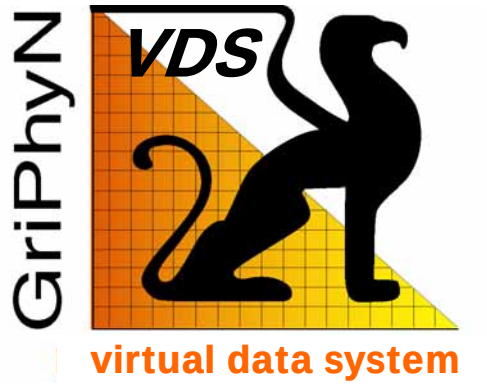
VDS
Virtual Data System

Users Guide

VDS Release 1.4



Contents

1 Getting Started with the VDS

Revision $Revision: 1.3 $ of file $RCSfile: VDSUG_GettingStarted.xml,v $.

1.1 Getting Set Up

The Virtual Data System (VDS) provides a means to describe a desired data product and to produce it in the Grid environment. The VDS provides a catalog that can be used by application environments to describe a set of application programs ("transformations"), and then track all the data files produced by executing those applications ("derivations"). The VDS contains the "recipe" to produce a given logical file, and on request produces a directed acyclic graph of program execution steps which will produce the file. (NEEDS MORE INTRO)

This is a quickstart guide to using the Virtual Data System. It is intended to give prospective users a quick guide of the release and to solicit feedback on the release's suitability for applications. It covers only the externally visible features of the release, not the details of the system's internal design.

Before we start using the vds and running on the grid, a few assumptions have been made. All of these steps must be complete before the user installs the latest version of the vds and can submit a workflow to the grid.

The user already has access to a machine with the virtual data toolkit (http://www.cs.wisc.edu/vdt/index.html) properly installed to use as a submit host.

A valid Department of Energy certification (http://www.doegrids.org/pages/cert-request.html) has been created and applied to a virtual organization.

If the user is new to the submit host server or have never run a workflow on the grid, then we recommend running a few tests to ensure that everything is properly configured. These simple steps will save the user future headaches.

Test 1. Check the version of java:

```
$ java -version
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_04-b05)
Java HotSpot(TM) Client VM (build 1.4.2_04-b05, mixed mode)
```

Test 2. Check if your GLOBUS_LOCATION is set

```
$echo $GLOBUS_LOCATION
/vdt/globus
```

If the environmental variable is not set or there is an error, ask your administrator for the information to set the environment variable. Then step is to set up the globus environment

```
$source $GLOBUS_LOCATION/etc/globus-user-env.sh
```

Test 3. Now generate a proxy certificate from your X509 credentials.

```
$grid-proxy-init
Your identity: /DC=org/DC=doegrids/OU=People/CN=Douglas Scheftner
123456
Enter GRID pass phrase for this identity:
Creating proxy ..................................................
Done
```

Test 4. Manually test if you can authenticate to the grid resource that you are trying to submit your jobs to.

```
$globusrun –a –r <hostname>/jobmanager-fork
```

If you get an authentication error check the /etc/grid-security/grid-map file on the machine you are trying to authenticate to and see if your x509 cert DN is mapped to the user. If not ask your administrator to add you to the grid-mapfile.

Test 5. The next test is using the globus-url-copy command to ensure that file transfer is possible. Take a file that you want to test with and execute the following command.

```
$globus-url-copy gsiftp://server/YourHomeDirectory/your_test_file
gsiftp://remote_server/YourHomeDirectory/your_test_file
echo $?
0
```

If the echo does not return a value of 0, please contact your administrator for help. For more information on globus-url-copy, please refer to http://www.globus.org/toolkit/docs/4.0/data/gridftp/

Test 6. To make sure the user can submit jobs into condor try checking the status of the condor queue

```
$condor_q
-- Submitter: ept.uchicago.edu : <123.456.789.01:23456> :
ept.uchicago.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

0 jobs; 0 idle, 0 running, 0 held
```

If the above tests work you are in good shape, otherwise please contact your friendly neighborhood administrator.

1.2 Downloading and Configuring the VDS

The first step to using the Virtual Data System is to download the vds-download script from the our twiki located at http://evitable.uchicago.edu/twiki/bin/view/VDSWeb/VDSQuickStart. This script was designed to simplify the installation and configuration for the new user. The following

is a quick synopsis of what the script will accomplish. The user has the option to download the nightly version or the official release. The VDS team recommends downloading the nightly version. This will ensure that all of the latest features are included in the release. Once the version to download is chosen, the script will create, if it doesn't already exist, a directory called vds-version under $HOME. Under vds-version a secondary directory that will be created. If the nightly version is downloaded, the directory will be named after the current date. If the download is the official release, the name of the directory will be the #.#.# version. This is to organized the nightly and official downloads. If the version in the nightly build, the vds-download script will then download and untar the proper version according to the glibc version and cpu architecture of the server. Once the proper version is downloaded, untarred and the proper symlink is created, the user has the option to execute a second script called vds-config. This script is located under $HOME/vds/bin/ will configure and test your vds distribution to ensure that it is functioning properly. The vds-config script will create a valid tc.data, pool.config, .wfrc and .vdsrc files with the user input. The script will also ask for some database information that will be used to set up your virtual data catalog.

```
dscheftn@linux:~> ./downloadvds
The version of the glibc on this system is 2.3.2
The architecture of this system is i686
The version of the vds that your system needs is vds-binary-1.3.9-
linux-i686-glibc232-20050809.tar.gz
Will download vds-binary-1.3.9-linux-i686-glibc232-20050809.tar.gz to
/home/dscheftn
Proceed with download? [y/n] ?y
--14:10:20--  http://vds.isi.edu/cvs-nightly/vds-binary-1.3.9-linux-
i686-glibc232-20050809.tar.gz
           => `vds-binary-1.3.9-linux-i686-glibc232-20050809.tar.gz'
Resolving vds.isi.edu... done.
Connecting to vds.isi.edu[128.9.176.20]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 20,284,341 [application/x-tar]

100%[=================================================================
==============================================================>]
20,284,341   222.25K/s    ETA 00:00

14:11:49 (222.25 KB/s) - `vds-binary-1.3.9-linux-i686-glibc232-
20050809.tar.gz' saved [20284341/20284341]

Untarring vds-binary-1.3.9-linux-i686-glibc232-20050809.tar.gz in vds-
versions/20050809
Do you want to configure your vds? [y/n]y
vds-config: Error: Your GLOBUS_LOCATION environment variable is not
set.
Do you want the VDS environmental variables set up in your login
script? [y/n]y
Do you use the (1) bash ,(2) sh ,(3) csh, (4) tcsh?
2
Do you want to test your vds configuration? [y/n]y

Hi, this is a small test program in form of a perl script. Whenever I
need to know something from you, you can accept the defaults by hitting
enter. Sometimes I just stop, to let you catch up reading on what I did
Let's get started.
```

```
[hit enter]

<<< This part was omitted >>>

 ------------------------------------------------------
The installation and configuration of the vds is complete
 ------------------------------------------------------
 Please use the helloworld.vdl found in
/home/dscheftn/vds/examples/vdl/helloworld for the HelloWorld example
```

If you make it to this point without much trouble, then you are in good shape. You should be able to run a simple hello world job using the Euryale planner from your submit host to the sofagrid.

1.3 Running "helloworld" on the grid

Included in the VDS distribution are two example workflows the user should run to ensure that the grid software is running properly and to give a feel for what the VDS can do. The first workflow is aptly named "helloworld". This workflow will create a file that will says "Hello World". The example job that we will use is located at $VDS_HOME/examples/vdl/helloworld/helloworld.vdl. It is recommended that the user should copy this file into their own working directory to use.

The first step involved in preparing a workflow for the grid is to convert the textual virtual data language into its xml equivalent. If your VDL is syntactically correct, this command will be silent. Otherwise, you will see syntax error messages to indicate approximately where in the input file the error occurred. The next step is to insert your .xml definitions into your virtual data catalog. The database information is added into the database that you specified during the config-vds script. Insertvdc will only insert new definitions into the virtual data catalog. Use the command updatevdc if the user wants to update a definition that already exists in the catalog.

A simple way to test what definitions are in the database is to query it using searchvdc.

```
$ searchvdc -n ivdgl
2005.06.08 13:44:38.944 CDT: [app] searching the database
2005.06.08 13:44:38.947 CDT: [app] wildcard search requested
2005.06.08 13:44:40.468 CDT: [app] found 1 matches total
ivdgl::hello->tut::echo
```

The next step is to create the abstract direct acyclic graph in xml, better known as a "dax" file. In order to obtain a workflow, the dependencies have to be converted into an "abstact DAG in XML" or "dax". The dax file contains logical descriptions of jobs to run and files to tranfer, but no concrete information like the application name or the physical filename. For a quick explanation of the arguments for gendax, -l is used to give the workflow a label or name. A label is needed to distinguish this workflow from other workflows running from your submit host. Since it is possible within the VDS to run simultaneous workflows, if two workflows have the same label, then they will interfere with each other. The name given to the -o argument will be the name of the dax file that is produced. The -D argument is used to request which derviation that you want to use in your workflow. All of these steps may appear to be a bit complicated, but

fortunately, there is a command that will take care of all of these steps and produce the helloworld.dax file in one command. This command is vdlc.

```
$ vdlc helloworld.vdl -u -o helloworld.dax
2005.08.15 10:37:58.121 CDT: [app] parsing "helloworld.vdl"
2005.08.15 10:37:59.118 CDT: [app] Trying to add TR tut::echo
2005.08.15 10:37:59.132 CDT: [app] Trying to add DV dscheftn::hello
2005.08.15 10:37:59.188 CDT: [app] requesting dscheftn::hello (1/1)
2005.08.15 10:37:59.217 CDT: [app] saving output to helloworld.dax
```

Now we need to create a concrete dag and plan the run using the .dax file we previously created and the command vds-plan. vds-plan uses as an argument the -b for the base directory in which to you want all of the files generated by the vds suite of tools to be located. The rest of the grid planning is done automatically. The user should receive similar output when the vds-plan command is used.

```
$ vds-plan -g ivdgl1 helloworld.dax
# using wfrc location default
# parsing properties in /home/dscheftn/.wfrc...
# parsing properties in /home/dscheftn/.vdsrc...
# VDS_HOME=/home/dscheftn/vds
# using sft file /home/dscheftn/vds/contrib/Euryale/grid3.sft
# using d2d from /home/dscheftn/vds/bin/d2d
# 14 lines in dax file
# workflow label test
# job count 1

Info: You use a workflow label of "test", how boring! It makes
distinguishing
your workflow from others difficult. Provenance becomes meaningless.
Please
consider using the --label or -l switch of gendax next time. Even using
the name
"helloworld" may help.

# good, base directory is absolute
# using base directory /home/dscheftn/run
# found /home/dscheftn/run/ivdgl1
# found /home/dscheftn/run/ivdgl1/test
# using rundir basename run0002
# created rundir /home/dscheftn/run/ivdgl1/test/run0002
# copied DAX into /home/dscheftn/run/ivdgl1/test/run0002/helloworld.dax
# /home/dscheftn/vds/bin/d2d --template
/home/dscheftn/vds/contrib/Euryale/grid3.sft --dir
/home/dscheftn/run/ivdgl1/test/run0002
/home/dscheftn/run/ivdgl1/test/run0002/helloworld.dax
2005.08.15 10:39:07.837 CDT: [default] starting
2005.08.15 10:39:08.768 CDT: [default] using 0 directory levels
2005.08.15 10:39:08.819 CDT: [default] created 1 structured filenames.
2005.08.15 10:39:08.819 CDT: [default] created 1 flat filenames.
# ran d2d successfully
# created job state log
/home/dscheftn/run/ivdgl1/test/run0002/jobstate.log
# created brain dump in
/home/dscheftn/run/ivdgl1/test/run0002/braindump.txt
# registered workflow into workman
```

```
I have concretized your abstract workflow. The workflow has been
entered
into the workflow database with a state of "planned". The next step is
to execute your workflow. The invocation is thus (or similar):

vds-run  /home/dscheftn/run/ivdgl1/test/run0002
```

The next step is to actually run your helloworld grid job on the sofagrid. The argument to use was given in the last output line of the vds-plan.

```
$ vds-run --debug /home/dscheftn/run/ivdgl1/test/run0002/
# parsing properties in /home/dscheftn/.wfrc...
# parsing properties in /home/dscheftn/.vdsrc...
Warning: run directory mismatch, using
/home/dscheftn/run/ivdgl1/test/run0002/
# slurped /home/dscheftn/run/ivdgl1/test/run0002/braindump.txt
# found /home/dscheftn/vds/bin/tailstatd
# GLOBUS_LOCATION=/vdt/globus
# found /vdt/globus/bin/grid-proxy-info
# GLOBUS_TCP_PORT_RANGE=49152,53247
# found /vdt/globus/lib
# grid proxy has 43197 s left
# found /home/dscheftn/vds/bin/vds-submit-dag
# found /home/dscheftn/run/ivdgl1/test/run0002/test-0.dag
# /home/dscheftn/vds/bin/vds-submit-dag -d 0
/home/dscheftn/run/ivdgl1/test/run0002/test-0.dag
# parsing properties in /home/dscheftn/.wfrc...
# parsing properties in /home/dscheftn/.vdsrc...

Checking all your submit files for log file names.
This might take a while...
Done.
-----------------------------------------------------------------------
File for submitting this DAG to Condor           :
/home/dscheftn/run/ivdgl1/test/run0002/test-0.dag.condor.sub
Log of DAGMan debugging messages                 :
/home/dscheftn/run/ivdgl1/test/run0002/test-0.dag.dagman.out
Log of Condor library debug messages             :
/home/dscheftn/run/ivdgl1/test/run0002/test-0.dag.lib.out
Log of the life of condor_dagman itself          :
/home/dscheftn/run/ivdgl1/test/run0002/test-0.dag.dagman.log

Condor Log file for all jobs of this DAG         : /tmp/test-7275.log
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster 7046.
-----------------------------------------------------------------------
# dagman is running
# /home/dscheftn/vds/bin/tailstatd
/home/dscheftn/run/ivdgl1/test/run0002/test-0.dag.dagman.out
# parsing properties in /home/dscheftn/.wfrc...
# slurped /home/dscheftn/run/ivdgl1/test/run0002/braindump.txt
# /home/dscheftn/vds/bin/tailstatd is running
# updated state in workman

I have started your workflow, committed it to DAGMan, and updated its
```

```
state in the work database. A separate daemon was started to collect
information about the progress of the workflow. The job state will soon
be visible. Your workflow runs in base directory

cd /home/dscheftn/run/ivdgl1/test/run0002/
```

A good command to use to watch the progress of the workflow in the condor queue is tail condor_q (FIX THIS)

```
$ condor_q

-- Submitter: evitable.uchicago.edu : <128.135.152.48:50298> :
evitable.uchicago.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
7046.0   dscheftn        8/15 10:41   0+00:00:31 R  0   2.6
condor_dagman -f -
7047.0   dscheftn        8/15 10:41   0+00:00:00 I  0   0.2  kickstart
-R 'evit

2 jobs; 1 idle, 1 running, 0 held

$ condor_q
-- Submitter: evitable.uchicago.edu : <128.135.152.48:50298> :
evitable.uchicago.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
7046.0   dscheftn        8/15  10:41   0+00:01:05 R  0   2.6
condor_dagman -f -
7047.0   dscheftn        8/15  10:41   0+00:00:04 R  0   0.2  kickstart
-R 'evit
2 jobs; 1 idle, 1 running, 0 held

$ condor_q
-- Submitter: evitable.uchicago.edu : <128.135.152.48:50298> :
evitable.uchicago.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
7046.0   dscheftn        8/15 10:41   0+00:02:09 R  0   2.6
condor_dagman -f -

$ condor_q
-- Submitter: evitable.uchicago.edu : <128.135.152.48:50298> :
evitable.uchicago.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

0 jobs; 0 idle, 0 running, 0 held
```

From the output of condor_q, the workflow is complete. The question that readily comes to mind is “how do we know it is complete?” and “where is the output of the workflow”. To answer these questions we will use the tool called vds-report. This tool is used to show the user all the errors and mishaps that can occur while running a workflow on the grid.

```
$vds-report -d run0002

**/test/run0002/euryale.log
   2 - 08/15 - 10:41:29.899 -  [14486] ID000001 projected for evitable

**/test/run0002/ID000001.dbg
  47 - 08/15 - 10:41:29.846 - [14486] chose site "evitable"
```

The results of vds-report indicate that the run was a success because vds-report did not find any errors. From the information given, vds-report found in the euryale.log and ID000001.dbg logs found that the job ID000001 successfully ran on evitable. We know since the job ran successfully on the server evitable, that we should look for the results in our directory on evitable. The place to look for the output depends on the cluster. Some cluster administrators setup a scratch or temporary area for users to run their jobs. Other administrators require the user to have an account on that cluster. If the administrators require an account, the user can find their work under their $HOME.

From our helloworld.vdl file, we know that the output file will be called dscheftn.hw.txt.

```
$ cd $HOME

$ ls -la dscheftn.hw.txt
-rw-r--r--    1 dscheftn dscheftn       12 Aug 15 10:41 dscheftn.hw.txt

$ less dscheftn.hw.txt
Hello World
```

Our helloworld run was successful.

1.4 Diamond DAG Example

Shown above (*to be inserted*) is the workflow of what we call the "diamond" DAG. The "document" symbols represent logical files (f.a, f.b.1, etc), and the bubbles represent jobs. The derivation name is printed bold while the transformation name is printed below. The workflow illustrates the fanning in and fanning out of data, for example, for the parallel execution and reuse of transformations for similar computations. The workflow uses one input file (f.a), which must be registered with the replica manager. Note that "replica manager" here is an abstract term for the mechanism that maps between logical and physical filenames.

```
$ vdlc blackdiamond.vdl -o blackdiamond.dax
2005.08.17 12:53:25.445 CDT: [app] parsing "blackdiamond.vdl"
2005.08.17 12:53:27.018 CDT: [app] Adding dscheftn::preprocess:1.0
2005.08.17 12:53:27.028 CDT: [app] Adding dscheftn::findrange:1.0
2005.08.17 12:53:27.031 CDT: [app] Adding dscheftn::analyze:1.0
2005.08.17 12:53:27.035 CDT: [app] Adding dscheftn::top:1.0
2005.08.17 12:53:27.052 CDT: [app] Adding dscheftn::left:1.0
2005.08.17 12:53:27.057 CDT: [app] Adding dscheftn::right:1.0
2005.08.17 12:53:27.068 CDT: [app] Adding dscheftn::bottom:1.0
2005.08.17 12:53:27.201 CDT: [app] requesting dscheftn::top:1.0 (1/4)
2005.08.17 12:53:27.239 CDT: [app] requesting dscheftn::right:1.0 (2/4)
2005.08.17 12:53:27.248 CDT: [app] requesting dscheftn::left:1.0 (3/4)
2005.08.17 12:53:27.250 CDT: [app] requesting dscheftn::bottom:1.0
(4/4)
2005.08.17 12:53:27.252 CDT: [app] saving output to blackdiamond.dax
```

1.5 Glossary of VDS Terminology

Abstract dag A workflow of VDL transformation calls ("derivations") in the form of a directed acyclic graph (aDAG) which uses logical names to represent both executables and files.

Abstract planner The VDS decision making engine which generates abstract DAGs in XML

Catalog host The catalog host (CH) has access to the Virtual Data Catalog. The abstract planning takes place on the catalog host.

CLI Command-line interface: An application that typically runs in a shell window.

Compute element Grid enabled cluster of compute hosts all of which share access to the file system of a Storage Element.

Concrete planner The Pegasus planner, which takes a DAX and generates DAGs.

Condor University of Wisconsin High Throughput Job Scheduler

Condor-G An extension to Condor that supports Globus

Concrete dag A concrete condor style DAG which can be submitted to the Condor DAGMan workflow executor. A concrete dag contains specific executable and input/output file locations.

DAG A directed acyclic graph that expresses the order in which a sequence of jobs is to run, and the dependencies between the jobs

DAGMan Condor's directed acyclic graph manager

DAX An abstract DAG expressed in XML

Derivation Recipe for or record of an instantiation of a transformation with specific arguments

Hint Hints, provided in the VDC and thus copied into the DAX, enable the VDS system to override default behaviors. For example, the pfnHint, if specified, overrides the data in the transformation catalog.

Logical file (LFN) A set of data contained in a file and given a name (the "logical file name") that is independent of the files location (i.e., on which site in the grid the file resides)

LRC The Local Replica Catalog contains knowledge for the LFN mapping.

Pegasus The Planning and execution system, that instantiates the abstract DAG and produces a concrete DAG, which it can send to DAGMan for execution.

Profile A profile allows to pass configuration information to later stages in the planning process in a system-independent manner.

Properties The configuration parameters for the VDS

PTC The Provenance Tracking Catalog, which keeps invocation records

Replica nodes The nodes which are added to a concrete DAG to transfer the materialized data from the execution pool to the output pool, and register the materialized data in the Replica Catalog.

RSL The Globus resource specification language, not to be confused with RLS

RLI The Replica Location Index, an index server that knows which LRCs have an LFN. Note, false positives must be checked with the LRCs that are referred

RLS The Replica Location Service, describes the set of RLI and one or more LRCs

Storage element Gridftp-enabled host that shares its file system(s) with one or more Compute Element.

Submit host The submit host executes the constructed DAG. Usually, it will be identical to the catalog host.

Transformation A VDL function definition, which defines the interface to any executable or script

Virtual data Denotes data objects whose method of production (or reproduction) is accurately known and represented in a metadata repository (specifically, this case, the virtual data catalog).

Virtual data language (VDL) A language that describes how a virtual data object is produced.

VDC Virtual data catalog. In this Beta release, the VDC consists of an ordinary file that stores a set of XML elements.

VDLt A textual representation of the VDL, intended to be human readable and producible.

VDLx An XML representation of the VDL, intended for application-to-application interchange.

VDS The virtual data system, including VDL processing tools, and the Pegasus and Euryale Grid workflow planners.

Wrapper Scripts Scripts that implement VDS command-line tools. These are provided in the bin directory of the VDS release, and call the underlying Java classes.

Transfer nodes The nodes which are added to a concrete DAG to get the data to the execution pool from the locations as returned from the replica catalog.

1.6 Directory Structure of the VDS

bin	contains the shell wrappers to the java classes
contrib	some unsupported contributions
com	sharable config files that frequently change (empty)
doc	documentation base directory
doc/javadoc	javadoc of all classes as far as known
doc/schemas	The generated documentation of the various XML schemas
etc	Single-machine configuration files that rarely change
example	some examples – not much there yet, refer to test/test?
lib	jar files necessary to run and/or compile
man	formatted manual pages for the CLI applications
man/man1	troff manual pages for the CLI applications
share	sharable config files that rarely change (empty)
sql	maintenance files for the SQL database backends
src	the source tree, only in the development distribution
test	an evolving battery of tests – not much there yet
var	Single-machine data files that frequently change

1.7 Major files of the VDS

etc/properties	Java VDS property file

etc/vdl-*n.nn*.xsd	VDLx XML schema definition, currently vdl-1.20.xsd
etc/dax-*n.nn*.xsd	DAX XML schema definition, currently dax-1.5.xsd
etc/iv-*n.nn*.xsd	XML schema definition for invocation records, currently iv-1.1.xsd
etc/poolconfig-*n.nn*.xsd	XML schema for the new pool.config file in XML, currently 1.4
etc/pool.config	The pool configuration file (to be set up by the user)
var/vds.db*	The VDLx database file, pointer, and backups
var/tc.data	The transformation catalog (to be set up by the user)
var/rc.data	The replica catalog stand-in for the shell planner only

2 The Virtual Data Language

Revision $Revision: 1.2 $ of file $RCSfile: VDSUG_VDLReference.xml,v $.

The Virtual Data Language (VDL) comes in two flavors, textual and XML. This document deals with the textual version of VDL, also known as VDLt.

The language syntax is shown using railroad diagrams. In these diagrams, rectangular boxes show terminals of the language; that is, tokens that must appear exactly as shown in the box. Round elements show non-terminals of the language. Non-terminals are place-holders for another rule.

2.1 Definitions

Each *Definition* is either a *Transformation* or *Derivation*. These two elements may be repeated in any order any number of times. The term *Definition* applies when talking about either element.

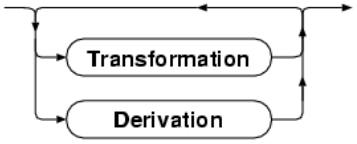


Figure 1: Definition, Transformation and Derivation.

2.2 Transformations

A transformation starts with the keyword **TR** followed by a fully qualified definition identifier (*fqdi*). This identifier comes in various shapes, and is explained in the next section.

The transformation identifier is followed by an open parenthesis, a list of formal arguments (*farg*-list), and a closing parenthesis. This list of formal arguments is optional. The parentheses are mandatory. These elements constitute the header of a transformation.

The body of a transformation constitues an opening curly brace, a list of optional transformation body elements, and a closing curly brace. The transformation body itself may be empty; however the curly braces are mandatory.

Transformations come in two flavors, simple or compound. The body elements determine the flavor of a transformation. If there is a *call* statement inside the body, the transformation becomes a compound transformation (*tr-comp-body*). The presence of an *argument* statement signals a simple transformation (*tr-simple-body*). While the *profile* body element may appear in both flavors of transformations, the *argument* and *call* element are mutually exclusive. The absence of both implies a simple transformation.

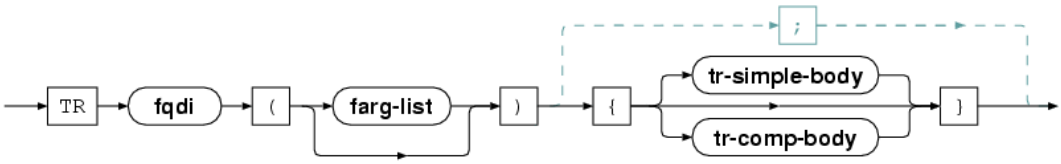


Figure 2: Make-up of a Transformation.

Note: Future versions may permit an empty transformation body by specifying a semicolon instead of an empty set of curly braces. The gray dashed line denotes this fact.

2.3 The Fully-qualified Definition Identifier

The *fqdi* uniquely identifies a *definition*. It consists of three parts:

1. a namespace,
2. a name (sometimes called identifier) within this namespace, and
3. a version number.

The namespace and version portions of the fully qualified definition identifier are optional. If a namespace is present, two colons[1] separate the namespace from the name. If a version is present, a single colon separates it from the name. There mustn't be any white-spaces between the different parts of a fully qualified definition identifier nor around the colons.

Figure 3: A fully qualified definition identifier.

It is highly recommended to use only characters in the namespace and name portion which are also permissible in the C programming language. We strongly suggest refraining from using characters not found in C identifiers. The following set of regular expressions shows the permissible characters in a TR or DV identifier:

```
# currently permissable characters
namespace   :== [a-zA-Z_./-][a-zA-Z0-9_./-]*
name        :== [a-zA-Z_./-][a-zA-Z0-9_./-]*
version     :== [0-9][.0-9]*
```

The following set of regular expressions shows the recommended characters in a TR or DV identifier:

```
# recommended characters to actually use
namespace   :== [a-zA-Z_][a-zA-Z0-9_]*
name        :== [a-zA-Z_][a-zA-Z0-9_]*
version     :== [0-9]+
```

[1] The two colons must be one token, no spaces between them.

Please also avoid periods, hyphens and slashes in identifiers. The period or the hyphen is a candidate in future releases to partition a namespace hierarchy. They would thus become a forbidden character. A hyphen in identifiers is problematic for the scanner, and is thus slated for removal in future releases.

```
/some/name/space::mydv   # namespace::name
some.other.scheme::tr2   # namespace::name
mimi:12                  # name:version
dv2                      # just name
```

2.3.1 The Namespace

The *namespace* must start with an alphabetical character, a slash, or an underscore. Alphanumerical characters, slashes, underscores, periods and hyphens may follow it. The inclusion of slashes and periods allows a user for self-imposed hierarchy creation in namespaces. There is no explicit support for namespace hierarchies yet.

Warning: Only such characters as are also permissible in C language identifiers should be used. The use of hyphens is strongly discouraged! Periods and slashes are discouraged unless used to denote a namespace hierarchy.

It is recommended to always use a namespace with any fully qualified definition identifier.

2.3.2 The Name

The *name*, sometimes also called *identifier*, must start with an alphabetical character. Alphanumerical characters, underscores or hyphens may follow it. The name is the minimal and mandatory member of any fully qualified definition identifier.

Warning: The use of periods, slashes and hyphens in the name is strongly discouraged!

2.3.3 The Version

The *version* must start with a numerical digit. Further digits or periods may follow it. Unfortunately, there are no user callbacks to compare versions correctly - versions are currently compared using string comparison. The version comparison becomes important when mapping a *derivation* to its *transformation* during the abstract planning process.

It is recommended to use only non-negative integers for the version number. Future releases are slated to make the version a true natural number. There are a couple of useful practices for the version number:

- Monotonously increasing integers from 0 or 1 upwards.
- A UTC timestamp, e.g. 1112031789.
- An ISO 8601 style timestamp, e.g. 20050828112233

2.4 The Formal Argument List

The formal argument list is optional. If formal arguments exist, each formal argument constitutes a *type* followed by the identifier of the formal argument *varname*. The *type* may be specified in a long or alternative short form. Permissible values for the type field are:

long	short	meaning

none		This type denotes a symbolic argument, either a string or a number that is to be passed verbatim. The omission of a type also denotes the symbolic argument.
input	**in**	The **input** type denotes filenames that are consumed, at least logically.
output	**out**	The **output** type denotes filenames that are produced.
inout	**io**	The **inout** type is reserved for compound transformations to denote temporary glue filenames. Glue Filenames are produced by a *call* in a compound transformation, and are consumed by another *call* in the same compound. The inout type is only permissable for argument declarations, not for *LFNs* themselves.

The formal argument identifiers follow the syntax of the C programming language for identifiers. The scope of formal arguments pertains to the *transformation* itself. If the identifier is associated with a list value, it must be followed by the open bracket and close bracket tokens.

It is permissible to assign default values to any formal argument. Default arguments are not, unlike C++, limited to the final formal argument declarations. They may occur where needed.

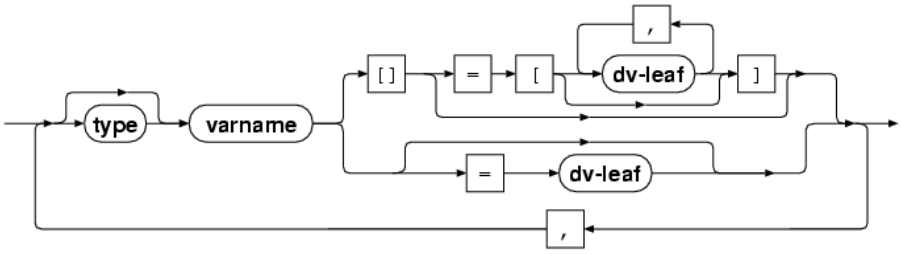


Figure 4: List of Formal Arguments

The rules for default arguments are similar to *actual arguments* in a *derivation*. If a simple scalar formal argument is of type none, the default argument must be a *text* element. If the scalar formal argument has any other type, the default argument must an *LFN*. Similar rules apply to list formal arguments. With list values, the list of default argument is enclosed in brackets. Each element within the list must be of the correct element. The list may be empty, denoted by a closing bracket immediately following an opening bracket.

Commas separate multiple formal arguments in the formal argument list. The following example just illustrates various ways to specify formal arguments, omitting the transformation body:

```
TR t1:1( foo, out bar ) {..}
TR t1:2( foo = "2.0", out bar ) {..}
TR t1:3( foo="2.0", out bar=@{output:"f1"} ) {..}
TR t1:4( foo, out bar = @{output:"f1"} ) {..}
```

The previous examples all illustrate passing a **none** and an **output** argument. The order of the arguments is at the time of this writing of no concern. In the first case, a caller must supply values

for both arguments, *foo* and *bar*. If no default exists, a binding must always be provided by the *derivation's* actual arguments.

In the next case, a caller may omit the binding of text for *foo*. In this case, the supplied default value is taken. If the caller choses to overwrite the default value, the caller must supply a value (binding) for *foo*.

2.5 The Simple Transformation Body

The simple transformation consists of any number (including zero) of *argument* and *profile* statements. Each statement must be terminated by a semicolon Pascal style.

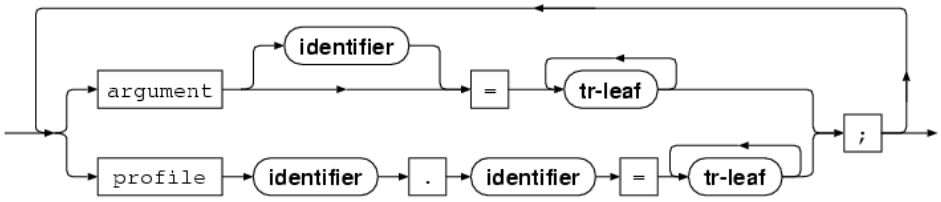


Figure 5: Body of a Simple Transformation.

An identifier may follow the **argument** keyword; however, it is optional in almost all cases. Since the identifier serves currently no purposes, it should best be left out. The equals character is mandatory, followed by one or more *tr-leaf* non-terminals. Permissible are the *text* constant, or a reference to a formal argument identifier (*use*). These are defined in a later section.

The **profile** keyword is followed by a the identifier for a profile's namespace. A period separates the namespace of the profile from the key within the namespace. Future release may need to modify this -- the double colon is being discussed. The mandatory equals character introduces one or more *tr-leaf* non-terminals. Permissable are *text* constants and references to formal argument identifiers. These are defined in a later section.

```
TR t1() { }     # simplest
TR t2( in f1 ) {
  argument = "-i " f1;
}
TR t3( in f1, out f2 ) {
  argument = "-i " f1;    # short
  argument = "-o " ${f2}; # or longer
  profile env.HOME = "/home/snej";
}
```

2.6 The Compound Transformation Body

A compound transformation consists of at least one *call* statement and any number of *profile* statements. Each statement must be terminated by a semicolon Pascal style.

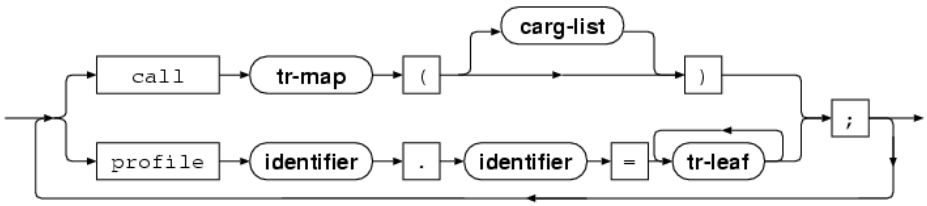


Figure 6: Body of a Compound Transformation.

The *call* statement is effectively an anonymous *derivation*. In other words, it behaves like a *derivation*, with some minor differences. A *call* does not feature a *fully qualified definition identifier* for itself. Since a *call* applies to a *transformation*, it offers the soft binding to matching *transformations* through the *tr-map* non-terminal described below.

Mandatory parenthesis follow the **call** reserved word. The list of call actual arguments (*carg-list*) is optional. Commas separate multiple entries. The *carg-list* non-terminal permits either string constants (*text*) or references to the enclosing transformation's formal argument identifiers (*use*).

The **profile** keyword is followed by a the identifier for a profile's namespace. A period separates the namespace of the profile from the key within the namespace. Future release may need to modify this -- the double colon is being discussed. The mandatory equals character introduces one or more *tr-leaf* non-terminals. Permissable are *text* constants and references to formal argument identifiers. These are defined in a later section.

```
TR t4( in f1, io f2, out f3 ) { # compound
  call t3( f1=${f1}, f2=${out:f2} );
  call t3( f1=${in:f2}, f2=${f3} );
}
```

2.6.1 The Compound Argument List

The actual argument binding inside a *call* statement only allows elements that are permissible inside a *transformation*: Either verbatim strings (*text*) or bound formal argument references (*use*).

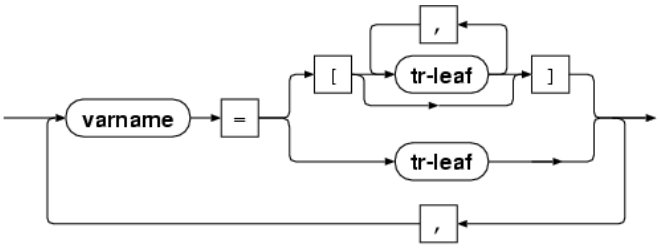


Figure 7: Argument list of Call Statements.

Each argument inside a *call* names the formal argument *varname* in the called *transformation*. This process is called binding. The equal character is followed either by a list or a scalar. If the called formal argument is a scalar, only the lower path may be taken. If the formal argument in the called transformation is a list, either the elements are enumerated between brackets, or a bound variable reference from the encompassing transformation may be used to pass the list.

2.7 Permissable Elements Inside a TR

The *tr-leaf* specifies the permissible arguments inside any *transformation*. The *tr-leaf* element permits either verbatim *text* or referencing bound variables (*use*). Please note in the railroad

diagrams which exhibit *tr-leaf* usage, if the *tr-leaf* element is a single element, a possible concatenation of *tr-leaf* elements, or a comma-separated list of simple *tr-leaf* elements.

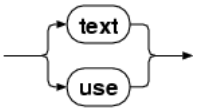


Figure 8: The tr-leaf *non-terminal.*

2.7.1 The Text Element

The *text* element is a C-style string in quotes. A backslash can be used to quote both, the quote and the backslash.

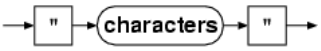


Figure 9: The text *element.*

```
""
"some text"
"\"quoted\" quote"
"\\\\W2K\\C:\\WINNT"
```

The first example shows an empty string. The next example shows a string with some content. The 3rd example shows how to escape quotes inside a string. The final example shows how to escape backslashes inside a string.

