs, code and binaries are cached for optimized uploads and operation
- `packages`
  Area where packages are actually build and installed
- `session-`*`sessionUniqueID`*
  Working area for session *sessionUniqueID*
- `data`
  Area to store files created by users during processing. By default is created under the sandbox, but can be redirected by the administrator to some other device.
- `queries` (master only)
  Area where the processing results are stored
- `datasets` (master only)
  Area where meta-information about the datasets is stored. By default is created under the sandbox, but can be redirected by the administrator to some other device.

2. The working directory

Each PROOF session has, under the sandbox, a unique working directory identified by the unique ID. The session unique ID is in the form

> *master_hostname-master_creation_time-master_process_ID*

where

- *master_hostname*
  The host name of the master node, e.g. lxbsql1409
- *master_creation_time*
  The creation time of the master in seconds since The Epoch, i.e. 1/1/1970
- *master_process_ID*
  Unix process ID of the master process

The sessionUniqueID is created on the master and communicated to the workers where is used to create the working area. Since more workers can be started on the same worker node, each worker gets another ID identifying it in unique way. This ID has a similar form as the session ID:

> worker-*ordinal-worker_hostname-workers_creation_time-worker_process_ID*

Here *ordinal* is the ordinal number of the worker, the remaining components have the same meaning as above.

We can have a look with Exec() at a real sandbox. First on the master:

```
root [1] proof->SetParallel(0)
PROOF set to sequential mode
(Int_t)0
root [2] proof->Exec(".!pwd")
/pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334473238-22773/master-0-lxbsq1409-1334473238-22773
(Int_t)0
```

In this case the administrator has chosen to set the user sandboxes under '/pool/PROOF-AAF/proof/proofbox'. The user sandbox is obtained by the sanbox path by adding the user name: '/pool/PROOF-AAF/proof/proofbox/ganis'. The content of the user sandbox is:

```
root [3] proof->Exec(".!ls -lt /pool/PROOF-AAF/proof/proofbox/ganis")
total 160
drwxr-xr-x 3 ganis z2 4096 Apr 15 11:51 session-lxbsq1409-1334483468-8890
lrwxrwxrwx 1 ganis z2   71 Apr 15 11:51 last-master-session -> /pool/PROOF-AAF/proof//proofbox/ganis/session-lxbsq1409-1334483468-8890
drwxr-xr-x 3 ganis z2 4096 Apr 15 11:51 queries
drwxr-xr-x 2 ganis z2 4096 Apr 15 11:50 session-lxbsq1409-1334473238-22773
drwxr-xr-x 3 ganis z2 4096 Apr 15 09:01 session-lxbsq1409-1334469543-9357
drwxr-xr-x 2 ganis z2 4096 Apr 15 08:56 cache
drwxr-xr-x 2 ganis z2 4096 Apr 15 07:59 session-lxbsq1409-1334421396-5704
drwxr-xr-x 2 ganis z2 4096 Apr 14 18:37 session-lxbsq1409-1334421276-4923
drwxr-xr-x 2 ganis z2 4096 Apr 13 14:05 packages
drwxr-xr-x 2 ganis z2 4096 Apr  9 16:13 session-lxbsq1409-1333980696-18601
```

We see that there is a symlink 'last-master-session' pointing to the latest created session directory. We also notice that the 'data' and 'datasets' directories are created elsewhere. We can examine the content of the cache:

```
root [4] proof->Exec(".!ls -lt /pool/PROOF-AAF/proof/proofbox/ganis/cache")
total 16
-rw-r--r-- 1 ganis z2 378 Apr 15 09:05 getEnvOrd.C
-rw-r--r-- 1 ganis z2  59 Apr 15 08:05 getROOTSYS.C
(Int_t)0
```

which contains the recently run macros, and of the packages directory:

```
root [5] proof->Exec(".!ls -lt /pool/PROOF-AAF/proof/proofbox/ganis/packages")
total 0
(Int_t)0
```

which is currently empty. Now we look at what is inside the session directory:

```
root [6] proof->Exec(".!ls -lt /pool/PROOF-AAF/proof/proofbox/ganis/last-master-session/")
total 40
-rw-r--r-- 1 ganis z2 3085 Apr 15 11:56 master-0-lxbsq1409-1334483468-8890.log
drwxr-xr-x 2 ganis z2 4096 Apr 15 11:51 master-0-lxbsq1409-1334483468-8890
-rw-r--r-- 1 ganis z2 1145 Apr 15 11:51 master-0-lxbsq1409-1334483468-8890.env
-rw-r--r-- 1 ganis z2 1932 Apr 15 11:51 master-0-lxbsq1409-1334483468-8890.rootrc
lrwxrwxrwx 1 ganis z2  113 Apr 15 11:51 session.rootrc -> /pool/PROOF-AAF/proof//proofbox/ganis/session-lxbsq1409-1334483468-8890/master-0-lxbsq1409-1334483468-8890.rootrc
(Int_t)0
```

there are three files and one directory, all named in the form 'master-0-sessionUniqueID' and the relevant extension (except for the directory). The directory is the real working place of the session. The extensions explain the meaning of the files: the session log file (.log), the session specific environment settings (.env) and the session specific ROOT-rc directives (.rootrc) (there is a symlink to the ROOT-rc file for internal convenience.

