AstroGrid-D

Deliverable 3.3

Distributed File Management

Tests of the Data- and Replica-Management Software for Selected Use Cases[1]

Deliverable	D3.3
Authors	Ralf Wahner, Thomas Brüsemeister, Mikael Högqvist, Jürgen Steinacker, Rainer Spurzem, Frank Breitling, Art Carlson, Robert Engel, Harry Enke, Iliya Nickelt, Thomas Radke
Editors	
Date	March 24th, 2008
Document Version	1.0.0
Current Version	1.0.0
Previous Versions	0.7.0, 0.6.1, 0.6.0, 0.5.1, 0.5.0–0.2.0, 0.1.7–0.1.0

A: Status of this Document

Deliverable 3 of working group 3.

B: Reference to project plan

[1]This work is part of the AstroGrid-D project and D-Grid.The project is funded by the German Federal Ministry of Education and Resarch (BMBF).

This is the third deliverable of the working group *Distributed File Management*. The work refers to task III-5 *Software test by adapting community applications* and to Milestone M24.

C: Abstract

The first chapter of this document summarizes the requirements for distributed file management in **AstroGrid-D**, based on both, the aforementioned questionaire and Deliverable 3.2; see appendix A or [7], respectively. The second chapter documents to which extent the requirements have been implemented in the current operation of three selected use cases, namely **Dynamo**, **NBody6++** and **GEO600**. Finally, the appendix presents the questionaire to its whole extent.

D: Changes History

Version	Date	Name	Brief summary
0.1.0	Oct. 4th, 2007	Jürgen Steinacker	Working Draft Creation
0.1.1	Oct. 23rd, 2007	Ralf Wahner	Contribution about handling dead links
0.1.2	Oct. 24th, 2007	Ralf Wahner	Figure illustrating the solution for the dead link problem. Document structure revised.
0.1.3	Dec. 3rd, 2007	Ralf Wahner	Section *On the purpose of this deliverable*
0.1.4	Dec. 6th, 2007	Ralf Wahner	Section *Alternative Approach: AstroGrid Data Management* (Draft created)
0.1.5	Dec. 11th, 2007	Ralf Wahner	Section *Alternative Approach: AstroGrid Data Management* (Draft finished)
0.1.6	Dec. 13th, 2007	Ralf Wahner	Section *Discussion: RLS versus ADM* (Draft created). Figure illustrating the database used for the AstroGrid-D Data Management.
0.1.7	Dec. 14th, 2007	Ralf Wahner	Section *Discussion: RLS versus ADM* (Draft revised)
0.2.0	Dec. 15th, 2007	Ralf Wahner	**Draft version of Deliverable 3.3 published** as requested for deadline on December 15th.
0.3.0	Dec. 19th, 2007	Ralf Wahner	Document structure has been revised.
0.4.0	Jan. 10th, 2008	Ralf Wahner	Section *Approach 1: AstroGrid-D Data Management*. Draft version revised.
0.5.0	Jan. 21st, 2008	Harry Enke	**Restructuring the document**.
0.5.1	Jan. 25th, 2008	Ralf Wahner	Subsection *Dynamo*: Markup for xml source and file- and directory names.
0.6.0	Feb. 6th, 2008	Ralf Wahner	Preview of new draft version published.
0.6.1	Feb. 8th, 2008	Ralf Wahner	**New Draft version published** due to refusal of previous draft (0.2.0, on Dec. 15th, 2007) for discussion in workgroup 3 and architecture group until Feb. 25th, 2008 (phase one).
0.7.0	Feb. 25th, 2008	Ralf Wahner	**New Draft version published** due to refusal of previous draft (0.2.0, on Dec. 15th, 2007), for discussion in the AstroGrid-D community until Mar. 17th, 2008 (phase two).
1.0.0	Mar. 24th, 2008	Ralf Wahner	Typos fixed. Example for `globus-url-copy`. Autorship indicated by means of footnotes.

Contents

1 Testing requirements due to the previous Deliverable 3.2[2]

The purpose of this deliverable, abbreviated by D3.3, is to describe and evaluate technical approaches specified by the previous deliverable, [7], in order to meet certain situations common to scientific computing within **AstroGrid-D**, e.g. staging of input and output data between execution host and storage location, cleaning up the execution host as well as replica management. The following two sections describe staging/cleanup and replica management from a general perspective, i.e. without referring to a certain use case. The application of the methods specified in [7] to use cases and the implications are presented in chapter 2 *Test of Use Cases Scenarios*.

1.1 File staging and cleanup

Staging and cleanup are supposed to utilize the **Globus Resource Specification Language** (**RSL**) while replica management is supposed to implement the **Globus Replica Location Service** (**RLS**). Carefully distinguish **RLS** from **RSL**: Whereas the former one, **RLS**, cares for the maintenance of file replica, i.e. identical copies of a file on different storage locations, the latter one, **RSL**, is used to specify file staging, cleanup and command line arguments for job submission. Therefore **.rsl* files are also known as *job descriptions*. The following code is a sample job decription:

```
<job>
  <!-- specify executable on execution host -->
  <executable>my_program</executable>
  <argument>ARGUMENT_VALUE_1</argument>
  <argument>ARGUMENT_VALUE_2</argument>
  <stdout>my_program.out</stdout>
  <stderr>my_program.err</stderr>
  <!-- transport a file from submission host to execution host -->
  <fileStageIn>
    <transfer>
      <sourceUrl>gsiftp://[hostname]/[path]/my_program</sourceUrl>
      <destinationUrl>file:///${GLOBUS_USER_HOME}/my_program</destinationUrl>
    </transfer>
  </fileStageIn>
  <!-- transport files back from execution host to submission host -->
  <fileStageOut>
    <transfer>
      <sourceUrl>file:///${GLOBUS_USER_HOME}/my_program.out</sourceUrl>
      <destinationUrl>gsiftp://[hostname]/[path]/my_program.out</destinationUrl>
    </transfer>
    <transfer>
      <sourceUrl>file:///${GLOBUS_USER_HOME}/my_program.err</sourceUrl>
      <destinationUrl>gsiftp://[hostname]/[path]/my_program.err</destinationUrl>
    </transfer>
```

