Implementations: Practical Policy Working Group, February 2015

Version: February 4, 2015

Abstract

The RDA Practical Policy Working Group was founded in Sept. 2012. The following goals were reached:

- Collection of policies in the RDA Wiki
- Categorization of policies
- Survey of the 11 most popular policy sets in institutions
- Description of policy templates for these 11 policy sets
- English Language Descriptions and implementation examples for selected policies

Furthermore an additional Interest Group will be founded to provide various testbeds for all RDA WG/IGs to demonstrate implementations of policy sets and interoperability.

In this document English Language Descriptions and implementation details of selected policies are described.

Table of Contents

1. Introduction

This document describes implementation examples of the policies provided in the document "Outcomes Policy Templates: Practical Policy Working Group, September 2014".
The example policies include computer actionable rules written in the integrated Rule Oriented Data System rule language, policies defined using an XML schema, and policies used on GPFS. Since GPFS is a file system, not a Data Management System, there are some pitfalls such as non-existence of event-based operations and no metadata collection and storage. Pitfalls will be mentioned within each affected policy.

The generic policy areas, common to almost all data management systems, are:

1. Contextual metadata extraction
2. Data access control
3. Data backup
4. Data format control
5. Data retention
6. Disposition
7. Integrity (including replication)
8. Notification
9. Restricted searching
10. Storage cost reports
11. Use agreements

Each of the generic policy areas actually represents a set of policies. Policies are needed to set environmental variables that control the execution of the policy; to enforce desired collection properties; and to validate assessment criteria.

Each policy example can be modified to implement the specific policy required by an institution. Thus the policies should be treated as examples of approaches for controlling a desired property of a data management system.

In Appendix A, the XML schema used by EUDAT to define policies is listed. The schema lists the attributes and elements that are combined to define policy terms. The associated EUDAT policies are listed in the policy examples for each policy area.

2. Contextual metadata extraction policies

2.1 Example GPFS extraction policy

In GPFS there is no straightforward way to store provenance and descriptive metadata, since GPFS is simply a filesystem. The only way you can directly store some metadata with a file is by using extended attributes, e.g.:

```
mmchattr --set-attr name=value yourfile.txt
```

This way you can store up to 4KB of text data with the file.

2.2 Examples iRODS extraction policies

Rule examples are provided that are written using the iRODS rule language [1]. Each rule that is run interactively has a rule name, a rule body enclosed in braces, INPUT variables, and OUTPUT variables. Note that "ruleExecOut" on an OUTPUT line will copy the output information to the user's screen.

Rules that are applied at Policy-Enforcement-Points have a standard rule name related to the specific action that is being controlled. The INPUT variables are replaced with session variables that track who is executing an external action. Rules can query a metadata catalog to retrieve information about the collection, the users, the storage systems, and user-defined metadata. In many of the following examples, a query is made to the metadata catalog, a "foreach" loop is then used to process the rows returned from the query, parameters are extracted from the row using a "." structure, and information is output using a writeLine micro-service. More information on the iRODS rule language can be found at http://irods.org.

2.2.1 Contextual metadata extraction through pattern recognition

English language description:

Pattern matching operations can be applied to text to extract contextual metadata. A template for pattern matching can be created that defines triplets:

<pre-string-regexp, keyword, post-string-regexp>.

The triplets are read into memory, and then used to search a data buffer. For each set of pre and post regular expressions, the string between them is associated with the specified keyword and can be stored as a metadata attribute on the file.

In the example, the template file has the format:

```
<PRETAG>X-Mailer: </PRETAG>Mailer User<POSTTAG>
</POSTTAG>
<PRETAG>Date: </PRETAG>Sent Date<POSTTAG>
</POSTTAG>
<PRETAG>From: </PRETAG>Sender<POSTTAG>
</POSTTAG>
<PRETAG>To: </PRETAG>Primary Recipient<POSTTAG>
</POSTTAG>
```

```
<PRETAG>Cc: </PRETAG>Other Recipient<POSTTAG>
</POSTTAG>
<PRETAG>Subject: </PRETAG>Subject<POSTTAG>
</POSTTAG>
<PRETAG>Content-Type: </PRETAG>Content Type<POSTTAG>
</POSTTAG>
```

The end tag is actually a "return" for unix systems, or a "carriage-return/line-feed" for Windows systems. The example rule reads a text file into a buffer in memory, reads in the template file that defines the regular expressions, and then parses the text in the buffer to identify presence of a desired metadata attribute.

iRODS implementation:

```
myTestRule {
# Input parameter is:
#  Tag buffer
# Output parameter is:
#  Tag structure

 # Read in first 10,000 bytes of file
 msiDataObjOpen(*Pathfile,*F_desc);
 msiDataObjRead(*F_desc,*Len,*File_buf);
 msiDataObjClose(*F_desc,*Status);

 # Read in tag template
 msiDataObjOpen(*Tag,*T_desc);
 msiDataObjRead(*T_desc, 10000, *Tag_buf);
 msiReadMDTemplateIntoTagStruct(*Tag_buf,*Tags);
 msiDataObjClose(*T_desc,*Status);

 # Extract metadata from file using tag template
 msiExtractTemplateMDFromBuf(*File_buf,*Tags,*Keyval);

 # Write result to stdout
 writeKeyValPairs("stdout", *Keyval," : ");

 # Add metadata to the file
 msiGetObjType(*Outfile,*Otype);
 msiAssociateKeyValuePairsToObj(*Keyval,*Outfile,*Otype);
}
INPUT *Tag="/$rodsZoneClient/home/$userNameClient/test/email.tag",
*Pathfile="/$rodsZoneClient/home/$userNameClient/test/sample.email",
*Outfile="/$rodsZoneClient/home/$userNameClient/test/sample.email", *Len=10000
OUTPUT ruleExecOut
```

2.2.2 Load metadata from an XML file

English language description:

Metadata can be loaded into a data grid directly from an XML file. This policy assumes a specific structure for the XML file of the form:

iRODS implementation:

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
 <AVU>
  <Target>/$rodsZoneClient/home/$userNameClient/XML/sample.xml</Target>
  <Attribute>Order ID</Attribute>
  <Value>889923</Value>
  <Unit />
 </AVU>
```

```
  <AVU>
   <Target>/$rodsZoneClient/home/$userNameClient/XML/sample.xml</Target>
   <Attribute>Order Person</Attribute>
   <Value>John Smith</Value>
   <Unit />
  </AVU>
 </metadata>
```

Note that this specifies the target file to which the metadata is added. Each metadata attribute, value, and unit is formed into an AVU that is attached as metadata to the file.

iRODS implementation:

```
myTestRule {
# Input parameters are:
# targetObj   - iRODS target file that metadata will be attached to, null if Target is specified
# xmlObj      - iRODS path to XML file that metadata is drawn from
#
#    xmlObj is assumed to be in AVU-format
#    This format is created by transforming the original XML file
#       using an appropriate style sheet as shown in rulemsiXsltApply.r
#    This micro-service requires libxml2.
  # call the micro-service
  msiLoadMetadataFromXml(*targetObj, *xmlObj);
  # write message to the log file
  writeLine("serverLog", "Extracted metadata from *xmlObj and attached to *targetObj");
  # write message to stdout
  writeLine("stdout", "Extracted metadata from *xmlObj and attached to *targetObj");
}
INPUT *xmlObj="/$rodsZoneClient/home/$userNameClient/XML/sample-processed.xml", *targetObj=""
OUTPUT ruleExecOut
```

3. Data access control policies

3.1 Example GPFS policy for data access control:

GPFS has support for POSIX or NFSv4 access control lists, which can be used to control access to files. You simply set an appropriate ACL for a file or directory.

Give user *Bob* and group *audit* full access to file *project.txt* but exclude others:

```
mmputacl -i project.acl project.txt
```

where file *project.acl* contains:

```
user::rwx
group::---
other::---
mask::rw-
user:bob:rwx
group:audit:rwx
```

3.2 Examples iRODS policies for data access control:

These policies can be applied interactively to files within a collection, or can be automated as part of a file ingestion process.

3.2.1 Find the User_ID associated with a User_name:

English language description:

Since identifiers for users may be set as either strings (USER_NAME) or integers (USER_ID), a policy that allows a person to find the USER_ID for their USER_NAME is useful. This policy queries a metadata catalog, and retrieves the USER_ID for the person who is running the rule. The output is written to the screen.

iRODS implementation:

```
myTestRule {
#List information about the person running the rule
  *Query = select USER_ID where USER_NAME = '$userNameClient';
  foreach (*Row in *Query) {
    *userid = *Row.USER_ID;
    writeLine("stdout", "User: $userNameClient  UserID: *userid");
  }
}
INPUT null
OUTPUT ruleExecOut
```

3.2.2 Find the File_ID associated with a file name:

English language description:

Since identifiers for files may be set as either strings (DATA_NAME) or integers (DATA_ID), a policy that finds the DATA_ID for a file is useful. This policy queries a metadata catalog, and retrieves the DATA_ID for a specified file name that is input to the rule. The result is written to the screen.

iRODS implementation:

```
myTestRule {
# find the DATA_ID associated with a file name
 *Coll = "/$rodsZoneClient/home/$userNameClient/" ++ *RelativeCollectionName;
 *Query = select DATA_ID where DATA_NAME = '*File' and COLL_NAME = '*Coll';
 foreach(*Row in *Query) {
  *Dataid = *Row.DATA_ID;
  writeLine("stdout", "Collection *Coll, File *File, File ID *Dataid");
 }
}
INPUT *File = 'foo1', *RelativeCollectionName = 'test'
OUTPUT ruleExecOut
```

3.2.3 Set write access control for a user:

English language description:

An administrator can set an access control on a file by specifying the file name, the desired access control, and the user name. This policy reads as input the user name, the collection and file on which the access control is set, and the desired access control. A metadata catalog is updated to record the change in access control.

iRODS implementation:

```
myTestRule {
# Input parameters are:
#  Recursion flag
#    default
#    recursive  - valid if access level is set to inherit
#  Access Level
#    null
#    read
#    write
#    own
#    inherit
#  User name or group name who will have ACL changed
#  Path or file that will have ACL changed
 *Home="/$rodsZoneClient/home/$userNameClient/";
 *Path= *Home ++ *RelativeCollection ++ "/" ++ *File;
 msiSetACL("default", *Acl,*User,*Path);
 writeLine("stdout","Set owner access for *User on file *Path");
}
INPUT *User="testuser", *RelativeCollection="test", *File="foo1", *Acl = "write"
OUTPUT ruleExecOut
```

3.2.4 Set inheritance of access controls on a collection

English language description:

Access controls on a file can be inherited from the collection into which the file is organized. This rule reads as input the collection name and then sets an "inherit" flag on the collection. Files that are deposited into the collection will "inherit" the access controls that were set on the collection.

iRODS implementation:

```
myTestRule {
# Input parameters are:
#  Recursion flag
#    default
#    recursive  - valid if access level is set to inherit
#  Access Level
#    null
#    read
#    write
#    own
#    inherit
#  User name or group name who will have ACL changed
#  Path or file that will have ACL changed
  *Home="/$rodsZoneClient/home/$userNameClient/";
  *Path= *Home ++ *RelativeCollection;
  msiSetACL("recursive", *Acl,*User,*Path);
  writeLine("stdout","Set inheritance of access on collection *Path");
}
INPUT *RelativeCollection="test", *Acl = "inherit", *User=""
OUTPUT ruleExecOut
```

3.2.5 Set operations that are allowable for the user "public"

English language description:

This policy controls the operations that "public" users are allowed to execute. Only 2 operations are allowed -"read" - read files; "query" - browse some system level metadata. This uses the micro-service "msiSetPublicUserOpr" to specify what types of public access operations are allowed. The micro-services is called from a policy enforcement point associated with setting Public User Policy.

iRODS implementation:

```
acSetPublicUserPolicy {msiSetPublicUserOpr("read%query"); }
```

3.2.6 Check the access controls on a file:

English language description:

This policy is intended for use as a subroutine within other policies. This rule reads as input a collection and file for which access controls will be checked. The desired access permission is compared with the access permissions set on the file. If the access control is not found, an error message is written.

iRODS implementation:

```
myTestRule {
#Input parameters are:
#  Name of object
#  Access permission that will be checked
#Output parameter is:
#  Result, 0 for failure and 1 for success
 *Path = "/$rodsZoneClient/home/$userNameClient/" ++ "*Coll" ++ "/" ++ "*File";
 msiCheckAccess(*Path,*Acl,*Result);
 if(*Result == 1) {
  writeLine("stdout","Access is allowed");
 }
 else {
  writeLine("stdout","Access is not allowed");
 }
}
INPUT *Coll =$"Rules", *File =$"ruleintegrityACL.r", *Acl =$"own"
OUTPUT ruleExecOut
```

3.3 EUDAT policies

The EUDAT schema is listed in Appendix A. This defines the terms that are used to specify policies within the EUDAT federated environment. Example EUDAT policies can then be written using the terms from the EUDAT schema. Each EUDAT policy is interpreted, and converted into a computer actionable rule.

3.3.1 EUDAT persistent identifier policy

English language description:

This policy specifies that a persistent identifier will be created when objects are copied to another site. The specification includes information about the collection and the second site, whether object de-duplication is needed, how to find the PID of the original file, and the process to update the PID of the copied file.

EUDAT implementation in iRODS:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   this is a policy template to preserve PID after object copies internally to site B. The sync is performed on copy.
   1) check object duplication
   2) search PID related to original copy
   3) update URL of PID record
   4) remove original copy
-->

<policy name="data movement - PID preservation" version="1.0" author="Claudio Cacciari"
 uniqueid="eb9f34e0-0r27-45e8-8132-eceahf70d40d" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
 xmlns="http://eudat.eu/2013/policy" xmlns:irodsns="http://eudat.eu/2013/iRODS-policy">
  <dataset>
   <collection id="0">
    <location xsi:type="irodsns:coordinates">
     <irodsns:site type="EUDAT"><!-- B --></irodsns:site>
     <irodsns:path><!-- /path/to/destination1 --></irodsns:path>
     <irodsns:resource><!-- defaultResc --></irodsns:resource>
```

```
    </location>
            </collection>
   <!--
   add here further collections, if needed
   -->
  </dataset>
  <actions>
   <action name="check object duplication">
    <type>object search</type>
    <trigger type="action"><!-- onCopy --></trigger>
    <targets>
     <target id="1">
      <location xsi:type="irodsns:coordinates">
       <irodsns:site type="EUDAT"><!-- B --></irodsns:site>
       <irodsns:path><!-- /path/to/destination2 --></irodsns:path>
       <irodsns:resource><!-- defaultResc --></irodsns:resource>
      </location>
     </target>
    </targets>
   </action>
   <action name="search PID related to original copy">
    <type>URL PID search</type>
    <trigger type="action"><!-- check object duplication --></trigger>
    <targets>
     <target ref="1"></target>
    </targets>
   </action>
   <action name="update URL of PID record">
    <type>URL PID update</type>
    <trigger type="action"><!-- search PID related to original copy --></trigger>
    <!--
    assuming the PID comes from the action "search PID related to original copy"
    -->
    <targets>
     <target ref="0"></target>
    </targets>
   </action>
   <action name="remove original copy">
    <type>remove</type>
    <trigger type="action"><!-- update URL of PID record --></trigger>
    <targets>
     <target ref="1"></target>
    </targets>
   </action>
  </actions>
</policy>
```

4. Data backup policies

4.1 Example GPFS backup policy

To protect against potential data loss, GPFS comes with a utility called `mmbackup`. This utility is used to back up a filesystem. The `mmbackup` utility requires a Tivoli Storage Manager (TSM) Server accessible on the network and a Tivoli Storage Manager Backup-Archive Client must be installed locally and configured to use the TSM server.

Once this setup is in place `mmbackup` scans the filesystem and compares the results with the previous scan. Newly created or modified files are backed up. Files that were deleted are expired by the TSM Server. A common practice is to set up a cron job to periodically run the `mmbackup` utility in order to ensure required backup schedule.

Given a filesystem *research* and TSM Server called *SERVER1*, run `mmbackup` to do a backup of that filesystem:

```
mmbackup research --tsm-servers SERVER1
```

4.2 Example iRODS backup policy

Within the iRODS data grid, backups, copies, and replicas can be supported. The difference is the set of state information that is needed for each type of entity. A backup is a time-stamped copy of a file. A replica is an additional copy of a file that is stored on a separate storage system. The replica number is tracked and whether the original has been changed. Generic state information includes a creation time for the data object, the location where the data object is stored, the owner of the data object, modification time stamps, and access controls. An outcome of this approach is that it is possible to use the same client to access backups, copies, and replicas.