We now look at a worker sandbox. We activate only one worker for browsing convenience:

```
root [7] proof->SetParallel(1)
PROOF set to parallel mode (1 worker)
(Int_t)1
root [8] proof->Exec(".!pwd")
/pool/PROOF-AAF/proof/proofbox/ganis/session-lxbsq1409-1334483468-8890/worker-0.13-lxbsq1240-1334483470-340
(Int_t)0
```

We notice that the sandbox path on the worker machine is the same (typically clusters have the same configuration patterns) and we see the session directory is named after the sessionUniqueID that we found already on the master. If we look at the content of the sandbox:

```
root [12] proof->Exec(".!ls -lt /pool/PROOF-AAF/proof/proofbox/ganis")
total 152
drwxr-xr-x 3 ganis z2 4096 Apr 15 11:51 session-lxbsq1409-1334483468-8890
lrwxrwxrwx 1 ganis z2   71 Apr 15 11:51 last-worker-session -> /pool/PROOF-AAF/proof//proofbox/ganis/session-lxbsq1409-1334483468-8890
drwxr-xr-x 2 ganis z2 4096 Apr 15 11:50 session-lxbsq1409-1334473238-22773
drwxr-xr-x 2 ganis z2 4096 Apr 15 08:59 cache
```

```
drwxr-xr-x 3 ganis z2 4096 Apr 15 07:59 session-lxbsq1409-1334469543-9357
drwxr-xr-x 2 ganis z2 4096 Apr 15 07:58 session-lxbsq1409-1334421396-5704
drwxr-xr-x 2 ganis z2 4096 Apr 14 18:36 session-lxbsq1409-1334421276-4923
drwxr-xr-x 2 ganis z2 4096 Apr 13 22:35 packages
drwxr-xr-x 2 ganis z2 4096 Apr  9 16:11 session-lxbsq1409-1333980696-18601
```

we notice that the symlink is called now 'last-worker-session' and that there is no 'queries' directory in the sandbox. The session directory

```
root [13] proof->Exec(".!ls -lt /pool/PROOF-AAF/proof/proofbox/ganis/last-worker-session/")
total 40
-rw-r--r-- 1 ganis z2 1953 Apr 15 12:09 worker-0.13-lxbsq1240-1334483470-340.log
drwxr-xr-x 2 ganis z2 4096 Apr 15 11:51 worker-0.13-lxbsq1240-1334483470-340
lrwxrwxrwx 1 ganis z2  115 Apr 15 11:51 session.rootrc -> /pool/PROOF-AAF/proof//proofbox/ganis/session-lxbsq1409-1334483468-8890/worker-0.13-lxbsq1240-1334483470-340.rootrc
-rw-r--r-- 1 ganis z2 1149 Apr 15 11:51 worker-0.13-lxbsq1240-1334483470-340.env
-rw-r--r-- 1 ganis z2 1771 Apr 15 11:51 worker-0.13-lxbsq1240-1334483470-340.rootrc
(Int_t)0
```

also in this case three files and one directory, which are named using the worker unique ID, as mentioned above.

Examining the sandbox has also allowed to illustrate additional usage of Exec.

Processing in PROOF

In this section we will have a first look at PROOF processing starting from simple generations of random numbers. Historically this is not the way all this was developed, but we think it is easier to understand the various components starting from this use case.

---

1. TSelector framework

The reason behind PROOF is to increase processing performance by executing in parallel a number of independent tasks. To steer parallel execution PROOF uses the TSelector framework which provides an initialization phase, a processing phase and a termination phase. The processing phase is the one which can be parallelized.

The following table shows the name of the related TSelector methods and where/when they are called:

TSelector calling sequence

Phase	Client	Workers	Description
Client Init	Begin() SlaveBegin()		Client initialization
Worker Init		SlaveBegin()	Worker initizlization
Process		Process()	Called N times
Worker Terminate		SlaveTerminate()	Worker termination
Client Terminate	Terminate()		Client termination

where

- Begin()
  Method called on the client only; may be used to configure PROOF with specific settings; in general is left empty.
- SlaveBegin()
  Method called on client AND workers. This is the place where to create the objects to be filled while processing, like histograms; typically is also the place where objects are added to the output list though this is not mandatory at this stage.
- Process()
  Method called for each task (or event) to be executed.This is the place where actual processing is done and output objects filled.
- SlaveTerminate()
  Method called after processing is finished on each worker. This the place where to close or cleanup temporary things, like open files. This is also the place where to add objects to the output list, if not already done. Often this meathd is empty.
- Terminate()
  Method called on the client. It has access to the full output list. This is the place where some final analysis or drawing is done. May be empty of the output object are analyzed elsewhere.

To process something in PROOF we need to provide our code in the form of a derivation of TSelector. From the considerations above the minimal set of methods to overload are SlaveBegin() - for the output definition - and Process() - fr the actual task execution.

2. Filling a 1D histogram with random numbers

We now illustrate the various steps by creating an TSelector implementation which creates a 1D histogram filled with gaussian random numbers and displays it. In this case the task is the generation of a gaussian random number and its filling in the histogram; this task will be repeated for a number N of times.

We start from the template file SelTemplate.h. We rename this to ProofFirst.h, replacing all internal occurrences of 'SelTemplate'. The first thing to do is to define the histogram. We need the object across methods (in Process and in SlaveBegin), so the more convenient way is to have it as data member of the selector. We will therefore have something like this:

```
//////////////////////////////////////////////////////////
//
// TSelector template
//
//////////////////////////////////////////////////////////

#ifndef ProofFirst_h
#define ProofFirst_h

#include

class TH1F;
class TRandom;
class ProofFirst : public TSelector {
public :

   // Define members here
   TH1F   *fH1F;             //! Output histogram
   TRandom *fRandom;  //! Random number generator

   ProofFirst();
   virtual ~ProofFirst();
   virtual Int_t   Version() const { return 2; }
   virtual void    Begin(TTree *tree);
   virtual void    SlaveBegin(TTree *tree);
   virtual Bool_t  Process(Long64_t entry);
   virtual void    SetOption(const char *option) { fOption = option; }
   virtual void    SetObject(TObject *obj) { fObject = obj; }
   virtual void    SetInputList(TList *input) { fInput = input; }
   virtual TList  *GetOutputList() const { return fOutput; }
   virtual void    SlaveTerminate();
   virtual void    Terminate();

   ClassDef(ProofFirst,2);
};
#endif
```

The object is not streamed when streaming the selector because we want it to live on the worker only (selector streaming is an advanced feature which may encounter later).