2.7.2 The Use Element

The *use* element is a reference to a formal argument. The typical old-style *use* element starts with the dollar sign. Its argument, the identifier of the formal argument, is set in curly braces. A simplified version uses just the bound variable identifier, as commonly used in procedural programming languages.

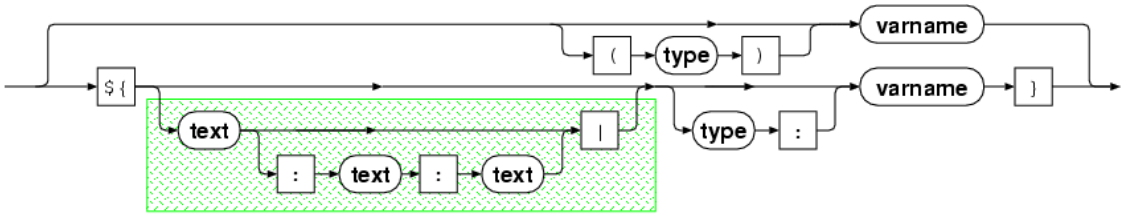


Figure 10: The use *element.*

The simplified reference to formal argument identifiers (also known as bound variables) is shown in the top path. If the bound variable is used with the same input/output type as declared in the formal argument type, just the identifier needs to be specified. If type casting is required, C-style type casts can be used for simple casts. A C-style type cast puts the target type into paranethesis in front of the identifier.

While list type variables may be used with the simple notation, any special rendering requires the old-style notation.

The optional rendering, marked green, changes the default appearance in the command-line arguments and profile components, when rendering a list value. If a rendering is present, the

vertical bar character separates it from the other parts of the *use* element. A rendering consists either of one or three *text* elements. Each *text* element is a quoted string.

If your rendering requires to specify just the separating string between list elements, the 1-string case applies. In this case, the string is the only rendering element.

If you also require to specify the introduction (prefix) string of a list, or the the end (suffix) of a list, you need to use the 3-string case. In the 3-string case, colons separate all strings. The contents of the first string is placed before the first list element, the separator string in the second string between all list elements, and the contents of the third string as suffix after the list.

The *type* is described in the formal argument list. If a *type* is used, it must be suffixed by a colon character. The type casts to either **input** or **output** is usually necessary, if the bound variable's type is **inout**.

The final and mandatory part is the name of the formal argument identifier (*varname*) that is referenced.

```
simplest
${simple}
(out) more
${out:more}

${"-"|list1}
${"-"|out:list2}
${" [ ":", ":" ] "|list3}
```

The first case shows the simplest reference to a bound variable by just using its name. The next line shows an equivalent form, using old-style dollar-brace notation. The 3rd line shows a simple type-cast variable reference, with the 4th line showing the equivalent case.

The first list case shows a rendering which will render each list item separated by just a hyphen, no spaces. The second list also casts the type of the list. The final list item will render the list by putting it between brackets, and separating each element with a comma. If the *list3*contained three elements a, b, and c, the output would look like this:

```
 [ a, b, c ]
```

2.8 Derivations

A *derivation* starts with the reserved keyword **DV** followed by a fully qualified definition identifier (see above). The identifier uniquely describes the *derivation*.

The arrow operator, which consists of a hyphen immediately followed by a less-than character, separates the derivation identifier from the transformation mapping (*tr-map*). The transformation mapping allows a flexible binding of the given *derivation* to a set of matching *transformations* by their versioning. The in-depth support in the abstract planner is lacking, though.

Mandatory parenthesis enclose the actual argument list (*aarg-list*). The actual argument list is optional. It binds formal arguments to actual values. A *derivation* specification is terminated with a semicolon.

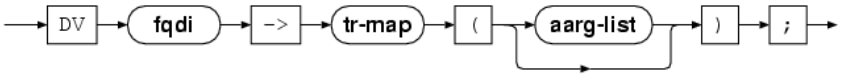


Figure 11: The Derivation.

2.9 The Transformation mapping *tr-map*

The *tr-map* non-terminal maps the current *derivation* or *call* onto a set of matching *transformations*. It is very similar to a fully qualified definition identifier, but differs with the version specification. As with the fully qualified definition identifier, white-spaces anywhere within the *tr-map* non-terminal are forbidden.

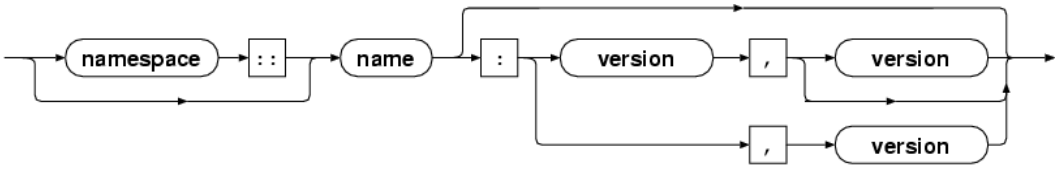


Figure 12: Mapping to a Transformation.

The namespace and name rules are the same as for *fqdi*. The version left of the comma is a minimum inclusive version number. The syntax of this version is the same as the fqdi version. The version to the right of the comma is a maximum inclusive version number. The syntax of this version is the same as the *fqdi* version.

The syntax diagram looks more complicated than the matter actually is. There are essentially four cases to distinguish:

1. If a colon does not follow the name, there is no version specification, and no version matching will be done. In this case, all versions match.
2. If the name is followed by a colon, one of the following mappings must be applied:
    a. The regular case specifies two version number, a minimum inclusive version followed by a comma, followed by a maximum inclusive version. An example is "name:10,20".
    b. The case with an open maximum version only specifies the minimum version followed by a comma, e.g. "name:10,".
3. The case with an open minimum version only specified a comma followed by the maximum permissible version, e.g. "name:,20".

```
name          #  1: unrestricted versioning
name:10,20    # 2a: version range requirement
name:10,      # 2b: minimum version requirement
name:,20      #  3: maximum version requirement
```

2.10 The Actual Argument List

The list of actual argument in a *derivation* binds values to formal arguments of a *transformation.* The syntax is similar to the argument list for a *call*. The difference is in the permissible elements in the list. Being inside a *derivation*, only quoted strings (*text*) and logical filenames *LFN* may be passed.

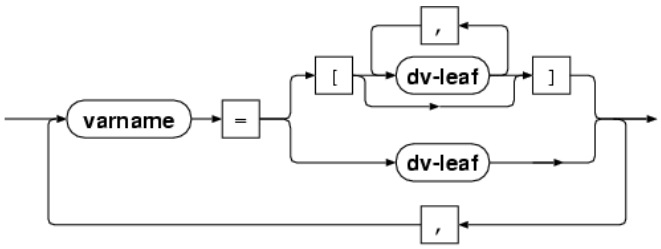


Figure 13: Actual Arguments in Derivations.

Each actual argument names the formal argument in the called transformation to bind to (*varname*). Either a list or a scalar follows the mandatory equals character. If the bound formal argument is a scalar, only the lower path may be taken. If the formal argument in the called transformation is a list, the elements are enumerated between brackets,

2.10.1 Permissable Elements Inside a DV

The *dv-leaf* specifies the permissible arguments inside a *derivation*. Inside any *derivation*, only verbatim textual elements (*text*) and *LFNs* are permitted. Please note in the railroad diagrams which exhibit *dv-leaf* use, if it is a single *dv-leaf* element, a concatenation of *dv-leaf* elements, or a comma-separated list of *dv-leaf* elements.

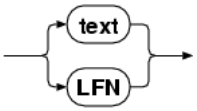


Figure 14: Elements permitted in dv-leaf.

2.10.2 The Text Element

The *text* element is a C-style string in quotes. A backslash can be used to quote both, the quote and the backslash. Please refer to section 2.7.1 for details.

2.10.3 The LFN Element

The *LFN* element describes a logical filename. Any LFN is introduced by the "at" character, with further arguments in curly braces.

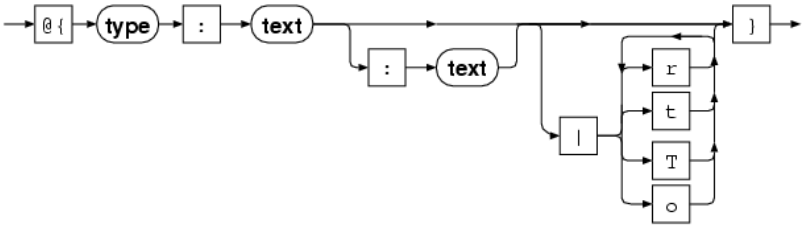


Figure 15: The LFN element.

The mandatory *type* describes a logical file as either **input**, **output**, or **inout**. The latter is usually reserved for transient files. A colon separates the logical filename itself from the type. Please note the use of the *text* element for the filename - it must be enclosed in quotes.

In case of transient filenames, the pattern to construct temporary filenames can be specified as follows: A colon separates the pattern string, a *text* element again, from the filename. The pattern is optional. In the presence of the pattern, the filename is deemed transient. Transient files are neither transferred nor cataloged. It will not participate in the staging process, except for inter-

pool transfers as necessary. In the absence of the pattern, the file is assumed to be a fully tracked, transferable and registrable file. However, this behavior can be modified as shown below.

```
@{in:"lfn1"}
@{io:"lfn2":"tmp-XXXXXX"}
```

The first file specification shows a typical input file. The 2nd file specifies a transient file, which will neither be registered nor transferred to the final staging area. However, it may be transferred to different sites, if the consuming jobs require so.

Recent additions permit to specify a finer control over the handling of the file. The options are the last part of any logical filename syntax. A vertical bar separates the options from any preceding items like the filename or pattern. Zero or more one-character codes flag how a concrete planner is supposed to deal with the file.

flag	**meaning**	**present**	**absent**
r	register file	register file in RC	do not register file
t	transfer file	transfer file to output	only inter-pool transfers
T	no-fail transfer	don't fail if transfer fails	mutually exclusive with 't'
o	optional file	don't fail, if not in RC	fail if not found in RC

The following equivalences apply to short-cuts and compatibility with previous versions of VDL:

```
@{in:"lfn1"}      <=>  @{in:"lfn1"|rt}
@{in:"lfn2":"x"}  <=>  @{in:"lfn2":"x"|}
```

3 Conventions for Running VDS Workflows on the Grid

Revision $Revision: 1.4 $ of file $RCSfile: VDSUG_RunningOnGrid.xml,v $.

This document describes the conventions for workflow execution and job management that one needs to know in order to express your workflow needs in VDL and run them across multiple Grid sites, using either the Pegasus or Euryale planners. We focus primarily on issues that are common to both planners. The details of actually creating and running the workflow with a specific planner are covered in documents specific to each planner.

3.1 The General Nature of a Workflow

We first need to define how file names are referenced and processed. The VDS is concerned with three kinds of file names:

LFN: the logical filenames used in VDL programs (and hence in the DAX file as well – relevant for users that code DAX files directly). LFNs may be relative or absolute, and may or may not have multiple directory levels in them. In the VDS 1.4 release, Pegasus does not support structured LFNs. Support may be added in the future.

When a LFN is looked up in a replica catalog, it is looked up in its relative or absolute form, exactly as it is coded in the DAX file. Similarly, when an instance of an LFN is produced at a Grid site, it is stored in the RC exactly as it was coded in the VDL and/or DAX.

SFN: the physical "storage" filenames, as files are referenced by user jobs running at a grid site

TFN: the physical "transfer" filenames (i.e., complete URLs), as files are accessed by the GridFTP protocol, both inside and outside_ the site.

LFNs and SFNs are issues that the user needs to be concerned about. For the most part, the formation and use of the TFN happens automatically in the code generated by the VDS planners. The user does not need to be aware of this coding of file names, other than to make sure that the GridFTP server is properly configured and identified to the Grid catalog mechanism.

The important concept to understand here is how these names are used as a workflow is translated from VDL and executed on the Grid.

File names are translated to SFNs by the planner (or by its generated code), and used by the programs that are executed in the workflow.

Input files are looked up in the Replica Catalog and copied to the site where the job will run, if they don't already exist at that site. In the VDS 1.4 release, RLS is the only implementation of a replica catalog. However, more implementations are forthcoming.

Programs start in their working directory (i.e., their "current working directory" is set to the working directory before the job is started).

When jobs execute, they reference their input files using names that have been translated into physical "storage file names" or SFNs. They create new output files at or below this current working directory.

When jobs complete, any files they create that were designated in the VDL as "output" files are cataloged as existing at the site. In some cases, these files may be moved within the site for persistence.

We anticipate that some jobs will need to have a "bigger" workspace, and may create files above or below their CWD. It is possible, but rare, that jobs will exist that cannot be accommodated by this model. Sometimes it may be necessary to adjust a job's parameters in order to get it to conform to this model.

For programs that expect input files:

1. the files should be pre-cataloged in the Replica Catalog by the logical name (LFN) exactly as coded in the VDL
2. the PFN in the Replica Catalog must be a TFN that can be used verbatim to retrieve the file. For LIGO exceptions please refer to section 3.6.
3. the file is copied by planner-inserted code to the SDIR, under the name relative to the LFN (regardless of whether the LFN is relative or absolute)
4. the SFN produced by the planner and placed in the command line is the file's relative name within the SDIR

3.2 The Layout of a Grid Site

3.2.1 How the Grids Define Sites

A Grid site, typically described by GridCAT or an equivalent description mechanism, consists of the following storage areas:

$DATA: where users can create and leave persistent files, typically under VO-specific directory hierarchies.

$APP: where users can create and leave application files and directories, typically under VO-specific directory hierarchies.

$TMP: place to create temporary directories for the duration of a job or workflow. Typically shared by all users of the Grid site, and reachable and sharable by all worker nodes.

$WN_TMP: place to create temporary directories on the local disk of the worker node. The local disk provides speed-up to IO intensive file operations, e.g. database searches.

$GRID: location of the OSG (or VDT) software stack, under which we can expect the base directory for the Globus and VDS worker tools installation.

3.2.2 How the VDS site catalog describes sites.

The VDS Grid Execution Model (VDS GEM) defines the following concepts:

Each site has a grid storage directory ("sdir")

- contains files that persist at the site
- all files in the *sdir* are tracked in the Replica Catalog. The files that are transferred as part of the workflow description to the sdir are automatically registered in the Replica Catalog. The user can however explicitly ask them not to be registered using the transience attribute (dR) in the VDL/DAX.
- must be accessible via gridftp to VDS submit hosts that run jobs at the site

Jobs are run at each site in (or below) a "working directory" (workdir)

- the *workdir* is created when the site is set up for the user
- a "job directory" may be created within the *workdir* for the duration of one or more jobs and destroyed when all jobs are done.
- A job starts with its CWD set to the *jobdir* (or the *workdir*). Hence relative pathnames referenced by a job are relative to this CWD.

As of this writing, a submit host must be able to access any site's *sdir*, *workdir*, and *jobdir* via gridftp.

These directories are specified in the site catalog as follows (using XML as the example here):

```
<pool handle="chalant" gridlaunch="/home/wilde/vds/bin/kickstart"
      sysinfo="cpuarch">
    <lrc url="rls://terminable.uchicago.edu"/>
    <gridftp url="gsiftp://chalant.uchicago.edu"
             storage="/home/wilde/vds" …/>
    <workdirectory >/home/wilde</workdirectory>
    <jobmanager ..… />
</pool>
```

In XPath syntax we have the following fields of interest:

sdir = pool/gridftp@storage
gftp = pool/gridftp@url
workdir = pool/workdirectory

We also need to define how file names are referenced and processed. The VDS is concerned with three kinds of file names.

LFN: the logical filenames used in VDL programs. These may be relative or absolute, and may or may not have multiple directory levels in them.

SFN: the physical "storage" filenames, as files are referenced by user jobs running at a grid site. It is the internal view of a physical filename (PFN).

TFN: the physical "transfer" filenames (i.e., complete URLs), as files are accessed by the GridFTP protocol, both inside and outside the site. It is the external view of a physical filename.

While the planners look identical in the way they use these two names, the subtle differences are explained in subsequent sections.

	LFN	**SFN**	**TFN**
Euryale	a/b/c	<workdir>/a/b/c	<gftp><workdir>/a/b/c
Pegasus	a/b/c	<workdir>/a/b/c	<gftp><workdir>/a/b/c

Euryale assumes that *sdir* is equivalent to *wdir*: *sdir* is the internal view and *wdir* the external view of the **same** underlying directory.[2] Euryale ignores any setting of *sdir* in the site catalog. The **vds-get-sites** tool sets *sdir* to $data by default.

Pegasus assumes that *workdir* is accessible, and that *sdir* is a separate storage location it can use to stage data to/from. Pegasus does not distinguish between internal and external paths to the same data. The external path to *workdir* is constructed as gftp + workdir.

Ideally, we want to have both ways, *sdir* and *wdir*, with internal and external access paths to both. This is shown in section 3.5.

3.3 Euryale File Management Conventions

We describe what happens at run time in a Euryale DAG:

- for setup,
- for stage-in,
- for compute job filenames,
- for stage-out,
- for replicas,
- for inter-pool transfers and
- for cleanup.

We also specify related aspects of the compute job (filename arguments - how are they created) and replica registration (what TFN is registered). Given the following site catalog excerpt:

- **site 1**
  - *wdir1*
  - *sdir1*
  - gridftp=gsiftp://HOST1
- **site 2**
  - *wdir2*
  - *sdir2*
  - gridftp=gsiftp://HOST2

3.3.1 Setup Job

Euryale uses a setup job to create the remote *jdir1* from *wdir1*. The job directory is created by appending a random string to the *wdir* directory. Thus, the *jdir* is a function of *wdir*:

- *jdir1* := *wdir1* + random

Additionally, all relative LFNs associated with the job, both input and output, are searched for directory components. These directories extracted from relative LFNs are created underneath the *jdir1* before any other jobs by the setup job.

3.3.2 Stage In

Euryale only uses 3rd party transfers (3pt) and job directories. Before each compute job, all input files are pulled in as necessary.

It is helpful to distinguish between source TFN (sTFN) and destination TFN (dTFN). If the LFN is relative, e.g. "a/b/c":

[2] Euryale should have used *sdir* in the TFN construction.

- sTFN := pfn_from_replica_catalog(LFN)
- dTFN := gsiftp://HOST1 + *jdir1* + LFN

If the LFN is absolute, e.g. "/d/e/f":

- sTFN := pfn_from_replica_catalog(LFN)
- dTFN := gsiftp://HOST1 + LFN

3.3.3 Compute Job

For the compute job, the remote CWD is set to *jdir*. Absolute LFNs are always used as is - their SFN is identical to the LFN.

- SFN := LFN

The command-line arguments that derive from filenames are resolved relative to the CWD. Thus, relative LFNs, whose SFN contains the *jdir*, yield relative filenames on the command-line.

- SFN := *jdir1* + LFN

The property *wf.use.relative*, if set to *false*, creates absolute filenames for relative LFNs. In this mode of operation, the SFNs are used as is. However, due to length limits in the Condor submit file, this mode is not recommended.

3.3.4 Stage Out

Euryal uses 3rd party transfer (3pt) mode only. It moves files into a replicated directory structure underneath *wdir* that previously existed in *jdir*. The clean-up script, run at a later point in time, executes the necessary move operations. For relative LFNs we distinguish between sTFNs and dTFNs again:

- sTFN := gsiftp://HOST1 + *jdir1* + LFN
- dTFN := gsiftp://HOST1 + *wdir1* + LFN
- SFN := *jdir1* + LFN

For absolute LFNs, all directory components are part of the LFN. Absolute LFNs will not be moved, since they might overwrite themselves or system files:

- sTFN := dTFN := gsiftp://HOST1 + LFN
- SFN := LFN

The sTFN designates the source of the file before the move, the dTFN where the file can be found after the move. The dTFN location will be registered into the replica catalog.

The property *wf.final.output* designates a final resting place for data to which it is replicated. Without this final resting place, no stage-out takes place. The value for the property is a gsiftp URI which may include path components where files are to be replicated to.

If a final resting place URI is specified, e.g. gsiftp://HOST3/PATH, files and directories are replicated there. If the tool *uberftp* is found in the PATH on the submit host, full deep directory structures are recreated during replication. If the tool does not exist, destination files are flattened by stripping preceding directory components from the LFN (like the Unix tool *basename*):

- sTFN := gsiftp://HOST1 + *jdir1* + LFN
- dTFN := gsiftp://HOST3/PATH + LFN

This is the only stage-out that is performed by Euryale.

3.3.5 Replica Registration

Euryale uses the dTFN from the previous step to register files. A product file can have two registrations, if the final restring place exists, one of the site, and one for the final resting place. With appropriate tools like *T2*, the duplicate registration and availability of a file helps, if a site goes down which contains a necessary input file for a later step.

- gsiftp://HOST1 + *wdir1* + LFN
- gsiftp://HOST3/PATH + (*uberftp* ? LFN : basename(LFN))

Euryale does not implement Pegasus's "local RC cache logic" yet. Thus, all product files are *always* registered, so that later steps can find them

3.3.6 Inter-pool Transfer

The conditions thar usually require inter-pool transfers are explained in section 3.4.6. These do not apply to Euryale.

Inter-pool transfers are a Pegasus notion to transfer files that are shared between partitions. Euryale uses a pull-model and its partitions encompass exactly one job. Each file, even untracked ones, are put into the replica catalog. Each job pulls in all necessary files during the stage-in phase as necessary. Thus, Euryale does not need inter-pool transfers.

3.3.7 Clean-up Job

Euryale's cleanup job moves all relative output LFNs from the *jdir* to the *wdir*, replicating any directory structures as necessary.

- sSFN := *jdir1* + LFN
- dSFN := *wdir1* + LFN

3.4 Pegasus File Management Conventions

We describe what happens at run time in a Pegasus DAG:

- for setup,
- for stage-in,
- for compute job filenames,
- for stage-out,
- for replicas,
- for inter-pool transfers and
- for cleanup.

We also specify related aspects of the compute job (filename arguments - how are they created) and replica registration (what TFN is registered). Given the following site catalog excerpt:

- **site 1**
  - *wdir1*
  - *sdir1*
  - gridftp=gsiftp://HOST1
- **site 2**
  - *wdir2*

- *sdir2*
- gridftp=gsiftp://HOST2

In addition, Pegasus allows the users to specify a relative path or an absolute path in the properties file that applies to all the sites. Given the following properties excerpt

- vds.dir.exec *pwdir*
- vds.dir.storage *psdir*

Depending upon whether *pwdir/psdir* is relative or absolute the *wdir/sdir* are either appended or replaced.

	pwdir	***psdir***
***XXdir* is relative**	*wdir1 += pwdir*	*sdir1 += psdir*
***XXdir* is absolute**	*wdir1 == pwdir*	*sdir1 == psdir*

All subsequent filename resolutions below assume that the above transformations have taken place.

Pegasus does not distinguish between relative and absolute LFNs. Furthermore, directory structures inside any LFN are discouraged and are not supported. Pegasus expects the LFN to be just a base-name (no directory component in the LFN). For clarity purposes, such a LFN is referred to as the **BFN.**

3.4.1 Setup Job

Pegasus uses a setup job to create the remote *jdir1* from *wdir1* for a partition of a workflow. A partition can be as large as the full workflow, or as small as a single job. By default the full workflow constitutes one partition.

The job directory is created by appending a random string to the *wdir* directory. Thus, the *jdir* is a function of *wdir*:

- *jdir1* := *wdir1* + random

One setup job is created for each execution site, where the portions of a partition have been scheduled. All jobs of the same partition and scheduled for the same site share the same *jdir1.*

In the case, where Pegasus is not invoked with the --randomdir option, there is no setup job created. The *jdir1* is identical to *wdir1*.

- *jdir1* := *wdir1*

In this case, all jobs share the same directory at each site. Additionally, jobs from other concurrent workflows may share the same directory.

3.4.2 Stage In

Pegasus distinguishes between direct (peer to peer) transfers and 3rd-party transfers (3pt). For the replica catalog look-ups, it uses the LFN to find entries. However, Pegasus expects the LFN to be unstructured (identical to the BFN).

In direct transfer mode, one or more transfer jobs run on the gatekeeper to pull files:

- sTFN := pfn_from_replica_catalog(LFN)

- dTFN := file:// + *jdir1* + BFN

In 3rd party transfer mode (set by specifying the property *vds.transfer.thirdparty.sites*), the transfers are initiated from the submit host between the remote source servers and destination server:

- sTFN := pfn_from_replica_catalog(LFN)
- dTFN := gsiftp://HOST1 + *jdir1*+ BFN

3.4.3 Compute Job

All the compute jobs are run in the *jdir* directory. The *jdir* directory is determined as explained in 3.4.1.

All references to filenames on the command-line are relative to the *jdir1*. Since filenames are flattened, the following translation into SFNs applies.

- SFN := *jdir1* + BFN

The filenames actually used are stripped of their *jdir1* prefix to generate relative paths to files. The command-line arguments that derive from filenames are resolved relative to the CWD, and contain only the BFN.

3.4.4 Stage Out

The stage-out distinguished between direct (peer to peer) transfers and 3rd party transfers (set by specifying the property *vds.transfer.thirdparty.sites*). For direct transfers, files movement is initiated on the remote site between

- sTFN := file:// + *jdir1* + BFN
- dTFN := gsiftp://HOST2 + *sdir2* + BFN

In 3rd party transfer mode, file transfer is directed from the submit host between:

- sTFN := gsiftp://HOST1 + *jdir1* + BFN
- dTFN := gsiftp://HOST2 + *sdir2* + BFN

In the above, the site2 stands in as the output pool where appropriately tagged result files are to be transferred to.

3.4.5 Replica Registration

Files marked for registration are registered with the replica catalog, usually an LRC. The LRC where to register is picked up from the site catalog and corresponds to the output site that the user specifies at runtime. The file to register for a given LFN uses the dTFN of the stage-out.

- gsiftp://HOSTxxxx + *sdirX* + BFN

The replica that is registered in the replica catalog, is the one residing on the output site i.e the replica that was staged to the output site by the stageout job. The replica's that are on the execution sites are not catalogued currently, as they are usually on scratch space that can be purged according to the site's policy.

3.4.6 Inter-pool Transfer

Inter-pool transfer jobs are a variant of stage-in jobs, and are created under the following conditions:

- A job X is scheduled at site1.
- Its immediate parent P(X) is scheduled at site2 unequal site1.
- Job X requires an input file that is generated by job P(X).

Again, in direct (peer to peer) transfers, the transfer job or jobs are executed on the destination site with the following filename translations:

- sTFN := gsiftp://HOST1 + *jdir1* + BFN
- dTFN := file:// + *jdir2* + BFN

In 3rd party transfer mode, the following transfers are directed from the submit host:

- sTFN := gsiftp://HOST1 + *jdir1* + BFN
- dTFN := gsiftp://HOST2 + *jdir2* + BFN

The *jdir* directory is determined as explained in section 3.4.1.

3.4.7 Clean-up Job

In random directory mode, when Pegasus setup jobs created partition-specific remote job directories, a clean-up DAG is generated. The submit files for the clean-up DAG are generated in a cleanup directory in the partition's or workflows' submit directory. The clean-up DAG consists of clean-up jobs for each execution site, where the portions of the partition were scheduled and run.

The clean-up DAG is not submitted automatically, and has to be submitted manually by the user.

3.5 Future Extensions to the Site Catalog

We need the separate storage dir for staging and storing files, and a work directory to run jobs. We need to know, how to access these directories both, from the inside and from the outside. Thus, we need, in addition to what the site catalog provides today,

- a storage dir element for the inside view.
- a workdir gridftp path for outside view of the workdir.

Some sites may not permit outside access to the workdir, or inside access to the storage dir, e.g. LCG2. Such a site requires 2nd-level staging. We will deal with these another time.

```
<directory type="working" url="gsiftp://host1" internal="dir1"
 external="dir2">
<directory type="storage" url="gsiftp://host2" internal="dir3"
 external="dir4">
```

In Euryale, use *storage* for storing (moving, linking) files away instead of using *working* as we now do. It may require 2nd level staging to get files from *working* to *storage.*

Pegasus constructs the external view of the working directory by adding the site catalog's *wdir1* to the gridftp base URI. Pegasus constructs transfer filenames (TFN) according to the XPath syntax from section 1.2.3 as per the following rule

- TFN := pool/gridftp@url + pool/workdirectory+ BFN

The TFNs constructed according to the above rule are listed by transfer job type below:

- Stage in jobs : dTFN
- Interpool jobs : both sTFN and dTFN

- Stageout jobs : sTFN

Bringing in a notion of explicitly defining a gridftp server with the storage-dir and work-dir will solve this problem: We do not have to use the gsiftp URI from the storagedir to get the outside view to the workdir.

The external view of the *wdir1* can now be correctly constructed from the correct knowledge. In XPath syntax from the above excerpt, the resulting filename becomes:

- TFN := directory[@type="working"]@url + directory[@type="working"]@external + BFN

Since we know how to access each file system inside as well as from the outside, it will be a much better and more flexible design.

3.6 The LIGO PFN Issue and how VDS Solves It

LIGO defines three pfns in the Replica Catalog for the same logical file.

1) Inside view only (may be local or shared file system)

   e.g. file://localhost/foo/bar

2) Outside view via one machine only (local file access only)

   e.g. gsiftp://host1/local/foo/bar

3) Outside view via the gridftp server (shared file system)

   e.g. gsiftp://host2/shared/foo/bar

In the case where the PFN is tagged in the replica catalog with an attribute "pool" matching the job's execution site handle, symlinks are created instead of physically transferring the files. In LIGO's case "pool" attributes are added only to those PFN entries in the replica catalog which are accessible via a shared file system on all the worker nodes.

Pegasus transfer jobs access the working directory externally, constructing the TFN from the gridftp server entry's storage directory, as shown in section 3.4. However, not all gridftp servers mentioned in the site catalog mount all sharable file systems. Some may just mount the storage directory, but not the working directory. Others may mount the working directory in their stead. Thus, the construction algorithm may incorrectly refer to a non-existing directory, causing certain transfer cases to fail.

The solution proposed in section 3.5 will allow users to specify distinct gridftp servers and access paths for each directory. Therefore, the assumption that all gridftp servers mount both directories, storage and working, can be lifted.

4 Database Server Administration Guide

Revision $Revision: 1.3 $ of file $RCSfile: VDSUG_DatabaseAdmin.xml,v $.

This section deals with the advanced issues of managing a relation database system like PostGreSQL or MySQL.

For the access and proper configuration to a relation database management system (rDBMS), a number of configuration variables need to be gathered from your database administrator. Many sections will refer to these variables:

- **dbhost** The database host. If you use the database on your local machine, ignore this option. If you use a remote machine, check with the database- and network administrator that you can access the machine. Both PostGreSQL and MySQL require special configurations to permit remote hosts to access database accounts.
- **dbport** The database server's listening port. If your database is set to use the standard port for the engine, don't worry about this option. The standard port for PostGreSQL is 5432 and for MySQL 3306.
- **dbname** The identifier for the space that the administrator assigned to your account. Each database engine may maintain multiple databases. You will have to know which database you are actually accessing. Often, the name of the database is identical with the name of your local user account.
- **dbuser** The database user account name. Often, the name of the login account is used.
- **dbpass** The database user account password. This is not, and should never be, your login account password.

4.1 Structural Overview

The database setup of the VDC and related catalogs is highly flexible and customizable. The VDC itself can work with a single XML file that stores all virtual data definitions, or with a powerful relational database management system (rDBMS) to store millions of entries. This guide talks about the rDBMS approach.

Part of the flexibility derives from a layered approach which separates the quirks specific to each rDBMS from the catalog interactions. Figure 16 shows the tiering of the different access layers.

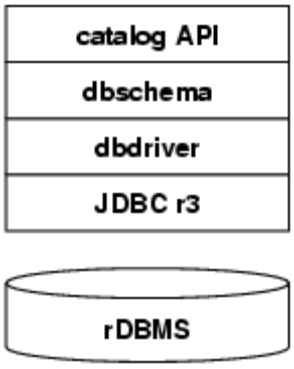


Figure 16: Layers of the VDS access to databases.

At the highest level is the catalog access API, which is used by other portions of VDS to communicate with a given catalog. The schema uses methods from the database driver layer to reach the database. The database driver layer in turn requires JDBC release 3 to eventually talk to the database management system. It also abstracts the differences between database implementations to present a more uniform view to the schema layer.

The layering approach was chosen for two reasons:

1. The separation of the driver and schema permits each schema to optionally connect to a different database. Or all schemas can connect to the same database.
2. The database driver on top of JDBC was necessary to mediate between JDBC3 optional features, work-around implementations for specific databases, and general non-conformity of data manipulation language (DML) implementations by the various vendors.

4.2 Virtual Data Catalog (VDC)

```
vds.db.vdc.schema = ( AnnotationSchema | ChunkSchema )
```

The VDC comes in two flavors. Previous releases of the VDS chose the efficient and fast chunk schema. Research guided us to provide an alternative schema as default for this release, which is based on the chunked schema, but permits annotations to transformations, derivations, filenames, and formal arguments.

```
vds.db.vdc.schema.xml.url = ( URL | file )
```

The VDC stores the majority of the data of a VDL definition as XML character large object. When reading back the definition into memory, the XML requires parsing. The schema URL control which location is passed as the XML schema document which describes the VDLx. This option might be required in rare circumstances to read back definitions entered into the VDC with an older version of the VDS.

4.3 Provenance Tracking Catalog (PTC)

```
vds.db.ptc.schema = InvocationSchema
```

If defined, the schema describes the provenances tracking catalog. This is the catalog where the data extracted from *kickstart* records will be stored. Note that not everything in a kickstart record will be recorded due to the sheer amount of information.

4.4 Transformation Catalog (TC)

```
vds.db.tc.schema = Database
```

The transformation catalog can be stored in a database to simplify the ease of manipulation and persistence. The *tc-client* VDS tool permits access to the rDBMS transformation catalog.

4.5 Experimental Replica Catalog (RC)

```
vds.rc              JDBCRC
vds.rc.url          jdbc:…
vds.rc.driver       ( org.postgresql.Driver | com.mysql.jdbc.Driver )
vds.rc.user         dbuser
vds.rc.password     dbpass
```

The database-driven replica catalog alternative is an experimental feature that has not (yet) been fully integrated into the tiers of the drivers and schemas. For this reason, it may be simpler to make it work with other database engines.

The *url* property is database engine specific, and explained in the database-specific sections below. The *driver* is the name of the Java class which implements the JDBC driver.

4.6 Workflow Management (WF)

```
work.db                 ( Pg | mysql | SQLite2 )
work.db.host            dbhost
work.db.database        dbname
work.db.user            dbuser
work.db.password        dbpass
```

The workflow manager is an experimental new feature that has not (yet) been fully integrated into the tiers of drivers and schemas. Its configuration usually happens in the *$HOME/.wfrc* property file instead of the regular VDS property file. Currently, access happens mostly through Perl, not Java. However, in the near future, it is expected to integrate into the tiering system. The expected property set will be *vds.db.wf.**.

The workflow management catalog is maintained by the workflow monitoring daemon *tailstatd* as well as the high-level VDS scripts *vds-plan* and *vds-run*. The planning script enters a successfully planned workflow with state “planned” (-2) into the table of known workflows. The start script changes the workflow state to “running” (-1), after successfully submitting the workflow to DAGMan. The workflow monitor updates the workflow state to the exit code of DAGMan between 0 and 127.

For each workflow, the workflow monitor keeps track of each job. Initially, all jobs have the state “unready”. As they are started and run, their state change is reflected in the job state table of the workflow catalog.

Additionally, the workflow monitor attempts to store some metrics garnered from observations of the behavior of sites, and stores the job turn-around in a site catalog. This feature to aide site selection may be dropped in future releases.

4.7 The Relational Database Management Systems

The database driver level uses several properties to configure. The database driver selection is shown in greater detail to show what applies to the other database driver configuration options

```
vds.db.<catalog>.driver = ClassName
```

```
vds.db.*.driver = ClassName
```

The database driver class is dynamically loaded. The driver is required by the schema a.k.a. catalog. Currently, only the classes *Postgres* and *MySQL* are supported. The spelling of class names is case sensitive. The respective JDBC3 driver for the databases is provided as part of VDS.

A user may provide their own implementation, derived from the VDS Java class *org.griphyn.vdl.dbdriver.DatabaseDriver*, to support a database of their choice.

For each <*catalog*> in *vds, ptc, tc, rc* and *wf*, a driver is instantiated separately. The instance has the same prefix as the catalog. The separate instances may result in multiple connections to the same rDBMS backend.

To cut down on configuration options, the special <*catalog*> asterisk (*) configures the database driver for all catalogs. However, this is a fall-back. If a more specific driver configuration is found, the more specific one is applied. For instance,

```
vds.db.*.driver         Postgres
vds.db.ptc.driver       MySQL
```

will use a separate instance of the PostGreSQL driver implementation for all catalogs except the *ptc* schema. The order of property specifications does not matter.

After the driver is selected, the JDBC connection URI determines the database host, port and name. However, since the URI is highly specific to the database's JDBC implementation, it will be covered in the sections below.

```
vds.db.<catalog>.driver.url = jdbc:…
vds.db.*.driver.url = jdbc:…
```

Most database administrator require a database account name and password:

```
vds.db.<catalog>.driver.user = dbuser
vds.db.*.driver.user = dbuser

vds.db.<catalog>.driver.password = dbpass
vds.db.*.driver.password = dbpass
```

Further configurations per driver depend also on the driver implementation.

4.8 PostGreSQL

The GriPhyN Virtual Data System supports Postgres versions 7.4.* and 8.0.*. If you need to install a new database, either use the last production release from the 7.4 series, or the latest 8.0 series. Many Linux distributions ship Packages, albeit often outdated.

```
psql -d dbname [-h dbhost [-p dbport]] -U dbuser [-f script]
```

The above statement shows the various options required to connect to a Postgres engine. If coming via a TCP/IP connection, most configurations will require you to type your password.

In script mode, the name of a script can be passed as argument to the **-f** option of the **psql** client. However, for regular maintenance, we recommend interactive mode.

On the interactive prompt of the client, you can execute various script files from the *$VDS_HOME/sql* directory using the backslash-i command. For instance, to install all schemas into a new table space:

```
\i create-pg.sql
```

To exit the client, use the backslash-q command.

```
\q
```

More information is available using the backslash-questionmark command.

```
\?
```

4.8.1 Installation

For guidance to a Postgres installation, please refer to their manuals. The Postgres manuals will talk about compile-time optimization options, and an installation that requires root privileges.