4.2.1 Data backup staging policy

English language description:

This rule creates a time-stamped backup directory, and copies all of the files from the source directory to the backup directory. The rule reads from input the collection for which the backup will be done, the storage location where the backups will be stored, and the destination collection that will hold the backup. Within the destination collection, a time-stamped sub-directory is created to hold each backup set. The rule checks the input, checks that each operation completes correctly, and writes information to a server log.

iRODS implementation:

```
myTestRule {
```

```
# Test delayed execution
 *Home = "/$rodsZoneClient/home/$userNameClient/";
 *Source = *Home ++ *Collrel;
 writeLine("stdout","Backup collection *Source");
 delay("<PLUSET>1s</PLUSET><EF>30s</EF>") {
  msiGetSystemTime(*Time,"human");
# Create backup collection with name *Dest/Check-Timestamp
#=============get current time, Timestamp is YYY-MM-DD.hh:mm:ss  ======================
  msiGetSystemTime(*TimeH,"human");
  *Dest = *Home ++ *Destrel;
#========== check whether a destination collection was specified ==================================
  msiIsColl(*Dest,*Result, *Status);
  if(*Result == 0 || *Status < 0) {
   writeLine("stdout","Input path *Dest is not a collection");
   fail;
  }     # end of status check
#============ create a collection for backup if it does not exist ===============
  *Lpath = *Dest ++ "/" ++ *TimeH;
  msiIsColl(*Lpath,*Result, *Status);
  if(*Result == 0 || *Status < 0) {
   msiCollCreate(*Lpath, "0", *Status);
   if(*Status < 0) {
    writeLine("serverlog","Could not create backup collection");
    fail;
   }  # end of check on status
  }  # end of backup collection creation
  writeLine("serverLog","Created Backup for *Source at *TimeH");
  msiCollRsync(*Source,*Lpath,*Resource, "IRODS_TO_IRODS", *Status);
  if(*Status < 0) {
   writeLine("serverlog","Backup failed at *TimeH");
  }
 }
}
INPUT *Collrel = "test", *Destrel = "back1", *Resource = "testResc"
OUTPUT ruleExecOut
```

5. Data format control policies

5.1 Example GPFS format control policy

We are using the utility `mmapplypolicy` with a custom made policy and a custom made script. `mmapplypolicy` scans the filesystem for files with the specified extension (wildcard character '%' stands for an arbitrary number of characters) and then runs the script to notify the administrator. It must be run explicitly, since GPFS doesn't provide necessary triggers.

The following policy checks the staging area of a filesystem (filesystem is called '*research*') for files with extension "*.r*" and notifies administrator via e-mail (sends the list of paths to the files that are found). E-mail sending is done by an external script called '*/tmp/execScript*'.

The list of file paths selected by the policy is passed to this script by defining an '*EXTERNAL LIST*' rule.

GPFS policy file (called *extension.policy*):

```
RULE EXTERNAL LIST 'doSomething' EXEC '/tmp/execScript'

RULE 'ListFilesWithExtension' LIST 'doSomething'
WHERE(PATH_NAME LIKE '/research/staging/%.r')
```

Script (called *execScript*):

```
#!/bin/bash

cmd=$1
list=$2
tempfile=$(mktemp)

if [[ "$cmd" = "TEST" ]]; then
       exit 0
fi

if [[ "$cmd" = "LIST" ]]; then
       /bin/awk '{print $5}' $2 > $tempfile
       /bin/mail -s "Files with .r extension in staging area"
"administrator@example.com" < $tempfile
fi

rm -f $tempfile
```

You can run the example by issuing this command (filesystem is called '*research*'):

```
mmapplypolicy research -P extension.policy
```

5.2 Example iRODS format control policy

File format type is stored in a state information variable called DATA_TYPE_NAME. Queries can be issued against the metadata catalog to retrieve files with a given format type. Operations are supported for extracting the file format type of a file, based on the file extension.

5.2.1 Identify and archive specific file formats from a staging area

English language description:

This policy examines a staging area for files with a specific format type. The file format is determined from the file extension. Files that have a desired extension, in this case an extension ".r", are moved into a specified collection. This makes it possible to sort files by file format type. The collection that correponds to the staging area and the collection that corresponds to the destination collection are read from input.

iRODS implementation:

```
myStagingRule {
# Loop over files in staging area, /$rodsZoneClient/home/$userNameClient/*stage
# Put all files with .r into collection /$rodsZoneClient/home/$userNameClient/*Coll
 *Src = "/$rodsZoneClient/home/$userNameClient/" ++ *Stage;
 *Dest= "/$rodsZoneClient/home/$userNameClient/" ++ *Coll;
 *Query = select DATA_NAME where COLL_NAME = '*Src' and DATA_NAME like '%.r';
 foreach(*Row in *Query) {
  *File = *Row.DATA_NAME;
  *Src1 = *Src ++ "/" ++ *File;
  *Dest1 = *Dest ++ "/" ++ *File;
#Check whether file already exists
  msiIsData(*Dest1,*DataID,*Status);
# Move file and set access permission
  if(*DataID == "0") {
   msiDataObjAutoMove(*Src1,*Src,*Dest, $userNameClient, "true");
   msiSetACL("default","own",$userNameClient, *Dest1);
   writeLine("stdout", "Moved file *Src1 to *Dest1");
  }
 }
}
INPUT *Stage =$"stage", *Coll=$"Rules"
OUTPUT ruleExecOut
```

6. Data retention policies

6.1 Example GPFS data retention policy

This policy archives the files older than a month to a special storage pool called '*archive*'. Files are originally located in the '*system*' storage pool. It must be run explicitly, since GPFS doesn't provide necessary triggers.

We use the `mmapplypolicy` utility to scan the filesystem (here called '*research*'). Files older than a month are then migrated by the `mmapplypolicy` utility to a storage pool designed for archiving.

GPFS policy file (called *archive.policy*) :

```
RULE 'MigrateToArchive' MIGRATE
FROM POOL 'system'
TO POOL 'archive'
WHERE KB_ALLOCATED > 0
AND (DAYS(CURRENT_TIMESTAMP) - DAYS(ACCESS_TIME)) > 30
```

You can run this by issuing the following command (filesystem is called '*research*') :

```
mmapplypolicy research -P archive.policy
```

6.2 Examples iRODS cache purge policies

The iRODS data grid specifies a data expiration date in the metadata attribute "DATA_EXPIRY". The expiration date is stored as a Unix time variable. Information about the creation time of each file is stored in the metadata attribute DATA_CREATE_TIME.

6.2.1 Purge policy to free storage space

English language description:

This policy manages a cache to ensure that a minimum amount of free space is available for deposition of new files. The policy runs periodically, every 24 hours. An information catalog is queried to find the total amount of storage space that is being used. This is compared to an input parameter that specifies the maximum allowed space. Additional input parameters specify the collection and the storage resource names. A second query retrieves information about the file names, file sizes, and creation time. The result set is ordered by the creation date, making it possible to loop over the files, deleting the oldest files until the required free space is available.

iRODS implementation:

This policy was developed by Jean-Yves Nief of the French National Institute for Nuclear Physics and Particle Physics Computer Center. This rule could be modified to purge old backup directories.

```
purgeDiskCache {
  delay("<PLUSET>30s</PLUSET><EF>24h</EF>") {
    *Q1 = select sum(DATA_SIZE) where RESC_NAME = '*CacheRescName';
    foreach (*R1 in *Q1) {
      *TotalSize = *R1.DATA_SIZE;
    }
    *usedSpace = double(*TotalSize);
    *MaxSpAlwd = *MaxSpAlwdTBs * 1024^4;
    if ( *usedSpace > *MaxSpAlwd ) then {
      msiGetIcatTime(*Time,"unix");
      *Q2 = select DATA_NAME, COLL_NAME, DATA_SIZE, order(DATA_CREATE_TIME) where DATA_RESC_NAME =
'*CacheRescName' AND COLL_NAME like '*Collection%';
      foreach (*R2 in *Q2) {
        *D = *R2.DATA_NAME;
        *C = *R2.COLL_NAME;
        *S = *R2.DATA_SIZE;
        *usedSpace = *usedSpace - double(*S);
        if ( *usedSpace < *MaxSpAlwd ) {
          break;
        }
        msiDataObjTrim(*C/*D,"*CacheRescName","null","1","1",*status);
        writeLine("stdout","*C/*D on *CacheRescName has been purged");
      }
      if ( *usedSpace < *MaxSpAlwd ) {
        break;
      }
    }
  }
}
input *MaxSpAlwdTBs = $1, *Collection = "/tempZone", *CacheRescName = "demoResc"
output ruleExecOut
```