Next comes the implementation file. After copying SelTemplate.C into ProofFirst.C (and replacing all internal occurrences of SelTemplate) we need to include TH1F.h and TRandom3.h (we will use the TRandom3 implementation of the random generator), initialize the pointers to 0 in the constructor and destroy the random number generator in the destructor:

```
#include "ProofFirst.h"
#include "TH1F.h"
#include "TRandom3.h"

//______________________________________________________________________________
ProofFirst::ProofFirst()
{
   // Constructor
   fH1F = 0;
   fRandom = 0;
}

//______________________________________________________________________________
ProofFirst::~ProofFirst()
{
   // Destructor
   if (fRandom) delete fRandom;
}
```

Note that we do not explicitly destroy the histogram, as it will be owned by the output list. We then create the histogram and the random generator instance in SlaveBegin():

```
//______________________________________________________________________________
void ProofFirst::SlaveBegin(TTree * /*tree*/)
{
   // The SlaveBegin() function is called after the Begin() function.
   // When running with PROOF SlaveBegin() is called on each slave server.
   // The tree argument is deprecated (on PROOF 0 is passed).

   // TString option = GetOption();

   // Histogram
   fH1F = new TH1F("FirstH1F", "First TH1F in PROOF", 100, -10., 10.);
   fOutput->Add(fH1F);

   // Random number generator
   fRandom = new TRandom3(0);
}
```

The initialization of the random generator with seed 0 means a unique seed. The next step is to add the relevant instructions to Process():

```
//______________________________________________________________________________
Bool_t ProofFirst::Process(Long64_t)
{
   // The Process() function is called for each entry in the tree (or possibly
   // keyed object in the case of PROOF) to be processed. The entry argument
   // specifies which entry in the currently loaded tree is to be processed.
   // It can be passed to either ProofFirst::GetEntry() or TBranch::GetEntry()
   // to read either all or the required parts of the data. When processing
   // keyed objects with PROOF, the object is already loaded and is available
   // via the fObject pointer.
   //
   // This function should contain the "body" of the analysis. It can contain
   // simple or elaborate selection criteria, run algorithms on the data
   // of the event and typically fill histograms.
   //
   // The processing can be stopped by calling Abort().
   //
   // Use fStatus to set the return value of TTree::Process().
   //
   // The return value is currently not used.

   if (fRandom && fH1F) {
      Double_t x = fRandom->Gaus(0.,1.);
      fH1F->Fill(x);
   }

   return kTRUE;
}
```

Finally we display the result in terminate:

```
 //______________________________________________________________________________
void ProofFirst::Terminate()
{
   // The Terminate() function is the last function to be called during
   // a query. It always runs on the client, it can be used to present
   // the results graphically or save the results to file.

   // Create a canvas, with 100 pads
   TCanvas *c1 = new TCanvas("c1", "Proof ProofFirst canvas",200,10,400,400);
   fH1F = dynamic_cast(fOutput->FindObject("FirstH1F"));
   if (fH1F) fH1F->Draw();
   c1->Update();
}
```

We are now ready to process this selector. We do it invoking the method TProof::Process(const char *selector, Long64_t nentries):

```
root [0] TProof *plite= TProof::Open("lite://")
 +++ Starting PROOF-Lite with 2 workers +++
Opening connections to workers: OK (2 workers)
Setting up worker servers: OK (2 workers)
PROOF set to parallel mode (2 workers)
(class TProof*)0x101891600
root [1] plite->Process("ProofFirst.C+", 10000000)
 Info in <:setqueryrunning>: starting query: 1
Info in <:setrunning>: nwrks: 2
Info in <:aclic>: creating shared library /Users/ganis/dropbox/Private/Tutorial/root-tutorial/tutorial/./ProofFirst_C.so
Lite-0: all output objects have been merged
(Long64_t)0
```

We did the processing in PROOF-Lite. The resulting canvas should look like this:

Terminology

We start with reminding the relevant terminology used in the PROOF context:

---

1. TProof

The interface class to a PROOF session. It is a sort of PROOF shell allowing high- and low-level interaction with the system. In contains a few static methods, namely TProof::Open to start a session. See reference documentation for TProof.

2. Client

ROOT session run by the end-user of PROOF on her/his laptop or desktop; from this ROOT session the PROOF session is started by creating an instance of TProof with TProof::Open("*master_url*").

3. Master

PROOF node running a ROOT application in charge of coordinating the work between workers and of merging the results. The network hostname of this machine is the string to be passed to TProof::Open to start a session. The client must therefore know the hostname of the master machine (and also the network port where the master listens, if the latter is different from the default 1093).

4. Worker (or Slave)

PROOF node running a ROOT application doing the actual work. These nodes are known by the master. The client may ignore their hostnames.

5. PROOF session

The set composed by {client, master, workers} started by a call to TProof::Open .

6. Query

Process request submitted by the client to the master. Consists by a selector and a dataset.

7. Selector

A class deriving from TSelector providing the code to be executed.

8. Dataset

List of files containing the TTree to be processed. It can be in the form of TChain, TFileCollection or TDSet. It can also just be a name of a TFileCollection previously registered on the master node.

9. Package / PAR file

Additional code needed by the selector - not available on the PROOF cluster - loaded as a separate library. The code is distributed as a PAR file (Proof ARchive), a gzipped tar-ball containing all what needed to enable the package.

10. PROOF-Lite

Special version of PROOF designed for multi-core machines. It collapses the client and the master into the client and does not need any daemon to startup. PROOF-Lite is started passing "" or "lite://" to TProof::Open.