[2]Contributed by: Ralf Wahner (*rwahner@ari.uni-heidelberg.de*)

```
    </fileStageOut>

    <!-- clean up files and directories on the execution host -->
    <fileCleanUp>
      <deletion>
        <file>file:///${GLOBUS_USER_HOME}/my_program.out</file>
      </deletion>
      <deletion>
        <file>file:///${GLOBUS_USER_HOME}/my_program.err</file>
      </deletion>
      <deletion>
        <file>file:///${GLOBUS_USER_HOME}/my_program</file>
      </deletion>
    </fileCleanUp>

  </job>
```

First off, the job description *resides* permanently on the submission host but *runs* temporarily on the execution host. The above job description specifies by means of the `<executable>`-Tag, that `my_program` is to be handed over to the shell on the execution host together with two command line parameters, namely `ARGUMENT_VALUE_1` and `ARGUMENT_VALUE_2`. Furthermore, the output and error channel of `my_program` is redirected to the files *my_program.out* and *my_program.err*.

A common cycle of scientific computing begins on the *submission host*, where the sourcecode or binary version of the program and its input data reside *permanently*. By contrast the data is usually *volatile* on the *execution host*, because after the program terminates the outcome is transferred back to the submission host and the hard disk space on the execution host is released. The first section of the job description, namely `<fileStageIn>`, specifies the file transfer from submission host to execution host. For each file transferred, the `<fileStageIn>`-Tag contains a `<transfer>`-Tag inside of which source and target location are declared by means of `<sourceUrl>` or `<destinationUrl>`, respectively. Again, the job description executes on the execution host, so the *remote* source file path is prefixed with a protocol name, namely `gsiftp://`, while the *local* copy of the source file uses the `file://` prefix, where "remote" and "local" are understood with respect to the execution host. Finally, `${GLOBUS_USER_HOME}` is a user specific environment variable that points to the users home directory on the respective grid resource.

The meaning of the second section of the job description, namely `<fileStageOut>` is supposed to be self-explanatory, since the files are merely transported in reverse direction, i.e. from execution host to submission host, after the programm has terminated. The `<fileCleanUp>` section cares for tidying up the execution host after the output has been transferred back to the submission host. Each file to delete has a `<deletion>/<file>`-Tag on its own. Since the job description executes on the execution host all files to be deleted are local files with respect to the execution host and are therefore prefixed by `file://`. The cycle completes after cleaning up the files on the execution host.

Let the above job description be written to the file *job_description.rsl*. During job submission via `globusrun-ws` the job description is included and transferred by means of the `-f` switch:

```
globusrun-ws -submit -F <factory> -S -Ft <factory type> -f job_description.rsl
```

(A description of the operation mode and the switches of `globusrun-ws` is beyond the scope of this text. See *http://www.globus.org/toolkit/docs/4.0/execution/wsgram/rn01re01.html* for details about globusrun-ws.)

Remarks: (1) Even though the **RSL** syntax resembles the eXtensible Markup Language (XML), **RSL** actually is not XML compliant, since **RSL** allows to let away the XML declaration which is mandatory, due to the XML specification of the W3C; see *http://www.w3.org/XML*.
(2) According to [7], directory staging does not work whith **GridGateWay**. Due to the current recherche status, **GridGateWay** is still unable to stage directories.

1.2 Replica management

A replica is a copy of one file on a different storage location. Replica are needed for backup as well as for reducing the traffic load across the network between the storage facility and the actual resource that carries out the scientific computation on that data. Replica management is supposed to implement the **Globus Replica Location Service** (**RLS**). According to [7], **RLS** is basically a bidirectional look-up table, i.e. **RLS** allows to query for all values associated with a given key as well as to query for all keys associated with a given value, by means of indexing both, the keys and the values. In this context a "key" is an abstract identifier *merely representing* a physical file and the "value" associated with that key is a list of strings, each of which is a fully qualified path to an actual copy of the file on a real-life storage location. **RLS** maintains those meta data only, i.e. **RLS** does *not* care for file transfer and for the consistency of its data content with the actual file storage locations. Assume, that the aforementioned job description contains the abstract identifier instead of a fully qualified path, then the identifier must be resolved to a physical file name before job submission. The following subsections describe the key concepts behind **RLS** as well as the usage of the command line client program globus-rls-cli in a tutorial like manner.

1.2.1 Understanding logical and physical filenames and mappings

Regarding file administration by means of **RLS** there are two essential terms to understand. Given a file in an arbitrary location, say the user-defined bash configuration file *.bashrc*, the **physical filename** (PFN) of *.bashrc* is compiled from a protocol identifier, the hostname of the location where the filesystem resides and the fully-qualified name of the file, i.e. its path and basename, for example:

```
# Physical filename (PFN)
gsiftp://alnitak.ari.uni-heidelberg.de/home/Agrid/agrid064/.bashrc
```

Here, the protocol identifier is gsiftp. Other valid identifiers are gsiscp, ftp or http, to name just a few. The hostname and fully-qualified filename are considered to be self-explanatory. Contrary to the PFN, the **logical filename** (LFN) represents *.bashrc* from the perspective of **RLS**. The LFN must be unique with respect to **RLS** and can be choosen arbitrarily by the user, e.g. *agrid064-dotbashrc* in order to identify *.bashrc* owned by grid user agrid064:

```
# Logical filename (LFN) for the above PFN
agrid064-dotbashrc
```

Putting a file under **RLS** control is a two-step-process, namely registering the file with **RLS** and transporting the file to its desired location within the grid. In order to get a clear picture of what

RLS does and how it operates, the next issue to understand is the 1:n-mapping of one LFN (with respect to **RLS**) to one or more PFNs (with respect to files in the real world).