6.2.2 Data expiration policy

English language description:

This policy checks the date specified by an expiration metadata attribute that has been assigned to the file, and creates a list of all files that have expired. Input parameters are used to specify the collection that is being checked and whether expired files should be found. A query is made to the information catalog to get a list of the DATA_EXPIRY date for each file. This is compared to the current Unix time. Files that have expired are listed and the total number is counted.

iRODS implementation:

```
integrityExpiry {
#Input parameter is:
#  Name of collection that will be checked
#  Flag for "EXPIRED" or for "NOT EXPIRED"
#Output is:
#  List of all files in the collection that have either EXPIRED or NOT EXPIRED

 #Verify that input path is a collection
 msiIsColl(*Coll,*Result, *Status);
 if(*Result == 0) {
  writeLine("stdout","Input path *Coll is not a collection");
  fail;
 }
```

```
  *Count = 0;
  *Counte = 0;
  msiGetIcatTime(*Time,"unix");

  #Loop over files in the collection
  *Q1 = select DATA_ID,DATA_NAME,DATA_EXPIRY where COLL_NAME = '*Coll';
  foreach(*R1 in *Q1) {
   *Attrname = *R1.DATA_EXPIRY;
   if(*Attrname > *Time  && *Flag == "NOT EXPIRED") {
    *File = *R1.DATA_NAME;
    writeLine("stdout","File *File has not expired");
    *Count = *Count + 1;
   }
   if(*Attrname <= *Time && *Flag == "EXPIRED") {
    *File = *R1.DATA_NAME;
    writeLine("stdout","File *File has expired");
    *Counte = *Counte + 1;
   }
  }
  if(*Flag == "EXPIRED") {writeLine("stdout","Number of files in *Coll that have expired is *Counte");}
  if(*Flag == "NOT EXPIRED") {writeLine("stdout","Number of files in *Coll that have not expired is *Count");}
}
INPUT *Coll = "/$rodsZoneClient/home/$userNameClient/sub1", *Flag = "EXPIRED"
OUTPUT ruleExecOut
```

7. Disposition policies

7.1 Example GPFS disposition policy

This policy checks the filesystem (here called '*research*') and automatically deletes files older than a week.

We use the `mmapplypolicy` utility to scan the filesystem and choose files older than one week. `mmapplypolicy` then deletes chosen files. It must be run explicitly, since GPFS doesn't provide necessary triggers.

GPFS policy file (called *retention.policy*) :

```
RULE 'DeleteOlderThanWeek' DELETE
WHERE((DAYS(CURRENT_TIMESTAMP) - DAYS(ACCESS_TIME)) > 7)
```

You can run this by issuing this command (filesystem is called '*research*') :

```
mmapplypolicy research -P retention.policy
```

7.2 Example iRODS disposition policy

Files in the iRODS data grid can be tagged with additional metadata attributes. For example, a metadata attribute with the name “Retention_Flag” can be added to each file, along with a metadata attribute value such as “EXPIRED” or “NOT_EXPIRED”. By using metadata to track the status of each file, it is possible to separate the retention policy from the disposition policy. The retention policy can set the metadata attribute, and the disposition policy can read the metadata attribute.

7.2.1 Disposition policy for expired files

English language description:

This rule migrates files to an archive that have a metadata attribute with the name “Retention_Flag” that has the value ”EXPIRED”. The rule reads as input the name of the collection that will be checked and the name of the destination collection. The collection names are verified. A query is then issued to the information catalog to retrieve the names of the files in the collection that have the “EXPIRED” value for the “Retention_Flag”. All of the returned files are moved to the destination collection. Note that the access controls on the file will need to be reset after the move.

iRODS implementation:

```
rule-disposition {
#Input parameter is:
#  Name of collection that will be checked
#  Retention_Flag with value "EXPIRED" or "NOT EXPIRED"
#Output is:
#  Migration of "EXPIRED" files to an archive collection
  *Coll = "/$rodsZoneClient/home/$userNameClient/" ++ *Collrel;
```

```
  *Dest = "/$rodsZoneClient/home/$userNameClient/" ++ *Archiverel;
  #Verify that input path is a collection
  msiIsColl(*Coll,*Result, *Status);
  if(*Result == 0) {
    writeLine("stdout","Input path *Coll is not a collection");
    fail;
  }
  #Verify that archive path is a collection
  msiIsColl(*Dest,*Result, *Status);
  if(*Result == 0) {
    writeLine("stdout","Archive *Dest is not a collection");
    fail;
  }
  *Count = 0;
  #Loop over files in the collection
  *Q1 = select DATA_ID,DATA_NAME where COLL_NAME = '*Coll' and META_DATA_ATTR_NAME = 'Retention_Flag'
and META_DATA_ATTR_VALUE = 'EXPIRED';
  foreach(*R1 in *Q1) {
    *File = *R1.DATA_NAME;
  # move the file to the archive
    *SourceFile = *Coll ++ "/" ++ *File;
    *DestFile = *Dest ++ "/" ++ *File;
    msiDataObjRename(*SourceFile,*DestFile,"0",*Status);
    if (*Status < 0) {
      writeLine("stdout","File *SourceFile could not be archived");
    }
    else { *Count = *Count + 1;}
}
  writeLine("stdout","Migrated *Count files to the archive *Dest");
}
INPUT *Collrel = "sub2", *Archiverel = "archive"
OUTPUT ruleExecOut
```

8. Integrity policies

8.1 Example GPFS integrity policy

This policy checks the filesystem (here called '*research*') for files which are under-replicated and re-replicates them if needed.

We use the `mmapplypolicy` utility to first scan the filesystem for under-replicated files and second to execute a special script, which then calls another utility (`mmrestripefile`) to do the re-replication. It must be run explicitly, since GPFS doesn't provide necessary triggers.

GPFS policy file (called *check_replicas.policy*) :

```
RULE EXTERNAL LIST 'doSomething' EXEC '/tmp/execScript'

RULE 'UnderReplicated' LIST 'doSomething'
WHERE (MISC_ATTRIBUTES LIKE '%J%')
```

Script (called *execScript*) :

```
#!/bin/bash
cmd=$1
list=$2
tempfile=$(mktemp)

if [[ "$cmd" = "TEST" ]]; then
        exit 0
fi

if [[ "$cmd" = "LIST" ]]; then
        /bin/awk '{print $5}' $2 > $tempfile
        /usr/lpp/mmfs/bin/mmrestripefile -F $tempfile -r
fi

rm -f $tempfile
```

You can run this by issuing this command (filesystem is called '*research*') :

```
mmapplypolicy research -P check_replicas.policy
```

8.2 Examples iRODS integrity policies.

Policies are typically created to verify the integrity of files by comparing the current checksum with a saved value of the checksum. However, integrity policies can also

be created to verify access controls on a collection, verify the presence of required metadata, verify file distribution, etc.

8.2.1 File integrity policy for access controls

English language description:

This rule analyses the files in a collection to verify that a required access control is present on each file. The input includes the name of the collection that will be verified, the type of access control that is required, and the name of a person for which the access control is set. The rule verifies the collection name, retrieves a USER_ID for the named person, and retrieves a DATA_ACCESS_DATA_ID number for the type of access control. A loop is made over the files in the collection, with a sub-loop that verifies the access control on each file. The results are printed to the screen.