More advanced examples

In this pages we will dissect the PROOF tutorials and steering macros available under tutorials/proof. The tutorials directory is available under $ROOTSYS fro non-prefixed installations or under *docdir* for prefixed installations, defaulting to *prefix*/share/doc/root .

See also the slides for a one-day PROOF tutorial: Introduction, Basic hands-on .

---

1. Run the available examples: runProof.C

The macro 'runProof.C' steers the running of the tutorials. It starts a PROOF session, sets the required options, and runs the required tutorial. The signature of the runProof macro is the following:

```
void runProof(const char *what = "simple",
              const char *url = "proof://localhost:40000",
              Int_t nwrks = -1)
```

The first argument *what* is a string composed by the name of the tutorial to run and, optionally, by one or more arguments in the form 'name(arg1,arg2,...)'; arguments can be general (applying to all tutorials) or specific to a given tutorial. Available tutorials and general arguments are described in the tables below. The second argument url is the place where to run the tutorial, used as first argument to getProof. The third argument *nwrks* is the number of workers to be used for the tutorial: it is used as a second argument to the call to getProof and as argument of a call to TProof::SetParallel .

The available tutorials are shown in the table:

Available tutorials

Name	Type	Output	Description
simple	cycle	histograms 1 canvas	Generate gaussian random numbers and fill histograms
simplefile	cycle	histograms 2 canvases	Generate gaussian randoms; saved in two directories; merge via file

event	cycle	histograms
		1 canvas
			Generate Event entries;
			run simple analysis;
			uses 'event.par' PAR file

pythia8	cycle	histograms
		1 canvas
			Generate Pythia8 Monte Carlo events;
			run simple analysis;
			uses 'pythia8.par' PAR file

ntuple	cycle	ntuple 1 canvas	Generate simple ntuple; merge via file
dataset	cycle	dataset 1 canvas	Same as 'ntuple' but dataset creation instead of file merging; uses TProof::DrawSelect
friends	data	histograms 1 canvas	Generate some simple main TTree trees and their friends; run simple analysis
h1	data	histograms 2 canvases	Usual H1 analysis

Name	Type	Output	Description
eventproc	data	histograms 1 canvas	Simple analysis of files with Event entries; uses 'event.par' PAR file

The 'Type' refers to the type of processing. The keyword 'data' means data-driven; this happens when processing a TTree, whose entries steer the distribution of work. The type 'cycle' indicates that a cycle of tasks has to be repeated the specified number of times and there is no steering TTree involved; this happens, for example, when generating Monte Carlo events.

The general arguments are shown in the table:

General arguments

Argument	Description
debug=[what:]level	Set verbosity to 'level'; optionally select the scope with 'what' (same names as in TProofDebug)
nevt=N	Process N entries
first=F	Start processing from entry F (when processing data files)
asyn	Run in non-blocking (asynchronous) mode
nwrk=N	Set the number of active workers to N (may not always be successful)
punzip	Use parallel unzip in reading files

cache=*bytes*
cache=*kbytes*K
cache=*mbytes*M
	Change the size of the TTree cache;
	use

submergers[=S]	Enable merging via submergers (number set to S or to the default)
rateest=average	Use the measured average to estimate the current processing speed reported by the progress bar
perftree=*perftreefile.root*	Generate the performance tree and save it to *perftreefile.root*

In addition to these common arguments, there is a way to control the ACLiC mode used to build the selectors: by default '+' is used, i.e. compile-if-changed. However, this may lead to problems if the available selector libs were compiled in previous sessions with a different set of loaded libraries (this is a general problem in ROOT). When this happens the best solution is to force recompilation (ACLiC mode '++'). To do this just add '++' to the name of the tutorial, e.g. runProof("event++") .

2.1 Cycle-driven processing

The PROOF tutorials include three examples of cycle-driven processing, i.e. of a generic task executed in parallel N times.

2.1.1 "simple": ProofSimple

Selector: tutorials/proof/ProofSimple.h, tutorials/proof/ProofSimple.C
PAR file: *none*

This is an example of a random gaussian generation filling a configurable number of 1D histograms.

2.1.2 "event": ProofEvent

Selector: tutorials/proof/ProofEvent.h, tutorials/proof/ProofEvent.C
PAR file: tutorials/proof/event.par

This is an exampe of a simple Monte Carlo generation creating 'event'-like structures (see $ROOTSYS/test/Event.h). This example shows how to use a PAR package.

2.1.3 "pythia8": ProofPythia

Selector: tutorials/proof/ProofPythia.h, tutorials/proof/ProofPythia.C
PAR file: tutorials/proof/pythia8.par

This is similar to the previous example but with a real Monte Carlo generator, Pythia 8. It shows how to use a PAR package and how to set the relevant external variables.

2.2 Data-driven processing

The PROOF tutorials include three examples of data-driven processing, i.e. of processing steered by an exting TTree.

2.2.1 "h1": h1analysis

Selector: tutorials/tree/h1analysis.h, tutorials/tree/h1analysis.C
PAR file: *none*

This is the famous H1-analysis available since a long time under tutorials/tree. By default the data are read from the ROOT HTTP server. However, one can an change the location of the source.

2.2.2 "eventproc": ProofEventProc

Selector: tutorials/proof/ProofEventProc.h, tutorials/proof/ProofEventProc.C
PAR file: tutorials/proof/event.par

This is an example of data-driven analysis using a PAR package. It also shows how to vary the fraction of the event read for the analysis.

2.2.3 "friends": using TTree friends

Selector: tutorials/proof/ProofFriends.h, tutorials/proof/ProofFriends.C
PAR file: *none*

2.3 Large-output handling via files on workers

The PROOF tutorials include three examples of handling of large outputs via intermediate saving of the worker outputs on files local to the workers.