However, it is possible to install Postgres completely without root privileges. To do so, chose an installation directory where you have permissions to write. When adopting a user-installation, chose a default port that is not the standard Pg port in order to not interfere with the system. You must have Java, ant, Perl, C and C++ installed, available in your *PATH*, and *JAVA_HOME* correctly set:

```
# user installation at alternate port
./configure --prefix=$HOME/pgsql \
--disable-nls --with-pam --with-java --with-perl --with-cxx \
--with-openssl --with-pgport=12345 \
--enable-thread-safety --enable-integer-datetimes
```

The above compiles a Pg8 into your *$HOME* directory at port 12345. You can follow the rest of the compilation and setup instructions laid out in the Postgres manual from here.

4.8.2 Setup and Configuration

Before you can install users, you may want to check your PostGreSQL server configuration files. These files usually require either root privileges, or privileges of the user that the database was installed as.

Two files control essential properties of the server's behavior. In the *$PGDATA* location you will find the file *postgresql.conf*. This file should include the following options:

```
tcpip_socket = true                 # permit internet logins
listen_addresses = *                # [8.*] or an interface address
port = 5432                         # as applicable
hostname_lookup = false             # speed up connects
show_source_port = true             # [7.*] tracing
shared_buffers = 256                # min 16, typically 8KB each
max_fsm_relations = 4096            # min 10, ~40 bytes each
max_fsm_pages = 131072              # min 1000, ~6 bytes each
max_locks_per_transaction = 64      # min 10
wal_buffers = 32                    # min 4, typically 8KB each
sort_mem = 32768                    # min 64, size in KB
vacuum_mem = 32768                  # min 1024, size in KB
fsync = false                       # slightly unsafe, a lot faster
log_connections = true              # check your log info
```

If your server already features higher numbers, do not decrease. Options only available in certain versions are marked thus. Detailed configuration options are documented for Pg8 and Pg7.4.

In addition, network access requires a special setup. By default, Pg8 will install with clamped down networking, and many Linux distributions install Postgres without networking. In order to enable access from Internet sources, **including localhost which is required by VDS**, you will need to adjust the *pg_hba.conf* file in your *$PGDATA* directory.

```
local all   all                                   trust
host  all   all   127.0.0.0   255.0.0.0           md5
host  all   all   <network>   <netmask>           md5
```

You should only trust your local connections that come in through the UNIX socket interface (1st line). Sometimes, it is better not to trust even them, e.g. if the database is on a common login host.

You should always include your complete loopback network 127/8. Postgres and many distributions limit loopback to exactly 127.0.0.1/32. Do not use the *trust* option, use *md5.* Older versions may show a value of *crypt.* However, if your Postgres supports *md5,* do use it - it is safer than *crypt.*

Finally, you need to enter one line for each network and netmask that you need to connect from onto this database.

Once you configured your database, if it is already running, shut it down and restart it. If it wasn't started, it is ready to be started now.

4.8.3 Account Setup

While users may share a VDC, it is often good to give users separate spaces to set up their VDC. To create a database user account, use the following steps:

1. Connect as database super-user to your Postgres instance *template1*. The specific steps largely depend on your setup. If your database resides on a different host, you need to use additional options to connect, and may be prompted for a password.
2. Create the database user with the password. Grant *createdb* privileges, but no *createuser* privileges.
3. Change your default identity on the connection from the super-user to the newly created database user.
4. As the new user, create a database.

```
psql -U <dbsuperuser> template1
create user <dbuser> password '<dbpass>' createdb;
\c - <dbuser>
create database <dbname>;
```

As the database administrator, you will need to pass the above information, along with the database hostname and listening port, to the user.

The user should create the PL/SQL stored procedure language once, and once only per *dbname.* While the stored procedures are not yet used, it is useful to have them activated. The user needs to type the following command on the shell, adding additional connection-related options as necessary:

```
createlang plpgsql <dbuser>
```

Finally, the user can install the database as described by the rest of this documentation.

4.8.4 Special VDS Configurations

With JDBC, a database is represented by a URL (Uniform Resource Locator). The *vds.db.*.driver.url* property reflects this URL. If PostGreSQL runs locally, you can omit the host portion. If your database server runs remotely, please make sure that you can access it. With PostGreSQL, the URL to access any database takes one of the following three forms:

```
jdbc:postgresql:<dbname>
jdbc:postgresql://<dbhost>/<dbname>
jdbc:postgresql://<dbhost>:<dbport>/<dbname>
```

with the meaning of the template variables as the section's introduction:

<*dbhost*>	The host name of the server. Defaults to localhost.
<*dbport*>	The port number the server is listening on. Defaults to the PostGreSQL standard port number (5432).
<*dbname*>	The database name.

The *Postgres* database driver implementation understands the following extra properties:

```
vds.db.*.driver.ssl
vds.db.*.driver.protocolVersion
vds.db.*.driver.logLevel
vds.db.*.driver.charSet
vds.db.*.driver.compatible
vds.db.*.driver.allowEncodingChanges
vds.db.*.driver.prepareThreshold
```

The options may also be applied separately to just one specific <*catalog*> by replacing the asterisk with the catalog name. Please refer to the PostGreSQL JDBC documentation for details on above configuration options.

4.9 MySQL

The GriPhyN Virtual Data System supports MySQL versions 4.0,* and 4.1.*. If you need to install a new database, either use the last production release from the 4.0 series, or the latest 4.1 series. Many Linux distributions ship Packages, albeit often out-dated versions.

```
mysql [-h dbhost [-P dbport]] -u dbuser -p dbname [ < script ]
```

The above statement shows the various options required to connect to a MySQL engine. Usually, most engines will require you to type your password. If your account is set up with a password, connection attempts will not work without the **-p** option. Please note that **-p** does *not* take a password argument.

In script mode, the script to run can be connected to the *stdin* of the **mysql** client using the shell's redirection character. However, for regular maintenance, we recommend interactive mode.

On the interactive prompt of the client, you can execute the various script files using the backslash-dot command. For instance, to install all schemas into a new table space:

```
\. create-my.sql
```

To exit the client, use the backslash-q command.

```
\q
```

4.9.1 Installation

Sorry, we don't have support information about a user-installed instance of MySQL.

4.9.2 Setup and Configuration

For Gaurang and/or Doug to solve.

4.9.3 Account Setup

For Gaurang and/or Doug to solve.

Talk about privileges, how to add a user, which permissions, *flush privileges*

4.9.4 Special VDS Configurations

With JDBC, a database is represented by a URL (Uniform Resource Locator). The *vds.db.*.driver.url* property reflects this URL. If MySQL runs locally, you do not need to set the host portion, although JDBC always accesses MySQL through its networked interface. If your database server runs remotely, please make sure that you can access it. With MySQL, the URL to access any database takes one of the following three forms:

```
jdbc:mysql:///<dbname>
jdbc:mysql://<dbhost>/<dbname>
jdbc:mysql://<dbhost>:<dbport>/<dbname>
```

with the template variables shown in the section's introduction:

<dbhost>	The host name of the server. Defaults to localhost.
<dbport>	The port number the server is listening on. Defaults to the MySQL standard port number (3306).
<dbname>	The database name.

The *MySQL* database driver implementation understands the following extra properties:

```
vds.db.*.driver.databaseName
vds.db.*.driver.serverName
vds.db.*.driver.portNumber
vds.db.*.driver.socketFactory
vds.db.*.driver.strictUpdates
vds.db.*.driver.ignoreNonTxTables
vds.db.*.driver.secondsBeforeRetryMaster
vds.db.*.driver.queriesBeforeRetryMaster
vds.db.*.driver.allowLoadLocalInfile
vds.db.*.driver.continueBatchOnError
vds.db.*.driver.pedantic
vds.db.*.driver.useStreamLengthsInPrepStmts
vds.db.*.driver.useTimezone
vds.db.*.driver.relaxAutoCommit
vds.db.*.driver.paranoid
vds.db.*.driver.autoReconnect
vds.db.*.driver.capitalizeTypeNames
vds.db.*.driver.ultraDevHack
vds.db.*.driver.strictFloatingPoint
vds.db.*.driver.useSSL
vds.db.*.driver.useCompression
vds.db.*.driver.socketTimeout
vds.db.*.driver.maxReconnects
vds.db.*.driver.initialTimeout
vds.db.*.driver.maxRows
vds.db.*.driver.useHostsInPrivileges
vds.db.*.driver.interactiveClient
vds.db.*.driver.useUnicode
vds.db.*.driver.characterEncoding
```

Please refer to the MySQL JDBC documentation for a detailed specification of the available options. VDS ships MySQL's JDBC driver version 3.0.11, which may not contain all available options.

4.10 Others

A number of further rDBMS were reviewed over the years. However, none of them are supported. Oracle 10i will require another major change of the name of schema columns due to its abundance of reserved words. SQL Server requires a 3rd party JDBC3 driver -- the version shipped from Microsoft is insufficient.

Site selectors and other Perl programs often find SQLite2 useful. The advantage is that the full SQLite installation becomes part of the Perl database driver module, and once compiled, does not depend on other system information like the other database modules. Unfortunately, a JDBC3 driver for SQLite does not exist.

4.11 The "sql" Directory

This section deals with manual creation of schemas for the various catalogs. Assuming that a database account exists, it is necessary to change the current working directory to *$VDS_HOME/sql*. This is necessary because some files often include other files, and will not work correctly, if your working directory is wrong.

db	database
pg	PostGreSQL 7.4.* and 8.0.*
my	MySQL 4.0.* and 4.1.*
sl	SQLite 2.* (not supported)
or	Oracle 8 or above (not supported)

Table 1: Database support.

Table 1 shows the databases that are supported, or for which there is support to be envisioned. However, in the absence of either a license for the rDBMS, or a decent JDBC3 driver, only PostGreSQL and MySQL can be supported.

The files residing in the *$VDS_HOME/sql* directory follow for the most part a simple template

<work>-<rdbms>-<schema>.sql

with the following value sets:

- *<work>* is one of "create" to instantiate a schema, "delete" to drop the schema and all data, or "update" to migrate an existing database from a previous schema to the current schema.
- *<rdbms>* uses "pg" for PostGreSQL, or "my" for MySQL.
- *<schema>* references the table set or schema for a catalog.

schema	cat.	used for
init	-	metadata upon which the other schemas depend.

chunk	VDC	Virtual Data Catalog (deprecated)
anno	VDC	Virtual Data Catalog (recommended)
ptc	PTC	Provenance Tracking Catalog
tc	TC	Transformation Catalog
rc	RC	Simple Replica Catalog (prototype)
wf	WF	Workflow Manager

Table 2: Schemas and Catalogs

The *init* schema is crucial. It must exist or be created before any of the other schemas can be added. When deleting, it is the last schema to go. The *vds-config* tool uses information stored in the metadata to determine the schema version information.

Table 2 shows two *vdc* schemas. These are mutually exclusive in their usage. With recent changes, they can both peacefully co-exist. However, since both constitute a valid VDC, you can only work with one of them at one time. Unless you are a developer, we do not recommend that you create both VDC table-sets.

The file name template shown above each creates a single schema for exactly one catalog. The recommended order of catalog creation is kept in the following templates:

<work>-<rdbms>.sql

which creates, deletes, or updates all our recommended set of schemas and catalogs. We recommend the *Annotation* schema for the VDC. Thus, the above command will deal with the schemas *init, anno, ptc, tc, rc* and *wf.*

4.11.1 Creating a Schema

If you are a new user, and you want to create a new schema in a fresh database tablespace that contains no previous VDS schemas, you should use the *create-<rdbms>.sql* script, where *<rdbms>* is either *pg*, if you use a PostGreSQL database, or *my* for a MySQL database.

The following steps show the typical setup of a full VDC:

1. Change into directory *$VDS HOME/sql*.
2. Start the appropriate database client in interactive mode.
3. Execute the *create-<rdbms>.sql* script.

After adding the schema, you will have to update your property settings to match your new databases.

4.11.2 Deleting a Schema

In order to delete all catalogs from a database, follow the steps of creating a schema from section 4.11.1. The section describes which *create* script to use. However, for deletion, use the *delete* script instead.

You can selectively delete a schema by using the catalog-specific delete file from Table 2. However, the **init** schema must only be removed after all other schemas were removed.

4.11.3 Updating All Schemas from 1.2.* and 1.3.5b to 1.4.0

The update from versions 1.2.* … 1.3.5b to 1.4.0 is possible. However, it is untested, and will thus not be described here. If you need to upgrade an existing catalog, please request support.

4.12 Matching Property Setup

Each process in the VDS is intimately tied with properties that contain many of the information you used to create a database and schemas. The use of the *$HOME/.chimerarc* has been deprecated, and the file should be named *$HOME/.vdsrc*. These files deal with properties that have a *vds* prefix.

The database engine parameters from *dbhost* through *dbpass* need to be entered in a special format into these property files. For Postgres, the driver options are entered as follows:

```
vds.db.*.driver Postgres
vds.db.*.driver.url jdbc:postgresql://dbhost:dbport/dbname
vds.db.*.driver.url jdbc:postgresql://dbhost/dbname
vds.db.*.driver.url jdbc:postgresql:dbname
vds.db.*.driver.user dbuser
vds.db.*.driver.password dbpass
```

Below are the properties for a MySQL setup:

```
vds.db.*.driver MySQL
vds.db.*.driver.url jdbc:mysql://dbhost:dbport/dbname
vds.db.*.driver.url jdbc:mysql://dbhost/dbname
vds.db.*.driver.url jdbc:mysql:///dbname
vds.db.*.driver.user dbuser
vds.db.*.driver.password dbpass
```

In both cases, please use only one of the URL properties. If your database engine runs on a non-standard port, you must use the first. If your engine runs on a different host, use the second. If your database engine runs locally on the standard port, use the third.

For the installed schemas, you need to record all schemas into your properties. If you are using a freshly created database according to our recommendations, you will use the *AnnotationSchema* as VDC. If you upgraded an old database, you use either the *ChunkSchema* or *AnnotationSchema*. Your VDC catalog schemas are independent of the database driver:

```
vds.db.vdc.schema  AnnotationSchema
vds.db.vdc.schema  ChunkSchema
vds.db.ptc.schema  InvocationSchema
#vds.tc            Database
```

Please use only of VDC schema, as befits your installation.

An additional property set is kept in the file *$HOME/.wfrc*. It deals with properties of the *wf* and *work* prefixed properties. You will have to repeat the database setup for now:

```
work.db Pg
work.db mysql
work.db.host dbhost
work.db.port dbport
work.db.database dbname
work.db.user dbuser
work.db.password dbpass
```

You have to chose one database driver, either *Pg* or *mysql*. Please note that the spelling is case sensitive. If you connect locally, you can omit the host and port.

The work properties rely on Perl. Perl abstracts the operations on a relational database into its DBI module, and provides drivers for different database in the DBD module family. To check your Perl for its DBD driver and that your DBI is recent enough, run the following command (shown with output):

```
$ perl -MDBI -e 'DBI->installed_versions'
  Perl            : 5.008006    (i386-linux-thread-multi)
  OS              : linux       (2.4.30)
  DBI             : 1.48
  DBD::mysql      : 2.9004
  DBD::Sponge     : 11.10
  DBD::SQLite2    : 0.33
  DBD::Proxy      : 0.2004
  DBD::Pg         : 1.41
  DBD::Multiplex  : 1.95
  DBD::File       : 0.33
  DBD::ExampleP   : 11.12
  DBD::DBM        : 0.02
```

If the command fails, your host administrator will have to install the DBI module:

```
$ perl -MCPAN -e shell
…
cpan> install Bundle::DBI
```

If you cannot find the appropriate Perl module for your Database in the DBI output, "Pg" for Postgres, or "mysql" for MySQL, your host administrator will have to install the necessary missing modules. We recommend to download the necessary modules from http://www.cpan.org/ and to compile them manually.

5 Site Catalog Administration

Revision $Revision: 1.3 $ of file $RCSfile: VDSUG_SiteCatalog.xml,v $.

Concrete planners of the Virtual Data System query various catalogs in order to convert an abstract workflow into an executable workflow. This document describes the site catalog. The site catalog describes the configuration and layout of remote sites.

A site is a collection of compute machines sharing a network and disk space, also known as a cluster. A site is comprised of

- one or more Globus gatekeepers (head nodes),
- one or more cluster schedulers (e.g. Condor, PBS, LSF, etc.) per gatekeeper,
- one or more GridFTP server (storage element), and
- one or more compute hosts (worker nodes), sharing file systems[3] with the other elements.

The site catalog maintains the essential configuration about various sites. It keeps system information like architecture and OS vendor, information about various Globus job-managers, GridFTP servers and local replica catalogs (LRC) for each a particular site. The site catalog also stores site-specific information such as environment variables and some other configuration options. It maintains information about the shared temporary (*work-dir*) and permanent storage (*storage-dir*) available to a site.

Customarily, a number of tools can generate a site catalog from information sources. Information sources include MDS4, local files, or GridCat. Eventually, the generators will be united under the roof of the *vds-get-sites* client. Other clients permit the manipulation of a site catalog, and format conversions. However, manual intervention is sometimes required.

5.1 Implementations

A site catalog can be maintained one of two formats:

- XML format (recommended), or
- multi-line textual format.

The user should choose the format that best suits her needs. However, we recommend the XML format.

[3] Future releases of VDS strive to relax the shared file system requirements.

5.1.1 System Information

Each site needs to declare its *System Information*. System information is the architecture, operating system, distribution vendor and glibc version number of the machines that the site is comprised of. VDS assumes homogeneous clusters.

The format to define the System info is

ARCH::OS[:OS-VENDOR[:GLIBC-VER]]

There are currently five supported architectures:

1. INTEL32,
2. INTEL64,
3. SPARCV9,
4. AMD64, and
5. SPARCV7.

Only two operating systems are supported:

1. LINUX, and
2. SUNOS.

For example, the system information

```
INTEL32::LINUX:RH9:232
```

translates that the site has worker nodes of x86 architecture running RH9 Linux OS with glibc 2.3.2 installed. In the absence of any system information, the default is **INTEL32::LINUX.**

The system information of a site is matched against the system information of a transformation selected to run on that site. The match determines whether a site can run a given transformation. Currently, only exact matches are made between a site's system information and the transformation's system information. It is important to ensure that the system information defined for the site and transformation are correct in order to avoid runtime or planning errors.

5.1.2 Profiles

Profiles are the VDS way of abstracting information that is required to execute a job. The primer on Profiles describes their occurrences, priorities and inter-actions.

5.2 Multi-line File format (Text)

The "Text" site catalog format is a user-friendly readable format. The multi-line format can either be used directly, or converted into the XML format.

An example for the multi-line format looks like this:

```
pool BNL_ATLAS_2 {
  profile env "app"  "/usatlas/projects/OSG"
  profile env "data" "/usatlas/prodjob/share"
  gridlaunch "/opt/OSG/vds/bin/kickstart"
  lrc "rls://evitable.uchicago.edu"
  gridftp "gsiftp://gridgk02.racf.bnl.gov/usatlas/prodjob/share/jsv" "2.4.3"
  workdir "/usatlas/prodjob/share/jsv"
  universe vanilla  "gridgk02.racf.bnl.gov/jobmanager-condor" "2.4.3"
  universe transfer "gridgk02.racf.bnl.gov/jobmanager-fork" "2.4.3"
}
```

A more abstract description shows the reserved words in bold, and required arguments:

```
pool sitehandle {
  profile namespace "key" "value"
  gridlaunch "path to $VDS_HOME/bin/kickstart"
  lrc "URL to local replica catalog"
  gridftp "GridFTP URL to storage location" "GT version"
  workdir "base path to execution directory"
  universe type "jomanager URL" "GT version"
}
```

5.2.1 The Site Handle

```
pool identifier { … }
```

The site handle is an unique identifier for a particular site. It is unique only within the VDS catalogs (RC, SC, TC). The site handle is an identifier, which consists of 1 or more alphanumeric characters including the underscore. The regular expression

```
[A-Za-z_][A-Za-z0-9_]*
```

describes legal identifiers for the site handle, similar to C or Java identifiers. The site handle *local* is reserved, see section 5.4.

5.2.2 Site Profile Entries

```
profile identifier string string
```

A site may specify as many profiles as necessary, including none. Please refer to section 5.1.2. To specify environment settings which apply to every job run on that site, e.g.:

```
profile env "VDS_HOME" "/software/vds"
profile env "LD_LIBRARY_PATH" "/software/globus/lib"
```

5.2.3 The Grid Launch Entry

```
gridlaunch string
```

The grid launch entry is optional, but recommended. The entry points to the VDS *kickstart* wrapper executable, which runs VDS jobs. The argument points to the installed location of VDS kickstart, as visible on the worker node. The launcher collects additional job statistics like worker node resource usage, the application's exit code, and remote file statistics. The use of a grid launcher is highly recommended, because it also aides grid debugging.

The VDS grid launcher is incapable of launching Condor standard universe jobs and MPI jobs.

```
gridlaunch "/shared/vds/bin/kickstart"
```

In the absence of a grid launch setting, jobs are run directly without the wrapper. However, with GT2, it is impossible to detect remote job failure without launcher.

5.2.4 LRC Entries

```
lrc url
```

The LRC entry specifies the replica catalog contact for data stored or produced at this site. The URL is generally one of *rls://hostname* or *rlsn://hostname*, e.g.

```
lrc "rls://replica.isi.edu"
```

Support for multiple LRC entries is forthcoming.

5.2.5 GridFTP Entries

```
gridftp string string
```

A site should have one or more GridFTP URLs pointing to permanent storage locations available on that site. The URL is usually of schema *gsiftp.* The contact may contain port numbers. The path component of the URL points to the storage directory.

A Globus version is required along with the URL. The version is of the form *major.minor.patchlevel*, e.g. "2.4.3" or "4.0.1". Multiple GridFTP servers are permitted, e.g.:

```
gridftp "gsiftp://my.host.edu" "2.4.3"
gridftp "gsiftp://my.host.edu:2812" "4.0.1"
```

5.2.6 The Working Directory Entry

```
workdir string
```

The working directory is a fully-qualified path to the (temporary) execution directory base. The directory must reside on a shared file system. The directory must be visible to all worker nodes, the storage element, and the head node.

The working directory can be modified using the *vds.dir.exec* property, see section 5.5.

```
workdir "/nfs/scratch/username"
```

Due to the shared nature of grids, we recommend that you create your own niche in the form of a sub-directory on the remote site.

5.2.7 Universe Entries

```
universe identifier string string
```

The universe entries describe the Globus GRAM contacts of a site. Each site usually has at least two contacts, but may have more. The first argument is the type of VDS universe. The site catalog supports two types of VDS universes:

1. The *transfer* universe names those job manager contacts that run auxiliary and transfer jobs. If the worker nodes on the site reside on a private non-routed network, the *transfer* contact must point to a batch job manager. Otherwise, it should point to the fork job manager.
2. The *vanilla* universe names those job manager contacts that run compute jobs. It does not imply a Condor vanilla universe.

The 2nd argument is the job manager contact URI. This is the full hostname of the gatekeeper plus the job manager service name. The final argument is the Globus version.

If multiple job managers of each type are specified, e.g. in the presence of several gatekeepers, load balancing is applied.

```
universe vanilla  "machine1.isi.edu/jobmanager-condor" "3.2.1"
universe vanilla  "machine2.isi.edu/jobmanager-pbs" "2.4.3"
universe transfer "machine1.isi.edu/jomanager-fork" "3.2.1"
```

5.3 The XML Format

The XML file format of the site catalog is the VDS default implementation. The XML format augments the information of the multi-line format by several attributes, like the number of CPUs

available on a remote cluster, or storage capacities. The schema for the XML file is provided in *$VDS_HOME/etc/gvds-poolcfg-1.5.xsd.*

An example for one site's entry in the XML format looks like this:

```
<pool handle="BNL_ATLAS_2"
      sysinfo="INTEL32::LINUX"
      gridlaunch="/opt/OSG/vds/bin/kickstart">
  <profile namespace="env" key="app">/usatlas/projects/OSG</profile>
  <profile namespace="env" key="data">/usatlas/prodjob/share</profile>
  <lrc url="rls://evitable.uchicago.edu"/>
  <gridftp url="gsiftp://gridgk02.racf.bnl.gov"
           storage="/usatlas/prodjob/share/jsv" major="2" minor="4" patch="3"/>
  <workdir>/usatlas/prodjob/share/jsv</workdir>
  <jobmanager universe="vanilla" url="gridgk02.racf.bnl.gov/jobmanager-condor"
              total-nodes="2712" major="2" minor="4" patch="3"/>
  <jobmanager universe="transfer" url="gridgk02.racf.bnl.gov/jobmanager-fork"
              total-nodes="2712" major="2" minor="4" patch="3"/>
</pool>
```

Please note that XML is not HTML. All attributes must be placed inside quotes. All elements must be properly closed. The order of elements, as described in the sub-sections below, is important to XML. XML is case-sensitive.

5.3.1 The Pool Element

The pool element encapsulates the configuration elements for a given site. It is itself a child element of the **config** element, which comprises the site catalog.

- **handle** uniquely identifies a site for the VDS system. The value is an identifier similar to a C or Java language identifier. The site *local* is reserved, see section 5.4.
- **sysinfo** describes the site's architecture, OS vendor and other system related information, see section 5.1.1.
- **gridlaunch** is an optional attribute to point to the grid launch application installed on the remote site. Please refer to section 5.2.3 for details on the behavior and requirements of a grid launcher.

The ordering of elements inside the site catalog is important to XML.

5.3.2 The Profile Element

Zero or more profile elements encapsulate configuration data specific the execution environment.

- **namespace** notes the profile namespace.
- **key** is a key within the namespace.

The content of the element is the value for the profile. Please refer to primer on profiles.

5.3.3 The Lrc Element

The LRC entry specifies the replica catalog contact for data stored or produced at this site. The URL attribute is generally one of *rls://hostname* or *rlsn://hostname*. Support for multiple LRC entries is forthcoming.

5.3.4 The Gridftp Element

A site should have one or more GridFTP URLs pointing to permanent storage locations available on that site.

- **url** is usually of schema *gsiftp* with host and optional port.
- **storage** is the path component pointing to the storage directory.
- **major, minor** and **patch** describe the Globus version. The patch is optional.

The gridftp element may contain sub-elements refining the configuration. Please refer to the site catalog's XML schema documentation for details.

5.3.5 The Workdir Element

The working directory is a fully-qualified path to the (temporary) execution directory base. The directory must reside on a shared file system. The directory must be visible to all worker nodes, the storage element, and the head node. The working directory can be further refined using the *vds.dir.exec* property, see section 5.5.

Due to the shared nature of grids, we recommend that you create your own niche in the form of a sub-directory on the remote site.

5.3.6 The Jobmanager Element

The universe elements describe the Globus GRAM contacts of a site. Each site usually has at least two contacts, but may have more.

- **universe** refers to the VDS universe. The *transfer* universe is used for auxiliary jobs. The *vanilla* universe is used for compute jobs.
- **url** is the job manager contact URI, comprised of the full hostname of the gatekeeper plus the job manager's service name.
- **total-nodes** is an optional but recommended entry specifying the number of virtual or real CPUs available on a site. Alternatively, one can use the number of worker nodes, as long as it is done consistently for all sites.
- **major, minor** and **patch** describe the Globus version. The patch is optional.

5.4 The Local Special Site

The site *local* is required in any site catalog. The *local* site always refers to the submit machine. While the job manager entries are customarily not used, their presence is required. A local site does not need to have a GRAM gatekeeper installed; you can use dummy entries. A GridFTP server and LRC are only required if staging final data back to the submit machine.

5.5 Properties

Four properties configure the site catalog:

```
vds.sc              XML | Text
vds.sc.file         path to site catalog file
vds.dir.exec        modifier to working directory for all sites
vds.dir.storage     modifier to storage directory for all sites
```

While all properties above are optional, the first two properties have default settings. The format defaults to XML in the absence of a configuration. The site catalog file location defaults to

$VDS_HOME/etc/sites plus a format-specific suffix. In order to avoid confusion, we recommend to always set them explicitly.

The *vds.dir.exec* property modifies the working directory on all sites. The value of the property determines the behavior:

- If the path to *vds.dir.exec* is absolute, it overwrites the working directory setting with the specified path.
- If the path in *vds.dir.exec* is relative, it is appended to the working directory of the remote site. The working directory is a part of the site's configuration in the site catalog.

The *vds.dir.storage* property works similarly to the *vds.dir.exec* property. However, it modifies the storage directory instead of the working directory.

For additional information on how to use properties and what properties are available, please refer to the property guide *$VDS_HOME/docs/properties.pdf.*

5.6 Clients

Currently, two clients exist to manipulate the site catalog. The *genpoolconfig* client converts site catalogs from one format to another. The *vds-get-sites* generates site catalogs by querying grid information sources like GridCat, MDS4 or other sources.

5.6.1 Genpoolconfig

The *genpoolconfig* client converts the various textual site catalogs into one another. The output is written to *stdout* unless redirected into a file. Some examples:

- To convert from a multi-line format to XML format:

```
genpoolconfig -f sites.txt -o sites.xml
```

- To convert from a XML format to multi-line format:

```
genpoolconfig -f sites.xml -o sites.txt
```

5.6.2 Vds-get-sites

The *vds-get-sites* client generates the site catalog and transformation catalog by querying the OSG GridCat database or other databases for different grids. Often, the transformation catalog as generated by *vds-get-sites* is not appropriate.

```
vds-get-sites --grid osg --workdir '$data/jsv' \
  --sc sites.xml --sc-style xml \
  --tc /dev/null --tc-style new
```

The above command creates a site catalog in XML format in the current working directory. The remote working directory is set to the site's *$data* entry plus sub directory *jsv.* The *$data* is a special GridCat variable which is site-specific. Please note that any directories you are referring to must exist before you can run jobs there. Most planners will not create the base working directories for you.

6 Transformation Catalog Administration

Revision $Revision: 1.5 $ of file $RCSfile: VDSUG_TransformationCatalog.xml,v $.

Concrete planners of the Virtual Data System query various catalogs in order to convert an abstract workflow into an executable workflow. This document describes the transformation catalog. The transformation catalog describes the mapping of logical transformations to executable applications.

The Virtual Data Language introduces the concept of logical transformations. A logical transformation is location independent, albeit not executable per se. The transformation catalog, given a transformation identifier and a remote site, returns the information necessary to create a job of an application that implements the transformation. The applications can be compiled executables or scripts, specified as fully qualified file system path, as GridFTP URL, or as HTTP URL. Additional metadata (profiles) can be applied, specific to the remote site and application.

6.1 Implementation Details

A transformation catalog can have one of three implementations:

- A Database driven implementation (*Database*),
- a multi-columned textual file in new format (*File*), or
- a multi-columned textual file in old format (*OldFile* – deprecated).

The user should choose the format that best suits her needs. If no implementation is configured with properties, the *File* implementation will be assumed. Because the old textual format *OldFile* is deprecated, it can only be upgraded to one of the new implementations.

Table 3: Implementation Comparisons.

Features	**Database**	**File**
Adds	YES	YES
Deletes	YES	YES
Queries	YES	YES
LFN Profiles	YES	NO

PFN Profiles	YES	YES
Transformation types	YES	YES
System Information	YES	YES
All Profile Namespaces	YES	YES

6.1.1 Logical transformation identifiers

The identifier for a logical transformation is VDL's fully qualified transformation identifier. The identifier at minimum consists of a name. Often, a namespace prefixes the identifier. Optionally, a version number suffixes the identifier.

```
some_tr
sdss::gengcd
atlas::transfig:20051003
```

Please refer to the VDL language guide for more details, examples and recommendations.

6.1.2 Transformation Types

Each transformation can be defined to be one of various types that are supported by VDS:

- **INSTALLED** denotes that the executable/ transformation is installed on the remote site and is available via a shared file system on all the nodes in the site.
- **STATIC_BINARY** denotes that the executable/transformation is compiled as a static binary for a specific platform (see section 5.1.1) and is available via a GridFTP or HTTP URL.

More types will be forthcoming in the future. If a transformation is defined without a type, the type defaults to **INSTALLED**.

6.1.3 System Information

Each transformation needs to declare its *System Information*. System information is the Architecture, Operating System, Distribution vendor or Version and Glibc version number of the machine for which the transformation is compiled.

The format to define the System info is

ARCH::OS[:OS-VENDOR[:GLIBC-VER]]

There are currently five supported architectures:

6. **INTEL32**,
7. **INTEL64**,
8. **SPARCV9**,
9. **AMD64**, and
10. **SPARCV7**.

Only two operating systems are supported as of this writing:

3. **LINUX**, and
4. **SUNOS**.

For example, the system information

```
INTEL32::LINUX:RH9:232
```

translates that the site has worker nodes of x86 architecture running RH9 Linux OS with glibc 2.3.2 installed. In the absence of any system information, the default is **INTEL32::LINUX.**

The system information of a transformation is matched against the system information of a site on which it is selected to run. The match determines whether a site can run a given transformation. Currently, only exact matches are made between a site's system information and the transformation's system information. It is important to ensure that the system information defined for the site and transformation are correct in order to avoid runtime or planning errors.

6.1.4 Profiles

Profiles are the VDS way of abstracting information that is required to execute a job. The primer on Profiles describes their occurrences, priorities and inter-actions.

6.2 The File Implementation

The *File* implementation, also known as the new textual multi-column format, supports most of the features the *Database* implementation provides. However, the *File* implementation does not provide profiles on logical transformations.

The file format consists of at least six white-space separated columns. Each row describes one transformation located at one remote site:

```
SLAC   scec::analyze:1.0     /opt/osg-0.2.1/vds/bin/keg    INSTALLED INTEL32::LINUX null
local  T2 /vdt/vds/bin/T2 INSTALLED INTEL32::LINUX env::LD_LIBRARY_PATH="/vdt/globus/lib"
```

1. The first column is occupied by the site handle. The site handle must match a site handle in the site catalog to generate valid jobs.
2. The 2nd column contains VDL's transformation identifier, see section 6.1.1. A fully qualified transformation name is written in the format ***NAMESPACE::NAME:VERSION***
3. The third column is the absolute path to the executable implementing the transformation. The physical transformation name, an application name in most cases, is the physical location of the transformation on a remote system.

   If the transformation is of type *STATIC_BINARY* the physical transformation name in the 3rd column is the GridFTP or HTTP URL relating to the transformation, e.g.

```
gsiftp://hostname.domain.com/storage/transformations/bin/date
http://hostname.domain.com/storage/transformations/bin/date
```

4. The type of the transformation is specified in the fourth column. It contains one of the key words shown in section 6.1.2. A type of NULL or "null" assumes the default type **INSTALLED**.
5. The system information of the transformation contains the system information shown in section 5.1.1. A system information of NULL or "null" assumes the default ***INTEL32::LINUX***.
6. The sixth column describes profiles associated with the job. Profiles use the format

   NAMESPACE::KEY="VALUE"

It is highly recommended to always place the value in double quotes. Quoting rules similar to C apply, using backslash-escaping. If double quote characters or backslash characters are part of the value, a backslash prefix escapes them. For instance, a profile

```
NS::K1="V11\"V12"
```

results in a value of

```
V11"V12
```

A comma separates multiple profiles within the same profile namespace, e.g.:

```
ns1::key="val1",key2="val2",key3="val3"
```

Multiple Namespaces are separated by semicoli, e.g.

```
ns1::key1=”val1”,key2=”val2”;ns2::key3=”val3”,key4=”val4”
```

6.3 The Database Implementation

The *Database* implementation provides the full set of features of a Transformation Catalog. As the name conveys, the VDS common database engine backend implements a transactional safe and persistent transformation catalog inside a RDBMS. Please refer to the Database Administration Guide or the Getting Started Guide for the set of supported database engines and setup. If you have already run the *vds-config* tool, the database tables should be ready for use.

The *tc-client* tool and the Java API methods manipulate entries in the *Database* transformation catalog implementation. Bulk operations on the database often use the *File* format, described in section 6.2, as the data interchange format for imports and exports.

If you chose the *Database* implementation of the transformation catalog, several properties need to configure your access to the RDBMS. Please refer to section 6.5 for the TC configuration properties.

6.4 The OldFile Implementation

The *OldFile* implementation is deprecated. The format is read-only for the purposes of upgrading to one of the other two implementations.

The *OldFile* implementation of a file-based transformation catalog supports only query features. It does not support addition or deletion of entries, LFN profiles, or PFN profiles other than environment settings. This catalog implementation does not support transformation types or system information. Hence, the *OldFile* implementation cannot be used for executable staging.

The file format consists of three to four white-space separated columns. Each row describes one transformation located at one remote site:

```
SLAC   scec::analyze:1.0      /opt/osg-0.2.1/vds/bin/keg
local  T2   /vdt/vds/bin/T2   LD_LIBRARY_PATH=/vdt/globus/lib
```

1. The site handle constitutes the first column. The site handle must match the handle in the site catalog.
2. The VDL logical transformation name is found in the 2nd column.
3. The third column is the absolute path to the executable implementing the transformation.
4. An optional 4th column may contain environment settings, or the reserved work "null". The environment settings become part of the job.

Any existing transformation catalog using the *OldFile* format can and should be converted into one of the other format. In order to convert it into the (new) *File* format, the *tc-client* tool requires the following arguments:

```
tc-client -Dvds.tc=OldFile -Dvds.tc.file=<old file> -q -B > <new file>
```

The *OldFile* implementation will be removed in the next major release of the VDS. Please upgrade your existing TC, and refrain from using the old format.

6.5 Properties

Various properties configure the transformation catalog. All TC-related properties have reasonable defaults, as explained in the Property Reference Guide.

The central property *vds.tc* determines the implementation of the Transformation Catalog.

```
vds.tc          (File | Database | OldFile)
```

In absence, it will default to the *File* implementation. The deprecated values *multiple* and *single* are synonyms for *OldFile*.