iRODS implementation:

```
integrityACL {
#Rule to analyze files in a collection
#  Verify that a specific ACL is present on each file in collection
#Input
#  Collection that will be analyzed
#  Name of person to check for presence of ACL on file
#  Required ACL value, expressed as an integer
#Output
#  Names of files that are missing the required ACL
# Generate home collection name for user running the rule
 *Coll= "/$rodsZoneClient/home/$userNameClient/" ++ *Coll;

 #Verify input path is a collection
 *Result = 1;
 msiIsColl(*Coll,*Result,*Status);
 if(*Result == 0) {
  writeLine("stdout","Input path *Coll is not a collection");
  fail;
 }

 #Get USER_ID for the input user name
 *Query = select USER_ID where USER_NAME = '*User';
 *Userid = "";
 foreach(*Row in *Query) {
  *Userid = *Row.USER_ID;
 }
 if(*Userid == "") {
  writeLine("stdout","Input user name *User is unknown");
  fail;
 }

 #Get DATA_ACCESS_DATA_ID number that corresponds to requested access control
 *Query2 = select TOKEN_ID where TOKEN_NAMESPACE = 'access_type' and TOKEN_NAME = '*Acl';
 foreach(*Row2 in *Query2) {
  *Access = *Row2.TOKEN_ID;
 }
 writeLine("stdout","Access control number of *Acl is *Access");
 *Count = 0;

 #Loop over files in the collection
 *Query3 = select DATA_ID,DATA_NAME where COLL_NAME = '*Coll';
 foreach(*Row3 in *Query3) {
  *Dataid = *Row3.DATA_ID;
  *File = *Row3.DATA_NAME;
```

```
  *Path = *Coll ++ "/" ++ *File
  #Find ACL for each file
  *Query4 = select DATA_ACCESS_TYPE, DATA_ACCESS_USER_ID where DATA_ACCESS_DATA_ID = '*Dataid';

  #Loop over access controls for each file
  *Attrfound = 0;
  foreach(*Row4 in *Query4) {
   *Userdid = *Row4.DATA_ACCESS_USER_ID;
   if(*Userdid == *Userid) {
    *Attrfound = 1;
    *Datatype = *Row4.DATA_ACCESS_TYPE;
    if(*Datatype < *Access) {
     writeLine("stdout","* Path has wrong access permission, *Datatype");
    }
   }
  }
  if(*Attrfound == 0) {
   writeLine("stdout","*Path is missing access controls for  *User");
   *Count = *Count + 1;
  }
 }
 writeLine("stdout","Number of files in *Coll missing access control for *User is *Count");
}
INPUT *Coll =$"Rules", *User=$"rods", *Acl =$"own"
OUTPUT ruleExecOut
```

8.2.2 Example iRODS policy for checking integrity and number of replicas of files in a collection

English language description:

This policy implements 17 basic operations needed for a production quality policy. The basic operations include:

1. Verifying all input parameters for consistency
2. Retrieving state information from the metadata catalog on each execution
3. Verifying integrity of eah file by comparing the saved checksum with the computed checksum
4. Updating all replicas to the most recent version
5. Minimizing the load on the production services through a deadline scheduler
6. Differentiating between the logical name for the file and the physical location of the replicas
7. Identifying missing replicas and documenting their absence
8. Creating new replicas to replace missing and corrupted files
9. Implementing load leveling to distribute files cross available storage systems
10. Creating a log file to record all repair operations and storing the log file in the data grid
11. Tracking progress of the policy execution
12. Initializing the rule for the first execution, including setting variables to track progress.
13. Enabling restart from the last checked file
14. Manipulating files in batches of 256 files at a time to handle arbitrarily large collections
15. Minimizing the number of sleep periods required by the deadline scheduler
16. Checking new files that have been added on a restart

17. Generating statistics about the execution rate and properties of the files that were checked.

Implementing all 17 operations increases the size of the production policy substantially. However, it is possible to show that the average time spent per file is still less than a disk rotation period, implying that the production rule is suitable for verifying integrity across arbitrarily large collections.

iRODS implementation:

Each of these basic operations is annotated in the following integrity rule.

```
schedulerReplicas {
# This rule requires iRODS version 3.1 (msiCloseGenQuery mods)
# The replicas for each file are updated to the most recent version
# Each file is checked to verify whether all required replicas exist and have valid checksums
# As replicas are created, the algorithm round robins through available storage vaults
# Checks that the number of storage resources used within a collection is greater than or
#   equal to the number of desired replicas.
# This uses a just in time scheduler that slows down the processing rate
#   to complete the task within the specified number of seconds (*Delt)
# Checks a TEST_DATA_ID parameter associated with the collection
#   to determine enable restarts after system interrupts
# Writes a log file stored as Check-Timestamp in directory *Coll/log
#=============get current time, Timestamp is YYY-MM-DD.hh:mm:ss  ======================
 msiGetSystemTime(*TimeS,"unix");
 msiGetSystemTime(*TimeH,"human");
 *NumBadFiles = 0;
 *NumRepCreated = 0;
 *NumFiles = 0;
 *Runsize = double(0);
 *Sleeptime = 0;
 *colldataID = "0";
#this is used to round robin through available storage resources
 *Jround = 0;
#========= check whether a collection was defined ================================
 msiIsColl(*Coll,*Result, *Status);
 if(*Result == 0 || *Status < 0) {
  writeLine("stdout","Input path *Coll is not a collection");
  fail;
 }     # end of status check
#============= create a collection for log files if it does not exist ===============
 *LPath = "*Coll/log";
 msiIsColl(*LPath,*Result, *Status);
 if(*Result == 0 || *Status < 0) {
  msiCollCreate(*LPath, "0", *Status);
  if(*Status < 0) {
   writeLine("stdout","Could not create log collection");
   fail;
  }  # end of check on status
 }  # end of log collection creation
#============= create file into which results will be written ==========================
 *Lfile = "*LPath/Check-*TimeH";
 *Dfile = "destRescName=*Res++++forceFlag=";
 msiDataObjCreate(*Lfile, *Dfile, *L_FD);
#============= check whether the attribute TEST_DATA_ID has been set from a prior execution ========
 *Val = "0";
 *Query1 = SELECT COUNT(META_COLL_ATTR_NAME) where COLL_NAME = '*Coll' and META_COLL_ATTR_NAME
= 'TEST_DATA_ID';
 foreach (*Row1 in *Query1) {
  *Val = *Row1.META_COLL_ATTR_NAME;
 }  # end of loop to count number of TEST_DATA_ID values
 if(int(*Val) == 0) {
  *Str1 = "TEST_DATA_ID=0";
  msiString2KeyValPair(*Str1,*kvp);
```

```
    msiAssociateKeyValuePairsToObj(*kvp,*Coll,"-C");
    writeLine("*Lfile","added TEST_DATA_ID attribute to collection *Coll");
   }  # end of creation of initial value for TEST_DATA_ID
#============= on a restart TEST_DATA_ID will be greater than 0 ============================
   *Query2 = select META_COLL_ATTR_VALUE where COLL_NAME = '*Coll' and META_COLL_ATTR_NAME =
'TEST_DATA_ID';
   foreach(*Row2 in *Query2) {
    *colldataID = *Row2.META_COLL_ATTR_VALUE;
   }  # end of retrieval of *colldataID
#======== *colldataID is the string identifier of the last file that has been checked =========
   msiMakeGenQuery("count(DATA_NAME), sum(DATA_SIZE)","COLL_NAME = '*Coll' and DATA_ID > '*colldataID'",
*GenQInp2);
#========= this counts all files that have not yet been checked including replicas =============
   msiExecGenQuery(*GenQInp2, *GenQOut2);
   foreach(*Row3 in *GenQOut2) {
    *num = *Row3.DATA_NAME;
    *sizetotal = *Row3.DATA_SIZE;
   }  # end of retrieval of number and size
   msiCloseGenQuery(*GenQInp2, *GenQOut2);
   *Size = double(*sizetotal);
   *Num = int(*num);
#========= expected execution time = 0.0161 (sec) * (number of files) + (total size) / (50 MBytes/sec) =========
   *Timeest = int(*Num / 62) + int(*Size / 50000000);
   writeLine("*Lfile","Estimated time is *Timeest seconds, total time is *Delt seconds, number of files is *Num,  and
total size is *Size bytes");
   writeLine("*Lfile","Number of required copies of a file is *NumReplicas");
   if(*Delt > 0 && *Size > 0) {
    *Fac = *Size / *Delt;
    writeLine("*Lfile", "Required analysis rate is *Fac bytes/second");
#============ identify the resources that were used for the collection ================
#============ use resources at which any files in the collection were stored =========
    *Query3 = select DATA_RESC_NAME where COLL_NAME = '*Coll';
    *Ir = 0;
    *Rlist = list();
    *Ulist = list();
    foreach(*Row3 in *Query3) {
     *Str1 = *Row3.DATA_RESC_NAME;
     *Rlist = cons(*Str1,*Rlist);
     *Ulist = cons("0",*Ulist);
     writeLine("*Lfile","Collection *Coll uses storage resource *Str1");
     *Ir = *Ir + 1;
    }  # end of set up of list of resources
    *Ulist0 = *Ulist;
    *Irm1 = *Ir - 1;
    if(*Ir < *NumReplicas) {
     writeLine("stdout","Required number of replicas, *NumReplicas, exceeds the number of storage vaults, *Ir");
     writeLine("*Lfile","Required number of replicas, *NumReplicas, exceeds the number of storage vaults, *Ir");
     fail;
    }  # end of check on number of available resources
#============== loop over all the files in the collection in batches of 256 ================
    msiMakeGenQuery("order(DATA_ID), DATA_SIZE, DATA_NAME, COLL_NAME, DATA_CHECKSUM","COLL_NAME =
'*Coll' and DATA_ID > '*colldataID'",*GenQInp);
    msiExecGenQuery(*GenQInp, *GenQOut);
    msiGetContInxFromGenQueryOut(*GenQOut,*ContInxNew);
    *ContInxOld = 1;
    while (*ContInxOld > 0) {
     foreach(*GenQOut) {
       msiGetValByKey(*GenQOut, "DATA_SIZE", *Sizedata);
       msiGetValByKey(*GenQOut, "DATA_ID", *newdataID);
       msiGetValByKey(*GenQOut, "DATA_NAME", *Name);
       msiGetValByKey(*GenQOut, "COLL_NAME", *Colln);
#======= before updating replicas, must verify that the replica has a valid checksum =========
#======= get all replica numbers for this file ===================================
       *Query5 = select DATA_REPL_NUM,DATA_CHECKSUM,DATA_RESC_NAME where COLL_NAME = '*Colln' and
DATA_NAME = '*Name';
       *Numr = 0;
       *Ulist = *Ulist0;
```

```
      foreach(*Row5 in *Query5) {
       *Numr = *Numr + 1;
       *Repln = *Row5.DATA_REPL_NUM;
       *Chk = *Row5.DATA_CHECKSUM;
       *Rescn = *Row5.DATA_RESC_NAME;
       msiDataObjChksum("*Colln/*Name", "replNum=*Repln++++forceChksum=", *Chkf);
       if(int(*Chk) == 0) {
        *Chk = *Chkf;
       } # end of set of checksum if not available
#======= save list of resources ================================================
       if(int(*Chk) == int(*Chkf)) {
        for(*J=0;*J<*Ir;*J=*J+1) {
         if(elem(*Rlist,*J) == *Rescn) {
          *Ulist = setelem(*Ulist,*J,"1");
          break;
         } # end of set of *Ulist for resource
        } # end of loop over resources
       } # end of processing good checksum
#======== check whether checksum is correct, delete file if bad checksum =========
       if (int(*Chk) != int(*Chkf)) {
        writeLine("*Lfile","Bad checksum for replica *Repln of file *Colln/*Name with DATA_ID *newdataID.");
        *NumBadFiles = *NumBadFiles + 1;
        msiDataObjUnlink("objPath=*Colln/*Name++++replNum=*Repln", *Status);
        writeLine("*Lfile","Deleted replica *Repln of file *Colln/*Name");
        *Numr = *Numr - 1;
       } # end of processing a bad checksum
      } # end of loop over replicas for a logical file
#========= update all replicas to the most recent version ======================
#========= this can be done because all remaining copies have good checksums ===
      msiDataObjRepl("*Colln/*Name","updateRepl=++++irodsAdmin=",*Status2);
      if(*Status2 != 0) {
       writeLine("*Lfile","Unable to update replicas to most recent version for *Colln/*Name");
      } # end of error message if not able to update replicas to most recent version
#========== pick resource to use as source =====================================
#========== we only select from resources that have good copies ================
      for(*J=0;*J<*Ir;*J=*J+1) {
       if(elem(*Ulist,*J) == "1") {
        *Resource = elem(*Rlist,*J);
        break;
       } # end of selection of resource with valid copy
      } # end of loop over all resources
#========= test whether the required number of replicas exists ==================
      if (*Numr != *NumReplicas) {
       *N = *NumReplicas - *Numr;
       if(*N > 0) {
        writeLine("*Lfile","File *Colln/*Name is missing *N replicas");
        for(*I = 0;*I<*N;*I=*I+1) {
#==== pick resource to use for storing replica, round robin through storage systems without a replica ======
         *Check = false;
         for(*L = 0;*L<*Ir;*L=*L+1) {
          *J = *L + *Jround;
          if(*J >= *Ir) {
           *J = *J - *Ir;
          } # end of reset of start location for load leveling
          *Stu = elem(*Ulist,*J);
          if(*Stu == "0") {
           *Resu = elem(*Rlist,*J);

msiDataObjRepl("*Colln/*Name","destRescName=*Resu++++rescName=*Resource++++irodsAdmin=",*Status1);
           *NumRepCreated = *NumRepCreated + 1;
           *Ulist = setelem(*Ulist,*J,"1");
           *Check = true;
           *Jround = *J + 1;
           if(*Jround >= *Ir) {
            *Jround = 0;
           } # end of reset of start location for load leveling
           if(*Status1 < 0) {
```

```
                *NumRepCreated = *NumRepCreated - 1;
                writeLine("*Lfile","Unable to create a replica for *Colln/*Name on resource *Resu");
                *Check = false;
               } # end of decrement of error check for creating replica
              } # end of creation of a new replica
              if(*Check == true) {
               break;
              } # end of test that were able to create a replica
             } # end of loop over storage resources
            } # end of loop over number of replicas to create
           } # end of check that additional replicas are needed
          } # end of check that the required number of replicas is not present
          msiCloseGenQuery(*GenQInp4,*GenQOut4);
#========= slow rate at which are processing collection to meet deadline ====================
          *Runsize = *Runsize + double(*Sizedata);
          msiGetSystemTime(*timei, "unix");
          *timerun = int(*TimeS) + *Runsize / *Fac;
          *delt = *timerun - int(*timei);
          if (*delt > 4) {
            msiSleep(str(*delt), "0");
            *Sleeptime = *Sleeptime + *delt;
          } # end of check on length of sleep time
          *NumFiles = *NumFiles + 1;
        } # end of loop over 256 rows of list of logical file names
        *Str1 = "TEST_DATA_ID=*colldataID";
        msiString2KeyValPair(*Str1, *kvp1);
        msiRemoveKeyValuePairsFromObj(*kvp1, *Coll, "-C");
        *colldataID = *newdataID;
        *Str2 = "TEST_DATA_ID=*colldataID";
        msiString2KeyValPair(*Str2, *kvp);
        msiAssociateKeyValuePairsToObj(*kvp, *Coll, "-C");
        writeLine("*Lfile", "Reset TEST_DATA_ID to *colldataID for collection *Coll");
        *ContInxOld = *ContInxNew;
        if (*ContInxOld > 0) {
          msiGetMoreRows(*GenQInp,*GenQOut,*ContInxNew);
        } # end of test to get more rows
      } # end of check on continuation index for another set of rows
      writeLine("*Lfile", "Number of logical file names tested is *NumFiles, total size checked is *Runsize bytes, and
total time slept is *Sleeptime seconds");
      writeLine("*Lfile", "Number of bad files is *NumBadFiles, and number of replicated files created is
*NumRepCreated");
#=======  reset TEST_DATA_ID status flag to zero ====================
      *Query6 = select META_COLL_ATTR_VALUE where COLL_NAME = '*Coll' and META_COLL_ATTR_NAME =
'TEST_DATA_ID';
      foreach(*Row6 in *Query6) {
        *colldataID = *Row6.META_COLL_ATTR_VALUE;
      } # end of loop to get *colldataID
      *Str1 = "TEST_DATA_ID=*colldataID";
      msiString2KeyValPair(*Str1, *kvp1);
      msiRemoveKeyValuePairsFromObj(*kvp1, *Coll, "-C");
      *Str2 = "TEST_DATA_ID=0";
      msiString2KeyValPair(*Str2, *kvp);
      msiAssociateKeyValuePairsToObj(*kvp,*Coll,"-C");
      writeLine("*Lfile", "Reset TEST_DATA_ID to 0 indicating a successful completion of the integrity check");
    } # end of check on evaluation bandwidth
#====== Calculate actual elapsed time ============================
    msiGetSystemTime(*TimeE, "unix");
    *Del = int(*TimeE) - int(*TimeS);
    writeLine("*Lfile","Total elapsed time is *Del seconds");
} # end of micro-service code
INPUT *Coll=$"/$rodsZoneClient/home/$userNameClient", *Delt=10, *NumReplicas = 2, *Res="demoResc"
OUTPUT ruleExecOut
```

8.3 Examples EUDAT schema for defining an integrity policy

The EUDAT schema is listed in Appendix A. Multiple examples are given for types of policies for data replication, data synchronization, and data distribution.

8.3.1 Replication of files from MPI-TLA to RZG

English language description:

This policy template specifies the persistent identifier for a collection, the operation that will be performed on the collection, the periodicity of application, and the target resource where files will be replicated.