2.3.1 "simplefile": merging histograms via files

Selector: tutorials/proof/ProofSimpleFile.h, tutorials/proof/ProofSimpleFile.C
PAR file: *none*

2.3.2 "ntuple": merge simple ntuple (TTree) via files

Selector: tutorials/proof/ProofNtuple.h, tutorials/proof/ProofNtuple.C

PAR file: *none*

This example shows how to generate in parallel a simple ntuple, to save it in local files on the workers, and to automatically merge the files as part of the finalization phase.

2.3.3 "dataset": access simple ntuple (TTree) files via a dataset

Selector: tutorials/proof/ProofNtuple.h, tutorials/proof/ProofNtuple.C
PAR file: *none*

This example is similar to the previous one, except that the files, instead of being merge, are left on the workers and a dataset is created and registered on the master so that it can be automatically used in a subsequent run. In the example, the 'subsequent runs' are examples of drawing operation via PROOF.

2. Start a PROOF session using getProof.C

In this section we describe the macro 'getProof.C', located under $ROOTSYS/tutorials/proof, whihc is used by 'runProof.C' (and '$ROOTSYS/test/stressProof.C') to start a PROOF session, either locally (lite or standard) or to a remote cluster. When relevant, the macro checks for the state of the relevant daemon. For local daemons, the macro makes the necessary configurations steps and automatically (re)starts the daemon.

Signature:

```
TProof *getProof(const char *url = "proof://localhost:40000",
                 Int_t nwrks = -1, const char *dir = 0,
                 const char *opt = "ask", Bool_t dyn = kFALSE,
                 Bool_t tutords = kFALSE)
```

The first argument *url* is the url of the master where to start the PROOF session; use 'lite://' for a PROOF-Lite session. The default is 'proof://localhost:40000', i.e. a standard cluster on the local machine on port 40000. The macro tries to start the daemon if not found. The second argument *nwrks* specifies the number of workers for the session; it can only be fulfilled if the daemon allows the requested number (or if the daemon needs to be started and, therefore, configured) or in PROOF-Lite.

The other arguments apply only in the case the local daemon on port 40000 needs to be started:

1. *dir*:
   directory to be used for the files and working areas. When starting a new instance of the daemon the relevant files and directories in this directory are cleaned. If left null, the default is used: '/tmp/user/.getproof';
2. *opt*: defines what to do if an existing xrootd uses the same ports; possible options are: "ask", ask the user; "force", kill the xrootd and start a new one; if any other string is specified the existing xrootd will be used. Default is 'ask'. Note that for a change in 'nwrks' to be effective you need to specify 'force';
3. *dyn*: this flag can be used to switch on dynamic, per-job worker setup scheduling. Default is off;
4. *tutords*: this flag can be used to force a dataset dir under the tutorial dir. Default is no.

Examples:

1. Standard session with startup of the local daemon

   ```
   root [0] .L tutorials/proof/getProof.C+
   root [1] TProof *p = getProof()
   getProof: working area not specified temp
   getProof: working area (tutorial dir): /tmp/ganis/.getproof
   SysError in <:unixtcpconnect>: connect (localhost:40000) (Connection refused)
   getProof: xrootd config file at /tmp/ganis/.getproof/xpd.cf
   getProof: xrootd log file at /tmp/ganis/.getproof/xpdtut/xpd.log
   (NB: any error line from XrdClientSock::RecvRaw and XrdClientMessage::ReadRaw should be ignored)
   getProof: waiting for xrootd to start ...
   getProof: xrootd pid: 11681
   getProof: start / attach the PROOF session ...
   Starting master: opening connection ...
   Starting master: OK
   Opening connections to workers: OK (4 workers)
   Setting up worker servers: OK (4 workers)
   PROOF set to parallel mode (4 workers)
   root [2]
   ```

2. PROOF-Lite session with 8 workers

   ```
   root [0] .L tutorials/proof/getProof.C+
   root [1] TProof *p = getProof("lite://", 8)
   getProof: trying to open a PROOF-Lite session with 8 workers
    +++ Starting PROOF-Lite with 8 workers +++
   Opening connections to workers: OK (8 workers)
   Setting up worker servers: OK (8 workers)
   ```

```
PROOF set to parallel mode (8 workers)
root [2]
```

Special PROOF tags and branches

The PROOF system is still in development and new features an fixes are provided regurarly in the trunk of the SVN repository. However, it is not always possible to update the full ROOT distribution to the trunk. In this pages we give information about the PROOF dev tags and branches. The purpose of these SVN branches is to integrate official ROOT tags with the latest PROOF.

ROOT Version v5-26-00 Proof

This branch consist of the latest 5-26-00-patches integrated with the latest PROOF develpments from the SVN trunk. The branch is developed at

```
https://root.cern.ch/svn/root/branches/dev/proof/branches/v5-26-00-proof
```

Tags on the branch are created under

```
https://root.cern.ch/svn/root/branches/dev/proof/tags
```

The version on the development branch has cycle +49 with respect to the reference ROOT tag, e.g. 5.26/49 . The first tag on the branch has version 5.26/50. The next development cycle 5.26/51. And so on. Tags have always even cycle numbers.

Changes in the head of v5-26-00-proof wrt v5-26-00-proof-04

- New functionality
- Improvements
  - Add support for the ROOTrc variable 'Proof.UseMergers' to trigger the use of submergers for merging. This has the same meaning of the parameter 'PROOF_UseMergers'. The setting can be always superseeded by the user setting of the 'PROOF_UseMergers' in the input list (or of the local ProofUseMergers setting).
  - Add test in stressProof for package argument setting.
- Fixes
  - Lock the package directory during the execution of the modified SETUP to avoid clashes between multiple workers on the same machine.
  - Correctly register the status code of building and loading packages on the workers, and transmit it to the client.
  - Fix problem with packet re-assignment in case of a worker death. Some packets were processed twice or more times.
  - Fix problem with transmission of EnablePackage arguments to workers.
  - Fix an issue with a semaphore closing actions and asynchronous processing.This should remove a problem sometimes happening at the end of stressProof where the main application goes into timeout when closing the PROOF session.