Any file-based transformation catalog, *File* or *OldFile*, should declare the location of the file representing the catalog. The location should be an absolute path:

```
vds.tc.file    ${vds.home}/var/tc.data
```

The file location defaults to *$VDS_HOME/var/tc.data* in the absence of a location property.

In case of the *Database* implementation, several properties configure the VDS database abstraction layer. For the *Database* setup on top of MySQL, you should configure the following properties:

```
vds.tc      Database
vds.db.tc.driver          MySQL
vds.db.tc.driver.url      jdbc:mysql://[dbhost[:dbport]]/dbname
vds.db.tc.driver.user     dbuser
vds.db.tc.driver.password dbpass
```

For the same setup on top of PostGreSQL, you should configure:

```
vds.tc      Database
vds.db.tc.driver          Postgres
vds.db.tc.driver.url      jdbc:postgresql:[//dbhost[:dbport]/]dbname
vds.db.tc.driver.user     dbuser
vds.db.tc.driver.password dbpass
```

The brackets contain optional values. The Database Administration Guide contains more information on the different possibilities for the value. The database driver may take additional optional configuration values. Depending on the RDBMS, the database account and password may not be required.

If all VDS databases use the same access information, as explained in the Database Administration Guide, you can substitute the "***tc***" portion of the key against an asterisk (*****).

6.6 Transformation Catalog Entry Requirements

Each row in a transformation catalog constitutes one transformation at one specific site. Certain entries are required irrespective of the chosen implementation of the Transformation Catalog.

The concrete planners require certain applications for their auxiliary jobs. These applications are shipped as part of the VDS worker package, should be installed at each remote site, and be visible to all worker nodes. The VDS binary package is a true super-set of the VDS worker package, and thus contains all required applications.

The required entries fall in two categories. The *vds-get-sites* client configures most of these settings appropriately.

6.6.1 Remote sites

Each remote site must declare the following two transformations.

LFN	PFN
transfer	path to $VDS_HOME/bin/transfer
dirmanager	path to $VDS_HOME/bin/dirmanager

Optionally, depending on transfer requirements, each remote site could also declare the following transformations:

LFN	PFN
T2	path to $VDS_HOME/bin/T2
globus-url-copy	path to $GLOBUS_LOCATION/bin/globus-url-copy

If applicable, the *type* is *INSTALLED*. If applicable, the *System Information* of the transformation should match the remote site's *System Information* to avoid any executable staging or planning errors.

6.6.2 Local site

The local site runs more auxiliary processes than any remote site. Thus, the local site must declare the following transformations:

LFN	PFN	Environment Profile
RLS_Client	path to $VDS_HOME/bin/rls-client	GLOBUS_LOCATION VDS_HOME JAVA_HOME CLASSPATH

The *type*, if applicable, is *INSTALLED*. The *Sytem Information*, if applicable, matches the submit host's *System Information*. Additionally, the *RLS_Client* transformation requires environment settings for correct execution. The *GLOBUS_LOCATION, VDS_HOME* and *JAVA_HOME* point to their correct locations on the submit host. The *CLASSPATH* must contain the following JAR files:

- $GLOBUS_LOCATION/lib/rls.jar
- $VDS_HOME/lib/rls.jar
- $VDS_HOME/lib/java-getopt-1.0.9.jar
- $VDS_HOME/lib/gvds.jar

Please note that the above jar paths employ environment variables for illustrative purposes only. You must not use environment variables in your transformation catalog. Please substitute the environment variables to their correct value on the submit host.

6.7 The tc-client tool

The *tc-client* command is a generic client that performs the three basic operation of adding, deleting and querying of any Transformation Catalog implemented by the TC API. The client implements all the operations supported by the TC API. It is up to the TC implementation whether they support all operations or modes.

The following three operations are supported by the *tc-client*. When invoking the client, one of the following operations has to be specified:

ADD: This operation allows the client to add or update entries in the *Transformation Catalog*. Entries can be added one by one on the command line or in bulk by using the *BULK* Trigger and providing a file with the necessary entries. Profiles can be added to either the logical transformation or the physical transformation.

DELETE: This operation allows the client to delete entries from the *Transformation Catalog*. Entries can be deleted based on logical transformation, by resource, by transformation type as well as the transformation system information. Profiles associated with the logical or physical transformation can be deleted.

QUERY: This operation allows the client to query for entries from the *Transformation Catalog*. Queries can be made for printing all the contents of the Catalog or for specific entries, for all the logical transformations or resources etc.

See the **Triggers 6.7.3** and **VALID COMBINATIONS 6.7.5** section for more details.

6.7.1 Usage

```
tc-client Operations Triggers [OPTIONS] [-h] [-v] [-V]
```

6.7.2 Operations

The client can perform one of three operations:

−a −−add Perform addition operations on the TC.

−d −−delete Perform delete operations on the TC.

−q −−query Perform query operations on the TC.

6.7.3 Triggers

Triggers modify the behavior of an operation. For instance, if you want to perform a bulk operation, you use a BULK Trigger. If you want to perform an operation on a logical transformation, you use the LFN Trigger.

The following seven triggers are available. See the **VALID COMBINATIONS** section for the correct grouping and usage.

−B Triggers a bulk operation.

−L Triggers an operation on a logical transformation.

−P Triggers an operation on a physical transformation

−R Triggers an operation on a resource.

−E Triggers an operation on a Profile.

−T Triggers an operation on a Type.

−S Triggers an operation on a System information.

6.7.4 OPTIONS

The following options are applicable for all the operations.

−l	−−lfn *logical TX*	The logical transformation to be added. The format is ***NAMESPACE::NAME:VERSION***. The name is always required, namespace and version are optional.
−p	−−pfn physical TX	The physical transformation to be added. For *INSTALLED* executables it is a local file path, for all others its a url.
−t	−−type type	The type of physical transformation. Valid values are : *INSTALLED, STATIC_BINARY, DYNAMIC_BINARY, SCRIPT, SOURCE, PACMAN_PACKAGE.*, see section 6.1.2.
−r	−−resource siteID/resourceID	The site identifier where the transformation is located.
−e	−−profile profiles	The profiles for the transformation. Multiple profiles of same namespace can be added simultaneously by separating them with a comma ",". Each profile section is written as ***NAMESPACE::KEY=”VALUE”,KEY2=”VALUE2”.*** An example is given below.
−s	−−system system-info	The architecture, OS, OS version and glibc version as applicable for the executable. Please refer to section 6.1.3 for details.
−v	−−verbose	Displays the output in verbose mode (lots of debugging info).
−V	−−version	Displays the version number of the Virtual Data System.
−h	−−help	Generates help

An example argument to the -e option can look like this:

```
tc-client -a -E -L -l date -e \
ENV::JAVA_HOME=”/usr/bin/java”,VDS_HOME=”/usr/local/vds”
```

To add multiple namespaces you need to repeat the −e option for each namespace. E.g.

```
-e ENV::JAVA_HOME=”/usr/bin/java” -e GLOBUS::JobType=”MPI”,COUNT=”10”
```

6.7.5 VALID COMBINATIONS

The following are a few examples of valid combinations of **OPERATIONS, TRIGGERS** and **OPTIONS** for the tc-client.

6.7.5.1 ADD

Add TC Entry - Adds a single entry into the transformation catalog.

```
-a -l lfn -p pfn -t type -r resource -s system [-e profiles..]
```

Add PFN Profile - Adds profiles to a specified physical transformation on a given resource and of a given type.

```
-a -P -E -p pfn -t type -r resource -e profiles ....
```

Add LFN Profile - Adds profiles to a specified logical transformation

```
-a -L -E -l lfn -e profiles ....
```

Add Bulk Entries - Adds entries in bulk mode by supplying a file in the new File format.

```
-a -B -f file
```

6.7.5.2 DELETE

Delete all TC - Deletes the entire contents of the TC. **Caution!** The client does not prompt again after you ask for dropping the contents of the TC nor does it create a backup.

```
-d -BPRELST
```

Delete by LFN - Deletes entries from the TC for a particular logical transformation and additionally a resource and/or type.

```
-d -L  -lfn [-r resource] [-t type]
```

Delete by PFN - Deletes entries from the TC for a given logical and physical transformation and additionally on a particular resource and or of a particular type.

```
-d -P -l lfn -p pfn [-r resource] [-t type]
```

Delete by Type - Deletes entries from TC of a specific type and/or on a specific resource.

```
-d -T -t type [-r resource]
```

Delete by Resource - Deletes the entries from the TC on a particular resource.

```
-d -R -r resource
```

Delete by SysInfo - Deletes the entries from the TC for a particular system information type.

```
-d -S -s sysinfo
```

Delete Pfn Profile - Deletes all or specific profiles associated with a physical transformation

```
-d -P -E -p pfn -r resource -t type [-e profiles ..]
```

Delete Lfn Profile - Deletes all or specific profiles associated with a logical transformation

```
-d -L -E -l lfn -e profiles ....
```

6.7.5.3 QUERY

Query Bulk - Queries for all the contents of the TC. It produces a file format TC which can be added to another TC using the bulk option.

```
-q -B
```

Query LFN - Queries the TC for logical transformation and/or on a particular resource and/or of a particular type.

```
-q -L [-r resource] [-t type]
```

Query PFN - Queries the TC for physical transformations for a give logical transformation and/or on a particular resource and/or of a particular type.

```
-q -P -l lfn [-r resource] [-t type]
```

Query Resource - Queries the TC for resources that are registered and/or resources registered for a specific type of transformation.

```
-q -R [-l lfn] [-t type]
```

Query Lfn Profile - Queries for profiles associated with a particular logical transformation

```
-q -L -E -l lfn
```

Query Pfn Profile - Queries for profiles associated with a particular physical transformation

```
-q -P -E -p pfn -r resource -t type
```

6.7.6 Properties

Please refer to section 6.5 for properties applicable to any transformation catalog.

6.7.7 Files

$VDS_HOME/var/tc.data is the default location for the file corresponding to a file-based Transformation Catalog.

$VDS_HOME/etc/properties is the location to specify properties to change what Transformation Catalog Implementation to use and the implementation related properties.

gvds.jar contains all compiled Java byte code to run the GriPhyN Virtual Data System.

6.7.8 Environment variables

For proper execution of the *tc-client*, the following environment variables need to be set:

- *$VDS_HOME* points to your VDS installation.
- *$JAVA_HOME* points to the JDK or JRE installation directory.
- *$CLASSPATH* contains the necessary JAR files for the VDS execution environment.

6.8 Developer API

http://vds.isi.edu/docs/vds-1.4.1-javadoc/ documents the Java API for the Transformation Catalog. Refer to *org.griphyn.cPlanner.catalog.TransformationCatalog* for the class description. It is possible to provide your own implementation of the API, and let the VDS dynamic class-loader interface load your own set of classes.

7 Profiles in GVDS

Profiles in the Griphyn Virtual Data System (GVDS) are triples of the form (namespace, key, value) that allow the user to control the behavior of the planners and runtime environment of jobs in the workflow. This document describes

- the various types of profiles
- levels of priorities of profiles
- how to specify profiles

7.1 Profile Namespaces

Each namespace refers to a different aspect of a job's runtime settings. Currently GVDS supports the following namespaces for profiles

- env execution time environment parameters .
- globus globus RSL parameters.
- condor condor configuration parameters
- dagman dagman configuration parameters
- vds planner configuration parameters

A profile's representation in the concrete plan (usually condor submit files) depends on the value of the namespace.

7.1.1 env

The env namespace allows the user to specify the environment variables that need to be set when a job is executed on the remote grid site. The key for a profile denotes the name of the environment variable, and the value denotes the it's value.

All the profiles(env,key,value) associated with a particular job are translated to a semi colon separated list of key=value pairs. This list is passed as the argument to the environment command in the condor submit file for the job. Some common environment variables that are often required to be set for VDS executables (like transfer, rls-client, rc-client) are VDS_HOME and GLOBUS_LOCATION.

```
##########################################################################
# GRIPHYN VDS SUBMIT FILE GENERATOR
# DAG : black-diamond, Index = 0, Count = 1
# SUBMIT FILE NAME : findrange_ID000002.sub
##########################################################################
globusrsl = (jobtype=single)
environment=GLOBUS_LOCATION=/shared/globus;LD_LIBRARY_PATH=/shared/glob
us/lib;
arguments = -n vahi::findrange:1.0 -N vahi::left:1.0 -R isi_condor
/nfs/asd2/pegasus/software/linux/vds/default/bin/keg  -a left -T60  -i
vahi.f.b1 vahi.f.b2 -o vahi.f.c1 -p 0.5
executable = /shared/software/linux/vds/default/bin/kickstart
```

```
globusscheduler = columbus.isi.edu/jobmanager-condor
remote_initialdir = /shared/CONDOR/workdir/isi_hourglass
universe = globus
…
…
queue
######################################################################
# END OF SUBMIT FILE
```

Figure 1: env profiles represented in the condor submit file

7.1.2 globus

The globus namespace allows the user to specify additional globus RSL parameters that may need to be set to run job on the remote scheduler. The key for a profile denotes the name of the RSL parameter and the value it's value.

All the profiles(globus,key,value) associated with a particular job are specified as the value to the globusrsl command in the condor submit file for the job.

```
######################################################################
# GRIPHYN VDS SUBMIT FILE GENERATOR
# DAG : black-diamond, Index = 0, Count = 1
# SUBMIT FILE NAME : findrange_ID000002.sub
######################################################################
globusrsl = (jobtype=single)(queue=fast)(project=nvo)
environment=GLOBUS_LOCATION=/shared/globus;LD_LIBRARY_PATH=/shared/glob
us/lib;
arguments = -n vahi::findrange:1.0 -N vahi::left:1.0 -R isi_condor
/nfs/asd2/pegasus/software/linux/vds/default/bin/keg  -a left -T60  -i
vahi.f.b1 vahi.f.b2 -o vahi.f.c1 -p 0.5
executable = /shared/software/linux/vds/default/bin/kickstart
globusscheduler = columbus.isi.edu/jobmanager-condor
remote_initialdir = /shared/CONDOR/workdir/isi_hourglass
universe = globus
…
…
queue
######################################################################
# END OF SUBMIT FILE
```

Figure 2: globus profiles represented in the condor submit file

Some of the common RSL parameters that a user may need to specify as profiles are listed below. For an authoritative list refer to the online RSL specification.

	Key	Description
1	count	the number of executions of the executable.
2	jobtype	specifies how the jobmanager should start the remote job. It needs to be set to mpi when running mpi jobs through GVDS.
3	maxcputime	the max cpu time for a single execution of a job.
4	maxmemory	the maximum memory in MB required for the job

5	maxtime	the maximum time or walltime for a single execution of a job.
6	maxwalltime	the maximum walltime for a single execution of a job.
7	minmemory	the minumum amount of memory required for this job
8	project	target the job to be allocated to a project account as defined by the scheduler at the defined (remote) resource.
9	queue	target the job to a queue (class) name as defined by the scheduler at the defined (remote) resource.

Table 1: Commonly used globus profile keys

Pegasus prevents the user from specifying certain RSL parameters as profiles, as they are automatically generated or can be overridden through some different means. The disallowed RSL parameters are listed below.

	Key	**Reason for not allowing**
1	arguments	the arguments to an executable can only be specified through DAX.
2	directory	determined from the site catalog and properties file.
3	environment	to specify an environment, use profiles with env namespace
4	executable	determined from the transformation catalog
5	stdin	can only be specified in DAX or is generated automatically
6	stdout	can only be specified in DAX or is generated automatically
7	stderr	can only be specified in DAX or is generated automatically

Table 2: Disallowed globus profile keys

7.1.3 condor

The condor namespace allows the user to set condor commands for a particular job. The key for a profile denotes the name of the condor command and the value it's value.

All the profiles(condor,key,value) associated with a particular job are translated to key=value lines separated by \n in the condor submit file for the job.

Some of the common condor commands that a user may need to specify are listed below. For an authoritative list refer to the online condor documentation. Note: The planners by default specify a lot of condor commands in the submit files depending upon the job, and where it is being run.

	Key	**Description**
1	universe	set it to standard, if you want to run jobs in standard universe
2	periodic_release	the number of times job is released back to the queue if it goes to

		HOLD. Default value of 3 is used if not specified by the user.
3	periodic_remove	the number of times a job is allowed to get into HOLD state before being removed from the queue. Default value of 3 is used if not specified by the user.
4	filesystemdomain	used for condor glidein's to ensure that jobs are actually run on nodes glided in from a particular site.

Table 3: Commonly used condor profile keys

Pegasus prevents the user from specifying certain condor commands as profiles, as they are automatically generated or can be overridden through some different means. The disallowed condor commands are listed below.

	Key	**Reason for not allowing**
1	arguments	the arguments to an executable can only be specified through DAX.
2	environment	to specify an environment, use profiles with env namespace
3	executable	determined from the transformation catalog
4	globusscheduler	determined by the site selector and the site catalog.
5	log	has to be same for the whole dag. The planners generate themselves.
6	remote_initialdir	determined from the site catalog and properties file.
7	stream_error	applicable only for globus/grid jobs. However can be set through boolean property vds.scheduler.condor.error.stream
8	stream_output	applicable only for globus/grid jobs. However can be set through boolean property vds.scheduler.condor.output.stream

Table 4: Disallowed condor profile keys

7.1.4 dagman

The dagman namespace allows the user to specify what post/pre scripts need to be run when a job is run by DAGMAN. By default, all the jobs on a grid site are launched by a gridstart (kickstart) executable , that tracks the execution of a job namely the exitcode with which the jobs exit. Kickstart generates an invocation record on it's stdout that is transferred back to the submit host by Condor. Each job is associated with a post script that parses this invocation record generated by kickstart to determine the exitcode. The exitcode is then passed to DAGMAN that allows it to detect failure of jobs.

However, for certain types of jobs(MPI and standard universe), kickstart cannot be used to launch them. In those situations the user will need to specify his own postscript that can determine the correct exitcode to be passed to DAGMAN, by parsing the stdout of the application that is staged back to the submit host by Condor.

All the profiles(dagman,key,value) associated with a particular job are translated to dagman commands for the job in the .dag file.

The keys for the profiles with namespace as dagman are listed below.

	Key	Description
1	PRE	the path to the pre script on the submit host that is to be invoked for the job, before DAGMAN submits the job.
2	PRE_ARGS	the arguments with which the prescript has to be invoked.
3	POST	the path to the pre script on the submit host that is to be invoked on a job's stdout(staged back by condor), after the job has finished execution on the remote site.
4	POST_ARGS	the arguments with which the postscript has to be invoked.
5	RETRY	the number of times DAGMAN retries the job in case of it detecting that job failed.

Table 5: Allowed dagman profile keys

7.1.5 vds

The vds namespace allows the user to specify extra options to the planners that can be applied selectively to a job or a group of jobs. These profiles can be used by site selectors to make better decisions.

The keys that Pegasus understands and what action it takes are listed below.

	Key	Description
1	workdir	sets the remote initial dir for a condor globus job. Overrides the work directory that is determined from the site catalog and properties file
2	bundle	the number of clustered jobs that need to be created per level of the workflow. Used in clustering.
3	collapse	the number of compute jobs that need to be in a single clustered job per level of the workflow. Used in clustering.
4	collapser	indicates the collapser executable that is used to run the clustered job on the remote site.
5	gridstart	determines the launching executable to be used for launching a job on the grid. User can specify Kickstart|NoGridStart Support for K2 is expected in the coming releases.
6	bundle_stagein	the number of stage in jobs that are required for a particular site. Can either be specified in the site catalog for the site or in the transformation catalog for the transfer executable that you are using.
7	group	allows the user to specify that multiple jobs are a part of the same group. The Group site selector then can schedule all the jobs in a group to the same site.
8	change_dir	triggers kickstart into changing the directory before launching the application. In this case, kickstart is run in the condor spool directory (if running in condor vanilla universe) or the gram

		scratch directory (if running in condor grid universe)

Table 6: vds profile keys

7.2 Specifying Profiles

Profiles (namespace, key, value) can be specified at various levels in the GVDS. The user has an option to associate profiles at

- VDL level (profiles are specified per job/transformation)
- Site catalog (profiles are specified per site and apply to all jobs that run on that site)
- Transformation Catalog (profiles are specified per transformation per site)

7.2.1 Profiles in VDL

The user can associate profiles with logical transformations in VDL. These profiles then appear in the DAX in the jobs that refer to those logical transformations.

The commands to insert a profile with a transformation is as follows. Here we are associating a profile in the globus namespace that indicates the maxwalltime of 10 minutes for the job.

The profiles appear in the DAX as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<adag xmlns="http://www.griphyn.org/chimera/DAX"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.griphyn.org/chimera/DAX
http://www.griphyn.org/chimera/dax-1.8.xsd"
version ="1.8" count="1" index="0" name="blackdiamond">

<!-- part 1: list of all files used (may be empty) -->
  <filename file="vahi.f.c1" link="input"/>
  <filename file="vahi.f.c2" link="input"/>
  <filename file="vahi.f.d" link="output"/>

<!-- part 2: definition of all jobs (at least one) -->

  <job id="ID000001" namespace="vahi" name="analyze" version="1.0"
level="1" dv-namespace="vahi" dv-name="bottom" dv-version="1.0">
    <argument>-a bottom -T 6  -i <filename file="vahi.f.c1"/> <filename
file="vahi.f.c2"/>  -o <filename file="vahi.f.d"/></argument>
    <profile namespace="globus" key="maxwalltime">10</profile>
    <uses file="vahi.f.c1" link="input" dontRegister="false"
dontTransfer="false"/>
    <uses file="vahi.f.c2" link="input" dontRegister="false"
dontTransfer="false"/>
    <uses file="vahi.f.d" link="output" dontRegister="false"
dontTransfer="false"/>
  </job>

<!-- part 3: list of control-flow dependencies(empty for single jobs) -
->
</adag>
```

7.2.2 Profiles in site catalog

The user can associate profiles with a site in the site catalog. The profiles specified in the site catalog for a particular site, apply to all jobs that are executed on that particular site. Usually, site specific information is specified here, like environment variables that need to be set for all jobs that are run on a particular site using profiles with namespace env. For e.g the user might want to set environment variables like VDS_HOME and GLOBUS_LOCATION in the site catalog. These are required to be set for most of the vds worker tools, and other grid tools that the user may commonly use.

Currently, there is no tool to insert profiles for sites into the site catalog. The only option is to hand edit the site catalog file (either the xml version or the multiline version).

In the XML version, to specify a profile (ns,keyname,value) the user needs to insert a profile element in the pool element

<profile namespace=”ns” key=”keyname”>value</profile>

The valid values for namespace attribute are

- env
- globus
- condor
- dagman
- vds

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.griphyn.org/chimera/GVDS-PoolConfig"
xsi:schemaLocation="http://www.griphyn.org/chimera/GVDS
http://www.griphyn.org/chimera/gvds-poolcfg-1.3.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.3">

  <pool handle="isi_condor"
gridlaunch="/home/shared/vds/bin/kickstart">
    <profile namespace="env" key="GLOBUS_LOCATION"
>/home/shared/globus/</profile>
    <profile namespace="env" key="LD_LIBRARY_PATH"
>/home/shared/globus/lib</profile>
    <lrc url="rls://sukhna.isi.edu" />
   <gridftp  url="gsiftp://smarty.isi.edu/"
storage="/home/shared/isi_condor/data" major="2" minor="4" patch="0" />
    <jobmanager universe="transfer" url="columbus.isi.edu/jobmanager-
fork" major="2" minor="4" patch="0" />
    <jobmanager universe="vanilla" url="columbus.isi.edu/jobmanager-
condor" major="2" minor="4" patch="0" />
    <workdirectory >/home/shared/isi_condor/exec</workdirectory>
  </pool>

</config>
```

Figure 3: Profiles in the XML version of the site catalog

Note that the order is important in the above example. Profile elements should appear first in the pool elements. If the order is wrong, you would see a lot of SAX Parser errors and warnings.

In the multi-line textual version of the site catalog, to specify a profile (ns,keyname,value) the user needs to insert *profile ns "keyname" "value"* in the description for the pool.

```
pool isi_condor {
  lrc "rls://sukhna.isi.edu"
  profile env "GLOBUS_LOCATION"  "/home/shared/globus"
  profile env "LD_LIBRARY_PATH" "/home/shared/globus/lib"
  gridftp "gsiftp://smarty.isi.edu/home/shared/isi_condor/data "2.4"
  gridlaunch "/home/shared/vds/bin/kickstart"
  workdir "/home/shared/isi_condor/exec"
  universe transfer "columbus.isi.edu/jobmanager-fork" "2.4"
  universe vanilla "columbus.isi.edu/jobmanager-condor" "2.4"
}
```

Figure 4: Profiles in the multi line textual version of the site catalog

Note that the order within the pool definition is not important. Profiles can appear anywhere not necessarily only after the lrc specification.

7.2.3 Profiles in Transformation Catalog

The user can associate profiles with a physical transformation (pfn,site) in the transformation catalog. Optionally, in the database version of the transformation catalog the user can associate profiles to a logical transformation.

Usually, user will want to specify those profiles in the transformation catalog that apply to a particular transformation on a particular site. For e.g. to run a particular transformation, some transformation specific environment variables may be required. Job clustering parameters are also specified in the transformation catalog.

The user can use the tc-client to specify a profile with a transformation. The tc-client works with both the database and textual versions of the transformation catalog.

```
tc-client -a -P -E -p /home/shared/executables/analyze -t INSTALLED -r
isi_condor -e env::GLOBUS_LOCATION="/home/shared/globus"
```

Figure 4: Specifying a profile with the physical transformation

The above example, adds an environment variable GLOBUS_LOCATION to pfn /home/shared/executables/analyze on site isi_condor. More details can be found in the transformation catalog guide.

7.3 Priority Ordering of Profiles

Irrespective of where the profiles are specified, eventually the profiles are associated with jobs. Thus, it is possible that for a job there can be more than one candidate value for a particular profile. For e.g a certain profile (ns,key,value) might be specified for a site X in the site catalog, and a profile (ns,key,value1)for a transformation Tx on site X in the transformation catalog. In this case, a job that refers to transformation Tx and runs on site X has two candidate values for the profile with namespace as ns and keyname as key. This section specifies the priority ordering of profiles.

In the GVDS, there are four levels of priorities for profiles, where a higher priority takes precedence over a lower level of priorities:

[1] user local profiles

[2] Transformation Catalog profiles

[3] Site Catalog profiles

[4] VDL/DAX profiles

Currently in this release of GVDS (1.4.0), the support for [1] does not exist, and hence the transformation catalog profiles have the highest priority.

8 Running Pegasus

Revision $Revision: 1.3 $ of file $RCSfile: VDSUG_RunningPegasus.xml,v $.

8.1 Introduction

Pegasus [1] is a configurable system for mapping and executing abstract application workflows over the Grid. An abstract application workflow is represented as a directed acyclic graph where the vertices are the compute tasks and the edges represent the data dependencies between the tasks. The input to Pegasus is a description of the abstract workflow in XML format. The mapping of tasks to the execution resources is done based on information derived for static and/or dynamic sources (site catalog, tc.data, MDS[3], RLS[2] etc). The output is an executable workflow (also called the concrete workflow) that can be executed over the Grid resources. In case the workflow tasks are mapped to multiple resources that do not share a file system, explicit nodes are added to the workflow for orchestrating data transfer between the tasks.

The format of the concrete workflow is determined by the execution engine used for executing the concrete workflow over the mapped resources. For example, if Condor DAGMan [5] is used as the enactment engine, then the concrete workflow consists of a DAG description file which lists the dependencies and Condor submit files for each task in the concrete workflow. Pegasus can also use other enactment engines such as GRMS [7] etc.

Pegasus has been mostly used with Condor DAGMan. The use of DAGMan does not imply the use of a Condor pool for executing the workflow. However, it is one of the possibilities. Instead, Condor-G [5] can be used to execute any task in the workflow on any Grid resource that provides a Globus GRAM [4] interface. The combination of DAGMan and Condor-G provides a powerful mechanism for executing the workflow over the Grid resources.

The purpose of this document is to give an overview of the various resource configurations that can be used with Pegasus when Condor DAGMan is used as the workflow enactment engine. The classification is done based on the mechanism used for submitting jobs to these resources. 8.2 gives an overview of the various resource configurations and 8.3 , 8.4 specifies the format of the generated Condor submit files for each of these configurations.

8.2 Resource Configurations

This section discusses the various resource configurations that can be used with Pegasus when Condor DAGMan is used as the workflow execution engine. It is assumed that there is a submit host where the workflow is submitted for execution. The following classification is done based the mechanism used for submitting jobs to the Grid resources and monitoring them.

The classifications explored in this document are

Using Globus GRAM

Using Condor pool

Both the above configurations use Condor DAGMan to maintain the dependencies between the jobs, but differ in the manner as to how the jobs are launched. A combination of the above mentioned approaches is also possible where some of the tasks in the workflow are executed in the Condor pool and the rest are executed on remote resources using Condor-G.

8.2.1 Using Globus GRAM

In this configuration, it is assumed that the target execution system consists of one or more Grid resources. These resources may be geographically distributed and under various administrative domains. Each resource might be a single desktop computer or a network of workstations (NOW) or a cluster of dedicated machines. However, each resource must provide a Globus GRAM interface which allows the users to submit jobs remotely. In case the Grid resource consists of multiple compute nodes, e.g. a cluster or a network of workstations, there is a central entity called the head node that acts as the single point of job submissions to the resource. It is generally possible to specify whether the submitted job should run on the head node of the resource or a worker node in the cluster or NOW. In the latter case, the head node is responsible for submitting the job to a local resource management system (PBS, LSF, Condor etc) which controls all the machines in the resource. Since, the head node is the central point of job submissions to the resource it should not be used for job execution since that can overload the head node delaying further job submissions. Pegasus does not make any assumptions about the configuration of the remote resource; rather it provides the mechanisms by which such distinctions can be made.

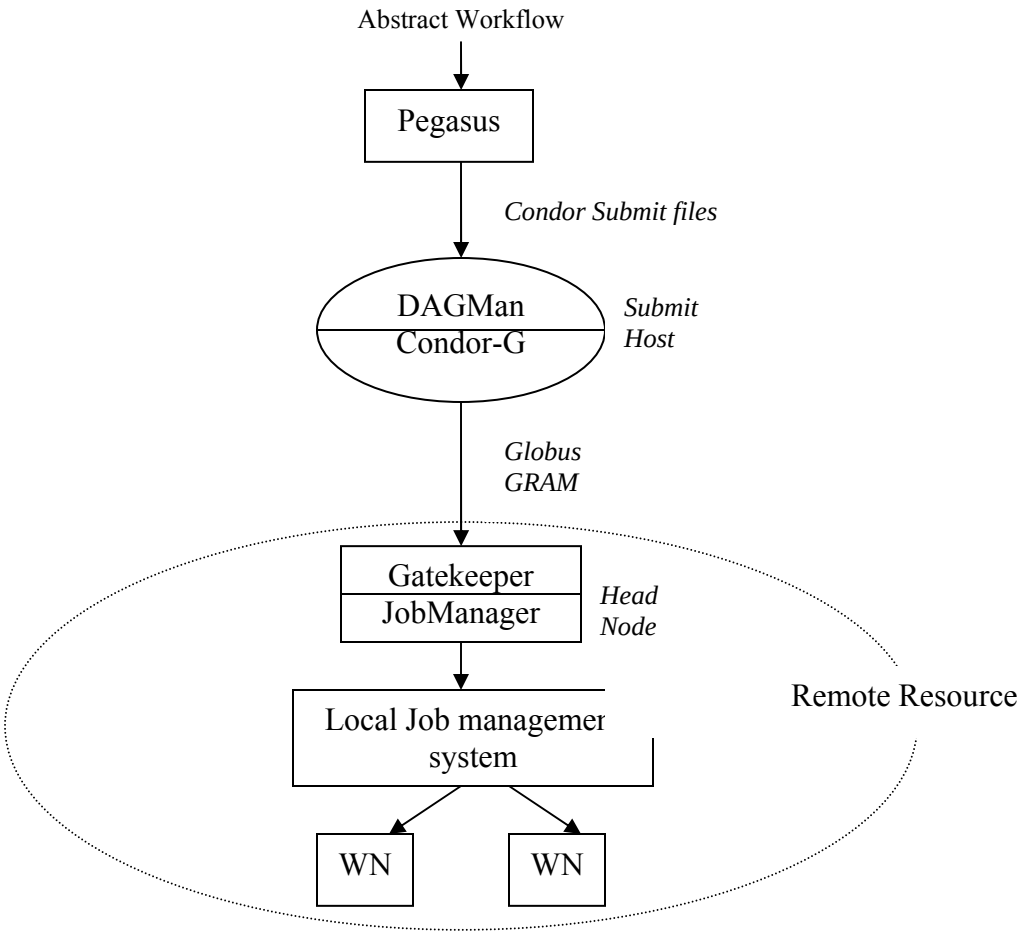


Figure 1: Resource Configuration using GRAM

In this configuration, Condor-G [5] is used for submitting jobs to these resources. Condor-G is an extension to Condor that allows the jobs to be described in a Condor submit file and when the job is submitted to Condor for execution, it uses the Globus GRAM interface to submit the job to the remote resource and monitor its execution. The distinction is made in the Condor submit files by specify the *universe* as *globus* and the *globusscheduler* attribute is used to indicate the location of the head node for the remote resource. Thus, Condor DAGMan is used for maintaining the dependencies between the jobs and Condor-G is used to launch the jobs on the remote resources using GRAM.

The implicit assumption in this case is that all the worker nodes on a remote resource have access to a shared filesystem that can be used for data transfer between the tasks mapped on that resource. This data transfer is done using files.

8.2.2 Condor pool

A Condor pool is a set of machines that use Condor for resource management. A Condor pool can be a cluster of dedicated machines or a set of distributively owned machines. Pegasus can generate concrete workflows that can be executed on a Condor pool.

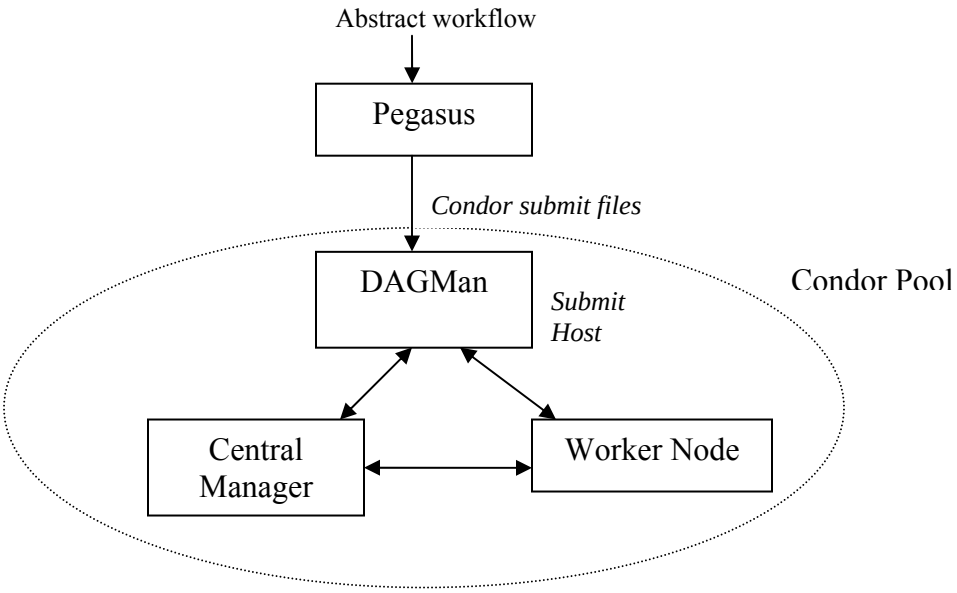


Figure 2. The Grid resources appear to be part of a Condor pool

The workflow is submitted using DAGMan from one of the job submission machines in the Condor pool. It is the responsibility of the Central Manager of the pool to match the task in the workflow submitted by DAGMan to the execution machines in the pool. This matching process can be guided by including Condor specific attributes in the submit files of the tasks.

If the user wants to execute the workflow on the execution machines (worker nodes) in a Condor pool, there should be a resource defined in the site catalog which represents these execution machines. The *universe* attribute of the resource should be *vanilla*. There can be multiple resources associated with a single Condor pool, where each resource identifies a subset of machine (worker nodes) in the pool. Pegasus currently uses the *FileSystemDomain* classad[] attribute to restrict the set of machines that make up a single resource.

To clarify this point, suppose there are certain execution machines in the Condor pool whose FileSystemDomain is set to "*ncsa.teragrid.org*". If the user wants to execute the workflow on these machines, then there should be a resource, say "NCSA_TG" defined in the site catalog and the FileSystemDomain and universe attributes for this resource should be defined as "*ncsa.teragrid.org*" and "*vanilla*" respectively. When invoking Pegasus, the user should select NCSA_TG as the compute resource.

```
<config xmlns="http://www.griphyn.org/chimera/GVDS-
PoolConfig"
xsi:schemaLocation="http://www.griphyn.org/chimera/G
VDS http://www.griphyn.org/chimera/gvds-poolcfg-1.3.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" vers
ion="1.3">
……

<pool handle="ncsa_tera"
gridlaunch="/opt/VDS/bin/kickstart" >
    <profile namespace="condor"
key="universe">vanilla</profile>
    <profile namespace="condor"
key="FileSystemDomain">ncsa.teragrid.org</profile>
    <gridftp  url="gsiftp://ncsa.teragrid.org/"
storage="/vds/data/" major="2" minor="4" patch="0" />
    <jobmanager universe="transfer"
url="ncsa.teragrid.org/jobmanager-condor" major="2"
minor="4" patch="0" />
    <jobmanager universe="vanilla"
url="ncsa.teragrid.org/jobmanager-condor" major="2"
minor="4" patch="0" />
    <workdirectory
>/opt/workspace/CONDOR/exec</workdirectory>
</pool>
…….
</config>
```

Figure 3: Specifying FileSystemDomain in Pool Configuration File

8.3 Running Jobs through Globus GRAM

This section describes how Pegasus runs jobs through Globus GRAM via Condor-G. Pegasus by default generates a concrete workflow in terms of condor submit files and dag containing workflow dependencies, that when submitted to DAGMan ends up running on remote resources.