The official website of **RLS**, *http://www.globus.org/toolkit/data/rls*, declares, that **RLS** "maintains and provides access to mapping information from logical names for data items to target names". Assume, that "data items" are just files like the above *.bashrc*, in order to simplify the consideration for this text. Straightly speaking, under **RLS** an LFN behaves like a reference pointing to an "array" of actual filenames; set more formally:

$$LRC : \texttt{agrid064-dotbashrc} \mapsto \left\{ \begin{array}{c} \texttt{gsiftp://alnitak/home/Agrid/agrid064/.bashrc,} \\ \texttt{gsiftp://mintaka/home/Agrid/agrid064/.bashrc,} \\ \vdots \\ \texttt{gsiftp://eridanus/home/Agrid/agrid064/.bashrc} \end{array} \right\},$$

where `alnitak` is shorthand for `alnitak.ari.uni-heidelberg.de` to save space (likewise for `mintaka` and `eridanus`). The above "formula" represents a **mapping** from `agrid064-dotbashrc` (LFN) to a set of PFNs. The notion of the so-called **local resource catalogue** (LRC) allows to group mappings that have one or more properties in common.

1.2.2 Initial mapping and insertion of replica

We are now prepared to tell **RLS** about the file we want to check in. First off, we are going to announce the *.bashrc* file, or more precisely *gsiftp://alnitak/home/Agrid/agrid064/.bashrc*, to **RLS** and afterwards register a handful of replica (i.e. copies of a file on different locations) of *.bashrc*. As a general rule, the first step is to create an *initial mapping* within the **RLS** "repository":

```
# Create an initial mapping (RLS)
# Command:  create <lfn> <pfn> <rls://server-name>
globus-rls-cli
    create agrid064-dotbashrc                        # logical filename (LFN)
    gsiftp://alnitak/home/Agrid/agrid064/.bashrc     # physical filename (PFN)
    rls://hydra.ari.uni-heidelberg.de                # RLS server hostname
```

The `globus-rls-cli` commands are multilined to improve readability and *not* in order to make them look unnecessarily complex. ;-) The client programm `globus-rls-cli` accepts up to four subcommands as well as up to five arguments; see **RLS** command reference in figure 1 on page 11. The above subcommand `create` expects three arguments, namely the user-defined unique logical filename (LFN) and the uniform resource locator (URL) pointing to the actual file on the hard disk. The last argument is the domain name of the host accomodating the **RLS** server. Note, that the server host must *always* be specified (no exception).

Let's immediately verify that the entry has been created:

```
# List all PFNs "behind" agrid064-dotbashrc (LFN)
# Command:  query lrc lfn <lfn> <rls://server-name>
globus-rls-cli
    query lrc lfn agrid064-dotbashrc
    rls://hydra.ari.uni-heidelberg.de

# Output
agrid064-dotbashrc:  gsiftp://alnitak/home/Agrid/agrid064/.bashrc
```

The **RLS** on `rls://hydra.ari.uni-heidelberg.de` contains one entry for `agrid064-dotbashrc` as we would expect. By the way, if we tried to query for `agrid064-dotbashrc` before the initial mapping was defined, the client would have answered:

```
# RLS error message: unknown LFN (query for undefined LFN)
globus_rls_client: LFN doesn't exist: agrid064-dotbashrc
```

Since **RLS** already knows that it's supposed to administer a couple of replica behind `agrid064-dotbashrc`, it's sufficient to just `add` (instead of `create`) new entries:

```
# Register two replicas with RLS
# Command: add <lfn> <pfn> <rls://server-name>
globus-rls-cli
    add agrid064-dotbashrc
    gsiftp://mintaka/home/Agrid/agrid064/.bashrc
    rls://hydra.ari.uni-heidelberg.de

globus-rls-cli
    add agrid064-dotbashrc
    gsiftp://eridanus/home/Agrid/agrid064/.bashrc
    rls://hydra.ari.uni-heidelberg.de
```

As an intermediate result we currently have three entries for our *.bashrc* file in our **RLS**:

```
# List again all PFNs "behind" agrid064-dotbashrc (LFN)
# Command: query lrc lfn <lfn> <rls://server-name>
globus-rls-cli
  query lrc lfn agrid064-dotbashrc
  rls://hydra.ari.uni-heidelberg.de

# Output
agrid064-dotbashrc: gsiftp://alnitak/home/Agrid/agrid064/.bashrc
agrid064-dotbashrc: gsiftp://eridanus/home/Agrid/agrid064/.bashrc
agrid064-dotbashrc: gsiftp://mintaka/home/Agrid/agrid064/.bashrc
```

Remark: Note, that until now, no file or replica has been transported to a storage location. All we did so far, was registering a file and two replica with **RLS**. In a **Globus**-based grid environment, file transfer is usually accomplished by means of `globus-url-copy`:

```
# File transfer by means of globus-url-copy
SOURCE=gsiftp://alnitak.uni-heidelberg.de/home/Agrid/agrid064/.bashrc
TARGET=gsiftp://storage.uni-heidelberg.de/home/Agrid/agrid064/.bashrc-copy

globus-url-copy $SOURCE $TARGET
```

According to the current status of knowledge, there is no `globus-url-move` in order to modify the file name or location and no `globus-url-delete` to wipe out files on the storage location.

Mappings, i.e. entries for replica locations, are removed from the "array" of actual filenames by means of the `delete` command:

```
# Delete a mapping
# Command: delete <lfn> <pfn> <rls://server-name>
```

```
globus-rls-cli
    delete agrid064-dotbashrc
    gsiftp://mintaka/home/Agrid/agrid064/.bashrc
    rls://hydra.ari.uni-heidelberg.de
```

Finally, after all replica mappings and the initial mapping have been deleted the LFN disappears from the **RLS** memory.

That said, we must not disesteem, that **RLS** is meant to be more than a mere file administration tool. Basically, **RLS** has been designed to administer any abstract data item that somehow belongs to the grid, among those, e.g. architecture and hardware equipment of individual grid resources. Now that we know how to set up mappings and add replica to **RLS**, let's find out how to comb through what **RLS** knows about our files. See table 1 on page 12 for a brief description of the placeholders used in `globus-rls-cli -help` or the full documentation under [8].