Tag v5-26-00-proof-04

Details

- Version number: 5.26/54
- SVN branch
  - https://root.cern.ch/svn/root/branches/dev/proof/tags/v5-26-00-proof-04
- Source tarball:
  - ftp://root.cern.ch/root/files/root_v5.26.00-proof-04.source.tar.gz
- Binaries:
  - SLC5, x86_64, gcc 4.3, LCG config + ApMon interface:
    - ftp://root.cern.ch/root/files/root_v5.26.00-proof-04.Linux-slc5_amd64-gcc4.3.tar.gz
-

Changes in v5-26-00-proof-04 wrt v5-26-00-proof-02

- New functionality
  - Identify which TSelector data members point to which output list object on the workers, and if consistent set the client's data members to point to the corresponding objects of the merged output list.
- Improvements
  - Add support for arguments in the SETUP function: it is possible now to pass a string
    Int_t SETUP(const char *opt)
    or a list of objects
    Int_t SETUP(TList *optls);
    Two interface methods have been added to support this:
    Int_t TProof::EnablePackage(const char *package, const char *opt, Bool_t notOnClient);
    Int_t TProof::EnablePackage(const char *package, TList *optls, Bool_t notOnClient);
  - Allow building packages in the global package directories if 'write' permissions are available; this facilitates management of global packages from an authorized account.
  - Optimize the validation step in the case not all the entries are required. The validation step is stopped as soon as the requested number of events is reached. If the parameter "PROOF_ValidateByFile" is set to 1, the number of files is exactly what needed; otherwise the number of files may exceed the number of fles needed by #workers-1.
  - New Xrootd version 20100602-0830 .
- Fixes
  - Make sure the return code from BUILD.sh is properly checked and execution flow stop on failure;
  - Fix a subtle bug affecting the (possibly rare) case when not all entries are required and # entries does not correspond to an complete subset of files (e.g. # entries = 1001000 with files of 100000 entries each).
  - Couple of fixes in h1analysis (see #33866);
  - In TXSocket, make sure that the socket cannot be closed (and deleted) while processing an asynchronous message (in a

separated thread).
    - Fix issue with the ROOT_VERSION_CODE preventing library compatibility with 5.26/00x.

Tag v5-26-00-proof-02

Details

- Version number: 5.26/52
- SVN branch
    - https://root.cern.ch/svn/root/branches/dev/proof/tags/v5-26-00-proof-02
- Source tarball:
    - ftp://root.cern.ch/root/files/root_v5.26.00-proof-02.source.tar.gz
- Binaries:
    - SLC5, x86_64, gcc 4.3, LCG config + ApMon interface:
        - ftp://root.cern.ch/root/files/root_v5.26.00-proof-02.Linux-slc5_amd64-gcc4.3.tar.gz
-

Changes in v5-26-00-proof-02 wrt v5-26-00-proof-00

- New functionality
    - Add possibility to disable the graphic progress dialog via a call to TProof::SetProgressDialog(kFALSE).
- Improvements
    - Updated man pages for the PQ2 tools
    - New tutorial 'friends' to illustrate the use of TTree friends
    - New packetizer TPacketizerFile to generating packets which contain a single file path to be used in processing single files. Used, for example, in tasks generating files, like in the forthcoming PROOF bench or the "friends" tutorial.
    - In TFileMerger, implement merging of THStack objects as done already in hadd
    - Remove data directory in TProofServ::Terminate (and alike) when empty: avoid having many empty directories around (patch #33693)
- Fixes
    - In TXProofMgr import fix #33639: fix problem affecting TProofMgr::Find.
    - Import fix #33640: fix a few issues affecting the usage of tree friends in PROOF and a bug affecting the locality check for files in TDSet.
    - In TProof::ClearData() import fix #33641: fix problem observed when the dataset repository is empty.
    - In TProof import fix #33648: fix an issue with the workers names in TSlaveInfo in PROOF-Lite .
    - Fix a few issue affecting test/ProofBench on MacOsX
    - In Xrootd, import fix #33677 fixing a fake authentication failure with proxies created by voms-proxy-init when the input certificates are PKCS12-formatted
    - In TProofLite, make sure that the default sandbox is always ~/.proof and that the package lock file is under gSystem->TempDirectory()

Tag v5-26-00-proof-00

Details

- Version number: 5.26/50
- SVN branch: https://root.cern.ch/svn/root/branches/dev/proof/tags/v5-26-00-proof-00
- SLC5/x86_64/gcc43 binaries:
    - Optimized: /afs/cern.ch/user/g/ganis/scratch0/proof/5-26-00-proof-00/x86_64-slc5-gcc43-opt/root
    - Debug:
-

Changes in v5-26-00-proof-00 wrt v5-26-00-patches

- New functionality
    - Add support for *processing many datasets in one go* in *TProof::Process(const char *dataset, ...)*.
      Two options are provided:
       - *'grand dataset'*: the datasets are added up and considered as a single dataset (*syntax*: "dataset1|dataset2|...")
       - *'keep separated'*: the datasets are processed one after the other; the user is notified in the selector of the change of dataset so she/he has the opportunity to separate the results. A new packetizer, TPacketizerMulti, has been developed for this case: it basically contains a list of standard packetizers (one for each dataset) and loops over them (*syntax*: "dataset1,dataset2,..." or dataset1 dataset2 ...").
      In both cases, entry-list can be applied using the syntax "dataset The datasets to be processed can also be specified on one or multiple lines in a text file.
      See http://root.cern.ch/drupal/content/working-data-sets for more details.
    - Add support for automatic download of a package when available on the master but not locally. The downloaded packages are store under */packages/downloaded* and automatically checked for updates against the master repository. If a local version of the same package is created (using the UploadPackage) the entry in *downloaded* is cleared, so that the behaviour is unchanged.
      The new functionality is described in http://root.cern.ch/drupal/content/working-packages-par-files .