In the configuration described in 8.2.1, Pegasus generates jobs in the condor *globus* universe. In the addition the condor submit file, identifies what globus scheduler (jobmanager contact) the job needs to be run on.

The jobmanager contacts associated with a resource are identified in pool configuration file. There are two types of jobmanager contacts associated with a resource/pool.

vanilla

transfer

The vanilla job manager is identified by specifying the attribute universe as vanilla.

Pegasus uses this jobmanager for executing compute jobs, when the site selector schedules those compute jobs to that pool.

The transfer jobmanager is identified by specifying the attribute universe as transfer. Pegasus uses this jobmanager for executing transfer jobs on that particular resource.

```
<config xmlns="http://www.griphyn.org/chimera/GVDS-
PoolConfig"
xsi:schemaLocation="http://www.griphyn.org/chimera/G
VDS http://www.griphyn.org/chimera/gvds-poolcfg-1.3.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" vers
ion="1.3">
……

<pool handle="isi_condor"
gridlaunch="/opt/VDS/bin/kickstart" >
    <profile namespace="env" key="GLOBUS_LOCATION">
/nfs/v6/globus/GT2/linux/STABLE</profile>
    <profile namespace="env" key="LD_LIBRARY_PATH">
/nfs/v6/globus/GT2/linux/STABLE/lib</profile>
    <gridftp  url="gsiftp://smarty.isi.edu/"
storage="/vds/data/" major="2" minor="4" patch="0" />
    <jobmanager universe="transfer"
url="columbus.isi.edu/jobmanager-condor" major="2"
minor="4" patch="0" />
    <jobmanager universe="vanilla"
url="columbus.isi.edu/jobmanager-condor" major="2"
minor="4" patch="0" />
    <workdirectory >/nfs/cgt-
scratch/vahi/CONDOR/exec</workdirectory>
</pool>
…….
</config>
```

Figure 4: Specifying Job managers in Pool Configuration File

The reason for having different jobmanagers for compute and transfer jobs is to account for the case where the worker nodes are behind a firewall. In that case, the transfer jobs need to run on a node, that has access to the outside world to stage-in the input files and stage-out the materialized data. The nodes that run the compute job do not need access the outside world, as the compute jobs work on data that has already been staged-in by the transfer jobs.

8.4 Running Jobs in Condor Pool

This section describes the various changes in the resource configurations files required for running the workflow jobs in a Condor pool as mentioned in 8.2.2. Pegasus by default uses Condor-G for job submissions (8.2.1) i.e. it generates jobs in the condor *globus* universe. For running jobs directly in a Condor pool, the jobs need to be generated for the *vanilla* universe. This can be done using the condor profile namespace in Pegasus, and associating it with the job or with the execution pool (compute resource). The recommended way is to tag your execution pool as a vanilla pool, which would result in all the jobs scheduled on that pool being run in the vanilla universe. Alternatively, the user could tag a particular job either in the transformation catalog or in the DAX (abstract representation of the workflow) as to be run in the vanilla universe.

In all the three cases, the user needs to insert a key universe with the value vanilla in condor profile namespace.

To tag the pool as a vanilla universe pool, the user needs to update his Resource Information Catalog a.k.a Pool catalog. Usually, this is the site catalog file in XML form.

```
<config xmlns="http://www.griphyn.org/chimera/GVDS-
PoolConfig"
xsi:schemaLocation="http://www.griphyn.org/chimera/G
VDS http://www.griphyn.org/chimera/gvds-poolcfg-1.3.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" vers
ion="1.3">
……

<pool handle="ncsa_tera"
gridlaunch="/opt/VDS/bin/kickstart" >
    <profile namespace="condor"
key="universe">vanilla</profile>
    <gridftp  url="gsiftp://ncsa.teragrid.org/"
storage="/vds/data/" major="2" minor="4" patch="0" />
    <jobmanager universe="transfer"
url="ncsa.teragrid.org/jobmanager-condor" major="2"
minor="4" patch="0" />
    <jobmanager universe="vanilla"
url="ncsa.teragrid.org/jobmanager-condor" major="2"
minor="4" patch="0" />
    <workdirectory
>/opt/workspace/CONDOR/exec</workdirectory>
</pool>
…….
</config>
```

Figure 5: Pool Configuration file for use with a Condor Pool

In the above example, the pool isi_condor has been tagged as a vanilla universe pool.

Alternatively, the user can tag a particular job in the DAX as a vanilla universe job. This is not recommended but the facility exists.

Note even though the jobmanagers are specified for this pool (due to XML schema constraints), they are not used in this mode.

```
<adag xmlns="http://www.griphyn.org/chimera/DAX"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.griphyn.org/chimera/DAX
http://www.griphyn.org/chimera/dax-1.8.xsd" count="1"
index="0" name="blackdiamond">

      …….

     <job id="ID000004" namespace="vds" name="analyze"
version="1.0" level="1" dv-namespace="vds" dv-name="bottom"
dv-version="1.0">

         <argument>-a bottom -T 6  -i <filename
file="f.c1"/> <filename file="f.c2"/>  -o <filename
file="f.d"/></argument>
         <profile namespace="condor"
key="universe">vanilla</profile>

          <uses file="f.c1" link="input"
dontRegister="false"
              dontTransfer="false"/>
           <uses file="f.c2" link="input"
dontRegister="false"
              dontTransfer="false"/>
           <uses file="f.d" link="output"
dontRegister="false"
              dontTransfer="false"/>

     </job>
    …….
</adag>
```

Figure 6: Modified DAX

In the above example, the job with logical name analyze has been tagged with the condor profile, which would result in it being run in the condor universe vanilla on whatever execution pool it is mapped to.

8.4.1 Changes to the Compute Jobs

The tagging of job as vanilla ends up triggering a couple of changes in the submit files for the compute jobs that are done automatically by Pegasus. This section explains what those changes are, and why they are done.

Removal of the following "***globus***" universe key value pairs from the condor submit files.

- globusscheduler

- remote_intialdir

- globusrsl

Addition of –w option to kickstart[4] for the compute jobs

Since, Pegasus is not specifying the remote directory in the submit file, condor runs each job in a unique spool directory on the remote execution pool. However, we need the jobs need to be run in the work directory specified by the user in his configuration files, as that is where all the data required by the job is staged in. Hence, the –w option is passed to kickstart that makes it change into that directory before executing the user executable as specified by the compute job.

Addition of the following condor key value pairs to the submit files

should_transfer_files=YES

when_to_transfer_output=ON_EXIT

8.4.2 Changes to the transfer jobs

The transfer jobs are different from the compute jobs in the sense that they rely on the underlying grid mechanisms (globus-url-copy mainly) to transfer the files in and out of the execution pools. The running of the transfer jobs on the remote pools requires the use of the user proxy that in case of the "***globus***" universe jobs is transported to the remote end by CondorG from the submit host.

In case of condor "***vanilla***" universe jobs, the transfer of proxy is handled by using the Condor file transfer mechanisms. This is done by the user either specifying the path to his user proxy in

[4] The kickstart executable is a light-weight program that is distributed as part of the Virtual Data System (VDS) which connects the stdin, stdout and stderr filehandles for jobs on the remote site. It sits in between the remote scheduler and the executable making it possible to gather additional information about the executable run-time behavior, including the exit status of jobs.

his pool configuration file for the **local** pool (submit host) or using the property **vds.local.env** in the properties file.

The user specifies the X509_USER_PROXY environment variable in the env profile namespace.

```
<config xmlns="http://www.griphyn.org/chimera/GVDS-
PoolConfig"
xsi:schemaLocation="http://www.griphyn.org/chimera/G
VDS http://www.griphyn.org/chimera/gvds-poolcfg-1.3.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" vers
ion="1.3">
……
<pool handle="local"
gridlaunch="/nfs/asd2/vahi/test/chimera/bin/kickstart">
    <lrc url="rls://localhost" />
    <gridftp  url="gsiftp://smarty.isi.edu/cgt-
scratch/gmehta/LOCAL" storage="/cgt-scratch/gmehta/LOCAL"
major="2"
minor="4" patch="0" />
    <jobmanager universe="transfer"
url="localhost/jobmanager-fork" major="2" minor="4"
patch="0" />
    <jobmanager universe="vanilla"
url="localhost/jobmanager-condor" major="2" minor="4"
patch="0" />
    <workdirectory >/nfs/cgt-
scratch/gmehta/LOCAL</workdirectory>
    <profile namespace="env" key="GLOBUS_LOCATION"
>/nfs/v6/globus/GT2/linux/STABLE</profile>
    <profile namespace="env" key="LD_LIBRARY_PATH"
>/nfs/v6/globus/GT2/linux/STABLE/lib</profile>
    <profile namespace="env" key="X509_USER_PROXY"
>/tmp/pegasus/x509.proxy</profile>
      </pool>

…….
</config>
```

Figure 7: Modifications to the Pool Configuration file

The user can optionally edit the property file and specify the value of the environment variable.

```
#GVDS PROPERTY FILE

vds.local.env X509_USER_PROXY=/tmp/pegasus/x509.proxy
```

Figure 8: Modifications to the Properties file

The tagging of job as vanilla ends up triggering the following changes in the submit files for the transfer jobs, that are done automatically by Pegasus.

Removal of the following "***globus***" universe key value pairs from the condor submit files.

- globusscheduler
- remote_intialdir
- globusrsl

Addition of condor key value pair

transfer_input_files= <*path to user proxy on submit node*>

The path to the user proxy is picked up from the pool configuration file or the properties files with the value from the properties file overriding the one in the pool configuration file. This tells Condor to transfer the proxy to the remote spool directory where it is going to run the job.

However, the job needs to know that the proxy is there. This is done by specifying the environment variable X509_USER_PROXY in the environment key in the submit file.

For e.g the submit file would contain

environment=X509_USER_PROXY=x509.proxy

There is no –w option to kickstart generated for transfer jobs as we want the jobs to be executed in the spool directory where condor launches them. This is because there is no means to determine in advance the spool directory that contains the transferred proxy. The input for the transfer jobs (transfer.in file) is staged via stdin and not via data files as is the case for compute jobs.

Addition of the following condor key value pairs to the submit files

should_transfer_files=YES

when_to_transfer_output=ON_EXIT

8.5 Condor GlideIn

As mentioned in 8.2.2, Pegasus can execute workflows over Condor pool. This pool can contains machines managed by a single institution or department and belonging to a single administrative domain. This is the case for most of the Condor pools. In this section we describe how machines from different administrative domains and supercomputing centers can be dynamically added to a Condor pool for certain timeframe. These machines join the Condor pool temporarily and can be used to execute jobs in a non preemptive manner. This functionality is achieved using a Condor feature called Glide-in [5] that uses Globus GRAM interface for migrating machines from

different domains to a Condor pool. The number of machines and the duration for which they are required can be specified.

In this case, we use the abstraction of a local Condor pool to execute the jobs in the workflow over remote resources that have joined the pool for certain timeframe. Details about the use of this feature can be found in the condor manual.

A basic step to migrate in a job to a local condor pool is described below.

```
condor_glidein -count 10 gatekeeper.site.edu/jobmanager-pbs

The above step glides in 10 nodes to the user's local
condor pool, from the remote pbs scheduler running at
gatekeeper.site.edu. By default, the glide in binaries are
installed in the users home directory.

More details can be found in the Condor manual.
```

Figure 9: GlideIn of Remote Globus Resources

It is possible that the Condor pool can contain resources from multiple Grid sites. It is normally the case that the resources from a particular site share the same file system and thus use the same FileSystemDomain attribute while advertising their presence to the Central Manager of the pool. If the user wants to run his jobs on machines from a particular Grid site, he has to specify the FileSystemDomain attribute in the requirements classad expression in the submit files with a value matching the FileSystemDomain of the machines from that site. For example, the user migrates nodes from the NCSA Teragrid cluster (with FileSystemDomain *ncsa.teragrid.org*) into a Condor pool and specifies FileSystemDomain == "*ncsa.teragrid.org*". Condor would then schedule the jobs only on the nodes from the NCSA Teragrid cluster in the local condor pool.

The FileSystemDomain can be specifed for an execution pool in the pool configuration file in condor profile namespace as follows

```
<config xmlns="http://www.griphyn.org/chimera/GVDS-
PoolConfig"
xsi:schemaLocation="http://www.griphyn.org/chimera/G
VDS http://www.griphyn.org/chimera/gvds-poolcfg-1.3.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" vers
ion="1.3">
……

<pool handle="ncsa_tera"
gridlaunch="/opt/VDS/bin/kickstart" >
    <profile namespace="condor"
key="universe">vanilla</profile>
    <profile namespace="condor"
key="FileSystemDomain">ncsa.teragrid.org</profile>
```

```
    <gridftp  url="gsiftp://ncsa.teragrid.org/"
storage="/vds/data/" major="2" minor="4" patch="0" />
    <jobmanager universe="transfer"
url="ncsa.teragrid.org/jobmanager-condor" major="2"
minor="4" patch="0" />
    <jobmanager universe="vanilla"
url="ncsa.teragrid.org/jobmanager-condor" major="2"
minor="4" patch="0" />
    <workdirectory
>/opt/workspace/CONDOR/exec</workdirectory>
</pool>
…….
</config>
```

Figure 10: Specifying FileSystemDomain in Pool Configuration File

Specifying the FileSystemDomain key in condor namespace for a pool, triggers Pegasus into generating the requirements classad [6] expression in the submit file for all the jobs scheduled on that particular pool.

For example, in the above case all jobs scheduled on pool isi_condor would have the following expression in the submit file.

requirements = FileSystemDomain == "ncsa.teragrid.org"

References

[1] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, Miron Livny *"Pegasus : Mapping Scientific Workflows onto the Grid"* Across Grids Conference 2004, Nicosia, Cyprus, 2004

[2] A. Chervenak, E. Deelman and et al. *"Giggle: A Framework for Constructing Scalable Replica Location Services."* Proceedings of Supercomputing 2002 (SC2002), November 2002

[3] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. *"Grid Information Services for Distributed Resource Sharing."* Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[4] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke. Proc. IPPS/SPDP '98 . *"A Resource Management Architecture for Metacomputing Systems."* Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998.

[5] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke. *"Condor-G: A Computation Management Agent for Multi-Institutional Grids."* Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[6] Rajesh Raman, Miron Livny, and Marvin Solomon, *"Matchmaking: Distributed Resource Management for High Throughput Computing"*, *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 28-31, 1998, Chicago, IL

[7] http://gridlab.org/WorkPackages/wp-9/index.html *"GRMS: Gridlab Resource Management System"*

9 Data Transfer Conventions for Pegasus

Revision $Revision: 1.2 $ of file $RCSfile: VDSUG_PegasusDataTransfer.xml,v $.

9.1 Introduction

Pegasus [ref] is a configurable system for mapping and executing abstract application workflows over the Grid. Pegasus maps an abstract workflow to a concrete workflow, by performing a series of workflow annotations on the workflow. One of the annotations, involve adding of transfer nodes to the workflow, that staging in input data required by the jobs to the remote executions pools, and stage out the materialized data to a storage output pool.

Pegasus provides various transfer mechanisms for the user. Most of them eventually end up using globus-url-copy to do the final transfers. However, they differ from each other in the way the structure of the workflow is modified, and the granularity of transfer jobs.

The purpose of this document is to give an overview of the transfer mechanisms in Pegasus, how they are configured and what executables they refer to.

9.2 Transfer Configurations

This section discusses the various transfer configurations available to the user. The user can specify the transfer mode by setting the property **vds.transfer**.

The following transfer modes are available currently in Pegasus.

- Single
- Multiple
- T2
- Chain
- Bundle
- Stork
- GRMS

Each job specified in the abstract workflow (DAX) can be associated with 3 different types of transfer jobs

stage in (transfers the input files required by the compute job to the remote pool from the locations specified in the replica catalog)

interpool (transfers the input files required by the compute job to the remote execution pool, from the locations where the parent jobs materialized those input files)

stage out (transfers the materialized data to an external permanent storage area specified by the user when running Pegasus)

By default, all the transfer configurations employ a pull (stage in and interpool) and push (stage out) mechanism. The transfer jobs are executed on the remote execution pools, where the compute jobs are executed, unless an execution pool is explicitly denoted as a third party pool in the properties file. In that case, all the transfers are run on the submit host, using the third party transfer mechanisms.

The various transfer configurations differ in the manner they construct the various transfer jobs, the underlying grid transfer tools they utilize and how they affect the concrete workflow structure.

9.2.1 Single

In this configuration, a transfer job is added for each file that needs to be transferred to or from a remote execution pool. Each transfer job results in a globus-url-copy invocation on the remote pool. There is no throttling of the transfer jobs being done on the execution pools. For large workflows, this can result in simultaneous invocation of hundreds of globus-url-copy jobs.

```
#GVDS PROPERTY FILE

vds.transfer single
vds.tc File
```

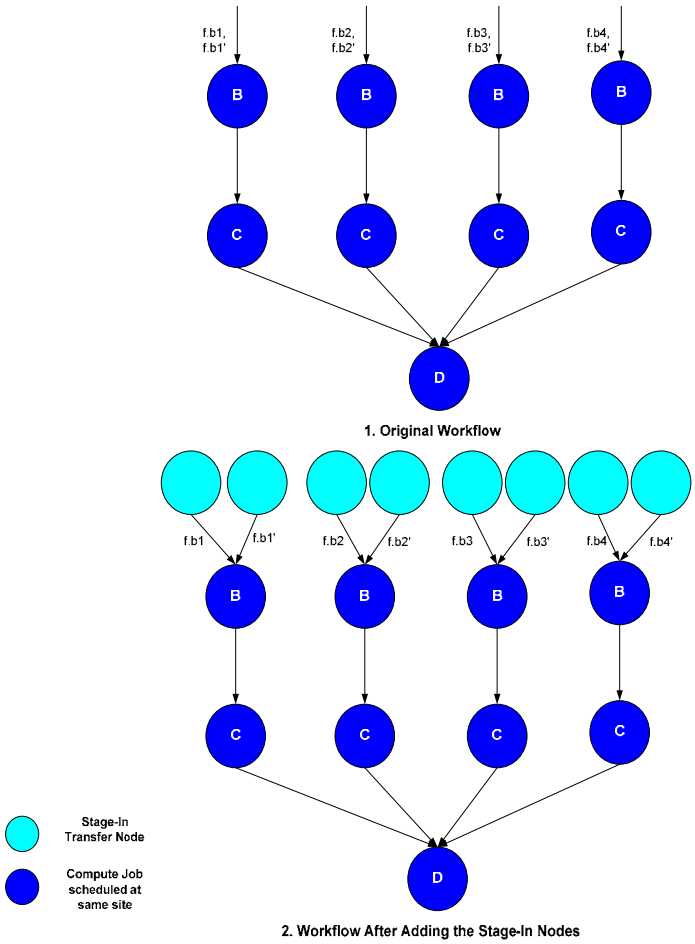


Figure 1: Adding Stage In Transfer nodes via the single transfer mode

```
#Entry in the file transformation catalog
isi_condor   globus-url-copy  /globus/bin/globus-url-copy
INSTALLED INTEL32::LINUX
ENV::GLOBUS_LOCATION=/globus/GT2/linux/STABLE;
```

The above entry should be on a single line, with each column separated by whitespace.

Pegasus looks for logical transformation ***globus-url-copy*** at the various execution pools in the transformation catalog in this mode. The latest statically built version of ***globus-url-copy*** is shipped in the VDS worker package, and can be found at **$VDS_HOME/bin/guc**. $VDS_HOME is the environment variable that points to the VDS installation on a remote pool.

9.2.2 Multiple

In this configuration a transfer job is added for each compute job. Thus, a compute job in this mode can at maximum have one stage in, one stage out and one interpool transfer job created for it. However, the compute job can still depend on more than one stage in jobs that have been created for other compute jobs to prevent ***file clobbering***. This happens in the case when a compute jobs share the same set of input files, and have been scheduled on the same remote execution pool by the site selector.

```
#GVDS PROPERTY FILE
```

```
vds.transfer multiple
vds.tc File
```

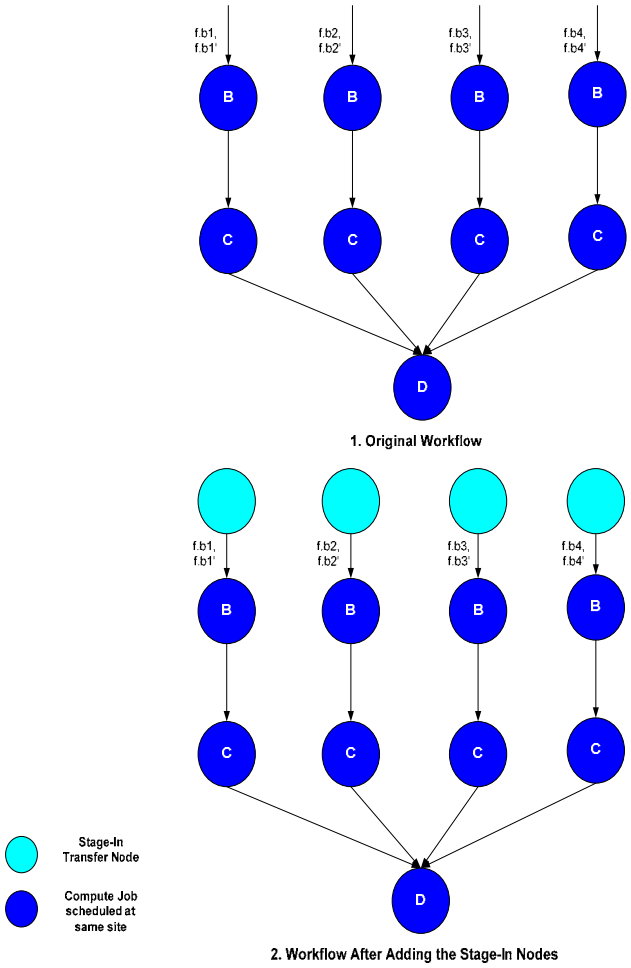


Figure 2: Adding Stage In Transfer nodes via the multiple transfer mode

```
#Entry in the file transformation catalog
isi_condor   transfer  /vds/default/bin/transfer INSTALLED
INTEL32::LINUX
ENV::GLOBUS_LOCATION=/nfs/v6/globus/GT2/linux/STABLE;ENV::V
DS_HOME=/vds/default
```

The above entry should be on a single line, with each column separated by whitespace.

Pegasus looks for logical transformation ***transfer*** at the various execution pools in the transformation catalog in this mode. The latest statically built version of ***transfer*** is shipped in the VDS worker package, and can be found at **$VDS_HOME/bin/transfer**. $VDS_HOME is the environment variable that points to the VDS installation on a remote pool.

The transfer executable is intelligent enough to pick up the more current globus-url-copy amongst the one found at $VDS_HOME/bin/guc and one at $GLOBUS_LOCATION/bin/globus-url-copy.

This mode is better than the single mode in handling large workflows because of the aggregation of transfer of multiple files in one transfer job. However, it still can result in a high load on the remote execution pools, if the site selector ends up mapping a large number of jobs at a particular level of the workflow to same execution pool.

9.2.3 T2

This configuration extends upon the ***multiple*** configuration. It differs from the multiple configuration with respect to the underlying grid transfer tool used to do the actual transfers.

In this configuration, Pegasus may associate multiple source and destination urls with a single transfer. Pegasus ends up generating these multiple urls if

1) there is more than one gridftp server associated with an execution pool. The underlying assumption in this case is that all the grid ftp servers at a pool share the same mount points.

2) there are more than one entry in the replica catalog for the same lfn.

The T2 executable, tries to transfer between the cartesian product of source and destination pairs, e.g. N x M and O x P pairs. Additional flags supplied during the invocation of T2 determine at what point and how to stop. For more information on T2, check the man page at $VDS_HOME/man/man1/T2.1.

By default T2 is invoked with the any option. In this mode, any source file transfer to any destination success determines the success of the section. If all the pair candidates are exhausted without success, the transfer fails. However, missing of a file on any source results in an immediate try for the next source. On the last source, it results in failure.

```
#GVDS PROPERTY FILE

vds.transfer T2
vds.tc File
```

```
#Entry in the file transformation catalog
isi_condor   T2  /vds/default/bin/T2 INSTALLED
INTEL32::LINUX
ENV::GLOBUS_LOCATION=/nfs/v6/globus/GT2/linux/STABLE;ENV::V
DS_HOME=/vds/default
```

The above entry should be on a single line, with each column separated by whitespace.

Pegasus looks for logical transformation ***T2*** at the various execution pools in the transformation catalog in this mode. The latest statically built version of ***T2*** is shipped in the VDS worker package, and can be found at **$VDS_HOME/bin/T2**. $VDS_HOME is the environment variable that points to the VDS installation on a remote pool.

9.2.4 Chain

In this configuration, the stage-in transfer jobs to a particular pool are chained in together sequentially. This ensures that at any moment there is only one stage-in transfer job executing on a remote execution pool. At the moment, there is no parameter to specify the number of chains of transfer jobs that the user wants for a particular pool.

This configuration extends upon the ***multiple*** configuration, leveraging its capability to prevent file clobbering while staging-in data. It differs with respect to the final concrete workflow

structure. Instead of the stage-in jobs on a particular execution pool being parallel (independent to each other), they are sequentially dependant on each other as in a chain.

```
#GVDS PROPERTY FILE

vds.transfer Chain
vds.tc File
```

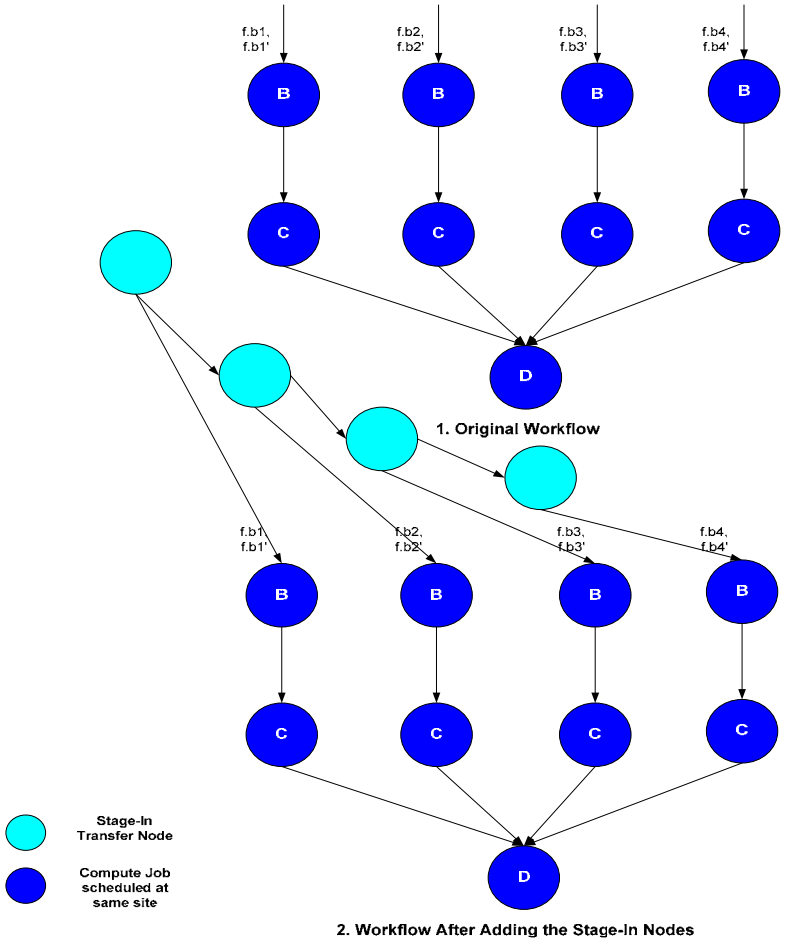


Figure 3: Adding Stage In Transfer nodes via the chain transfer mode

```
#Entry in the file transformation catalog
isi_condor   transfer  /vds/default/bin/transfer INSTALLED
INTEL32::LINUX
ENV::GLOBUS_LOCATION=/nfs/v6/globus/GT2/linux/STABLE;ENV::V
DS_HOME=/vds/default
```

The above entry should be on a single line, with each column separated by whitespace.

Pegasus looks for logical transformation ***transfer*** at the various execution pools in the transformation catalog in this mode. The latest statically built version of ***transfer*** is shipped in the VDS worker package, and can be found at **$VDS_HOME/bin/transfer**. $VDS_HOME is the environment variable that points to the VDS installation on a remote pool.

9.2.5 Bundle

In this configuration, the user can specify the number of bundles of transfer jobs that he wants for staging-in data to a remote execution pool. A bundle consists of one or more parallel jobs that are scheduled on the same execution pool.

The motivation behind this mode, is to leverage the scenarios where the input data is distributed over n number of gridftp servers. This allows the users to run large workflows without overwhelming either the gridftp servers by opening too many simultaneous connections to it or the remote execution pool by invoking too many globus-url-copy calls at the same time.

The user can specify the number of transfer jobs that are generated per execution pool by associating a ***bundle_stagein*** key in the VDS namespace with the transfer executable for a particular pool in the transformation catalog. Alternatively, the user can associate the same key with an execution pool in the pool configuration file. Pegasus uses this parameter as the upper limit on the number of stage in transfer nodes that are created for staging in the raw data to that particular execution pool.

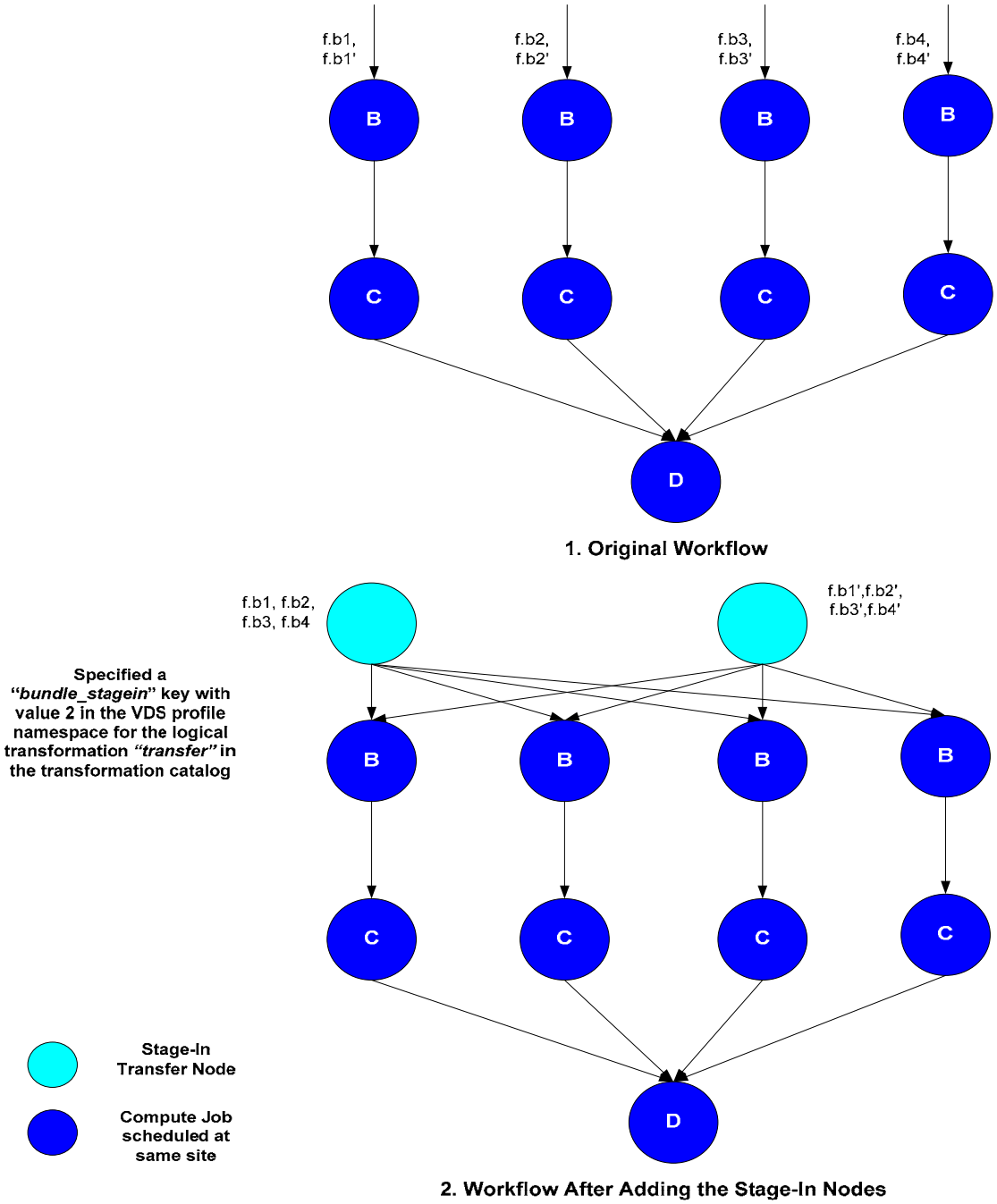


Figure 4: Adding Stage In Transfer nodes via the bundle transfer mode

```
#GVDS PROPERTY FILE

vds.transfer XIO3Bundle
vds.tc File
```

As shown in the figure 4 above, the input files for a single job are actually round robin distributed over all the transfer jobs in that bundle. This ends up creating an artificial synchronization point, that makes all the compute jobs to wait till all the data for the workflow has been staged-in to the

remote execution pools. This constraint due to the way the configuration has been implemented in Pegasus currently.

```
#Entry in the file transformation catalog
isi_condor   transfer  /vds/default/bin/T2 INSTALLED
INTEL32::LINUX
ENV::GLOBUS_LOCATION=/nfs/v6/globus/GT2/linux/STABLE;ENV::V
DS_HOME=/vds/default;VDS::bundle_stagin=2
```

The above entry should be on a single line, with each column separated by whitespace.

Pegasus looks for logical transformation ***transfer*** at the various execution pools in the transformation catalog in this mode. The latest statically built version of ***transfer*** is shipped in the VDS worker package, and can be found at **$VDS_HOME/bin/transfer**. $VDS_HOME is the environment variable that points to the VDS installation on a remote pool.

9.2.6 Stork

In this configuration , Pegasus uses Stork the data placement scheduler from Condor. It extends upon the single transfer configuration, as stork only supports one transfer per stork job. It uses all the features of the single configuration, except that the transfers instead of being specified in a condor submit file format are specified in the stork file format.

```
#GVDS PROPERTY FILE

vds.transfer StorkSingle
vds.tc File
```

9.2.7 GRMS

In this configuration, the user uses the underlying GRMS execution system to execute his workflows. In the GRMS concrete workflow description, the compute jobs are annotated with the transfers for staging-in and staging-out of data. The transfers are then taken care of by the remote GRMS/GridLab services as and when the compute jobs are executed.

```
#GVDS PROPERTY FILE

vds.transfer GRMS
vds.tc       File
vds.submit.writer
```

10 Clustering Jobs with Pegasus

Revision $Revision: 1.2 $ of file $RCSfile: VDSUG_PegasusJobClustering.xml,v $.

10.1 Motivation

A large number of workflows executed through the Virtual Data System, are composed of several jobs that run for only a few seconds or so. The overhead of running any job on the grid is usually 60 seconds or more. Hence, it makes sense to cluster small independent jobs into a larger job. This is done while mapping an abstract workflow to a concrete workflow. Site specific or transformation specific criteria are taken into consideration while collapsing smaller jobs into a larger job in the concrete workflow. The user is allowed to control the granularity of this clustering on a per transformation per site basis.

10.2 Implementation Details

The abstract workflow is mapped onto the various site by the Site Selector. This semi concrete workflow is then passed to the collapsing module. The collapsing of the workflow is done on a per level basis. The levels of the workflow are determined by doing a modified Breadth First Traversal of the workflow starting from the root nodes. The level associated with a node, is the furthest distance of it from the root node instead of it being the shortest distance as in normal BFS.

For each level the jobs are grouped by the site on which they have been scheduled by the Site Selector. Only jobs of same type are merged into a larger job.

10.2.1 Controlling the clustering granularity

The number of jobs that have to be clustered into a single large job, is determined by the value of two parameters associated with the smaller jobs. Both these parameters are specified by the use of a VDS namespace profile keys. The keys can be specified at any of the placeholders for the profiles (abstract transformation in the dax, site in the site catalog, transformation in the transformation catalog). The normal overloading semantics apply i.e. profile in transformation catalog overrides the one in the site catalog and that in turn overrides the one in the DAX. The two parameters are described below.

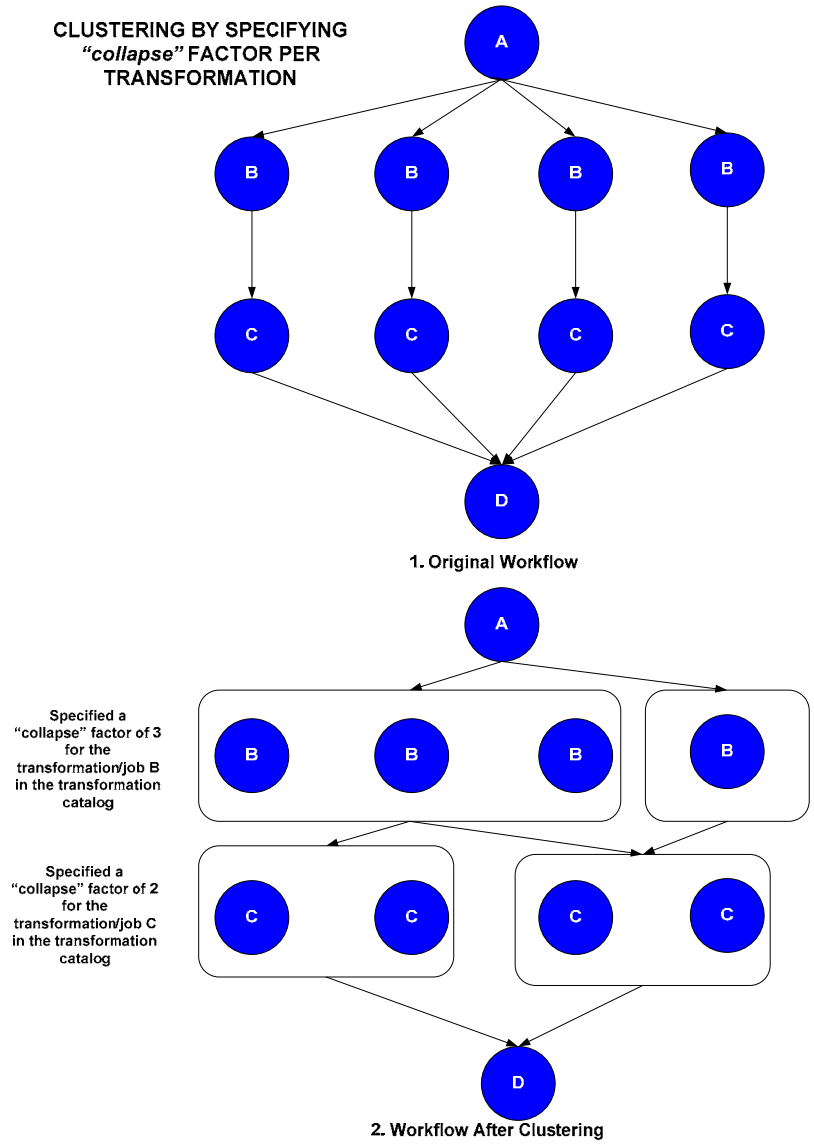
1. **collapse factor**

   The collapse factor denotes how many jobs need to be merged into a single clustered job. It is specified via the use of a VDS namespace profile key "**collapse**". for e.g. if at a particular level, say 4 jobs referring to logical transformation B have been scheduled to a

siteX. The collapse factor associated with job B for siteX is say 3. This will result in 2 clustered jobs, one composed of 3 jobs and another of 2 jobs.