1.2.3 Wildcard search

As time passes users often feel uncertain about the identifiers they took for their files. Therefore **RLS** has a rudimentary support of searching with wildcards. Assume, grid user `agrid064` is quite disciplined and strictly takes `agrid064` to prefix his LFNs. Then the following command displays all LFNs starting with this substring as well as all URLs associated with those LFNs (output abbreviated):

```
# Search for all LFNs starting with ”agrid064” (wildcard search)
# Command:  query wildcard lrc lfn <lfn-pattern> <rls://server-name>
globus-rls-cli
    query wildcard lrc lfn
    agrid064*
    rls://hydra.ari.uni-heidelberg.de

# Output
agrid064-dotbashrc:  gsiftp://alnitak/home/Agrid/agrid064/.bashrc
agrid064-dotbashrc:  gsiftp://eridanus/home/Agrid/agrid064/.bashrc
          ⋮
agrid064-dotemacs:  gsiftp://vulpecula/home/Agrid/agrid064/.emacs
```

Similarly, **RLS** allows for wildcard searching with a given PFN substring (reverse lookup). Let's see who else registered his *.bashrc* with **RLS** (output abbreviated):

```
# Search for all ”.bashrc” files registered with RLS (reverse lookup)
# Command:  query wildcard lrc pfn <pfn-pattern> <rls://server-name>
globus-rls-cli
    query wildcard lrc pfn
    *bashrc
    rls://hydra.ari.uni-heidelberg.de

# Output
agrid042-dotbashrc:  gsiftp://alnitak/home/Agrid/agrid042/.bashrc
          ⋮
```

```
Usage:  globus-rls-cli [-c] [-h] [-l result-limit] [-s] [-t timeout]
                       [-u] [-v] [command] rls-url

  add                            <lfn>          <pfn>
  attribute add                  <object>       <attr>     <obj type>   <attr type> <val>
  attribute bulk     add         <object>  ...  <attr>     <obj type>   <attr type> <val>
  attribute bulk     delete      <object>       <attr>     <obj type> ...
  attribute bulk     query       <attr>         <obj type> <object>   ...
  attribute define               <attr>         <obj type> <attr type>
  attribute delete               <object>       <attr>     <obj type>
  attribute modify               <object>       <attr>     <obj type>   <attr type> <val>
  attribute query                <object>       <attr>     <obj type>
  attribute search               <attr>         <obj type> <op>         <attr type> <val>
  attribute show                 <attr>         <obj type>
  attribute undefine             <attr>         <obj type>
  bulk      add                  <lfn>          <pfn> ...
  bulk      create               <lfn>          <pfn> ...
  bulk      delete               <lfn>          <pfn> ...
  bulk      exists   lrc lfn     <val>          <lfns...>
  bulk      exists   lrc pfn     <val>          <pfns...>
  bulk      exists   rli lfn     <val>          <lfns...>
  bulk      query    lrc lfn     <lfns...>
  bulk      query    lrc pfn     <pfns...>
  bulk      query    rli lfn     <lfns...>
  bulk      rename   lfn         <lfn>          <lfn> ...
  bulk      rename   pfn         <pfn>          <pfn>...
  clear
  create                         <lfn>          <pfn>
  delete                         <lfn>          <pfn>
  exists lrc                     <obj type>     <val>
  exists rli         lfn         <lfn>
  exit
  help
  query  lrc         lfn         <lfn>
  query  lrc         pfn         <pfn>
  query  rli         lfn         <lfn>
  query  wildcard    lrc lfn     <lfn-pattern>
  query  wildcard    lrc pfn     <pfn-pattern>
  query  wildcard    rli lfn     <lfn-pattern>
  rename lfn                     <lfn>          <lfn>
  rename pfn                     <pfn>          <pfn>
  set    clearvalues true|false
  set    reslimit    reslimit
  set    timeout     timeout
```

Figure 1: Reference card for the command line interface `globus-rls-cli` to the **Globus Replica Location Service** (**RLS**).

```
agrid064-dotbashrc:  gsiftp://alnitak/home/Agrid/agrid064/.bashrc
              ⋮
```

Obviously, grid user agrid042 also decided to put his *.bashrc* under the control of **RLS**. Just a second, ... grid user agrid042? How can user agrid064 view what belongs exclusively to user agrid042? This brings up an interesting question. What about user permissions with **RLS**? According to the current state of knowledge, **RLS** has no permission concept by default, but we can emulate access permissions with respect to user names by means of **RLS** attributes (see next section).

Placeholder	represents
`<lfn>`	an LFN, e.g. `agrid064-dotbashrc`
`<pfn>`	a PFN (URL), e.g. *gsiftp://alnitak/home/Agrid/agrid064/.bashrc*
`<attr>`	an attribute name, e.g. `file_owner`
`<attr type>`	an attribute type; valid are `date`, `float`, `int` and `string`
`<val>`	the attribute value
`<object>`	an "object" (LFN or PFN) that has the considered attribute
`<obj type>`	the object type and is either `lfn` or `pfn`
`<lfn-pattern>`	a wildcard pattern to search for LFNs
`<pfn-pattern>`	a wildcard pattern to search for PFNs
`<op>`	a relational operator; valid are =, !=, >, >=, < or <=
`<lfns...>`	[no description in the command reference]
`<pfns...>`	[no description in the command reference]

Table 1: Placeholders used in `globus-rls-cli -help`, taken from [8]. The order of rows respects, that a notion is introduced before it is used.