    - Add the possibility to remap the server for the files in a dataset. This allows, for example, to reuse the dataset information for the same files stored in a different cluster.
    - Add a local cache for TDataSetManagerFile. This is mainly used to improve the speed of TDataSetManager::ShowDataSets, which is run very often by users and may be very slow if the number of dataset is large. The cache is also used to cache frequently received dataset objects.
    - Add the possibility to audit the activity on the nodes via syslog. See http://root.cern.ch/drupal/content/enabling-query-monitoring.
- Improvements
    - Improve support for valgrind runs in PROOF-Lite
    - Add the possibility to add files to a dataset. This is achieved with a new option 'U' (for update) to RegisterDataSet.
    - Add methof TProof::GetStatistics to allow the client to retrieve the correct values of fBytesRead, fRealTime and fCpuTime at any moment; this will be used to setup a sort of ROOTmarks in stressProof .
    - Several improvements in the test program 'stressProof' and in the tutorials under 'tutorials/proof'
    - Avoid contacting the DNS when initializing TProofMgr as base class of TProofMgrLite: it is not needed and it may introduce long startup delays.
    - Make TProof::LogViewer("") start the viewer for a Lite session, in parallel to whats happen for TProof::Open("").
    - Several improvements in the handling of wild cards in the dataset manager; for example, issuing a GetDataSet(...) on a dataset URI containign wild cards will return a grand dataset sum of all the datasets matching the URI.
    - Add options to get a list of all dataset registered names from ScanDataSets (option kList; the result is a TMap of {TObjString, TObjString} with the second TObjString empty).
    - Improved version of the PQ2 scripts; the scripts now invoke a dedicated ROOT application (named *pq2*) available under $ROOTSYS/bin .
    - Add support for recursive reading of group config files via the 'include sub-file' directive. This allows to have a common part and, for example, customize differently the quotas.
- Fixes
    - Fix a bug in error status transmission which avoid session freezing in some cases
    - FIx a few issues in libXrdProofd.so with handling of connection used for admin operation: this should solve some cases where the daemon was not responding.
    - Fix a few memory leaks showing up when running several queries in the same session
    - Fix a few issues affecting the new sub-merging option
    - Fix an issue preventing proper real-time notification during VerifyDataSet
    - Fix an issue with TQueryResult ordering (was causing random 'stressProof' failures)
    - Fix an issue with TProof::AskStatistics (fBytesRead, fRealTime and fCpuTime were not correctly filled on the client; the values on the master, displayed by TProof::Print were correct).
    - Fix several small issues affecting the handling of global package directories
    - Fix an issue with socket handling in the main event-loop while sendign or receiving files via TProofMgr.
    - Fix a problem counting valid nodes in sequential or 'masteronly' mode, generating the fake error message "GoParallel: attaching to candidate!"
    - Fix a few issues with the length of Unix socket paths affecting PROOF-Lite and xproofd on MacOsX
    - Fix an issue with the release of file descriptors when recovering sessions .
    - Fix an issue with a fake error message `("`Error in <:cd>: No such file root:/") in PROOF-Lite when issuing TProof::SetParallel().
    - Fix a problem with negative values for 'workers still sending' in PROOF-Lite .
    - Fix locking issue while building packages locally.
    - Fix issue setting permission and ownership of the dataset user directories.

Benchmark results

Starting from dev version 5.33/02 (and version 5.32/01, 5.30/06), the TProofBench steering class provides the static method GetPerfSpecs which can be used - used after a CPU benchmark - to extract some numbers giving an idea of the cluster specs.

The selector used for the measurement is ProofSimple.h,.C which can be found undertutorials/proof . The selector fills 16 histograms with 1000000*workers Gaussian random numbers. Each 'unit' corresponds, therefore, to the generation of 16 gaussian number and the filling of 16 histograms. We call this a 'random generation' and we measure the perfomance specs in RNGPS, i.e. RaNdom Generations Per Second . We measure both the maximum RNGPS provided by the PROOF cluster and the normalized, per-worker, RNGPS .

This page is supposed to collect results from cluster where the measurement has been done.

Description host	**# workers**	**Max Rate (megaRNGPS)**	**Nwrks at max**	**Norm Rate (megaRNGPS/worker)**	**Comments**
PROOF-Lite plitehp24.cern.ch	24	11.007	24	0.450	24-core HP
PROOF plitehp24.cern.ch	24	11.000	24	0.453	24-core HP (standard PROOF, remote connection)
PROOF atlasui.lnf.infn.it	30	13.449	30	0.449	PoD w/ gLite on ATLAS IT Tier2
PROOF kiaf.sdfarm.kr	96	37.680	95	0.386	Alice AF at KISTI (8 x 12-core nodes)

Contacts

How to get help, send comments and/or submit requests for improvements

The Root Talk forum provides a dedicated section where problems and developments on PROOF are discussed.

The mailing list rootdev@cernSPAMNOT.ch are also available for discussion and request for help.

Finally, bugs and feature requests can be submitted via the ROOT JIRA portal (category PROOF).

The responsible and main developer of the PROOF project is Gerardo Ganis .

PROOF Frequently Asked Questions

This page is meant to collect answers to frequently raised issues. By its nature this page will be in continuous evolution.