The collapse factor can be specified in the transformation catalog as follows

```
#site   transformation   pfn              type          architecture       profiles
siteX    B     /shared/VDS/bin/jobB INSTALLED       INTEL32::LINUX  VDS::collapse=3
siteX    C     /shared/VDS/bin/jobC INSTALLED       INTEL32::LINUX  VDS::collapse=2
```

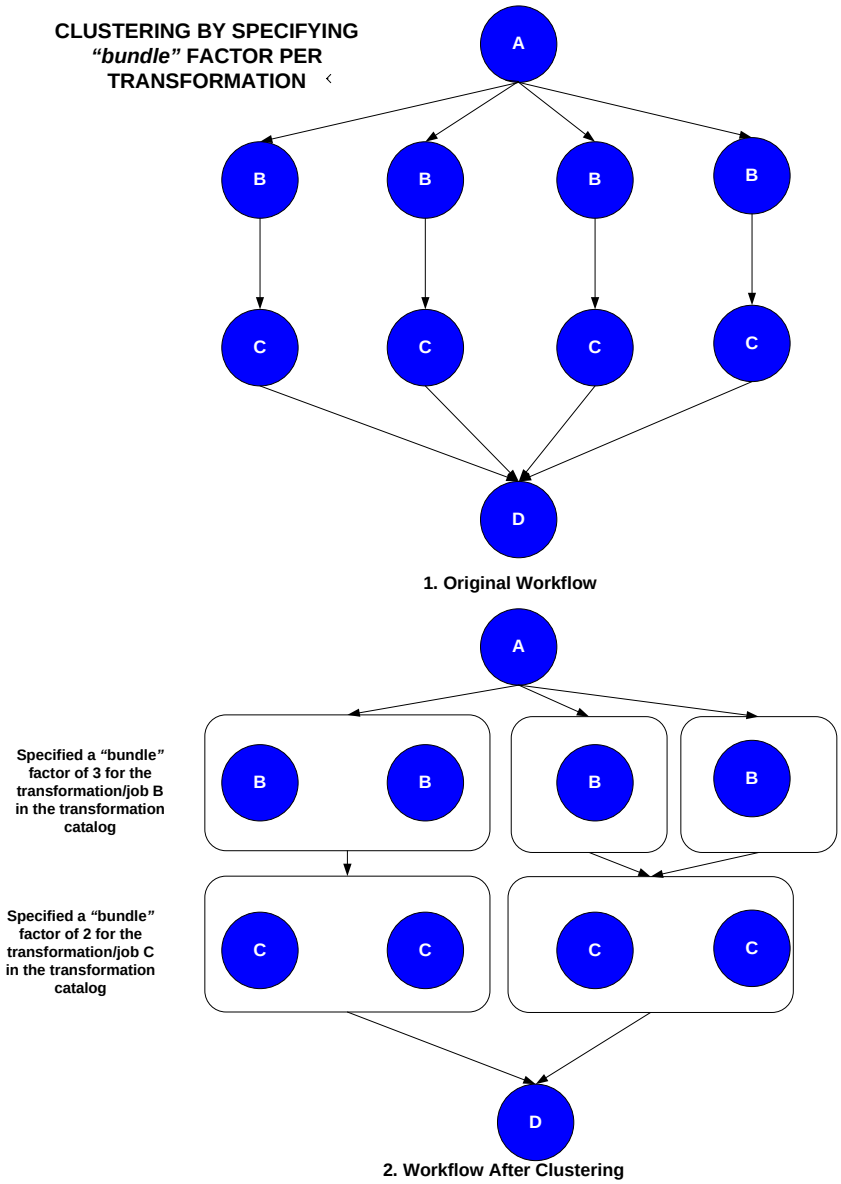


2. bundle factor

The bundle factor denotes how many clustered jobs does the user want to see per level per site. It is specified via the use of a VDS namespace profile key **"bundle**". for e.g. if at a particular level, say 4 jobs referring to logical transformation B have been scheduled to a siteX. The "**bundle"** factor associated with job B for siteX is say 3. This will result in 3 clustered jobs, one composed of 2 jobs and others of a single job each.

The bundle factor in the transformation catalog can be specified as follows

```
#site   transformation   pfn              type          architecture       profiles
siteX    B     /shared/VDS/bin/jobB INSTALLED       INTEL32::LINUX  VDS::bundle=3
siteX    C     /shared/VDS/bin/jobC INSTALLED       INTEL32::LINUX  VDS::bundle=2
```



In the case, where both the factors are associated with the job, the bundle value supersedes the collapse value.

```
#site   transformation   pfn            type          architecture       profiles
siteX    B     /shared/VDS/bin/jobB INSTALLED       INTEL32::LINUX
VDS::collapse=3,bundle=3
```

In the above case the jobs referring to logical transformation B scheduled on siteX will be clustered on the basis of **"bundle"** value. Hence, if there are 4 jobs referring to logical transformation B scheduled to siteX, then 3 clustered jobs will be created.

10.3 Execution of the Clustered Job

The execution of the clustered job on the remote site, involves the execution of the smaller constituent jobs either

- **Sequentially on a single node of the remote site**

  The clustered job is executed using seqexec, a wrapper tool written in C that is distributed as part of the VDS. It takes in the jobs passed to it, and ends up executing them sequentially on a single node.

- **On multiple nodes of the remote site using MPI**

  The clustered job is executed using mpiexec, a wrapper mpi program written in C that is distributed as part of the VDS. It is only distributed as source not as binary. The wrapper ends up being run on every mpi node, with the first one being the master and the rest of the ones as workers. The number of instances of mpiexec that are invoked is equal to the value of the globus rsl key nodecount. The master distributes the smaller constituent jobs to the workers.

  For e.g. If there were 10 jobs in the merged job and nodecount was 5, then one node acts as master, and the 10 jobs are distributed amongst the 4 slaves on demand. The master hands off a job to the slave node as and when it gets free. So initially all the 4 nodes are given a single job each, and then as and when they get done are handed more jobs till all the 10 jobs have been executed.

  Another added advantage of using mpiexec, is that regular non mpi code can be run via MPI.

Both the clustered job and the smaller constituent jobs are invoked via kickstart, unless the clustered job is being run via mpi (mpiexec). Kickstart is unable to launch mpi jobs. If kickstart is not installed on a particular site i.e. the gridlaunch attribute for site is not specified in the site catalog, the jobs are invoked directly.

10.3.1 Specification of the method of execution of clustered job

The method execution of the clustered job(whether to launch via mpiexec or seqexec) can be specified:

1) globally in the properties file

The user can set a property in the properties file that results in all the clustered jobs of the workflow being executed by the same type of executable.

```
#GVDS PROPERTIES FILE
vds.job.aggregator seqexec|mpiexec
```

In the above example, all the clustered jobs on the remote sites are going to be launched via the property value, as long as the property value is not overridden in the site catalog.

2) by associating profile key "collapser" with the site in the site catalog

```
<pool handle="siteX" gridlaunch = "/shared/VDS/bin/kickstart">
    <profile namespace="env" key="GLOBUS_LOCATION" >/home/shared/globus</profile>
    <profile namespace="env" key="LD LIBRARY PATH">/home/shared/globus/lib</profile>
    <profile namespace="vds" key="collapser" >seqexec</profile>
    <lrc url="rls://siteX.edu" />
    <gridftp  url="gsiftp://siteX.edu/" storage="/home/shared/work" major="2" minor="4"
patch="0" />
    <jobmanager universe="transfer" url="siteX.edu/jobmanager-fork" major="2" minor="4"
patch="0" />
    <jobmanager universe="vanilla" url="siteX.edu/jobmanager-condor" major="2"
minor="4" patch="0" />
    <workdirectory >/home/shared/storage</workdirectory>
  </pool>
```

In the above example, all the clustered jobs on a siteX are going to be executed via seqexec, as long as the value is not overridden in the transformation catalog.

3) by associating profile key "collapser" with the transformation that is being clustered, in the transformation catalog.

```
#site   transformation   pfn            type          architecture       profiles
siteX    B     /shared/VDS/bin/jobB INSTALLED       INTEL32::LINUX
VDS::collapse=3,collapser=mpiexec
```

In the above example, all the clustered jobs that consist of transformation B on siteX will be executed via mpiexec.

Note: The clustering of jobs on a site only happens only if

- there exists an entry in the transformation catalog for the clustering executable that has been determined by the above 3 rules
- the number of jobs being clustered on the site are more than 1

10.4 Generating the clustered concrete DAG

The clustering of a workflow, is activated by passing the –collapse|-C option to gencdag. The clustering granularity of a particular logical transformation on a particular site is determined as explained in section 2.1. The executable that is used for running the clustered job on a particular site is determined as explained in section 3.1

```
#Running gencdag to generate clustered workflows
```

```
gencdag --dax example.dax --dir ./dags –collapse –p siteX --output
local --verbose
```

The naming convention of submit files of the clustered jobs is **merge_NAME_IDX.sub** . The NAME is derived from the logical transformation name . The IDX is an integer number between 1 and the total number of jobs in a cluster.

Each of the submit files has a corresponding input file, following the naming convention **merge_NAME_IDX.in** . The input file contains the respective execution targets and the arguments for each of the jobs that make up the clustered job.

11 Conventions for Running Euryale jobs on the Grid

Revision $Revision: 1.2 $ of file $RCSfile: VDSUG_RunningEuryale.xml,v $

The Euryale system is a system to accomplish a simple-sounding but surprisingly complex task: Run workflows in The Grid. While many such tools exist today, Euryale features the following specific characteristics: It uses a deferred planning (late binding) approach, adding fault tolerance through re-planning. Each job is treated in isolation of other jobs, minimizing interference between jobs. Euryale generates Condor-G jobs and interfaces with Condor's DAGMan workflow manager, exploiting particular strengths of DAGMan.

11.1 Introduction

Figure 17 shows a Yourdon Flow Chart with data- and control flows. The key players are the *d2d* DAX to DAG convert, the *prescript* and the *postscript*. The majority of the Euryale logic resides in these three components. Each will be described in a section of its own.

The flow chart shows data storage used by the system between parallel lines. Actors of the Euryale system are shown in rounded boxes, whereas external actors are shown in square boxes. Data flows are shown by solid lines whereas controls flows use dashed lines. Shown in disk shape are VDS catalogs. Elements that were repeated in the diagram are marked with an asterisk. Shown in blue are the higher-level wrappers.

The user usually provides the DAX file as argument to *vds-plan*. Internally, *vds-plan* will call *d2d*. The *d2d* tool reads the DAX file, properties and a submit file template to produce the Condor submit files and DAGMan's DAG file.

In the next step, *vds-run* will submit the workflow do Condor DAGMan. DAGMan will decide which jobs first to run, and start for each ready job, up to a threshold, a PRE script before the job, and a POST script after the job. The heavy lifting happens in the *prescript*, while the *postscript* makes use of data supplied by the *prescript.*

The *prescript* is responsible for the following items:

- It calls out to the external site selector.
- It rewrites the submit file for the chosen site.
- It creates the setup and cleanup script for the job.
- It runs the setup job on the remote site to prepare for the job.
- It creates the lists of input- and output files.
- It transfers necessary input files to that site.
- It deals with re-planning upon failures.

After the pre script ran, DAGMan submits the job, described in the submit file rewritten by the pre script, to Condor. Condor in turn will run the job on the Grid. Once Condor detected that the job finished, regardless of remote failures, the *postscript* is responsible for the following items:

- It starts the remote cleanup job.
- It checks on the successful completion of the remote job.
    - It updates the provenance tracking catalog.
    - On failure it finishes early, signaling failure to DAGMan.
- It transfers output files to the collection area.
- It registers produced files in the replica manager.
- It may update file popularity.

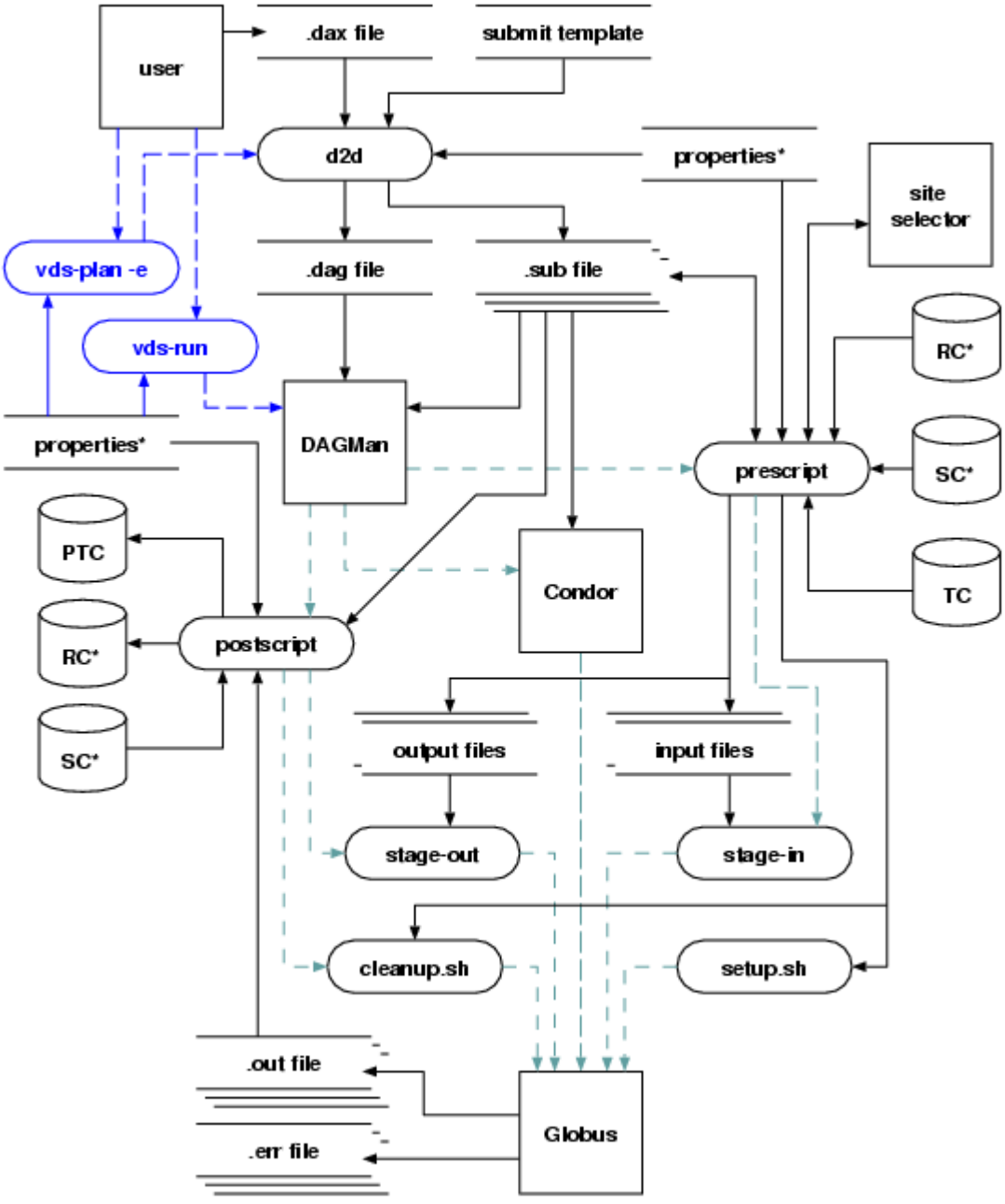


Figure 17: Data flow chart of Euryale.

11.2 Prerequisites, Preparing and Setup

Before attempting to run workflows with Euryale, you may want to check the issues pointed out in this section.

11.2.1 Software Versions

The Euryale run-time system is written entirely in Perl. While we strive to be as compatible as possible, a number of modules are required for a successful setup:

Module	Version	Description
Perl core	5.6.0	Perl run-time system. This includes many modules like Carp, POSIX, File::Spec, Getopt::Long, etc.
DBI	1.38	Perl database independence layer.
DBD::Pg	1.40	If you keep your WF database in PostGreSQL.
DBD::mysql	2.9003	If you keep your WF database in MySQL.
DBD::SQLite2	0.30	If you keep your WF database in SQLite 2.
Digest::MD5	*	MD5 checksum implementation.
File::Temp	0.14	Temporary file implementation.
Time::HiRes	1.51	High-resolution time, sleep and related functions.
XML::Parser	2.49	XML parser abstract base class.
XML::Parser::Expat	2.49	XML parser implementation via *expat.*
URI	1.32	Implementation of uniform resource identifiers.

Table 4: Perl Requirements.

Table 4 shows the Perl modules and versions required to run Euryale successfully. Note to self: This is fresh documentation; I may have missed some modules.

11.2.2 Properties

Euryale is configured through settings in a property file. Euryale is a mixed breed between Java and Perl. The Java portion of the *d2d* reads both property worlds, the VDS properties, and the Euryale *wfrc* properties. The VDS properties reference guide is “properties.pdf”, and the Euryale properties reference guide is “wfrc.pdf”.

This section repeats the introduction from the Euryale property guide, and emphasizes a couple of important properties. The *wfrc* file contains Java-style property settings which control the Euryale planner at run-time. Please note that the values rely on proper capitalization, unless explicitly noted otherwise.

Some properties can be made to rely on other properties. However, such a construction requires the dependent properties to be declared and visible. As a notation, the curly braces refer to the

value of the named property. For instance, *${user.home}* means that the value depends on the value of the *user.home* property plus any noted additions. You can use this notation to refer to other properties, though the extend of the substitutions are limited. Usually, you want to refer to a set of the standard system properties. Nesting is not allowed. Substitutions will only be done once. The following system properties are available:

Property	**Meaning**
file.separator	The string that separates file name components.
java.home	The value of environment variable *JAVA_HOME*, if it exists.
java.class.path	The value of environment variable *CLASSPATH,* if it exists.
java.io.tmpdir	The location of a directory for temporary files.
os.name	The name of the operating system, e.g. Linux.
os.version	The version of the OS kernel, e.g. 2.4.31
os.arch	The operating system architecture, e.g. i386.
user.dir	The current working directory.
user.home	The value of environment variable *HOME*, or the appropriate entry from the account database. Note, Java always uses the account database.
user.language	The value of environment variable *LANG*, if it exists, or “en” otherwise.
user.timezone	The value of environment variable *TZ*, if it exists.
vds.home	The value of environment variable *VDS_HOME*.

Table 5: System properties provided in Perl.

The *wfrc* properties only affect the Euryale planner, and certain portions of the VDS planning subsystem. All properties start with the letter "w", usually either "wf" or "work". The Pegasus planner is not affected by properties declared in the *wfrc* file.

The following example provides a sensible set of properties to be set by the user property file. These properties use mostly non-default settings:

```
wf.profile.vo.group ivdgl1
wf.script.pre     ${vds.home}/contrib/Euryale/prescript.pl
wf.script.post    ${vds.home}/contrib/Euryale/postscript.pl
wf.site.selector  ${vds.home}/contrib/Euryale/mock-const
wf.pool.style     xml
wf.pool.file      ${vds.home}/var/sites.xml
wf.tc.style       new
wf.tc.file        ${vds.home}/var/tc.data
wf.rc.style       LRC
wf.rc.lrc         rls://evitable.uchicago.edu
wf.base.dir       /work
work.db           Pg
```

```
work.db.database  ${user.name}
work.db.user      ${user.name}
work.db.password  ${user.name}
```

The above is an example only, and will not work for you!

Here are more useful configuration properties for Euryale:

```
wf.final.output         sitehandle
wf.final.output         gridftp-URI
```

By default, Euryale lets files stay wherever they are produced. If you specify a final output resource, all files marked as transferable will be copied to the final output site. The specification can either be a simple identifier. In this case, the identifier is assumed to be a site handle, and the site's GridFTP server will be determined from the site catalog. If the value starts with the “gsiftp” prefix, the output is a GridFTP server that is taken verbatim. Nevertheless, any path will be duplicated as necessary.

```
wf.transfer.program     <perlexpr>
```

By default, Euryale will use the program called transfer to grid-copy files between sites. *Transfer* is a simple wrapper around *globus-url-copy*. If the above property is set to anything (that evaluates to true in Perl), the program *T2* will be used instead. *T2* has the advantage that it can try multiple sources for one file, in case a site goes down. It should be considered to provide more robust transfers.

```
wf.permit.hardlinks     <boolean>
```

For each job, a setup script is executed on the fork job manager. This script, when optimizing away transfers, because a required input file already exists at the site, places a link to the file into the job directory. Since the (transient) job directory and the work directory are thought to always (guaranteed) share the same file system, it may be less confusing to some applications, if hard links are employed. By default, if missing or value "false", Euryale works with soft links. Soft links are required, if the site uses AFS. A "true copy" option is not implemented.

```
wf.exitcode.file        <filepath>
```

The VDS is distributed with an application *exitcode* that parses the kickstart record for failures of the remote site. If this option is set, and you should point it to your *$VDS_HOME/bin/exitcode*, Euryale uses the specified program and *exitcode* API to obtain the remote job's success. As a side-effect, *exitcode* will update the provenance tracking catalog (PTC). By default, if undefined, Euryale will use some internal logic to derive the remote job's success. However, *exitcode* is the recommended reference implementation.

```
wf.use.relative         true
```

To overcome limits on the command line length, it is recommended to use relative paths wherever possible on the command line. Some application, like fMRI, may require relative paths, because their computations may store file names within data products, to be referenced by subsequent steps. Only switch this property to false, if there is due cause

```
#wf.remote.job.queues   jazz=SC03
#wf.remote.job.projects jazz=ASDF
```

Some remote scheduling systems implement the notion of queues or projects. In order to enable jobs to run correctly on the site, the appropriate queue and project name can be supplied by

configuration. However, we recommend that you use the profiles of "globus" namespace with the site catalog instead.

The property value is a comma-separated list of key value pairs. The key is a site handle, and the value the queue name.

```
wf.job.retries          5
```

The property counts the number of retries that Euryale attempts before giving up on a job. Raise as needed.

```
wf.site.temp.unlink      true      # erase temp files
#wf.site.temp.unlink     false
#wf.site.temp.suffix      .lof      # default suffix
#wf.site.temp.dir         $TMP      # try /dev/shm on Linux
```

The miscellaneous items deal with temporary directories for temporary files, and the clean-up thereafter. Please turn on unlinking unless you are debugging Euryale call-outs.

11.2.3 Profiles

Profiles are a uniform abstraction of specific run-time environment options. Profiles are propagated in four priority levels through the VDS, please refer to section XXX.

Any property with the prefix *wf.profile* will set a VDS profile. The next component in the property key is the profile namespace. The fourth and final component in the property key is the profile key. Not all profiles are useful in Euryale:

Namespace	Planner	Usage
condor	all	Adds commands to a Condor submit file
dagman	Pegasus	Add configuration options to DAGMan's DAG file.
env	all	Sets remote environment variables.
hints	Pegasus	deprecated!
globus	all	Adds Globus RSL instructions.
selector	Pegasus	Passes information to the site selector.

Table 6: Available Profiles in Euryale.

11.2.4 Local Directory Layout

The workflow monitor *tailstatd* expects a certain directory hierarchy to find workflows and related files. We have found this hierarchy useful:

1. A *base directory* points to the base where the hierarchy starts ("b").
2. The *VO group name* is used as the next directory level ("v").
3. The *DAX label* is the directory level after that ("l").
4. The final directory is a *run directory* which gets automatically numbered ("r").

Together, these directories constitute the secondary key into the workflow database. They are necessary to uniquely identify any workflow and distinguish it from any other.

```
/home/voeckler/work/ivdgl1/400x1x7200/run0004
bbbbbbbbbbbbbbbbbbb vvvvvv lllllllll rrrrrrr
```

Using *vds-plan* will automatically create the right directory hierarchy while maintaining the workflow database.

The property *wf.base.dir* designates the base directory. It defaults to *${user.home}/run* if not specified – though we strongly encourage to specify a sane value.

11.2.5 Remote Site Requirements

Euryale exclusively works with pre-staged executables. The exception is *kickstart*, the remote application starter. Depending on the submit file template, you can chose a pre-staged *kickstart*, or a staged executable. The latter requires that the remote site has an operating system, architecture and (g)libc version compatible with our submit machine. Since this is rarely the case, we recommend using pre-staged *kickstart.*

Euryale's site model expects that the remote site layout shares a file-system between gatekeeper (GK), worker node (WN) and storage element (SE).

Furthermore, the working directory configured for each job is expected to reside on this shared file system. This is the internal view used to generate SFNs. The working directory, if appended to the base GridFTP URI, is the external view. The generation of TFNs uses the external view.

We are researching job and data scheduling models that will support reduced sharing of file systems. However, this is a complex problem.

11.3 Catalogs

Euryale uses the same set of catalogs as Pegasus. However, being Perl, it uses the Perl API to access the catalogs.

There are three catalogs central to the planning process. The site catalog (SC) contains knowledge about the site layout, where directories are, what gatekeeper (GK) and GridFTP server to use. The transformation catalog (TC) contains knowledge about where applications are installed and what extra information they may require to invoke them. Finally, the replica catalog (RC) keeps track of files that were produced.

Properties are passed as set of property values with a common prefix to their respective handling catalog. The *style* property usually determines the implementation. It derives from the Perl module implementing the API for the said catalog.

- The *wf.rc* prefixed properties configure the RC.
- The *wf.pool* prefixed properties configure the SC.
- The *wf.tc* prefixed properties configure the TC.
- The *work* prefixed properties configures the WF.

11.3.1 Replica Manager (RC)

The following values exist for *wf.rc.style:*

Style	State	Meaning

LRC	production	This style designates a local replica catalog from RLS to store replicas.
RLS	prototype	An implementation that uses the RLIs for look-up operations.
DBI	production	A simplistic implementation using a local RDBMS table for replicas.
vds	prototype	An implementation around the *rc-client* tool to use the VDS new RC API.

Each of the style has its own set of configuration properties (some may overlap).

Style	Property	Meaning
LRC RLS	*wf.rc.lrc*	The location of the LRC that will store all replicas. In LRC style, it will also be queried for replicas.
RLS	*wf.rc.rli*	The RLI that is queried for replicas. Defaults to *${wf.rc.lrc}*.
LRC RLS	*wf.rc.java.home*	This property overrides the setting of *$JAVA_HOME* to point to an alternative installation of Java.
LRC RLS	*wf.rc.globus.location*	This property overrides the setting of *$GLOBUS_LOCATION.* For most interactions, the *globus-rls-cli* tool is used.
LRC RLS	*wf.rc.vds.home*	The property overrides the setting of *$VDS_HOME.* Some interactions are handled by VDS's *rls-client* tool.
LRC RLS	*wf.rc.grc*	This property overrides the location of the Globus tool *globus-rls-cli*. Usually, there is no need to set this property.
LRC RLS	*wf.rc.rls.client*	This property overrides the location of the VDS tool *rls-client.* Usually, there is no need to set this property.
LRC RLS	*wf.rc.pool*	Do not use!
DBI	*wf.rc.uri*	The contact URI used by DBI to access a RDBMS. This is usually some string like "dbi:Pg:dbname=ourdb".
DBI	*wf.rc.dbuser*	The RDBMS account name.
DBI	*wf.rc.dbpass*	The RDBMS account password.

For the simplistic DBI replica manager, you only need to create one table in your favorite RDBMS. The VDS database setup will **not** do this for you:

```
create table RC_MAP (
  LFN VARCHAR(255),
  PFN VARCHAR(255),
  UNIQUE(LFN,PFN)
);
create index RC_MAX_IDX on RC_MAP(LFN);
```

11.3.2 Site Catalog (SC)

The following values exist for *wf.pool.style:*

Style	State	Meaning
old	production	The VDS < 1.2 multi-column site catalog file (deprecated).
new	production	The VDS > 1.1 multi-line site catalog file.
xml	production	The VDS > 1.1 site catalog as XML file (preferred).

The property *wf.pool.file* refers to the location of the file that contains the site catalog.

11.3.3 Transformation Catalog (TC)

The following values exist for *wf.tc.style:*

Style	State	Meaning
old	production	The VDS < 1.3 multi-column transformation catalog file.
new	production	The VDS > 1.2 multi-column transformation catalog file.
vds	prototype	An implementation accessing the RDBMS TC implementation via DBI.

In case of a file-based TC, the property *wf.tc.file* refers to the location of the TC file. In case of the RDBMS implementation, the regular VDS properties are parsed for the access data to the RDBMS[5].

11.3.4 Workflow Catalog (WF)

The workflow catalog does not come in different flavors. However its RDBMS access data has to be specified completely to enable Perl's DBI module to access the workflow database.

```
work.db                 Pg
work.db.host            my.database.host
work.db.database        ${user.name}
work.db.user            ${user.name}
work.db.password        XXXXXXX
```

The *work.db* property contains the base name of the Perl DBD module that implements a database. The other properties contain various access data to the RDBMS.

Additionally, the *wf.base.dir* property has an indirect influence onto the rows in the workflow database, as explained in section 11.2.4.

11.4 Site Selection

The site to run a compute job is selected dynamically at the point of time a compute job becomes run-able. The dynamic site selector is coded via a call-out to an external program. To talk to the site selector, a temporary file is generated, which contains simple key-value pairs:

[5] They may be overwritten in the constructor.

```
version=2.0
transformation=voeckler::generate:1.0
derivation=voeckler::d43.anon000001:1.0
job.id='ID000175'
wf.manager=dagman
wf.ujid=voeckler&&big-0&&20050128210617Z&&ID000175&&22193
wf.walltime=11
wf.diskspace=1
vo.name=ivdgl
vo.group=ivdgl1
recent.bad=PDSF
resource.id=FNAL_CMS3 gsiftp://cmssrv10.fnal.gov/
resource.id=OUHEP gsiftp://ouhep0.nhn.ou.edu/
resource.id=uofc gsiftp://e.cs.uchicago.edu/
resource.id=UWMadison gsiftp://cmsgrid.hep.wisc.edu/
```

The different keys strive to describe the job as best as possible to the site selector, and pass along a number of variables. Two multi-keys exist, the *resource.id* occurs once for each GridFTP server of a site (yes, repeating the site handle if necessary), and *input.lfn* lists the required input files as LFNs for a job. Note that *resource.id* only lists the GridFTP server hostname, and potentially a port, but no path component.

The prescript creates the above data in a temporary file. It executes the site selector application in a sub-process, passing the name of the temporary file as sole argument. The site selector application is specified in the workflow property *wf.site.selector*. The site selector in turn prints a string "SOLUTION:*sitehandle*" on its *stdout* filedescriptor. The chosen site should be a site handle from the *resource.id* section. It must be a site handle that is known to the site catalog.

Key	**Meaning**
version	The version number of the protocol. Must be 2.
transformation	Logical name of the VDL transformation that represents the job.
derivation	VDL derivation name that was chosen for the job.
job.id	Unique ID for the job within the workflow.
wf.manager	Name of the workflow manager, currently "dagman".
wf.name	The label given to the DAX file.
wf.stamp	The modification time of the DAX file.
wf.ujid	The unique job identifier, which is unique across all workflows
wf.walltime	The wall time requirements presented by the job.
wf.diskspace	The disk space requirements of the job.
vo.name	The name of the Virtual Organization the job belongs to.
vo.group	A group within the Virtual Organization (arbitrary).

resource.id	Lists one site candidates (multi-option).
recent.bad	Last bad site, if re-planning.
input.lfn	Lists one input LFN required by the job (multi-option).

Table 7: Typical keys exchanged with the site selector.

If a job is re-planned due to remote failure, bad sites are cached separately for each compute job, and omitted from the list of sites passed to the site selector. If no valid compute sites can be found, the prescript exits with an error.

The beauty of the external site selector is that it can be coded in any programming language that runs natively in the host environment. We recommend a scripting language with a light-weight interpreter[6], or a compiled language.

A demo site selector, which randomly chooses from the given site set, is included in the distribution. More elaborate schemes allow the integration of file popularity and monitoring feedback. If you intend to implement your own site selector in Perl, please refer to module Site::Selector, which ships with VDS.

The information contained in the temporary communication file between Euryale logic and external site selector may be subject to extension in the future. A site selector must not stumble over, or warn about unknown keys in the exchanged file.

11.5 Popularity Management

You probably don't need to worry about the popularity manager.

Each compute job that ran to a successful completion will update the popularity manager for input files. To talk to the dynamically popularity manager, a temporary file is generated, which contains just the list of logical input filenames. Note that this file may be empty.

The *postscript* invokes the popularity manager with the site and the file of files as sole command line arguments. It is not expected to return anything. The default popularity manager is a no-operation. More elaborate popularity managers can update the popularity of a file for an advanced site selector, and thus asynchronously push hot-spot files to likely candidates. Or they can just keep statistics.

11.6 Planning Euryale

There are two levels of conversion of a DAX file into the workflow DAG file and related submit files. The low-level tool *d2d* contains the conversion logic. At a higher level, the tool *vds-plan* wraps admittedly messy *d2d*, and employs configuration files for sensible defaults while maintaining a database of workflows.

The DAX file contains the abstract workflow description. The generation of DAX files is described elsewhere, and handled by the abstract planner. The DAX file, if generated from a VDC, is a build-style file containing knowledge how to build each intermediary product. However, the depth of the build can be limited. The DAX may also contain a forest in form of a disconnected graph, if multiple data products are requested simultaneously.

[6] Albeit heavy-weight, Java classes execute natively with Linux, if the *binfmt* kernel trappings are appropriately configured, see Linux kernel documentation.

D2d bridges between the VDS abstract level router and the Euryale concrete planner. While VDS is a pure Java system, Euryale is a Perl system. The *d2d* tool, as adaptor between the worlds, is controlled by two property files: The VDS properties, documented in section XXX, and the workflow property file, described in section YYY. Both property files are Java style.

The basic layout of each submit file is based on substitutions from a submit file template. The template contains a number of placeholders for different purposes. Through the template approach, the workflow can be easily adapted to different site requirements.

The template contains two kinds of place-holders (variables), using a distinct mark-up for each.

1. The **@@variable@@** notation is substituted by *d2d* itself. This point in the process is nick-named *compile-time.* Some variables may be filled with their final value, some with an empty string, and some with the **!!variable!!** variables for a 2nd round of deferred substitutions.
2. The **!!variable!!** place holders are substituted during *run-time* by the *prescript* logic. This is described in section 11.7.

Variable	**Substitution**
ARGS	This variable represents the job's command-line arguments. It may substitute to an empty value.
CONFIG	Experimental, don't use. It translates into the name of a *k.2* configuration specification.
LOGFILE	Name of the Condor common user log file. This file is shared by all Condor jobs. It should reside on a local file system.
DAGFILE	Is the name of the DAG file.
DAXLABEL	This is the value, in XPath notation, of adag@label from the abstract workflow. It is an arbitrary identifier for the workflow. We suggest to use a descriptive label.
DAXMTIME	This is the last modification time of the DAX file.
DV	This is the fully-qualified definition identifier of a VDL Derivation.
JOBID	This is the job identifier from the adag/job@id attribute. Each id is unique to a workflow, but may occur in other workflows.
LEVEL	This is the depth of the job in the breadth first search tree. The value is taken from the DAX attribute adag/job@level.
MAXPEND	This is the value from the workflow property *wf.max.pending.* It represents the maximum number of seconds a job is willing to wait in remote pending state. It defaults to 2 hours. A minimum of 10 minutes is enforced.
STDIN	This option translates into a later substituted LFN, possibly empty. It derives from the DAX element adag/job/stdin.

STDERR	This option translates into a later substituted LFN, possibly empty. It derives from the DAX element adag/job/stderr.
STDOUT	This option translates into a later substituted LFN, possibly empty. It derives from the DAX element adag/job/stdout.
SUBMIT	The name of the submit file itself.
SUBBASE	This is the name of the submit file without the ".sub" suffix.
TEMPLATE	This is the name of the file used as submit file template.
TR	This is the fully-qualified definition identifier of a VDL Transformation.
VERSION	The version of *d2d* that created the file sets.

Table 1: Substitutions for @@variable@@

Table 1 shows the identifiers recognized by *d2d*, and substitutions during the early replacement phase. However, a user rarely needs to concern himself with the low-level *d2d* tool.