EUDAT implementation in iRODS:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
this is a policy instance to replicate from MPI-TLA to RZG
-->
<policy name="replication to RZG" version="1.0" author="Willem Elbers" uniqueid="eb9f46e0-0b27-45e8-8732-
eceabf70d51d"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:irodsns="http://eudat.eu/2013/iRODS-policy"
xmlns="http://eudat.eu/2013/policy" xmlns:xs="http://www.w3.org/2001/XMLSchema/v1.1">
 <dataset>
  <collection id="0">
   <persistentIdentifier type="PID">1839/00-0000-0000-0001-2A70-B</persistentIdentifier>
  </collection>
 </dataset>
 <actions>
  <action name="replication onchange">
   <type>replication</type>
   <trigger type="action">weekly</trigger>
   <targets>
    <target id="0">
     <location xsi:type="irodsns:coordinates">
      <irodsns:site type="EUDAT">RZG</irodsns:site>
      <irodsns:path>/vzRZGE/eudat/clarin/</irodsns:path>
      <irodsns:resource>cacheResc</irodsns:resource>
     </location>
    </target>
   </targets>
  </action>
 </actions>
</policy>
```

8.3.2 Replication process control

English language description:

This policy specifies that when a file '.replicate' is written to a 'shared collection', a replication process will be started, replicating an object from a "source on repository A" to a 'destination on repository B'. Note that in this case, the action is triggered by the creation of a file in repository A.

EUDAT implementation in iRODS:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
   When a file '.replicate' is written in the 'shared collection'
   (typically <zone>/replicate), a replication process is started:
   the object with PID 'pid1' is replicated from 'source on repo A' to
   'destination on repo B'.
```

```
-->
<policy name="PID registration 2" version="1.0" author="Claudio Cacciari" uniqueid="eb9f46e0-0b27-45e8-8732-
eceabf70d51d"
xmlns:irodsns="http://eudat.eu/2013/iRODS-policy" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://eudat.eu/2013/policy">
 <dataset>
  <collection id="0">
   <location xsi:type="irodsns:coordinates">
    <irodsns:site type="EUDAT"><!--example:CINECA--></irodsns:site>
    <irodsns:path><!--example:/path/to/destination--></irodsns:path>
   </location>
            </collection>
  <!--
  add here further collections, if needed
  <collection id="1" type="other"></collection>
  -->
 </dataset>
 <actions>
  <action name="replication oncreation">
   <type>replication</type>
   <trigger type="action">oncreation</trigger>
   <sources>
                        <source id="0">
     <persistentIdentifier type="PID"><!--11100/6c8ac19e-c982-11e2-b3cb-e41f13eb41b2--
></persistentIdentifier>
    </source>
   </sources>
   <targets>
                        <target id="0">
                         <location xsi:type="irodsns:coordinates">
     <irodsns:site type="EUDAT"><!--example:B--></irodsns:site>
     <irodsns:path><!--example:/path/to/destination--></irodsns:path>
    </location>
   </target>
  </targets>
 </action>
</actions>
</policy>
```

8.3.3 Replication on triggers from source collection

English language description:

This policy template defines the replication of files from site A to site B, and from site A to site C when a file is changed on site A. The collection from which files will be replicated is specified using a persistent identifier.

EUDAT implementation in iRODS:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
  this is a policy template to replicate from site A to B and from site A
  to C triggering each change on the source collection.
-->
<policy name="replication - double copy ABAC" xmlns="http://eudat.eu/2013/policy"
xmlns:irodsns="http://eudat.eu/2013/iRODS-policy" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.0" author="Claudio Cacciari" uniqueid="eb9f46e0-0b27-45e8-8732-eceabf70d51d">
 <dataset>
  <collection id="0">
   <persistentIdentifier type="PID">11100/6c8ac19e-c982-11e2-b3cb-e41f13eb41b2</persistentIdentifier>
 </collection>
  <!--
  add here further collections, if needed
  -->
 </dataset>
```

```
 <actions>
  <action name="replication onchange">
   <type>replication</type>
   <trigger type="action">onchange</trigger>
   <sources/>
   <targets>
    <target id="0">
     <location xsi:type="irodsns:coordinates">
      <irodsns:site type="EUDAT"><!--example:CINECA--></irodsns:site>
      <irodsns:path><!--example:/path/to/destination--></irodsns:path>
      <irodsns:resource><!--defaultResc--></irodsns:resource>
     </location>
    </target>
    <target id="1">
     <location xsi:type="irodsns:coordinates">
      <irodsns:site type="EUDAT"><!--example:SARA--></irodsns:site>
      <irodsns:path><!--example:/path/to/destination--></irodsns:path>
     </location>
    </target>
   </targets>
  </action>
 </actions>
</policy>
```

8.3.4 Replication between two sets of sites based on change to source collection

English language description:

This policy template specifies the replication of files from site A to site B and then from site B to site C when a file is changed in site A. The collection from which the files will be replicated is specified by a persistent identifier.

EUDAT implementation in iRODS:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
   this is a policy template to replicate from site A to B and from site B
   to C triggering each change on the source collection.
-->
<policy name="replication - double copy ABC" xmlns="http://eudat.eu/2013/policy"
xmlns:irodsns="http://eudat.eu/2013/iRODS-policy" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.0" author="Claudio Cacciari" uniqueid="eb9f46e0-0b27-45e8-8732-eceabf70d51d">
 <dataset>
  <collection id="0">
   <persistentIdentifier type="PID"><!--11100/6c8ac19e-c982-11e2-b3cb-e41f13eb41b2--
></persistentIdentifier>
  </collection>
  <!--
  add here further collections, if needed
  -->
 </dataset>
 <actions>
  <action name="replication onchange 1">
   <type>replication</type>
   <trigger type="action">onchange</trigger>
   <sources/>
   <targets>
    <target id="0">
     <location xsi:type="irodsns:coordinates">
      <irodsns:site type="EUDAT"><!--example:CINECA--></irodsns:site>
      <irodsns:path><!--example:/path/to/destination--></irodsns:path>
      <irodsns:resource><!--defaultResc--></irodsns:resource>
     </location>
    </target>
   </targets>
```

```
  </action>
  <action name="replication onchange 2">
   <type>replication</type>
   <trigger type="action">replication onchange 1</trigger>
   <sources/>
   <targets>
    <target id="0">
     <location xsi:type="irodsns:coordinates">
      <irodsns:site type="EUDAT"><!--example:SARA--></irodsns:site>
      <irodsns:path><!--example:/path/to/destination--></irodsns:path>
     </location>
    </target>
   </targets>
  </action>
 </actions>
</policy>
```

8.3.5 Periodic synchronization policy

English language description:

This policy template specifies the periodic synchronization of files from a collection in site A to site B. The synchronization is performed every day at 1:15 AM. A persistent identifier specifies the collection from which the files will be synchronized.

EUDAT implementation in iRODS:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
   this is a policy template to ingest from site A to B. The sync is performed periodically
   once a day at 1:15 AM
-->

<policy name="ingestion - single copy AB" xmlns="http://eudat.eu/2013/policy"
xmlns:irodsns="http://eudat.eu/2013/iRODS-policy" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 version="1.0" author="Claudio Cacciari" uniqueid="eb9f46e0-0b27-45e8-8132-eceabf70d90d">
 <dataset>
  <collection id="0">
   <location xsi:type="irodsns:coordinates">
    <irodsns:site type="EUDAT"><!-- A --></irodsns:site>
    <irodsns:path><!-- /path/to/collection1 --></irodsns:path>
    <irodsns:resource><!-- resc01 --></irodsns:resource>
   </location>
            </collection>
  <!--
  add here further collections, if needed
  -->
 </dataset>
 <actions>
  <action name="ingestion">
   <type>ingestion</type>
   <trigger type="time"><!-- 15 1 * * * --></trigger>
   <targets>
    <target id="0">
     <location xsi:type="irodsns:coordinates">
      <irodsns:site type="EUDAT"><!-- B --></irodsns:site>
      <irodsns:path><!-- /path/to/destination1 --></irodsns:path>
      <irodsns:resource><!-- defaultResc --></irodsns:resource>
     </location>
    </target>
   </targets>
  </action>
 </actions>
</policy>
```

8.3.6 Ingestion policy to synchronize data between two sites hourly

English language description:

This policy template specifies the hourly synchronization of files from site A to site B. Both the source collection and the destination collection are specified.

EUDAT implementation in iRODS:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
   this is a policy template to ingest from site A to B. The sync is performed periodically
   each hour at minute 35
-->

<policy name="ingestion - single copy AB" xmlns="http://eudat.eu/2013/policy"
xmlns:irodsns="http://eudat.eu/2013/iRODS-policy" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 version="1.0" author="Claudio Cacciari" uniqueid="eb9f46e0-0b27-45e8-8132-eceabf70d90d">
  <dataset>
   <collection id="0">
    <location xsi:type="irodsns:coordinates">
     <irodsns:site type="EUDAT"><!-- A --></irodsns:site>
     <irodsns:path><!-- /path/to/collection2 --></irodsns:path>
     <irodsns:resource><!-- resc01 --></irodsns:resource>
    </location>
                </collection>
   <!--
   add here further collections, if needed
   -->
  </dataset>
  <actions>
   <action name="ingestion2">
    <type>ingestion</type>
    <trigger type="time"><!-- 35 * * * * --></trigger>
    <targets>
     <target id="0">
      <location xsi:type="irodsns:coordinates">
       <irodsns:site type="EUDAT"><!-- B --></irodsns:site>
       <irodsns:path><!-- /path/to/destination2 --></irodsns:path>
       <irodsns:resource><!-- defaultResc --></irodsns:resource>
      </location>
     </target>
    </targets>
   </action>
  </actions>
</policy>
```