1.2.4 File attributes

Traditional file systems provide meta data about their files and directories, e.g. ownership, permissions and timestamps concerning initial creation or recent access for each entry. Since **RLS** is not primarily an abstract file system these meta data are not present by default. **RLS** rather allows for maintaining a kind of data dictionary, i.e. a stock of independent properties which can be assigned to LFNs and PFNs. We want to create two attributes, namely `file_owner` and `file_creation_timestamp` and attach them to our replica:

```
# Create two attributes "file_owner" for string values and
# "file_creation_timestamp" for date values (format yyyy-mm-dd hh:mm:ss)
# Command:  attribute define <attr> <obj type> <attr type> <rls://server-name>
globus-rls-cli
    attribute define file_owner pfn string
    rls://hydra.ari.uni-heidelberg.de

globus-rls-cli
    attribute define file_creation_timestamp pfn date
    rls://hydra.ari.uni-heidelberg.de
```

RLS now knows about two attributes that may be referenced by PFNs. Note, that `file_owner` (type `string`) and `file_creation_timestamp` (type `date`) are defined as PFN-attributes, only, i.e. they cannot be assigned to LFNs. Put reverse, attributes are deleted from the "data dictionary" by just replacing `define` in the above command with `undefine`. Attributes are assigned to PFNs and initialized at the same time by means of:

```
# Assign two attributes to a PFN
# Command:  attribute add <object> <attr> <obj type> <attr type> <val>
#                         <rls://server-name>
globus-rls-cli
    attribute add                                   # assign an attribute
```

```
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc  # to this PFN (replicate)
        file_owner                                    # attribute name
        pfn                                           # entry (object) type
        string                                        # attribute type
        ”agrid064”                                    # attribute value
        rls://hydra.ari.uni-heidelberg.de

    globus-rls-cli
        attribute add
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc
        file_creation_timestamp
        pfn
        date
        ”2007-12-12 14:00:00”
        rls://hydra.ari.uni-heidelberg.de
```

The binding between an LFN or PFN and an attribute is dissolved by means of the `delete` sub-command. Assume, there were a redundant attribute `erroneously_assigned` and it contained the string ”attribute does not belong here”:

```
    # Display the properties of an assigned attribute
    # Command:  attribute query <object> <attr> <obj type> <rls://server-name>
    globus-rls-cli
        attribute query
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc    # this attribute has been
        erroneously_assigned                            # accidentally referenced
        pfn                                             # from this entry (object)
        rls://hydra.ari.uni-heidelberg.de

    # Output
    erroneously_assigned:  string:  attribute does not belong here
```

This output verifies, that PFN *gsiftp://alnitak/home/Agrid/agrid064/.bashrc* owns an attribute `erroneously_assigned`. Unbind the odd one out attribute from the PFN:

```
    # Delete an attribute from a PFN
    # Command:  attribute delete <object> <attr> <obj type> <rls://server-name>
    globus-rls-cli
        attribute delete
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc
        erroneously_assigned
        pfn
        rls://hydra.ari.uni-heidelberg.de
```

And verify, that the attribute has disappeared:

```
    # Try to display properties for a nonexistent attribute
    # Command:  attribute query <object> <attr> <obj type> <rls://server-name>
    globus-rls-cli
        attribute query
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc
        erroneously_assigned
        pfn
        rls://hydra.ari.uni-heidelberg.de

    # Output
    globus_rls_client:  Attribute doesn't exist:  gsiftp://alnitak/home/Agrid-
    /agrid064/.bashrc
```

Now that we know how to create attributes in and remove them from the "data dictionary", how to bind and unbind attributes to/from PFNs (similar for LFNs) and how to access attribute values, the last example shows how to change an already existing attribute value:

```
    # Current value of attribute file_owner
    globus-rls-cli
        attribute query
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc
        file_owner
        pfn
        rls://hydra.ari.uni-heidelberg.de

    # Output
    file_owner:  string:  agrid064

    # Change attribute value from agrid064 to agrid042
    # Command:  attribute modify <object> <attr> <obj type> <attr type>
    #           <rls://server-name>
    globus-rls-cli
        attribute modify
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc
        file_owner
        pfn
        string
        'agrid042'
        rls://hydra.ari.uni-heidelberg.de

    # New value value of attribute file_owner
    globus-rls-cli
        attribute query
        gsiftp://alnitak/home/Agrid/agrid064/.bashrc
        file_owner
        pfn
        rls://hydra.ari.uni-heidelberg.de

    # Output
    file_owner:  string:  agrid042
```

We are now prepared to give an answer to the question at the end of the previous section (*Wildcard search*), namely, how to filter the query output in order to display just the entries, that belong to a certain attribute value. The following command looks up all LFNs, where the `file_owner` attribute is set to `agrid064` (output abbreviated):

```
    # Search for all LFNs where the file_owner attribute equals "agrid04"
    # Command:  globus-rls-cli attribute search <attr> <obj type>
    #           <rls://server-name>
    globus-rls-cli
        attribute search file_owner lfn = string 'agrid064'
    rls://hydra.ari.uni-heidelberg.de

# Output
agrid064-dotbashrc:  gsiftp://alnitak/home/Agrid/agrid064/.bashrc
          ⋮
agrid064-dotbashrc:  gsiftp://eridanus/home/Agrid/agrid064/.bashrc
agrid064-dotbashrc:  gsiftp://mintaka/home/Agrid/agrid064/.bashrc
```

Note, that agrid064 is *not* a wildcard prefix in order to match LFNs beginning with that string, but an *attribute value*. According to the current state of knowledge, it seems to be impossible to combine wildcard search for LFNs or PFNs and filtering with respect to a certain attribute value, so that postprocessing of the query output is required, to meet the users needs. Furthermore it seems to be impossible to specify alternative values or patterns for attribute names, like

```
# *** Careful!  The following does NOT work!  ***
globus-rls-cli
    attribute search file_owner lfn = string 'agrid064|agrid042'
    rls://hydra.ari.uni-heidelberg.de

# Output
globus_rls_client:  Attribute with specified value doesn't exist:  file_owner
```

1.3 Drawbacks of RLS and Outlook

D3.3 is intended to subject the data management specified in [7], to thorough testing by means of selected use cases. The notion of data management concerns the supply of the software that is supposed to run on a grid resource together with the input data it might need (stage-in), the return transport of the output data after the program terminates (stage-out) as well as cleaning up the disk space on the grid resource afterwards. While several use cases implement the **Globus Resource Specification Language** (**RSL**) for job submission, apparently no use case has successfully implemented the **Globus Replica Location Service** (**RLS**) for replica management. The recent past provided evidence, that replica management by means of **RLS** for mapping logical file names (LFN) to physical file names (PFN) and globus-url-copy for file transfer is unsatisfactory for long-term file handling in **AstroGrid-D** for several causes.