---

1. **How to change the location of the sandbox (working directory) in PROOF-Lite**
The sandbox location is controlled by the ROOT configuration variable 'ProofLite.Sandbox'; the default is '~/.proof'. The value must be changed before starting the PROOF-Lite session, i.e.

```
root[] gEnv->SetValue("ProofLite.Sandbox", "/pool/proofbox")
root[] TProof *proof = TProof::Open("")
```

It can also be set for all runs in the $HOME/.rootrc file)or in the relevant 'rootrc' file):

```
# Changing the location of the PROOF-Lite sandbox
ProofLite.Sandbox: /pool/proofbox
```

2. **PROOF gets stuck when reading data via the DPM or dCache xrootd doors**
This misbehavior is caused by the fact that the XrdClientAdmin Locate function for those implemenations is not correctly implemented; as a consequence, during look-up the file URL is replaced by an URL which is not usable by PROOF, making all open attempts to end up in timeouts. The solution to this problem is to skip the look-up step. This can be achieved by setting the parameter "PROOF_LookupOpt" to "none" before running the query, i.e.

```
root[] TProof *proof = TProof::Open(...)
root[] proof->SetParameter("PROOF_LookupOpt", "none")
root[] proof->Process(...)
```

The setting can also be done for all queries in the relevant 'rootrc' file read by the master (or the client sesssion, in the case of PROOF-Lite) via the variable 'Proof.LookupOpt'; for example, for a standard cluster, the following setting can be add to the xproofd config file on the master:

```
# Disable lookup for all queries
xpd.putrc Proof.LookupOpt none
```

Note that priority is always given to the settings of the current query: if lookup is disabled by default and it is needed for one query (for pure xrootd back-ends is advised) it can be enforced with

```
root[] proof->SetParameter("PROOF_LookupOpt", "all")
```

3. **XROOTD-dependent modules disabled by configure; XROOTD binaries and libraries not found**
Starting with ROOT 5.32/00 the XROOTD package is not distributed anymore with ROOT. To build the ROOT plug-ins libNetx, libProofx and the ROOT binary xproofd, the relevant includes and libs are taken form an external standard installation. See the ROOT prerequisites page and the page dedicated to XROOTD.
4. **Is the TTreeCache enabled by default in PROOF?**
Yes. To disable it set the PROOF parameter 'PROOF_UseTreeCache' to 0 before exectuing Process, i.e.

```
root[] proof->SetParameter("PROOF_UseTreeCache", 0)
root[] proof->Process(...)
```

The cache size is controlled by the parameter 'PROOF_CacheSize' (value in bytes). See also the dedicated section in the PROOF parameters page.
You can also control the cache via the ROOTrc variables ProofPlayer.UseTreeCache and ProofPlayer.CacheSize to be set in the relevant .rootrc file or via xpd.putrc .
5. **Settings in TSelector::Begin are ignored on workers**
The Begin method of the selector is used for things which have exclusively to be done on the local (client) session; therefore is not called on the workers. The method SlaveBegin is called everywhere (locally and workers) and is the place where you should do the initialization. Some documentation is available at 'Developing a TSelector'.
6. **Sharing dataset meta-information**
To share meta-information about datasets, each user must initialize the TDataSetManager instance on the master from the same dataset repository directory. This is achieved via the directive xpd.datasetsrc in the master xproofd. For example:

```
if masternode
xpd.datasetsrc file url:/shared/dir/dataset opt:Av:Ar:
fi
```

allows to share the directory /shared/dir/dataset, with each user space under

```
/shared/dir/dataset/proofgroup/username
```

with *proofgroup* the PROOF group ('default' if not specified; see http://root.cern.ch/drupal/content/defining-groups-users). The directories /shared/dir/dataset/proofgroup must be created by the admin and should be writable and readable by all users; typically their are owned by an admin user. Then PROOF will create the user subdirs, owned by the *user*, where the dataset metafiles (a ROOT file datasetname.root containing two TFileCollection) are stored.

7. **Changing the ROOT version run by master and workers**
   Using the directive xpd.rootsys is possible to define a set of ROOT versions which can be run by the workers. See also Changing the default ROOT version.
8. **Checking the result of Process**
   After processing, the output list contains an instance of TStatus named 'PROOF_Status' which contains the exit status of the query, which can be obtained with TStatus::GetExitStatus():

   ```
   proof->Process(...);
   TStatus *st = (TStatus *) proof->GetOutputList()->FindObject("PROOF_Status");
   if (st->GetExitStatus() != 0) { //Failure ... }
   ```

   There are three possible values, with the following meaning: 0 = success, 1 = stopped, 2 = abort.
   In the case some files (or portion of files) were not processed, the output list contains a list 'MissingFiles' with the TFileInfo instances describing the missing files. The methods TProof::ShowMissingFiles and TProof::GetMissingFiles allow a faster access to the information.
9. **Limiting number of workers**
   It is possible to limit the number of workers started for a session using the 'workers=N' option to TProof::Open(...); for example, the following starts 10 workers:

   ```
   root [0] TProof *p = TProof::Open("master.some.wr", "workers=10")
   Starting master: opening connection ...
   Starting master: OK
   +++ Starting max 10 workers following the setting of PROOF_NWORKERS
   Opening connections to workers: OK (10 workers)
   Setting up worker servers: OK (10 workers)
   PROOF set to parallel mode (10 workers)
   root [1]
   ```

   In some cases, it may be convenient to start the master only (to test connection or to do dataset management operations); that can be achieved using option 'masteronly':

   ```
   root [0] TProof *p = TProof::Open("master.some.wr", "masteronly")
   Starting master: opening connection ...
   Starting master: OK
   PROOF set to sequential mode
   root [1]
   ```
10. **The 'xproofd' executable or the plug-ins libNetx, libProofx are not found or not available**
    Most likely XRootD was not available during the build; see this FAQ.
11. ...