11.6.1 Creating A Workflow

The *vds-plan* tool is a comfortable wrapper around *d2d*. It sanity checks a lot of the user's environment before committing a workflow. It also manages a directory sub-tree to place workflows into. If the workflow properties in *$HOME/.wfrc* are correctly set up, only the DAX file needs to be passed to *vds-plan*.

```
$ vds-plan --euryale blackdiamond.dax
2005.09.16 14:38:27.829 CDT: [default] starting
2005.09.16 14:38:29.139 CDT: [default] using 0 directory levels
2005.09.16 14:38:29.246 CDT: [default] created 4 structured filenames.
2005.09.16 14:38:29.246 CDT: [default] created 1 flat filenames.

I have concretized your abstract workflow. The workflow has been
entered into the workflow database with a state of "planned". The next
step is to start or execute your workflow. The invocation required is

vds-run  /work/ivdgl1/black-diamond/run0031
```

Vds-plan will tell you how to invoke *vds-run* to actually start the workflow.

11.7 Running Euryale

The generated DAG with its submit files needs to be executed by DAGMan. DAGMan will in turn invoke the *prescript*, which calls out the site selector[7]. The site selection logic selects one from the number of available sites.

The original set of site candidates is determined by the sites present in the site catalog (SC). The candidate set of sites is further limited by those sites in the transformation catalog (TC), with

[7] here are several ones available. The interface is file-driven and simple, see section [ref here].

regards to the computation to be run. The site selector choses a site from a given list of candidates depending on a variety of strategies.

The simplest strategy is constant, always choosing the same site. This is a good selector for specifically testing out software due to the remote site guarantee. Usually you will chose a site that you have easy access to. Also very simple, albeit not very efficient, are random-based strategies. Random strategies work well for trying out things.

Fancier algorithms, which take the presence of data at the destination site into account, or policies of remote sites, are presented separately by Kavitha and Catalin. We research a class of simple observation-based site selectors. However, most advanced site selectors will require knowledge about intentions of each instance, requiring adequate protection from race conditions and other concurrency conflicts. These are discussed separately.

For any selected site, the submit file is finalized by substituting the **!!variable!!** place holders. For the substitution, all VDS profiles are combined, with the VDL profiles at lowest priority, the PC profiles next, the TC profiles above that, and potential user profiles to override any lower priorities. Detailed overwrite, merge, or replace handling is not implemented - a higher priority profile always overwrites a lower priority value.

Input files are transferred always using submit-host-initiated 3rd party transfers. If the transfers fail for any reason, re-planning takes place, marking the failed site as "bad" for the job, and only the current job. Propagating the knowledge between the independently planned jobs is researched, but so far showed no promise, and interferes with smarter site selectors. Should the *prescript* run out of sites to run, a fatal error is generated, which will terminate the workflow.

If the prescript ran successfully, DAGMan will start the compute job via Condor-G. Once the compute job finished, DAGMan will run the post script, which parses the kickstart record, and transfers output files, if a central collection place is configured. Usually files are kept on the remote site. Having a central collection to put a replica robustifies workflows. Should the site go down which stored the initial output, the replica in the central collector is still known.

If the post script determines that a transfer failed, or the job failed, the postscript will fail, and DAGMan will attempt to run the job again, up to the number of user-configurable retries, starting with the prescript. The postscript also tags the failed site as "bad" to keep the site selector from choosing it again when re-planning.

Variable	**Substitution**
WORKDIR	This value is substituted with the working directory at the remote site. It is taken from the pool/workdir element of the site catalog.
JOBDIR	Each job runs in its own job directory. This value represents the job directory underneath the WORKDIR that a job will run in.
SITE	This value is the site handle from the site catalog. It uniquely identifies the site in the VDS catalogs.
UJID	This is a unique job identifier which is unique for the workflows, and with a very high probability even across multiple workflows. However, the identifier remains keeps its value, if a job is re-planned.

WALLTIME	Each job must declare an upper boundary on its computation time. The wall time is also required for the job monitor to find misbehaving jobs and sites.
DISKSPACE	This is the upper boundary of require disk space. Currently unused, but propagated.
KICKSTART	This is the location of the remote kickstart application. It is taken from the site catalog's pool@gridstart attribute.
TRANSFORMATION	Will be substituted with the logical transformation name. The name may contain colons, like any fully-qualified definition identifier.
APPLICATION	Will be substituted with the remote application's location. Euryale only works with out-of-band pre-staged executables.
GLOBUSSCHEDULER	This is replaced with the chosen job manager contact string for the remote site. It derives from the pool/jobmanager@url attribute of the site catalog.
STDIO	Should a job declare that it wants to track its own stdio, this string will convert into the appropriate kickstart invocation arguments.
GLOBUSRSL	This value will be created from "globus" namespace profiles.
CONDOR_GLOBUSRSL	This is similar to the GLOBUSRSL. However, if present, it will generate the Globus RSL with a prefix of "globusrsl = ". If no RSL values were found, nothing, not even an empty line, is generated.
CONDOR_ADDON	This value contains additional settings from the "condor" namespace profiles.
ENVIRONMENT	This value will extract remote environment settings from the "env" namespace profiles.
LFN:xxx	This special key substitutes each LFN xxx with its SFN.
ENV:xxx	This special key substitutes the environment value of xxx from the submit host's local environment.

Table 9: Substitutions for !!variable!!

11.7.1 Leaf Input Files

Before you attempt to actually run a workflow, you must consider leaf input files. These are files that are required to exist at some site, and their replicas be known to a replica manager.

Figure 18: Illustrating Leaf Input Files.

Figure 18 shows a simple workflow. In this workflow, LFN1 and LFN3 are leaf input files. No productions exist to generate them. Thus, their replica locations must be registered with a replica manager before a workflow is started. Otherwise, the workflow cannot succeed.

If you want to check manually for input files, extract all adag/filename elements. Those with an attribute "link" of "input" designate leaf input files for the given workflows. The "lfn" attribute shows you the LFN that must be known to your replica manager. For VDS-generated DAX files, you can use *grep*:

```
$ grep '  <filename ' blackdiamond.dax | grep 'link="input"'
  <filename file="username.f.a" link="input"/>
```

The *vds-verify* tool will check, among other things, that all leaf input files are known to your configured replica manager. It is more convenient, because it will check your replica manager for your for the existence of all necessary files. It will complain, if something is missing:

```
$ vds-verify --euryale blackdiamond.dax

All is well that ends well.
```

Running *vds-verify* for very large workflows can be expensive, as the XML representation of the workflow is kept in main memory.

11.7.2 Submitting a Workflow

There is one prime rule for Euryale workflows:

Never use *condor_submit_dag* with Euryale workflows!

While it may appear to work, depending on your workflow, you may overtax the resources on your submit system. VDS ships with the *vds-submit-dag* wrapper which sets suitable throttles for DAGMan. While the throttles default to numbers we found reasonable, a user can configure them.

The output from *vds-plan* shown in section 11.6.1 shows how to invoke *vds-run*. The *vds-run* tool itself is a wrapper around *vds-submit-dag*. Additionally, it manages workflow state, and starts a workflow monitoring daemon to watch over your workflow. The workflow monitor *tailstatd* kills misbehaving jobs while simultaneously maintaining the state of each job in the workflow database. It will also update the workflow database with the DAGMan's exit code, once it detects that DAGMan is gone.

```
$ vds-run  /work/ivdgl1/black-diamond/run0031
# parsing properties in .wfrc...
# parsing properties in .vdsrc...

Checking all your submit files for log file names.
This might take a while...
Done.
-----------------------------------------------------------------------
File for submitting this DAG to Condor  : /work/ivdgl1/black-diamond/run0031/black-diamond-0.dag.condor.sub
```

```
Log of DAGMan debugging messages       : /work/ivdgl1/black-diamond/run0031/black-diamond-0.dag.dagman.out
Log of Condor library debug messages    : /work/ivdgl1/black-diamond/run0031/black-diamond-0.dag.lib.out
Log of the life of condor_dagman itself : / work/ivdgl1/black-diamond/run0031/black-diamond-0.dag.dagman.log
Condor Log file for all jobs of this DAG: /tmp/black-diamond-50999.log
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster 26.
------------------------------------------------------------------------
# parsing properties in .wfrc...
# slurped /work/ivdgl1/black-diamond/run0031/braindump.txt

I have started your workflow, committed it to DAGMan, and updated its
state in the work database. A separate daemon was started to collect
information about the progress of the workflow. The job state will soon
be visible. Your workflow runs in base directory

cd /work/ivdgl1/black-diamond/run0031
```

For now, the output will still contain some amount of debug message. However, in the end, it tells you which directory to change into, if you want to watch the workflow closely. That base directory will contain the high-level files pertaining to your workflow.

11.8 Debugging Euryale

Pre- and postscript protocol verbosely what they are doing in a per job file with suffix ".dbg". Each jobs also logs certain information of global interest into a common, shared file "euryale.log". DAGMan maintains its own debug file. Condor tracks job state in the common user log file. The workflow monitor *tailstatd* logs into file "tailstatd.log", and also appends job state transitions into "jobstate.log".

11.8.1 The Job Debug File

Each job maintains a debug file in the same directory where the submit file resides. It uses the same base name, but a suffix ".dbg". Usually, the job debug files, albeit proliferous, contain the best clues why a job fails.

The structure of the job debug file is fixed in the first few columns to enable good parsing:

1. It starts with an ISO 8601 compliant local time stamp with millisecond extension. The time zone is omitted, though.
2. The next column contains the bracketed process ID of the writer. This is either the pre- or post script which is currently active.
3. The 3rd column contains a tag "PRE" to indicate a *prescript,* and "POST" to indicate a *postscript*. No other scripts should write to this debug file.
4. After follows a free format message. Some messages may be prefixed with Perl module identifiers or recognizable strings to indicate the urgency of the message. However, there is no coherent format to the rest of the line.

An example:

```
20050916T154049.561 [32539] PRE: starting /home/voeckler/vds/contrib/Euryale/prescript.pl
[1.74]
20050916T154049.562 [32539] PRE: open submit file ID000001.sub
20050916T154049.562 [32539] PRE: sub >> profile globus.jobtype=single
…
20050916T154049.565 [32539] PRE: sub << profile globus.jobtype=single
20050916T154049.565 [32539] PRE: sub << profile vds.diskspace=1
```

```
20050916T154049.566 [32539] PRE: sub << profile vds.walltime=2
20050916T154049.566 [32539] PRE: done with submit file ID000001.sub
20050916T154049.566 [32539] PRE: created new UJID voeckler&&black-diamond-
0&&20050824200700Z&&ID000001&&32539
20050916T154049.567 [32539] PRE: creating workflow config handle from
/home/voeckler/.wfrc
20050916T154049.570 [32539] PRE: creating site catalog handle
20050916T154049.586 [32539] PRE: found property style
20050916T154049.587 [32539] PRE: found property file
20050916T154049.681 [32539] PRE: reading from /home/voeckler/work/pool.gt2
20050916T154049.902 [32539] PRE: loaded GriPhyN::SC::xml [1.3]
20050916T154049.902 [32539] PRE: creating transformation catalog handle
…
```

11.8.2 The Common Euryale Log File

All *prescript* and *postscript* write important message to a shared file "euryale.log". This file resides in the same directory where the DAG file resides. Since the access is shared between concurrent instances, only few important message are written to this file. If a workflow fails, the first clue is usually found here.

The log file contains several columns:

1. The first column contains a simple 3-character tag. The purpose of this tag is to enable quick searches using *grep.* Some of these tags include:
   - "prs", "prf" to indicate *prescript* start and finish
   - "pos", "pof" to indicate *postscript* start and finish.
   - "sum" shows the remote exit information.
   - "die" marks the reason for a fatal failure.
2. contains an ISO 8601 compliant local time stamp with millisecond extension. The time zone is omitted, though
3. The next column contains the bracketed process ID of the writer. This is either the pre- or post script which is currently active.
4. The 4th column contains the job identifier. Each identifier is only unique within the workflow it belongs to.
5. The next column contains a tag "PRE" to indicate a *prescript*, and "POST" to indicate a *postscript*. No other scripts should write to this debug file.
6. After follows a free format message. Some messages may be prefixed with Perl module identifiers or recognizable strings to indicate the urgency of the message. However, there is no coherent format to the rest of the line.

The common log can be used to compute things like compute time, number of transfers, etc. One such file is associated with each workflow. For reasons of performance, the information in this file is few and restricted. While it may contain some clues about reasons for failure, one should always check the failing job's debug file for specifics. An example:

```
prs 20050916T154049.562 [32539] ID000001 PRE: started [1.74 1.37]
run 20050916T154050.535 [32539] ID000001 PRE: projected for terminable
sus 20050916T154054.294 [32539] ID000001 PRE: ran
terminable.uchicago.edu:2120/jobmanager-fork -s ID000001.setup.sh in 3.759 s
si1 20050916T154055.729 [32539] ID000001 PRE: transferred 1 input file(s) in 1.435 s
prf 20050916T154055.735 [32539] ID000001 PRE: finished after 6.173 s
```

11.8.3 The DAGMan Debug File

Sometimes, rarely but not impossible, DAGMan may get confused by the messages it detects. If the Euryale-related log files do not yield a conclusive reasons why a workflow failed, the DAGMan debug file may. It usually resides in a file with the same name as the DAG file plus the suffix ".dagman.out".

```
…
9/16 15:40:59 Event: ULOG SUBMIT for Condor Job ID000001 (27.0)
9/16 15:40:59 Of 4 nodes total:
9/16 15:40:59  Done     Pre   Queued    Post   Ready   Un-Ready   Failed
9/16 15:40:59   ===     ===      ===     ===     ===        ===      ===
9/16 15:40:59     0       0        1       0       0          3        0
9/16 15:41:09 Event: ULOG GLOBUS SUBMIT for Condor Job ID000001 (27.0)
9/16 15:41:09 Event: ULOG JOB HELD for Condor Job ID000001 (27.0)
9/16 15:41:44 Event: ULOG JOB RELEASED for Condor Job ID000001 (27.0)
9/16 15:46:14 Event: ULOG_GLOBUS_SUBMIT for Condor Job ID000001 (27.0)
9/16 15:46:14 Event: ULOG_JOB_HELD for Condor Job ID000001 (27.0)
9/16 15:46:44 Event: ULOG JOB RELEASED for Condor Job ID000001 (27.0)
9/16 15:47:48 Received SIGUSR1
9/16 15:47:48 Aborting DAG...
9/16 15:47:48 Writing Rescue DAG to /work/ivdgl1/black-diamond/run0031/black-diamond-
0.dag.rescue...
9/16 15:47:48 Removing submitted jobs...
9/16 15:47:48 Removing any/all submitted Condor/Stork jobs...
9/16 15:47:48 Executing: condor rm -const 'DAGManJobID == "26"'
9/16 15:47:48 Running: condor rm -const 'DAGManJobID == "26"'
9/16 15:47:48 WARNING: failure: condor rm -const 'DAGManJobID == "26"'
9/16 15:47:48   (pclose() returned 256)
9/16 15:47:48 Error removing DAGMan jobs
9/16 15:47:48 **** condor_scheduniv_exec.26.0 (condor_DAGMAN) EXITING WITH STATUS 2
```

The lines containing the HELD and RELEASED show that something fishy was going with this workflow. The SIGUSR1 indicates that the user chose to terminate DAGMan using *condor_rm.* The final exit code on the last line is what the workflow monitor *tailstatd* will record as status of the workflow before exiting itself.

Here is another DAGMan debug file, indicating an actual problem with DAGMan:

```
9/15 18:24:31 Max PRE scripts (20) already running; deferring PRE script of Job ID000135
9/15 18:24:41 Event: ULOG JOB TERMINATED for Condor Job ID000092 (13323.0)
9/15 18:24:41 EVENT ERROR: job 13323.0.0 ended; total end count != 1 (2)
9/15 18:24:41 WARNING: bad event here may indicate a serious bug in Condor -- beware!
9/15 18:24:41 Aborting DAG because of bad event (EVENT ERROR: job 13323.0.0 ended; total
end count != 1 (2))
9/15 18:24:41 Aborting DAG...
9/15 18:24:41 Writing Rescue DAG to test-0.dag.rescue...
9/15 18:24:41 Removing submitted jobs...
9/15 18:24:41 Removing any/all submitted Condor/Stork jobs...
9/15 18:24:41 Executing: condor_rm -const 'DAGManJobID == "13190"'
9/15 18:24:41 Running: condor rm -const 'DAGManJobID == "13190"'
9/15 18:24:41 Removing running scripts...
9/15 18:24:41 **** condor_scheduniv_exec.13190.0 (condor_DAGMAN) EXITING WITH STATUS 1
```

This DAGMan stumbled over duplicated "job terminated" events in Condor's common user log file. DAGMan can be told to ignore such event duplications, which may happen due to race conditions.

11.8.4 Condor's Common User Log

Condor writes its own log file, with one paragraph entry for each entry. If you use Perl to parse the records, you must set your input record separator $/ to "\n…\n".

Euryale puts the log file into a local temporary file system. The workflow monitor, upon start-up, places a symbolic link to the temporary file system file into the directory where the DAG file resides. It replaces the symbolic link with the true file once the workflow has finished.

```
000 (027.000.000) 09/16 15:40:59 Job submitted from host: <128.135.152.241:62374>
    DAG Node: ID000001
    pool:terminable
...
017 (027.000.000) 09/16 15:41:08 Job submitted to Globus
    RM-Contact: terminable.uchicago.edu:2120/jobmanager-condor
    JM-Contact: https://terminable.uchicago.edu:49163/11045/1126903263/
    Can-Restart-JM: 1
...
012 (027.000.000) 09/16 15:41:09 Job was held.
        Globus error 43: the job manager failed to stage the executable
        Code 2 Subcode 43
...
013 (027.000.000) 09/16 15:41:40 Job was released.
        "The PeriodicRelease expression '(NumSystemHolds <= 3)' evaluated to TRUE"
...
017 (027.000.000) 09/16 15:46:13 Job submitted to Globus
    RM-Contact: terminable.uchicago.edu:2120/jobmanager-condor
    JM-Contact: https://terminable.uchicago.edu:49163/11133/1126903568/
    Can-Restart-JM: 1
...
012 (027.000.000) 09/16 15:46:14 Job was held.
        Globus error 43: the job manager failed to stage the executable
        Code 2 Subcode 43
...
013 (027.000.000) 09/16 15:46:40 Job was released.
        "The PeriodicRelease expression '(NumSystemHolds <= 3)' evaluated to TRUE"
...
009 (027.000.000) 09/16 15:47:48 Job was aborted by the user.
        via condor_rm (by user voeckler)
...
```

The above log file excerpt contains Globus errors 43. The submit file for job ID000001 contained instructions to transfer the executable from the submit machine to the remote host, but the executable didn't exist where indicated. Condor retried the submission a 2nd time, still without success. Eventually, the user decided to terminate the workflow.

11.8.5 The Workflow Monitor Debug File

The workflow monitor *tailstatd* reports its own progress and failures into a file "tailstatd.log". The file is a simple format, started by an ISO 8601 local time stamp without time zone but millisecond extension. The next column contains a bracket line number from DAGMan's debug file that the workflow monitor daemon parses. Finally, the rest of the line constitutes the log message.

```
20050916T154050.343 [0]: starting [1.17], using pid 32541
20050916T154050.344 [0]: env: LD_LIBRARY_PATH=/opt/globus/latest/lib
20050916T154050.344 [0]: env: LD_LIBRARY_PATH+=/opt/pgsql/lib
20050916T154050.344 [0]: env: LD LIBRARY PATH+=/opt/condor/default/lib
20050916T154050.344 [0]: env: LD LIBRARY PATH+=/vdt/netlogger/lib
20050916T154050.344 [0]: env: LD LIBRARY PATH+=/opt/graphviz/lib/graphviz
20050916T154050.345 [0]: env: LD LIBRARY PATH+=/opt/gcc/3.3.4/lib
20050916T154050.345 [0]: env: GLOBUS_TCP_PORT_RANGE=61440,65535
20050916T154050.345 [0]: env: GLOBUS_TCP_SOURCE_RANGE=61440,65535
20050916T154050.345 [0]: env: GLOBUS LOCATION=/opt/globus/latest
20050916T154050.346 [3]: Using DAGMan version 6.7.10
20050916T154050.346 [5]: DAGMan runs at pid 32538
20050916T154050.351 [37]: Condor writes its logfile to /tmp/black-diamond-50999.log
20050916T154050.352 [37]: Trying to create local symlink to common log
20050916T154050.352 [37]: symlink /tmp/black-diamond-50999.log -> /work/ivdgl1/black-
diamond/run0031/black-diamond-0.log
20050916T154050.614 [54]: processed chunk of 3170 byte
```

```
20050916T154059.615 [55]: processed chunk of 65 byte
20050916T154100.615 [64]: processed chunk of 641 byte
20050916T154114.614 [64]: job ID000001 requests 120 s walltime
20050916T154114.614 [64]: job ID000001 is planned for site terminable
20050916T154114.668 [64]: inserted new site terminable
20050916T154114.676 [66]: processed chunk of 137 byte
20050916T154148.675 [67]: processed chunk of 70 byte
20050916T154607.673 [67]: Start: checking for starvation
20050916T154607.673 [67]: Done: checking for starvation
20050916T154637.674 [67]: job ID000001 requests 120 s walltime
20050916T154637.674 [67]: job ID000001 is planned for site terminable
20050916T154637.687 [69]: processed chunk of 137 byte
20050916T154646.684 [70]: processed chunk of 70 byte
20050916T154750.685 [80]: DAGMan finished with exit code 2
```

11.8.6 The Job State Transition Log

While the workflow monitor *tailstatd* maintains the current state of each job in the workflow database, the transition of job states is logged into the "jobstate.log" file. It resides in the same directory the DAG file is located in:

```
1126903248 INTERNAL *** TAILSTATD_STARTED ***
1126903248 ID000001 PRE_SCRIPT_STARTED - - -
1126903255 ID000001 PRE_SCRIPT_SUCCESS - - -
1126903259 ID000001 SUBMIT 27.0 - -
1126903269 ID000001 GLOBUS_SUBMIT 27.0 terminable 120
1126903269 ID000001 JOB_HELD 27.0 terminable 120
1126903304 ID000001 JOB_RELEASED 27.0 terminable 120
1126903574 ID000001 GLOBUS_SUBMIT 27.0 terminable 120
1126903574 ID000001 JOB_HELD 27.0 terminable 120
1126903604 ID000001 JOB_RELEASED 27.0 terminable 120
1126903670 INTERNAL *** TAILSTATD_FINISHED 2 ***
```

The file has at least three columns, but no more than 6 for job entries:

1. The 1st column is a UTC time stamp, seconds since epoch.
2. The 2nd column provides the job id within the workflow. The special identifier "INTERNAL" protocols event messages from the workflow monitor.
3. The 3rd column for jobs is the symbolic job state. The job states are mostly taken from Condor.
4. If available, the Condor ID of the job.
5. If available, the site handle where the job was intended to run.
6. If available, the wall time in seconds.

12 Interpreting Results from Kickstart

Revision $Revision: 1.4 $ of file $RCSfile: VDSUG_Kickstart.xml,v $.

This document shows how to interpret the results common found in the kickstart result record. This document assumes that you are familiar with some basic taxonomy of XML documents. You need to know what an element is and how to find attributes. You need some elementary grasp of UNIX programming concepts like the relation of *errno* to failed system calls, or how to invoke man pages to find out about system calls like the *wait* family.

12.1 An Example

To facilitate an example, please invoke kickstart on the command-line as follows:

```
kickstart -n unix::date -N isodate -R test /bin/date -Isec > date.out
```

The above will capture the output from kickstart into the file `date.out`. If your environment is correctly set up, you should not have seen any errors.

When viewing the example, several of the options passed to kickstart can be found in the resulting *provenance tracking record (PTR)* as the output is frequently is called. It is useful to extend the viewing terminal or editor beyond 80 columns for proper viewing.

The PTR is XML pretty-printed, with proper indentations to make it more legible to humans. The different sections in the PTR are somewhat repetitive and depend on how kickstart was invoked, and with what options.

The current description of the schema can be found at the GriPhyN web site:

http://www.griphyn.org/workspace/VDS/

This document describes the provenance tracking record XML schema in version 1.5:

http://www.griphyn.org/workspace/VDS/iv-1.5/iv-1.5.html

A local copy is shipped with the VDS distribution. It can be found in the `$VDS_HOME/doc/schemas` directory.

12.2 Preamble

The file starts with the XML preamble that designated the document as XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

This preamble is common do most properly formatted XML documents.

12.3 The *invocation* Root Element

The root element is **invocation**. Its initial set of attributes deals with stating that the present document is of a certain XML schema:

```
<invocation
 xmlns=http://www.griphyn.org/chimera/Invocation
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:schemaLocation=http://www.griphyn.org/chimera/Invocation
  http://www.griphyn.org/chimera/iv-1.5.xsd
 version="1.5" start="2005-08-30T10:40:30.160-05:00" duration="0.021"
 transformation="unix::date" derivation="iso" resource="test"
 hostaddr="128.135.152.241" hostname="griodine.uchicago.edu"
 pid="18554" uid="500" user="voeckler" gid="500" group="voeckler">
```

- **xmlns** associates an XML namespace identifier for all un-prefixed elements and attributes in the present document.
- **xmlns:xsi** associates an XML namespace identifier for all elements and attributes in the present document which are prefixed with **xsi**.
- **xsi:schemaLocation** associates the XML namespace identifier with a location of an XML schema document that describes elements falling into the XML namespace. For your convenience, the location pointed to is at the GriPhyN web site.[8]

The next set of attributes belongs to the PTR.

- **version** defines the version of this document instance.
- **start** is the timestamp when kickstart was started. Note that XML uses an ISO 8601 compliant timestamp format. Also, a time stamp is only as accurate as the clock of the worker node it is derived from.
- **duration** is the is the execution wall clock time of kickstart. It is in seconds with a millisecond resolution. To determine the finish time of kickstart, you can add the *duration* to the *start*. Since many UNIX systems still have a clock resolution of 10 ms, only two digits after the period may be significant.
- **transformation** defines the VDL transformation name for the current job. The value for this attribute is directly copied from the -n command-line argument. If the -n option is not present when kickstart is invoked, this attribute will not be generated.

  In our example case, the value is "unix::date", as passed on the command-line. The concrete planner is expected to set this option.
- **derivation** defines the VDL derivation name for the current job. The value for this attribute is directly copied from the value of the -N command-line option. If the -N option is not present when kickstart was invoked, this attribute will not be generated.

  In our example case, the value is "isodate", as passed on the command-line. The concrete planner is expected to set this option.
- **resource** marks the site handle where the present job was executed. The value for this attribute derives from the -R command-line option. If the -R option is not present when kickstart is invoked, this attribute will not be generated.

  In our example, the value is "test", as passed on the command-line. The concrete planner is expected to set this option.

[8] The VDS tools use a local copy shipped with VDS when processing XML to avoid network overhead.

- **hostaddr** is the numeric address of the primary interface of the host where kickstart was executed. Its format is the dotted quad.
- **hostname** is the symbolic version of *hostaddr*. However, should the name resolution fail for some reason, this attribute will not be generated.
- **pid** is the UNIX process id of the kickstart process itself.
- **uid** is the effective user id under which kickstart is executed.
- **user** is the symbolic translation of *uid.* Should the translation fail for some reason, this attribute will not be generated.
- **gid** is the effective group id under which kickstart runs.
- **group** is the symbolic translation of *gid*. Should the translation fail for some reason, this attribute will not be generated.

12.4 The job elements of *invocation*

Kickstart knows about five different jobs:

- **setup** describes the independent set-up helper job.
- **prejob** describes any helper job which is executed prior to the main job.
- **mainjob** describes the main compute job.
- **postjob** describes any helper job which is executed after the main job.
- **cleanup** describes the independent clean-up helper job.

The jobs have the following relationship and dependencies in shell pseudo notation:

```
setup ; prejob && mainjob && postjob; cleanup
```

It means that, if a *setup* job is present, it will be executed first. Regardless of its success, the chain of pre-, main- and postjob is executed next. If a pre job is present, it will be executed. If it fails, the chain ends there. If the pre job succeeds, or is not present, the main job will be executed next. If the main job fails, the chain ends. If a post script is present, it will be executed upon successful execution of the main job. Independently of the chain, if a cleanup job is present, it will be executed after the chain.

Due to the complexity of the execution path, you will usually see at least one resulting job element. However, there is no implied warrantee that you will always see a main job element in the PTR. In our current example, we will see just the main job:

```
<mainjob
 start="2005-08-30T10:40:30.161-05:00" duration="0.019" pid="18555">
  <usage utime="0.000" stime="0.000" minflt="14" majflt="133"
   nswap="0" nsignals="0" nvcsw="0" nivcsw="0"/>
  <status raw="0"><regular exitcode="0"/></status>
  <statcall error="0">
    <file name="/bin/date">7F454C460101010000000000000000</file>
    <statinfo mode="0100755" size="38588" inode="4251559"
     nlink="1" blksize="4096"
     mtime="2003-10-29T08:44:15-06:00"
     atime="2005-08-30T10:40:00-05:00"
     ctime="2004-06-28T16:26:44-05:00"
     uid="0" user="root" gid="0" group="root"/>
    </statcall>
    <arguments>-Isec</arguments>
</mainjob>
```

All the above jobs have an identical layout. Each job has the following set of attributes, similar to kickstart's own record:

- **start** is the wall time when the specific job was started. Note that XML uses an ISO 8601 compliant timestamp format.
- **duration** is the is the execution wall clock time of kickstart. It is in seconds with a millisecond resolution. To determine the finish time of the job, add the *duration* to *start*.
- **pid** is the UNIX process id under which the job was started.

Each job is comprised of the following four child elements:

- **usage** is an excerpt of the UNIX `struct rusage` record, which recorded the resource usage of the job. Because Linux only records a fraction of the potential, not all fields are passed back.
- **status** records the job exit results. It is most pertinent when trying to figure out the success of any job operation.
- **statcall** records the `struct stat` record about the application that was invoked.
- **commandline** records the argument vector of the job.

12.4.1 The *usage* element for jobs

The usage element capture the resource consumption of the specific job. Unfortunately, Linux is less than forthcoming with number, and thus only a subset of numbers are propagated in the PTR:

```
<usage utime="0.000" stime="0.000" minflt="14" majflt="133"
 nswap="0" nsignals="0" nvcsw="0" nivcsw="0"/>
```

Please refer to the UNIX manual page for the *getrusage* function to obtain what the attributes mean on your system. Generally the following interpretation is acceptable:

- **utime** is the time in seconds with millisecond resolution spent in user-land.
- **stime** is the time in seconds with millisecond resolution spent in kernel space.
- **minflt** is the number of page faults not requiring physical I/O.
- **majflt** is the number of page faults resulting in physical I/O.
- **nswap** is the number of swap operation, always 0 for Linux.
- **nsignals** is the number of signals received, always 0 for Linux.
- **nvcsw** is the number of voluntary context switches.
- **nivcsw** is the number of involuntary context switches (pre-emption).

12.4.2 The status element

The status element of any job exhibits the clues as to what happened to a job, e.g. if it started at all, how it finished, and whether it received a signal.

```
<status raw="0"><regular exitcode="0"/></status>
```

The **raw** attribute of the *status* element capture the raw exit code as obtained from the UNIX wait call family.

- If it is negative, some failure to start the application was perceived. Kickstart itself customarily generates a negative value, if it fails to start the application. The application itself is usually not started in this case.

- If it is 0, the application terminated successfully.
- If any the lower 7 bits are set, the application terminated on the signal number pertaining to the bit pattern.
- If the bit#7 is set, the application produced a UNIX core file upon exit.
- The upper eight bits constitute the exit code with which the application exited.

This distinction leads to one of the four possible child elements inside the *status* element:

- **failure** signals the inability to even start the application, which relates to the first case above. Kickstart itself detects this family of errors. The only attribute **error** contains the most recent value of the UNIX libc *errno* variable when the error was detected. The element's content pertains to the error condition.
- **regular** constitutes an application that ran until it finished by itself. Its **exitcode** attribute of the element captures the exit code value from the upper 8 bits in the raw exit code. A value of 0 means that the application returns an exit code of 0, etc.
- **signalled** marks the abnormal termination of an application by a signal. The content (value) of the XML element pertains to the symbolic cause. It features the following attributes:
    - **signal** records the signal number that cause the abnormal termination.
    - **corefile** is a Boolean value to indicate that a core file was generated. This attribute is optional, as not all systems may produce core file.
- **suspended** is a state that should never be visible here. Please contact *vds-support*, if you ever notice it in your kickstart output.

12.4.3 The statcall element

The *statcall* element of a job records i-node information about the application that was started. Its only attribute **error** records the result of the UNIX stat call on the executable's file name.

The *error* attribute is useful for debugging. If not 0, it provides UNIX's reason why it failed to start the application, e.g. the executable was not found where it was expected.

```
<statcall error="0">
  <file name="/bin/date">7F454C4601010100000000000000000</file>
  <statinfo mode="0100755" size="38588" inode="4251559"
   nlink="1" blksize="4096"
   mtime="2003-10-29T08:44:15-06:00"
   atime="2005-08-30T10:40:00-05:00"
   ctime="2004-06-28T16:26:44-05:00"
   uid="0" user="root" gid="0" group="root"/>
</statcall>
```

The *statcall* element as sub-element of a job is different from the root element's record, see section 12.6. The inner *statcall* record usually consists itself of two sub-elements:

- The **file** element records the name of the application in its **name** attribute. The content contains the hexadecimal coded first 16 (or less) byte of the application, if it was successfully run.
- The **statinfo** record is only present, if the stat call succeeded. Its attributes reflect a subset of the UNIX `struct stat` record:
    - **size** records the file size in byte. It is the only mandatory attribute.

- **mode** records the file permissions in the UNIX-typical octal notation. This attribute helps to distinguish, if a file was executable or not a file at all.
- **inode** records the i-node number of the application. Please note that the i-node number is specific to the file system the file resides upon.
- **nlink** records the hard link count of the file.
- **blksize** records the block size of the underlying file system in bytes.
- **atime** records the access time of the file as ISO 8601 timestamp. This is effectively the moment in time the stat call was executed.
- **mtime** records the last time the file's data was modified as ISO 8601 timestamp.
- **ctime** records the last time the file's status was modified as ISO 8601 timestamp.
- **uid** records the file's owner as UNIX user id.
- **user** records the symbolic translation of **uid** according to the executing host. If the translation fails, the attribute is not generated.
- **gid** records the file's group as UNIX group id.
- **group** records the symbolic translation of **gid** according to the executing host. If the translation fails, the attribute is not generated.

12.4.4 The arguments element

The **arguments** element records the argument vector as it is passed to the application. The content of the element is a space-separated list of the arguments passed to the application.

```
<arguments>-Isec</arguments>
```

Note to self: For some reason, the XML schema claims that the element name is **command-line**. This is wrong and requires fixing. For some reason, the XML parser never produced the necessary warnings.

12.5 The other elements of *invocation*

The *invocation* element feature a couple more sub elements to describe the remote execution. As you will notice, some elements are recycled from the job elements.

12.5.1 The *cwd* element

cwd records the current working directory in which kickstart was launched. This working directory is the same for all jobs.

```
<cwd>/home/voeckler</cwd>
```

12.5.2 The uname element

uname tries to capture information about the architecture and OS where kickstart was launched. This information is useful when debugging. It features the following attributes:

- **system** records the down-cased operating system name, see "uname -s".
- **archmode** is an artificial flag to record the architecture. Its values are currently hard-coded into kickstart, and derive from the compiler settings. Future versions may want to substitute the hardware platform or processor information.
- **nodename** is the name of the execution host as set by its system administrator. This may be just a hostname, or a fully qualified domain-name see "uname -n".

- **release** is the version of the operating system, see "uname -r".
- **machine** is the machine's hardware name, see "uname -m".
- **domainname** is optional for certain GNU systems. It records the domain-name of the executing host. Its presence depends on how kickstart was compiled.

The content of the *uname* element depends on the operating system, see "uname -v".

```
<uname
 system="linux"
 archmode="IA32"
 nodename="griodine.uchicago.edu"
 release="2.4.31"
 machine="i686">#1 SMP Thu Jun 9 12:55:56 CDT 2005</uname>
```

12.5.3 Kickstart's own *usage* record

usage is an excerpt of the UNIX `struct rusage` record, which recorded the resource usage of kickstart itself, but not its children. Because Linux only records a fraction of the potential, not all fields are passed back. The attributes of the *usage* record are discussed in section 12.4.1.

```
<usage utime="0.000" stime="0.000" minflt="35" majflt="115"
 nswap="0" nsignals="0" nvcsw="0" nivcsw="0"/>
```

12.6 The trailing *statcall* records

statcall records may appear multiple times as direct child of the *invocation* element. These record `struct stat` information of various kickstart-related file, as well as from files kickstart was explicitly asked about with the -s and -S options.

The format is slightly richer than the *statcall* records described in section 12.4.3. The attributes of the *statcall* element at this position needs to record which entity it pertains to. Thus, the set of attributes include:

- **error** records the result of the UNIX stat call on the executable's file name. The *error* attribute is useful for debugging. If not 0, it provides UNIX's reason why it failed to start the application, e.g. the executable was not found where it was expected.
- **id** record the entity the stat was invoked upon. This is a small set of identifiers with the following possible values:
    - *gridstart* pertains to kickstart itself. Kickstart tries to obtain its own stat information by attempting to stating argv[0].
    - *stdin*, *stdout* and *stderr* refer to the system default file descriptors pre-connected to standard input, standard output and standard error respectively. A remote scheduling system may chose to connect this to unexpected things.
    - *logfile* refers to the stat information about the logging file, if the -l option was used with kickstart. Otherwise, it pertains to kickstart's own *stdout* filedescriptor.
    - *channel* refers to the feedback FIFO kickstart creates to funnel data from kickstart-aware applications to the submit host.
    - *initial* and *final* refer to the stat records specifically asked for by the -S and -s option respectively. These stat records will also set the **lfn** attribute.
- **lfn** is only used in conjunction with the initial and final ids. The attribute records the logical filename for the physical file of which the stat information is recorded.