RLS is not primarily intended to emulate a file system. Therefore, communication between **RLS** and the storage facilities is *not* mandatory. Because of that missing connection, there is no guarantee that a file under **RLS** control is really accessible, moreover, there is actually no guarantee at all, that the file even *exists*. Merely deleting a physical file without changing the corresponding **RLS** entry leads to a "dead link" from the users/jobs point of view, because **RLS** pretends that there *is* a file as long as its memory is refreshed. The bottom line is, that since **RLS** and the storage facilities don't talk to each other, **RLS** has no clue about presence or absence of the actual file. Hence, users/jobs should (at least) not exclusively rely on **RLS**.

The previous section demonstrates, that the **RLS** command line client globus-rls-cli is difficult to handle and hence not suited for common use. The client has about 40 combinations of commands, subcommands and command options, where up to five arguments are not yet counted; see figure 1 on page 11. Unfortunately, the client has no command line switches that would allow users to supply the arguments in a more intuitive order than the current version forces, e.g.

```
    <object> <attr> <object type> <attr type> <value>
```

instead of more self-documenting

```
    <object type> <object> <attr type> <attr> <value>.
```

The parameter list has been taken from `attribute modify`. Experience has tought, that the mere multitude of commands, subcommands and options on the one hand as well as the partial cryptic form of appearance on the other hand complicates working with `globus-rls-cli` and causes increased error-proneness. Tentatively spoken, `globus-rls-cli` is not a good example for user-friendliness.

Despite the above critique concerning **RLS**, replica management is important for **AstroGrid-D** in the near future, due to several reasons. Obviously, the presence of replica is beneficial at storage location or network connection failures, since users can simply switch to a different copy of the data they need. Moreover, replica are expected to reduce the traffic volume between distant grid resources. Jobs with a low ratio between CPU time and transfer time might benefit from shorter transfer durations and job descriptions are more reusable, if they rely on LFNs rather than on PFNs. Regarding the situation, it might be worthwhile, to abandon **RLS** and vote for a different approach instead.

2 Test of Use Cases Scenarios

2.1 Dynamo[3]

Dynamo is an application for solving the induction equation with turbulent electromotive force modeling the turbulent **Dynamo** in planets, stars and galaxies. Further details about this use case are given in [6]. As grid application **Dynamo** is run in task farming mode, where atomic binaries are submitted to compute resources. An input data file of 0.1 to 1 GB with initial parameters is provided for each job. At runtime the program generates output files with a total size of 0.1 to 10 GB at predefined time steps. Data analysis and visualization is done after the simulation has been completed and the output files have been retrieved. For long running simulations the latest output files are periodically retrieved for intermediate analysis. Status and progress of the program is written to `stdout/stderr` in ASCII format to allow monitoring.

The necessary transaction for each job is specified by means of the **Job Submission Description Language** (**JSDL**) which is converted before submission into the **Globus Resource Specification Language** (**RSL**) via the eXtensible Stylesheet Language (for) Transformations (XSLT).

For organizing the staging process, parameter files and executable files are located in a tree of directories: one directory contains a set consisting of the binary and parameter files for one run. This way, a whole suite of jobs can be prepared on some host, from where the staging is being done. The results could go into another directory tree. In the case described below, the stage in and stage out are done on the same host.

The first transaction is stage in of the executable file (*Dynamo.x*) and the initial parameter files (*ener* and *FFELD*). The corresponding **RSL** description is:

[3]Contributed by: Harry Enke (*henke@aip.de*)

```
<fileStageIn>

    <transfer>
        <sourceUrl>gsiftp://HOST/DEMO_HOME/uploadX/Dynamo.x</sourceUrl>
        <destinationUrl>
            file:///${GLOBUS_USER_HOME}/Dynamo/Dynamo.x
        </destinationUrl>
    </transfer>

    <transfer>
        <sourceUrl>gsiftp://HOST/DEMO_HOME/uploadX/input</sourceUrl>
        <destinationUrl>
            file:///${GLOBUS_USER_HOME}/Dynamo/input
        </destinationUrl>
    </transfer>

    <transfer>
        <sourceUrl>gsiftp://HOST/DEMO_HOME/uploadX/ener</sourceUrl>
        <destinationUrl>
            file:///${GLOBUS_USER_HOME}/Dynamo/ener
        </destinationUrl>
    </transfer>

    <transfer>
        <sourceUrl>gsiftp://HOST/DEMO_HOME/uploadX/FFELD</sourceUrl>
        <destinationUrl>
            file:///${GLOBUS_USER_HOME}/Dynamo/FFELD
        </destinationUrl>
    </transfer>

</fileStageIn>
```

Next the **Dynamo** binary is executed and the output data is produced. When the run is finished the output data folder (*resultsX*) is staged out as follows:

```
<fileStageOut>
    <transfer>
        <sourceUrl>file:///${GLOBUS_USER_HOME}/Dynamo/</sourceUrl>
        <destinationUrl>gsiftp://HOST//DEMO_HOME/resultsX/</destinationUrl>
    </transfer>
</fileStageOut>
```

Finally the input files (*ener* and *FFELD*) are removed from the execution host:

```
<fileCleanUp>
    <deletion>
        <file>file:///${GLOBUS_USER_HOME}/Dynamo/ener</file>
    </deletion>
    <deletion>
        <file>file:///${GLOBUS_USER_HOME}/Dynamo/FFELD</file>
    </deletion>
</fileCleanUp>
```

In a simple scenario the **Dynamo** application only uses **Globus** Staging capabilities, including the third-party-transfer feature. In an elaborated scenario, e.g. for the demo run with **Dynamo**, additional features of **Globus** are used:

- Job monitoring data (start/run/end) is being uploaded to **Stellaris** from the submission host for each job, making use of the **Globus** monitoring feature via `epr`.
- The job monitoring information stored in **Stellaris** is shown via the timeline and the grid resource map
- if the Interface Definition Language (IDL) backend is employed to show actual runtime status of calculated data, a gsissh channel is opened up to pass the data back to the submission host.

Since **Dynamo** currently serves mainly as a demo application, the intended scientific use as a means to scrutinize in parallel a vast range of parameters is not fully implemented. For this purpose, a suitable distributed file management with automatic generation of metadata would be convenient.

2.2 NBody6++[4]

NBody6++ is a member of a family of high accuracy direct N-body integrators used for simulations of dense star clusters, galactic nuclei, and problems of planet formation. A more detailed description of that use case can be found in *nbody.pdf*; available either in the *3_3/misc* SVN directory or at *http://www.gac-grid.org/project-documents/UseCases/nbody.pdf*.

The application is not very disc or data expensive. The input and output files vary between 100MB and a few GB per run. For a new run the simulation requires a parameter input file and an optional file for initial data of mass m, radius r and velocity v, depending on the settings in the parameter input file (`KZ(22)`). Another option is to use the integrated particle generator whereas in this case the only file required is the parameter file.

When using the restart (checkpointing) facility of **NBody6++** a so called common block file needs to be transferred to the execution host which in principle is a shapshot of a previous run. These binary files are usually the biggest files produced by the application.

The number of output files generated by **NBody6++** depends on the settings in the parameter input file. This implies that the **Job Submission Description Language** (**JSDL**) or **Globus Resource Specification Language** (**RSL**) job description should be generated dynamically to minimize the effort for the user.

2.2.1 Race conditions

When submitting more than one simulation to a **Globus** resource race conditions can occur since input and output files are not protected against mutual overwriting. This can be avoided by creating a sandbox for every job on the grid resource.

2.2.2 Implementation

At the time of this writing the current approach is to transfer the files via **GRAM-WS/RFT/GridFTP** (or `globus-url-copy`/**GridFTP** when using **GridWay**) by means of a job description. A

[4]Contributed by: Thomas Brüsemeister (*tbruese@ari.uni-heidelberg.de*)

deployment package has been developed to perform a hot deployment of **NBody6++** and to submit the job using a job description generated by a script. It is also possible to submit jobs (transparently) to **GridWay** through **GridGateWay** which is primarily a **RSL** to job template (JT) translator. Unfortunately using **GridGateWay** works not completely transparent. When deploying applications one has to be aware of the side effects occuring due to the fact that **GridWay** creates its own sandbox and that files are treated differently depending on whether the file is specified as an absolute path or relative path in the job template.

```
=== The NBody6++ deployment package ===
$ ./submit.sh
Submits NBody6++ jobs to Globus nodes.
Usage:  ./submit.sh [options] <parameter-file>

Options:
  -d                    Delegate full credential.  [no]
  -g host               Submits the job to <host>.  [hydra.ari.uni-heidelberg.de]
  -h                    Print this help.
  -m                    Enable MPI. (Experimental).
  -n                    Disable batch mode.
  -q queue              Use queue <queue>.
  -s                    Enable job statistics (Experimental).
  -t job-manager        Use <job-manager> as Globus Job Manager.  [GW]

Stage-in Options:
  -fr file              Stage-in a common-block file for restart.
  -fd file              Stage-in a file for initial data of m,r,v.

Example:  ./submit.sh -d var/in1000.comment
          (Option -d must be used to provide a proxy for GridWay.)

NBody6++ deployment package for AstroGrid-D, v0.2.0-pre ($Revision:  35 $)
```

2.2.3 Future considerations

The deployment package will be extended to support the **Open Grid Forum** (**OGF**) standard **JSDL**. Furthermore, it is planned to use a replication management system to reduce the network costs (transfer time) and to improve reliability. When submitting **GridWay** jobs through **GridGateWay** the unnecessary overhead due to the two-hop filestaging will also be eliminated.

2.3 GEO600[5]

2.3.1 Distributed data management

Each **GEO600** task requires a single CPU core and has its own working directory of approximately 100MB in size. This directory is archived after each run has finished and staged-out to a storage location. Upon restart of the task, the working directory is again staged-in to the grid resource.

[5]Contributed by: Robert Engel (*engro@aei.mpg.de*)

2.3.2 Storage locations

In August 2007 we started production runs of **GEO600** on 32 different grid resources, 14 of these were large clusters. Our initial storage location was *gsiftp://buran.aei.mpg.de/store/GEO600/tasks* where all task archives were located. Staging in and out of data is specified entirely in the **Globus RSL** provided for each run and carried out by the **GRAM-WS** on each resource which relies on **GridFTP**.

We soon realized that the simultaneous start of more than a few ($\sim$10) tasks would not be possible due to network limitations (100MBit/s) and hard disk IO rate ($\sim$20MB/s) of the **GridFTP** server `buran`. These limitations were directed against scaling the use case to more than just a few runs a day.

It was therefore decided to move all archives to the astrodata storage location, which just became available at the time. Our initial assumption was, that the ten machines `astrodata-01.gac-grid.org ... astrodata10.gac-grid.org` would be able to load balance the stage-in and stage-out load and allow us to further scale up the use case by a factor 10. This goal could not be met, since the network IO of these 10 machines did not accumulate and that even though fast hard disk IO was available, network IO seemed to be as limited as on `buran`. Several problems due to availability of the astrodata storage location, made us soon look for a different solution.

The present solution is, that each large resource (cluster) usually provides sufficient storage space either in `$HOME` or at some Network Attached Storage (NAS) solution, that we use to keep our task archives on side. Further more these storage locations located at 14 different clusters accumulate in terms of network IO, hard disk IO and availability and can therefore be used by other grid resources as storage locations. In this way the german grid is in fact our distributed storage location, providing high availability and high IO rates.

2.3.3 Replica management

Each task we run on the grid is associated with an entry in our MySQL database running at `buran.aei.mpg.de:24999`. The information in the database keeps track of all our job submissions, collects statistics and further more acts as a simple service, storing the actual URI location of *stdout*, *stderr*, *log* and the archive for each task. If the task is running, the URI will point to the actual location of the *stdout*, *stderr* and *log* file. Otherwise it will point to the URI of the storage location.

2.3.4 Statistics

We are currently running 1500 jobs simultaneously on D-Grid resources. Each job runs for about 10 hours, which means that we submit 3000 jobs in average per day. Each job transfers *stdout*, *stderr*, *log* to *gsiftp://buran.aei.mpg.de/store/GEO600/tasks*. This equals a network stage-out load of 360MB per day to buran originating at the grid resources we use.