12.6.1 The file-related elements

The first and mandatory element in kickstart's own *statcall* elements are sub-elements describing a file or a file-like entity:

- The **file** element records the name of the application in its **name** attribute. The content contains the hexadecimal coded first 16 (or less) byte of the application, if it was successfully run.
- The **descriptor** element records a pre-opened file descriptor. Since UNIX does not provide a user-space and portable way to map a file descriptor to a filename, there is no file name. The attribute **number** records the descriptor number.
- The **temporary** element is used for temporary files which kickstart creates and destroys. Attributes record both, the file's **name** and **descriptor**.
- The **fifo** element pertains to named pipes. It records the file's **name** and **descriptor**, but also the number and extent of messages that passed through the FIFO.

12.6.2 The *statinfo* record

Please refer to section 12.4.3 for details on the *statinfo* record.

12.6.3 The *data* section

If the file was a temporary file and there was data captured in the temporary file, the first 64 pages of content are copied into the data contents. A page in Linux has 4kB, thus the first 256kB are usually reported. The example below will illustrate the details.

12.7 Finishing the example

There are a number of trailing *statcall* records towards the end of our example.

```
<statcall error="2" id="gridstart">
  <file name="kickstart"/>
</statcall>
```

This first record from the example refers by the *id* attribute to kickstart itself. It shows that kickstart was unable to record stat information for itself. The error number 2 means ENOENT in UNIX, and refers to the error message "file not found". This is correct, as a file called "kickstart" is unlikely in your current working directory. However, the shell which invoked kickstart found it in its PATH.

```
<statcall error="0" id="stdin">
  <file name="/dev/null"/>
  <statinfo mode="020666" size="0" inode="66938"
   nlink="1" blksize="4096"
   mtime="2003-01-30T04:24:37-06:00"
   atime="2003-01-30T04:24:37-06:00"
   ctime="2004-06-21T13:10:43-05:00"
   uid="0" user="root" gid="0" group="root"/>
</statcall>
```

Unless the concrete planner tells kickstart otherwise, the *stdin* file descriptor is typically connected to the `/dev/null` device. Please note how the mode shows that the file is a device and not a regular file.

```
<statcall error="0" id="stdout">
  <temporary name="/tmp/gs.out.G9acZ2" descriptor="3"/>
```

```
  <statinfo mode="0100600" size="25" inode="229165"
   nlink="1" blksize="4096"
   mtime="2005-08-30T10:40:30-05:00"
   atime="2005-08-30T10:40:30-05:00"
   ctime="2005-08-30T10:40:30-05:00"
   uid="500" user="voeckler" gid="500" group="voeckler"/>
  <data>2005-08-30T10:40:30-0500
</data>
</statcall>
```

Unless the concrete planner tells kickstart otherwise, the standard file descriptors *stdout* and *stderr* are connected to temporary files by kickstart. These temporary files capture the output an application may produce, but are removed when kickstart finishes. Note that with very verbose applications, you may flood a system's `/tmp` directory.

In the above case for *stdout*, kickstart captured contents produced on *stdout* into a file `/tmp/gs.out.G9acZ2`. The temporary file was also accessible with file-descriptor 3. Please note the new element **temporary**.

Additionally, for debugging purposes only, the first 64 pages (a page has 4kb on Linux) of content from the file are returned as content of the **data** element. In our case, we see the output from the date application that we ran. The result includes the trailing line-feed that date produces.[9]

```
<statcall error="0" id="stderr">
  <temporary name="/tmp/gs.err.gQtdyL" descriptor="4"/>
  <statinfo mode="0100600" size="0" inode="229446"
   nlink="1" blksize="4096"
   mtime="2005-08-30T10:40:30-05:00"
   atime="2005-08-30T10:40:30-05:00"
   ctime="2005-08-30T10:40:30-05:00"
   uid="500" user="voeckler" gid="500" group="voeckler"/>
</statcall>
```

The data section is only available with kickstart's temporary files, and only if data was actually produced. It is useful for debugging. For instance, if an application fails unexpectedly, it is often helpful to see what it printed on *stderr*. Since our date example did not fail, it did not print anything onto its *stderr*.

Both, Globus and remote scheduling systems, typically connect stdio to */dev/null*. Thus, kickstart facilitates grid debugging by providing access to otherwise lost data. However, the data element was not, and is never meant to pass production data from the remote site to the submit host. Please use the application feedback FIFO for such a purpose.

```
<statcall error="0" id="logfile">
  <descriptor number="1"/>
  <statinfo mode="0100644" size="0" inode="3434245"
   nlink="1" blksize="4096"
   mtime="2005-08-30T10:40:30-05:00"
   atime="2005-08-30T10:40:30-05:00"
   ctime="2005-08-30T10:40:30-05:00"
   uid="500" user="voeckler" gid="500" group="voeckler"/>
</statcall>
```

[9] Future versions of kickstart may encapsulate the content into an XML CDATA section.

The log file pertains to kickstart's own *stdout* filedescriptor, which we captured into the file *date.out* with shell redirections. Kickstart itself only knows the file descriptor, thus the **descriptor** element. Its only attribute is the file descriptor number.

```
<statcall error="0" id="channel">
  <fifo name="/tmp/gs.app.Mwci7t" descriptor="5"
   count="0" rsize="0" wsize="0"/>
  <statinfo mode="010640" size="0" inode="229531"
   nlink="1" blksize="4096"
   mtime="2005-08-30T10:40:30-05:00"
   atime="2005-08-30T10:40:30-05:00"
   ctime="2005-08-30T10:40:30-05:00"
   uid="500" user="voeckler" gid="500" group="voeckler"/>
</statcall>
```

The final *statcall* element in our example refers to the kickstart-aware application feedback channel. The channel is implemented as a UNIX FIFO (named pipe). The name of the FIFO is passed into the environment of all jobs. Whenever a jobs writes anything to the FIFO, it will be passed to the submit host on Globus's *stderr* channel. This way, a kickstart-aware application can communicate with the submit host. However, please note the best-effort transport of the data, which may not arrive on the submit host until the job has finished.

The **fifo** element contains several additional attributes besides the file's name and descriptor number. It records how many messages were received, how many bytes were read, and how many bytes in XML-encoded messages were sent.

13 Troubleshooting Guide

Revision $Revision: 1.2 $ of file $RCSfile: VDSUG_Troubleshooting.xml,v $.

As with any piece of complex software, there can be bugs or other unexpected behavior. Depending on the networking environment below also will introduce certain imponderables. All these issues combined will lead to behavior that is not quite the expected. Some problems may be of transient nature, some may be due to various mis-configurations, and some will be true bugs. This section helps you to help us troubleshoot causes and issues.

You are dealing with a multi-layered complex system. For debugging purposes it is easier, to work from the bottom up, and by-and-by add complexity. Often, even when a user suspects that the VDS is broken, in fact, some of the supporting fabric is experiencing problems.

13.1 Introduction to the Software Layers

It is important to understand the various software layers that have to interact in order to run a grid job through the Virtual Data System.

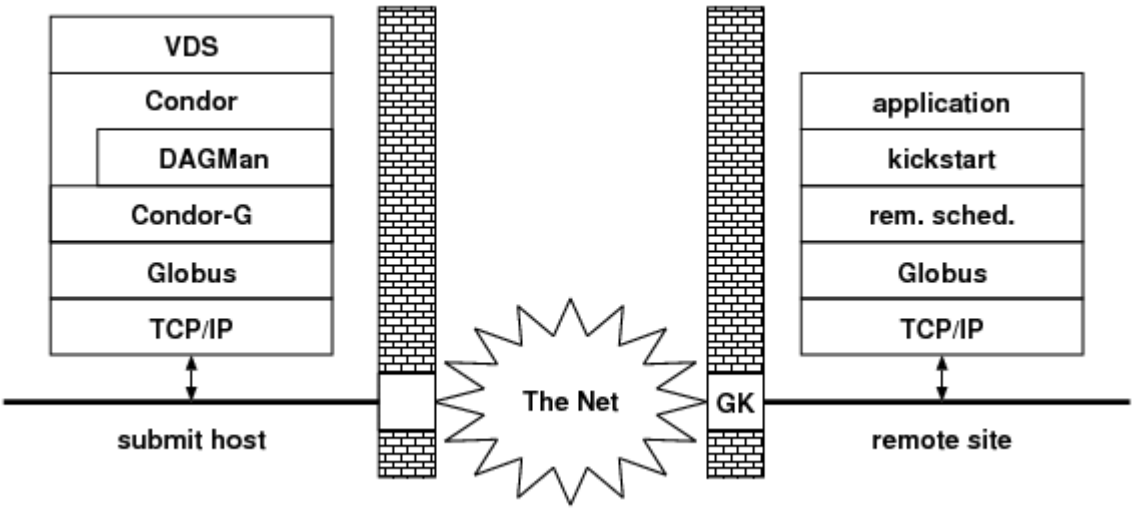


Figure 19: Layers of Software

Figure 19 shows the layers of software VDS depends upon. On the submit host, VDS generates submit files for Condor. Their run-order is captures into a Condor DAGMan control file. The DAGMan workflow manager is a Condor process itself. It submits the VDS-generated jobs to Condor, which will hand them to Condor-G. Condor-G is the adaptor between Condor and

Globus. Condor-G will translate the Condor submit file into Globus instructions. Globus in turn uses TCP/IP to contact any grid sites.

The submit host can be protected from the Internet by a firewall. Similarly, firewalls protect most remote sites. A remote firewall usually contains an opening for the gatekeeper port to enable access to the remote site's GRAM.

On the remote site, the software stack is (logically) traversed in the reverse direction. Globus picks up incoming requests on the port it listens. Globus hands off jobs to the remote site's scheduling system (e.g. PBS, LSF, and Condor). The remote scheduler may chose to run the actual job on a different host. The remote scheduler sees the "kickstart" tool as the executable. "kickstart" is a VDS tool which aids remote applications and provenance.

This complex system will be susceptible to various failures, transient and permanent. In order to track down the cause for a problem, we strongly urge the use of a bottom-up debugging approach. Often, even when you suspect that the VDS is broken, in fact, some of the supporting fabric experienced problems. If it is a genuine Condor or Globus problem, we will connect you with their respective support teams to resolve the issue.

A bottom-up approach will start with the least software layers, and gradually adds more software complexity, accessing from higher layers, until the problem can be reliably reproduced. Attempts to run VDS jobs will be futile effort, until the problem is fixed.

We do not recommend starting at the highest level, as higher levels often mask problems from lower levels. The following chapters deal with the layers starting from the lowest level. They show in bold face the user input, and in regular shade the expected output.

13.2 A Note on Firewalls

If your submit host employs firewalls, Globus must be told which range of ports can be used for incoming connections. Your submit host's site administrator must provide a range of ports that accept arbitrary incoming connections. As worst-case estimate, you will need 4 ports per grid jobs and 2 ports per gridftp transfer. If you site admin limits the incoming port range to 100 ports, you should not expect to reasonably be able to run more than 25 grid jobs.

You must have the environment variable *GLOBUS_TCP_PORT_RANGE* (called GTPR as shorthand) set to the range of ports a remote site may use to contact your submit host.

Most sites usually permit arbitrary outgoing connections. However, if your submit host's site also limits the outgoing connections on the firewall, you will have to tell Globus about this. For GT2, the range of ports must coincide with the range of incoming ports. The same variable GTPR applies to incoming and outgoing connections in GT2.

Starting with GT3, the environment variable *GLOBUS_TCP_SOURCE_RANGE* (called GTSR as shorthand) is used to bind the local port of an outgoing connection. With the GTSR variable, the port ranges for incoming and outgoing connection can be non-overlapping.

In the same sense that GTPR and GTSR applied to your submit site and your local firewall rules, the same variables GTPR and GTSR need to be set on any remote site in accordance with the remote site's firewall rules. The remote site's administrator copes with this task.

Incorrectly set GTPR and forgotten GTSR declarations are a source of infinite grief.

13.2.1 Firewalls for Remote Site Administrators

Please skip this section, unless you are setting up a remote site. VDT will not set up the firewall rules. You must manually correct the setup.

As shown in the introduction, the variables GTPR and GTSR must be correctly set up. For various reasons, this includes the following steps. Assume that the lowest permissible port is *min* and the highest permissible port in your port range is *max.* Assume that GTPR and GTSR cover the same port range.

1. To the bottom of your *$GLOBUS_LOCATION/etc/globus-user-env.csh* file, add the settings for GTPR and GTSR as follows:

```
setenv GLOBUS_TCP_PORT_RANGE "min,max"
setenv GLOBUS_TCP_SOURCE_RANGE "min,max"
```

2. To the bottom of your *$GLOBUS_LOCATION/etc/globus-user-env.sh* file, add the settings for GTPR and GTSR as follows:

```
GLOBUS_TCP_PORT_RANGE=min,max
GLOBUS_TCP_SOURCE_RANGE=min,max
export GLOBUS_TCP_PORT_RANGE GLOBUS_TCP_SOURCE_RANGE
```

3. To your xinetd configuration file for the Globus jobmanager, add the environment settings for GTPR and GTSR. If you use regular inetd, you must write a wrapper for each, or use the TCP wrapper daemon settings:

```
env += GLOBUS_TCP_PORT_RANGE=min,max
env += GLOBUS_TCP_SOURCE_RANGE=min,max
```

4. Do the same for your Globus gridftp server xinetd configuration.
5. Reconfigure the xinetd daemon by sending it the USR2 signal.
6. Since the Globus gatekeeper does not pass arbitrary environment variables to the Globus jobmanager, the *$GLOBUS_LOCATION/etc/globus-job-manager.conf* file must actively enable additional environment variables. Towards the end of the file, it should read:

```
-globus-tcp-port-range min,max
-extra-envvars LD_LIBRARY_PATH,GLOBUS_TCP_SOURCE_RANGE
```

For a submit host, steps 1 and 2 make sense to enable the configuration for all Globus clients.

13.3 Troubleshooting TCP/IP

At the lowest layer is the TCP/IP networking. Like the Stevens's literature, we use the *telnet* client to debug problems at this layer.

Occasionally a system administrator forgets to set up the correct path to the shared libraries required by Globus or other tasks to enable Grid services. You can check that the two important Grid services GRAM and GridFTP are accessible to you, and neither blocked nor failing. The test involves telnetting to the remote gatekeeper and gridftp server.

The following instruction checks the availability of a gatekeeper. Please replace <*gatekeeper*> with the fully-qualified domain-name (hostname) of the remote gatekeeper host:

```
telnet <gatekeeper> 2119
Trying <dotted-quad>...
Connected to <gatekeeper>.
Escape character is '^]'.
^]
telnet> close
Connection closed.
```

What you type is shown in boldface. The expected responses are shown in regular print. Note that the escape character "^]" for telnet is usually Ctrl+5 on Linux. Alternatively, you can press Enter a couple of times.

If you do not see the above responses, but instead a message about missing shared libraries, e.g.:

```
telnet <gatekeeper> 2119
Trying <dotted-quad>...
Connected to <gatekeeper>.
Escape character is '^]'.
globus-gatekeeper: error while loading shared libraries:
libglobus_gss_assist_gcc32dbg.so.0: cannot open shared object file: No
such file or directory
Connection closed by foreign host.
```

or a complaint about the wrong version of some shared library, the site administrator needs to set the LD_LIBRARY_PATH correctly.

To check the availability of a GridFTP service, try the following instructions. In this case, replace <*gridftpserver*> with the fully-qualified domain-name (hostname) of the remote GridFTP server:

```
telnet <gridftpserver> 2811
Trying <dotted-quad>...
Connected to <gridftp-server>.
Escape character is '^]'.
220 <host> FTP server (GridFTP Server 1.5 [GSI patch v0.5] <…>) ready.
QUIT
221 Goodbye.
Connection closed by foreign host.
```

If everything works as expected, you will see message similar to the ones above. Some sites (e.g. FNAL) may add banner messages to indicate federal systems. If you see a message about missing shared libraries, e.g.

```
telnet <gridftpserver> 2811
Trying <dotted-quad>...
Connected to <gridftp-server>.
Escape character is '^]'.
in.ftpd: error while loading shared libraries:
libglobus gss assist gcc32dbg.so.0: cannot open shared object file: No
such file or directory
Connection closed by foreign host.
```

or a complaint about the wrong version of a shared library, the site administrator needs to set the LD_LIBRARY_PATH correctly.

13.4 Troubleshooting GT2

This section deals with various Globus-level tests to verify that things are as they should be. Consequently, only Globus command-line tools are used to detect problems.

13.4.1 Verify that your certificate is permitted on the remote site

If you experience problems contacting a remote pool, the first thing to check is, if your certificate is acceptable to the remote site. Please replace the string "<*host*>" with the fully qualified domain-name (hostname) of the remote gatekeeper host:

```
globusrun -a -r <host>/jobmanager-fork

GRAM Authentication test successful
```

Boldface marks the user input. Regular font marks the expected response. There are multiple reasons why a remote site chooses to reject your authentication request or certificate:

- You forgot run initialize your user proxy certificate.

  Run *grid-proxy-init* **or** *voms-proxy-init* to initialize your user proxy certificate.

- The remote site does not know your certificate. There is no mapping in the remote gridmap file from your user certificate to a remote account.

  You will need to contact the remote site's admin to add your certificate subject string. Never send your full certificate nor your key file.

- There could be a mistake in the remote gridmap file, when the administrator added your user certificate. Simple typos can account for this error.

  If you suspect this rare error, you may want to have the remote site's administrator carefully check that your user certificate subject strings matches what he entered.

- A certificate authority (CA) signs any user certificate. The remote site may lack a copy of the CA's own certificate that signed your user certificate.

  You will need the remote site administrator's help to install a copy of your CA certificate. Please note that certificates from a SimpleCA are not tolerated by most sites.

- The host certificate of the gatekeeper has expired.

  In this case, neither jobs nor transfers start on the remote site. The remote site administrator will have to request a new host certificate.

- The CA certificate installed on the gatekeeper has expired.

  Usually the remote administrator receives ample warning about expiring host certificates. However, in rare cases, where the email account associated with the notification become obsolete, or in case of aggressive SPAM filters, a site administrator may lose vital messages. The Globus tool *grid-cert-info* permits to check any certificate's expiration.

- The certificate revocation lists associated with a CA certificate could not be retrieved.

  The gatekeeper contains a list of commonly used CA certificates. Often, the certificate's validity is limited using certificate revocation lists. If the revocation list could not be retrieved for too long of a time, the CA certificate becomes invalid. The remote site admin needs to ensure that the *edg-crl-updated* is continually running.

- The submit host misses a copy if the CA certificate that signed the host certificate of the remote site.

  The authentication of the Grid Security Infrastructure (GSI) is mutual. While the remote site must have a copy of the CA certificate that signed your user certificate, your submit host must have a copy of the CA certificate that signed the remote site's host certificate.

- No jobmanager service under the name *jobmanager-fork* is available.

  This should not happen with VDT nor OSG sites. Condor-G also requires that a *jobmanager-fork* service exists.

- The clocks between the two hosts differ by more than 300 seconds when converted into the same time zone.

  For security purposes, clocks must not differ by more than 300 seconds. Remote gatekeeper should use NTP to synchronize their clocks. Your submit host should, if possible, also use NTP to update its own clock.

- Your user certificate expired.

  Use *grid-cert-info* to determine the expiration date of your user certificate.

- You have multiple user certificates, but you are using the wrong one.

GSI errors are unfortunately not illuminative, and require either some degree of intuition to get to the bottom of, or a degree of perseverance.

13.4.2 Verify that you can run simple jobs

After checking that your user certificate authenticates, simple job executions are next on the checklist. These tests are two-parts.

First, you want to check that you can run a simple fork job. Fork jobs run on the remote gatekeeper host itself. For this reason, you must not use computationally or resource intensive jobs with the fork jobmanager. Please replace the <*host*> string below with the fully-qualified domain-name (hostname) of the remote gatekeeper host:

```
globus-job-run <host>/jobmanager-fork -l /bin/date
Fri Sep 12 15:52:59 CDT 2003
```

The above instructions run a simple Globus job in interactive mode. It should return quickly.

Usually, the remote site has at least one other jobmanager installed. The other jobmanager is usually a batch jobmanager, which points to a job scheduling system like Condor, PBS or LSF. The VDS will use batch jobmanagers to run compute jobs. Thus, you should verify that you could actually submit jobs to the remote batch scheduler.

The remote site usually publishes information which is the correct jobmanager contact string to use. Please check with the remote site for any required additional arguments like a project name for accounting, a queue name, or a minimum required job run-time request. Please refer to the Globus documentation on how to set these additional arguments:

```
globus-job-run -help 2>&1 | less
```

Note that Globus uses single hyphens for long options. Globus tools print their usage information on *stderr* instead of *stdout*.

The difference between the previous fork jobs and the next batch job is the suffix on the job manager contact. Please replace the <*sched*> string with the appropriate remote scheduling system.

```
globus-job-run <host>/jobmanager-<sched> -m 2 -l /bin/hostname –f
worker.node.in.pool
```

The -f option is only necessary for Linux. Please note that the execution of this command may take considerable time, so patience is required. If your queue priority is too low, it may starve for days. Thus, the -m option notifies the remote scheduling system that you don't expect your job to take longer than 2 minutes. The job will run almost instantaneously. However, some sites require a minimum run-time declaration of 2 minutes, and will deal with smaller declarations incorrectly.

If your jobs fail for weird reasons, there is a debug file created by Globus GRAM. GRAM is the part of Globus responsible for running jobs. For duration of a job, it appends to a log file at the remote site in the home directory of the account your user certificate is mapped to. The log file starts with "gram" and ends in ".log". Most sites are set up that a successful job run will remove the log file.

If multiple attempts failed, you will notice a plethora of these log files. Unless the remote account is shared, remove all gram*log files before submitting another test job. If you are the only user, this way, all gram*log files created will pertain to your job submission attempt. These log files provides valuable insights for anybody trying to debug jobs at the Globus level. Frequently, if Condor-G experiences problems, developers may ask for this file, too. Its information is admittedly arcane, and definitely not for the faint-of-heart.

13.4.3 Verify your run-time environment

Most users of grid software are astonished about the fact that the environment settings of their grid jobs significantly differ from interactive logins. It is a revelation to request the environment variables that are actually visible on a remote site:

```
globus-job-run <host>/jobmanager-fork -l /usr/bin/env | sort
GLOBUS_GRAM_JOB_CONTACT=https://<host>:61445/20636/1128549670/
GLOBUS_GRAM_MYJOB_CONTACT=URLx-nexus://<host>:61446/
GLOBUS_LOCATION=/vdt/globus
GLOBUS_TCP_PORT_RANGE=61440,65535
HOME=/home/<user>
LD_LIBRARY_PATH=/vdt/globus/lib
LOGNAME=<user>
X509_USER_PROXY=/home/<user>/.globus/.gass_cache/local/md5/dd/934a19e24368a7009
11ee476e8fba4/md5/75/40e0c53a79e7780586e646e067e510/data
```

Your output will vary. As you can see, there is no *PATH* environment variable set, nor any *LD_LIBRARY_PATH.* Your grid jobs should never assume that they "see" an environment you see on your interactive login.

13.4.4 Verify that you can transfer files

In order to provide data for your computations, the VDS stages files between sites. You should check that you could access file on the remote system using GridFTP. Please replace <*host*> in the instructions with the remote GridFTP server's fully qualified domain-name (hostname). Often, the GridFTP server is identical to the gatekeeper, but this is not a requirement:

```
date > test.out
globus-url-copy -vb file:///`/bin/pwd`/test.out \
 gsiftp://<host>/tmp/test.out
           29 bytes         0.03 KB/sec avg         0.03 KB/sec inst
echo $?
0
```

The above creates a small test file called "test.out". The next step transfers the freshly created file to the remote GridFTP server. Note the use of back-quotes to obtain the correct current working

directory. The *-vb* option prints additional information about the progress of the transfer. You must check the exit code ($?) of the run to verify that it indeed ran successfully.

If the transfer fails, it is often helpful to run with the additional option *-dbg* to request debug output. A common problem are firewall settings of the remote site as well as your submit host.

The debug output will often give valuable clues about firewall problems. As the debug output shows explicitly the ports that are used for any transfer, you can compare these port against the port ranges (GTPR and/or GTSR) that apply to your submit host and to the remote site.

13.5 Troubleshooting GT4

Support of GT4 is prototypical. We will add documentation here once GT4 support becomes mainstream VDS. For the pre-WS GT4 the tests in the previous section apply.

13.6 Troubleshooting Condor

This section deals with troubleshooting Condor. The Condor and DAGMan layer are usually a level above Globus. This section also addresses the adaptor between Condor and Globus, the Condor-G component.

13.6.1 Verify that you can run local Condor jobs

This section tries to verify your submit host environment. It is only necessary to verify once. The local submission in this section use the Condor scheduling system on your submit machine. Condor contains an extension Condor-G, which we will test in the following section.

To verify that your local Condor installation is capable of execution jobs correctly, please create a textual file called "try.sub" with the following content:

```
universe = scheduler
executable = /bin/date
output = try.out.txt
error = try.err.txt
log = try.log.txt
copy_to_spool = false
transfer_executable = false
notification = NEVER
queue
```

In the next step, submit this file to the Condor system. You only type the first line. The next three lines show the expected response from the Condor system, where <*ddd*> is an integer number:

```
condor_submit try.sub
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster <ddd>.
```

You can now occasionally poll the Condor queue to check on your job. Since the job will execute almost instantaneous[10], you will not see it in the Condor queue. It may be easier, though, to check the job's log file called "try.log.txt". The expected output file looks some *similar* to the following:

```
000 (ddd.000.000) 09/12 16:12:18 Job submitted from host: <128.135.11.143:50576>
...
001 (ddd.000.000) 09/12 16:12:18 Job executing on host: <128.135.11.143:50576>
...
```

[10] There is a bug in Condor 6.7.* which delays submissions into the "scheduler" universe by a couple of minutes. Condor 6.7.11 fixes the bug.

```
005 (ddd.000.000) 09/12 16:12:18 Job terminated.
        (1) Normal termination (return value 0)
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
        0  -  Run Bytes Sent By Job
        0  -  Run Bytes Received By Job
        0  -  Total Bytes Sent By Job
        0  -  Total Bytes Received By Job
...
```

Condor prefixes each log section with a status tag. Tag 000 means that a job was submitted, e.g. recognized by the Condor system. Tags 001 means that the job is executing. Tag 005 is generated after the job finished. The return value reported for *local* jobs is accurate, and should be zero. If you were getting an unsuccessful run, you will not see the return value of zero for local jobs.

The output file "try.out.txt" and the error file "try.err.txt", which were connected to standard file descriptors *stdout* and *stderr* respectively, provide insights on the reason of any failure of local jobs. In the above example, the "try.out.txt" file should contain a valid output from the Unix date command.

13.6.2 Verify that you can run Condor-G jobs

The Condor-G job facility is similar to plain Condor. The difference is, though, that it runs jobs in the Grid using Globus. This fact is captured by the Condor universe. In order to test these jobs, you need to modify the previous submit file slightly. Create a file "try2.sub" with the contents shown below. Again, you need to replace <*host*> with the gatekeeper's host name and <*sched*> with the correct scheduler name:

```
universe = globus
executable = /bin/hostname
globusscheduler = <host>/jobmanager-<sched>
remote_initialdir = /tmp
output = try2.out.txt
error = try2.err.txt
log = try2.log.txt
copy_to_spool = false
transfer_executable = false
notification = NEVER
queue
```

If the remote site requires special projects or queues, you may need to employ RSL syntax to add them using the "globusrsl" command in the submit file, and RSL syntax as content. If you do not know what this is, do not sweat it. Otherwise, if the remote site requires a projects, use the (project=<name>) string. If a queue name is required, use the (queue=<queue>) string. You will need to add the *globusrsl* configuration option into the submit file above right above the single word “queue”:

```
# this is an example, don’t use as such
globusrsl = (project=XYZ)(queue=ABC)(maxtime=2)
```

The next step submits this file to Condor, which will in turn process it using Condor-G. The first line again shows the input you need to type and the next three lines the expected output:

```
condor_submit try2.sub
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster <ddd>.
```

The above line shows that the job got the job id <*ddd*>, which is an integer number. When checking the progress of any job, use this number. For Globus jobs, it is easier than checking the log file. To check the status of any job, use the *condor_q* command:

```
condor_q <ddd>
-- Submitter: hamachi.cs.uchicago.edu : <128.135.11.143:50576> : hamachi.cs.uchicago.edu
 ID      OWNER            SUBMITTED     RUN TIME ST PRI SIZE CMD
 ddd.0   voeckler        9/12 16:32   0+00:00:00 I  0   0.0  date
```

The plain command shows all Condor jobs. For most of the job's lifetime, its status will be displayed as "idle". This is the *local* view that the Condor on the submit host has. Since Grid jobs are special, the option -globus to *condor_q* allows you to obtain better information about Condor-G jobs. This information contains the Globus state a job has on the remote gatekeeper:

```
condor_q -globus <ddd>
-- Submitter: hamachi.cs.uchicago.edu : <128.135.11.143:50576> : hamachi.cs.uchicago.edu
 ID      OWNER          STATUS  MANAGER  HOST                EXECUTABLE
 ddd.0   voeckler      UNSUBMITTED condor   <remote gatekeeper>     /bin/date
```

As time progresses, the job will change Globus state:

```
condor_q -globus <ddd>
-- Submitter: hamachi.cs.uchicago.edu : <128.135.11.143:50576> : hamachi.cs.uchicago.edu
 ID      OWNER          STATUS  MANAGER  HOST                EXECUTABLE
 ddd.0   voeckler      PENDING condor   <remote gatekeeper>   /bin/date
```

The first example shows the output from a freshly submitted job which has not made it into the remote scheduling system. The next example shows that the remote scheduling accepted the job, but did not start it yet. Various other remote Globus state exist. For grid debugging purposes, the remote jobs state is more interesting to us than the local Condor state.

Furthermore, for each user, all his or her grid jobs report into a Condor-G specific log file. If the submit host is a VDT setup, an extensive log can be found in the */tmp* directory. Its name starts with "GridmanagerLog" and ends with the submit host's user account name. Developers are very interested in this log file for any kind of grid problem in connection with Condor-G.

After a while, approximately five to ten minutes, the job should have finished. However, it may take longer on busy systems. If you check the job's log file "try2.log.txt", the information will be similar to what was shown in section 13.6.1, but longer:

```
000 (ddd.000.000) 09/12 16:32:51 Job submitted from host: <128.135.11.143:50576>
...
017 (ddd.000.000) 09/12 16:33:05 Job submitted to Globus
    RM-Contact: e.cs.uchicago.edu/jobmanager-condor
    JM-Contact: https://e.cs.uchicago.edu:49153/19305/1063402370/
    Can-Restart-JM: 1
...
001 (ddd.000.000) 09/12 16:38:14 Job executing on host: e.cs.uchicago.edu
...
005 (ddd.000.000) 09/12 16:38:24 Job terminated.
        (1) Normal termination (return value 0)
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Remote Usage
                Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
        0  -  Run Bytes Sent By Job
        0  -  Run Bytes Received By Job
        0  -  Total Bytes Sent By Job
        0  -  Total Bytes Received By Job
...
```

There are a few subtle differences worthy of your attention. There is an additional tag 017, which is specific to Globus jobs. Condor-G generates the tag 017 once it delivered the job to the remote gatekeeper.

Although the 005 tag contains a return value, the value reported by Condor-G is always 0 for GT2 and pre-WS GT4. This is effectively Globus' fault by lacking the appropriate reporting facility. The lacking return code is also the reason we highly recommend using some kind of grid launcher (kickstart, gridshell) to run remote jobs. The grid launcher will provide the submit site with the remote result code. GT4 WS-GRAM solved this problem.

There are now a lot more reasons possible for your job failing. Some errors are simple, e.g. if the job submission returns with the message that the proxy filename could not be determined, you grid proxy certificate has expired.

In many cases, VDS tells the Condor subsystem to retry job submissions a couple of times. These Condor-internal retries usually help to weed out transient grid failures. However, if all retries fail, or the error is hard, the job is put into Condor state "held". The "held" state is marked with an H in the *condor_q* output.

With the long option "-l" to condor_q, you can check in more detail, what went wrong with a failed job, if you provide the ID that Condor told you about the job:

```
condor_q –l <ddd> | sort
```

The result is a wealth of information. The *HoldReason* usually contains the Globus error, if Globus was at fault for holding a job. For instance, if the error reads that *stdout* or *stderr* could not be opened, most likely a firewall issue prohibits Globus from using GASS transfers:

```
condor_q –l | sort | grep ^Hold
```

The above returns all reasons any job was put on hold for all jobs. This is useful for workflows that VDS generates.

13.6.3 Troubleshooting DAGMan

The DAGMan component runs a VDS workflow using Condor. While DAGMan rarely exhibits bugs, in the past few years we did manage to find a few. DAGMan's own debug file contains the best information about DAGMan failures. The debug file ends in the suffix "*.dag.dagman.out*".

Given recent troubles with DAGMan, you want to ensure that you are running a recent version of Condor.

13.7 Troubleshooting VDS

The VDS problems are usually of two kinds. During what we call *compile time*, when a workflow is produced and concretized, most tools return with immediate failure. However, during *run time*, when a workflow executes on the grid, failure becomes more difficult to detect and propagate.

13.7.1 Troubleshooting the Abstract Plan

Talk about -Dvds.log.xxxx=yyyy and -Dvds.verbose=5 for abstract planning tools.

Until then, please contact the VDS support.

13.7.2 vds-verify between abstract and concrete planning

VDS provides a new tool *vds-verify*. It is meant to be run after abstract planning and before concrete planning commences. *Vds-verify* will look into the abstract plan and your runtime environment and check

- for the presence of a non-empty DAX file,

- variable settings in the user's environment,
- accessibility of basic directories for Globus, Java and VDS,
- presence of most commonly used executables from Globus, Condor, Java and VDS
- grid user certificate proxy problems,
- VDS property setup,
- presence and accessibility of the site catalog (SC), transformation catalog (TC) and replica catalog (RC),
- that SC and TC do not contain disjunctive sets of sites, e.g. at least one site handle from one catalog is also mentioned in the other catalog,
- essential and other Perl modules and versions which may be required during run-time,
- valid DAX XML:
    - dependency do not refer to non-existing jobs,
    - all input files are known to the replica manager,
    - all transformations have at least on entry in the TC.

For large DAX input, the checks may take noticeable time. Since the parsed XML is kept in memory, *vds-verify* may be inadequate for immense DAX files.

We will attempt to enhance *vds-verify* to catch more of the common errors that users stumble when submitting workflows into the grid.

To verify entries in SC and TC, and remote site sanity, the *vds-check-sites* tool will verify that a remote site is accessible. It implements many of the checks from sections 13.3 and 13.4. It is a prototypical tool, and you should read its manual page before attempting to use it.

13.7.3 Troubleshooting Pegasus

Please contact the VDS support for now.

13.7.4 Troubleshooting Euryale

Talk about "euryale.log" as first approach, then each job's debug file.

Until then, please contact the VDS support for now.

13.8 Summary

If any of the simple tests stated fails, you will not be able to run the VDS, because something in the fabric is broken. Attempts to run VDS jobs will be futile effort until the problem is fixed. The Condor and Globus documentation should provide additional information about the tools and methods used to identify, debug and sometimes fix the problem.

13.9 Submitting a bug report

The VDS facilitates many ways for users to interact with us. Frequently, we will also need to know more about your run-time environment. Every VDS tool reports its version when invoked with the --version option. There is also a separate *vds-version* tool. Since cutting-edge users use nightly CVS builds, we suggest to use the full output option for non-official release:

```
vds-version --full
```

```
1.3.10-200509140001l2Z
```

To obtain the Condor version, use

```
condor_version
$CondorVersion: 6.7.10 Aug  3 2005 $
$CondorPlatform: I386-LINUX_RH9 $
```

The Condor version is 6.7.10 in this case. The Globus version is difficult before GT4. Starting with GT4, the Globus version can be obtained using the *globus-version* tool:

```
globus-version
4.0.1
```

If you have a VDT setup, the VDT version will tell you about all installed software versions, including Condor, Globus and RLS:

```
vdt-version
You have installed a subset of VDT version 1.2.3:
    CA Certificates v4 (includes LCG 0.25 CAs)
    Virtual Data System 1.2.14
    ClassAds 0.9.7
    Condor/Condor-G 6.6.7
…
    Globus Toolkit 2.4.3
…
    RLS 2.1.5
…
```

13.9.1 Bugzilla

If you are utterly sure about a bug in VDS, or an enhancement request to VDS, you should use our bugzilla bug tracking system http://bugzilla.globus.org/vds/. The bug tracking facility permits you to see whenever a developer fixes something or changes state of the bug.

1. In order to use bugzilla, you will have to create an account. Note that bugzilla accounts from Globus work equally with VDS. If you already have an account, please use your existing account.
2. Select the "new bug" option from the start page.
3. Select the appropriate subsystem. In most cases, this will be the "Virtual Data System".
4. Please do select a component that most closely fits your problem component. You may also want to tell us which VDS version experiences the problem.

Please consider for your bug report the following details:

- Copy and paste the *exact* error message.
- Tell us exactly the command-line you used.
- Often, we will need to know which properties were set in your property files. Use our *show-properties* tool to report all properties that are visible to you.
- Many times, either we will need the VDL input file, or the DAX input file. Please attach the file to the bug, if you think them pertinent.

13.9.2 Support Mailing list

Sometimes, a bug can be tricky. Either it requires a multitude of files to show it, or you may not be sure, if it is a VDS bug. You can use the *vds-support at griphyn dot org* mailing list to submit bug reports.

Please use the *vds-bug-report* tool to gather all necessary files into one tar ball. Attach this tar ball as binary attachment into the email of your bug report. Please be as specific as possible in your bug report. That means you should tell us the exact and full error message, including context, and how you invoked the tools that generated the bug. Sometimes, the invocation of preceding VDS tools may also give us a clue. Please provide as many software versions as relevant to the problem.

13.9.3 Discussion Mailing list

If you have a general question that is unrelated to a bug, or an enhancement request that does not require immediate support, we suggest the *vds-discuss at griphyn dot org* mailing list.

13.10 Version mismatches

The *vds-version* tool comes with a special option *--match* to verify integrity. In rare cases, the VDS installation does not match the compiled library. The special option of *vds-version* will detect this.

Revision $Revision: 1.4 $ of master document file $RCSfile: VDSUG_Master.xml,v $.