Each task acts on about 100MB of data. We currently have 6000 tasks in our database, which together accumulate 600GB of distributed storage space on the grid resources we use.

Only 100 of these tasks run on smaller grid resources every day and actively stage-in and stage-out their archives from and to *gsiftp://astrodata09.gac-grid.org/store/01/aei/GEO600* where we use

4GB of space and create an IO of no more than 20GB per day.

2.3.5 Summary

GEO600 requires not a huge amount of storage space to operate in D-Grid (∼500GB). Due to the large number of jobs we run, **GEO600** creates a high IO load during stage-in and stage-out of data at the various grid resources. This IO load if directed to one storage location would create an IO rate of 600GB per day. Unfortunately this happens not to be a continuous rate, but happens to be in chunks of ∼10GB at a time, which of course can not be handled by a single storage location without receiving network connection timeouts. For this reason **GEO600** combines a number of grid resources to form a distributed storage location serving the **GEO600** use case. A simple MySQL database is keeping track of the various files at the various grid locations. Ever since we were able to scale up our use case from a few runs par day to 3000 job submissions per day at the present time. For all data transfers we use **GridFTP** through **GRAM-WS/RSL**.

A Questionaire and Input from Selected AstroGrid-D Use Cases about Distributed File Management[6]

A questionaire was sent to the users running the selected applications with the following questions.

1. *Question*: How are you currently transferring input and output data?
   A. **GridFTP**
   B. Hypertext Transfer Protocol (**HTTP**)
   C. File Transfer Protocol (**FTP**)
   D. Replica Location Service (**RLS**)
   E. `globus-url-copy`
   F: one hop mode of **GridGateWay**
   G: two hop mode of **GridGateWay**
   H: Other? In that case how?
2. *Question*: Are you using logical file names that is given by a namespace with a Uniform Resource Identifier (URI)-prefix? If no, do you have any plans on implementing support for this before the end of the project?
3. *Question*: What automated staging methods are you using?
   A. Input staging
   B. Output staging
   C. Input and output staging
   D. Input and output staging with intermediary storage
   E. If not any of the above, then why?
4. *Question*: Are you using **Stellaris** to store and query meta data? If not, how are you currently organizing your meta data?

[6]Contributed by: Jürgen Steinacker (*stein@mpia-hd.mpg.de*), Mikael Högqvist (*hoegqvist@zib.de*), Rainer Spurzem (*spurzem@ari.uni-heidelberg.de*), Frank Breitling (*fbreitling@aip.de*), Art Carlson (*awc@mpe.mpg.de*), Robert Engel (*engro@aei.mpg.de*), Harry Enke (*henke@aip.de*), Iliya Nickelt (*inickelt@aip.de*) and Thomas Radke (*tradke@aei.mpg.de*)

5. *Question*: Are you using the core **Resource Description Framework** (**RDF**) vocabulary suggested in D3.2? If not, are you describing your data-sets and how are you describing them?

6. *Question*: Do you have general comments or wishes concerning the data transfer or data management within your use case?

The feedback to questionaire is summarized in the following.

- **GridFTP**/`globus-url-copy` are basic tools used by all use cases to do input/output staging. Therefore, it will be important to guarantee that a submission by **GridWay**/**Globus** in the next step is working properly. Furthermore, it will have to be ensured that entire directories can be staged in and out without problems.
- Most use cases do not need logical file names (LFN). The advantage of removing the physical file location is more useful for output files that may be large and that will be re-used or that are stored at multiple locations for reliability. There is no guarantee that a file "hiding" behind an LFN will actually be stored at the system creating inconsistencies. One way of dealing with this is to create a separate service that "scrubs" the LFNs by checking if the files still exist.
- **Stellaris** is not used for metadata about files, more focus seems to be on how to record metadata on jobs. This may be useful from a monitoring perspective and also to find the data produced by a job. However, describing jobs can be generalized while the metadata for the produced data is more specific.
- Most use cases rely on a hierarchical naming structure which is sufficient for a low number of output-data. For a large number of executions on more sites, **Stellaris** or any way to search within the metadata becomes necessary.
- The use cases are in very different stages concerning the perspectives to the usage of a more general file management for their application. This will aggravate the implementation of the steps beyond the current usage of LFNs and **GridFTP**+`globus-url-copy`.
- Concerning XML-files, an interesting procedure for the file management is to store the files in the filesystem hierarchy and automatically index metadata from them into **Stellaris**. The files are then uploaded and retrieved via a service interface.

References

[1] AstroGrid-D Data Management (ADM) build-in documentation, accessible by means of `adm help` (general information) or `adm help <subcommand>` (manual page for individual subcommand).

[2] Cai, Min et al.: *A Peer-to-Peer Replica Location Service Based on A Distributed Hash Table*, see file *sc2004v15.pdf* located in directory *3_3/misc*.

[3] Chervenak, Ann L. et al.: *Performance and Scalability of a Replica Location Service*, see file *chervenakhpdc13.pdf* located in directory *3_3/misc*.

[4] Chervenak, Ann L. et al.: *Giggle: A Framework for Constructing Scalable Replica Location Services*, see file *giggle.pdf* located in directory *3_3/misc*.

[5] Collins-Sussman, Ben; Fitzpatrick, Brian W. and Pilato, C. Michael: *Version Control with Subversion*, for Subversion 1.4 (Compiled from r2866), see *http://svnbook.red-bean.com* or file *svn-book.pdf* located in directory *3_3/misc*.

[6] Deliverable 3.1: *AstroGrid-D Distributed File Management, Requirements Specification and Architectural Design*, (Version 1.0.0, changes according to feedback from the project).

[7] Deliverable 3.2: *Distributed File Management, Data- and Replica-management in AstroGrid-D*, (Version 1.0.0, public release with comments incorporated).

[8] The official RLS website: *GT Data Management: Replica Location Service (RLS)* at *http://-www.globus.org/toolkit/data/rls*. (Consider that the former website *http://www.globus.org/-rls/* is no longer valid.)