9. Notification policies

9.1 GPFS notification policy

GPFS is a filesystem and as such, it doesn't provide any meaningful events (like file deletion event or file creation event) to be used for notifications.

9.2 Example iRODS notification policy

Notification policies are implemented at Policy Enforcement Points, either before an action occurs or after the action is completed. A rule can be created that specifies the type of notification that will be used.

9.2.1 Notification policy for collection deletion

English language description:

This policy sends E-mail to an administrator on deletion of a collection. A session variable, $collName, is used to identify which collection is being deleted.

iRODS implementation:

```
acPreprocForRmColl {
  msiSendMail("admin@unc.edu","Collection deletion","Collection $collName is deleted");
}
```

10. Restricted searching policies

10.1 GPFS restricted searching policy

Since GPFS is a POSIX filesystem administrators can use standard POSIX permissions (user, group, other) to manage access to files and directories. If more fine grained restrictions are needed, administrators can use ACLs.

10.2 Example iRODS restricted searching policy

The most commonly requested restriction is to limit the ability of users to see any other user's files. This can be applied to all users, or applied to a specific user.

10.2.1 Strict access control

English language description:

A strict access control is implemented through the Policy Enforcement Point called acAclPolicy. The micro-service msiAclPolicy implements the restriction.

iRODS implementation:

```
acAclPolicy {msiAclPolicy("STRICT"); }
```

11. Storage cost policies

11.1 Example GPFS storage cost policy

This policy checks the filesystem (here called '*research*') and computes the sums of used storage space by each user for each device (storage pool).

We use the `mmapplypolicy` utility to scan the filesystem and generate a file containing needed information to compute the usage. Then a Python script iterates over that file and computes the usage. It must be run explicitly, since GPFS doesn't provide necessary triggers.

GPFS policy file (called *storage_cost.policy*):

```
RULE EXTERNAL LIST 'doSomething' EXEC '/tmp/execScript.py'

RULE 'StorageCost' LIST 'doSomething'
    SHOW(
         VARCHAR(FILE_SIZE) || ' ' || VARCHAR(USER_ID) || ' ' ||
VARCHAR(POOL_NAME)
        )
```

Script (called *execScript.py*) :

```
#!/usr/bin/python

import sys
from subprocess import call

costs = {}
# examples of storage costs for each device (storage pool)
media_costs = { 'tape': 1, 'maid': 5, 'system':10 }

if __name__ == "__main__":
   cmd, filelist  = sys.argv[1:]

   if cmd == 'TEST':
       exit(0)

   if cmd == 'LIST':
       with open(filelist, 'r') as fl:
           for entry in fl:
               size,uid,media = entry.split(' ')[4:7]
                   try:
                       costs[(uid, media)] += int(size)
                   except KeyError:
                       costs[(uid, media)] = int(size)

       with open('/tmp/output', 'w') as output:
```

```
            for key, size in costs.iteritems():
                uid, media = key
                size = int(size/1024/1024)
                cost = media_costs[media] * size

                output.write("User UID %s: %d MB on %s
(cost=%d)\n"
                    % (uid, size, media, cost))
```

You can run this by issuing this command (filesystem is called '*research*') :

```
mmapplypolicy research -P storage_cost.policy
```

11.2 Examples iRODS storage cost policies

The basic approach is to calculate the amount of storage used by each type of device and then to generate a cost by multiplying usage by the charge per storage for the device type. This can be refined to implement a separate cost per storage device. The cost information can be stored as a metadata attribute that is associated with each storage resource.

11.2.1 Usage report by user name and storage system

English language description:

This rule sums the amount of storage used for each device by each user. A query is issued to the information catalog that sums the storage for each home directory in the data grid. The result is written to the screen.

iRODS implementation:

```
ruleStorage {
# Total the size of files on each storage vault for each user
# Get list of users
 *Quser = select USER_NAME;
 foreach (*Row in *Quser) {
  *User = *Row.USER_NAME;
  writeLine("stdout","Storage for *User");
  *Path = "/$rodsZoneClient/home/*User/%";
  *Q = select sum(DATA_SIZE),DATA_RESC_NAME where COLL_NAME like '*Path';
  foreach (*R in *Q) {
   *Size = *R.DATA_SIZE;
   *V = *R.DATA_RESC_NAME;
   writeLine ("stdout", "    Storage on *V is *Size");
  }
 }
}
INPUT null
OUTPUT ruleExecOut
```

11.2.2 Cost report by user name and storage system

English language description:

A cost algorithm is implemented by storing a "cost per byte" metadata attribute on each storage resource. The "cost per byte" attribute is stored as the metadata attribute called "Storage_Cost", with the attribute value equal to the storage cost per byte. A query is issued to the information catalog to get a list of the users. Then for each user, a query is issued to sum the storage for each user for each storage device. The storage cost per byte is retrieved by a query, and the storage cost is calculated.

iRODS implementation:

```
ruleStorage {
# Total the cost for files on each storage vault for each user
# Get list of users
 *Quser = select USER_NAME;
 foreach (*Row in *Quser) {
  *User = *Row.USER_NAME;
  writeLine("stdout","Storage for *User");
  *Path = "/$rodsZoneClient/home/*User/%";
  *Q = select sum(DATA_SIZE),DATA_RESC_NAME where COLL_NAME like '*Path';
  foreach (*R in *Q) {
   *Size = *R.DATA_SIZE;
   *V = *R.DATA_RESC_NAME;
   *Qresc = select META_RESC_ATTR_VALUE where RESC_NAME = '*V' and
META_RESC_ATTR_NAME = 'Storage_Cost';
    foreach (*Rowc in *Qresc) {
     *Cost = int(*Rowc.META_RESC_ATTR_VALUE);
    }
    *Scost = *Cost * int(*Size);
    writeLine ("stdout", "    Storage cost on *V is *Scost");
  }
 }
}
INPUT null
OUTPUT ruleExecOut
```

12. Use agreement policies

12.2 GPFS use agreement policy

- none available -

12.2 Examples iRODS use agreement policies

A metadata attribute can be defined for each user to designate receipt of a signed user agreement. This is an example of a user-defined metadata attribute that can be associated with each user name.

12.2.1 Set receipt of signed use agreement

English language description:

This policy uses the metadata attribute "Use_Agreement" to store a value of "RECEIVED" when a use agreement is confirmed. The policy sets the use agreement for a specified user.

iRODS implementation:

```
ruleSetUse {
# Sets metadata attribute Use_Agreement to RECEIVED
 msiAddKeyVal(*Keyval,"Use_Agreement","RECEIVED");
 msiAssociateKeyValuePairsToObj(*Keyval,*User,"-u");
 writeLine("stdout","Set use agreement for *User");
}
INPUT *User = "rods"
OUTPUT ruleExecOut
```

12.2.2 Identify users without signed use agreement

English language description:

This policy queries all user names to find users who either do not have a "Use_Agreement" metadata attribute name, or have a value that is not "RECEIVED". If either case is found, a message is written to the screen.

iRODS implementation:

```
ruleCheckUse {
# Checks whether the  metadata attribute Use_Agreement has been set for each user
 *Quser = select USER_NAME;
 foreach (*Row in *Quser) {
  *User = *Row.USER_NAME;
  *Quse = select count(META_USER_ATTR_NAME) where USER_NAME = '*User' and
META_USER_ATTR_NAME = 'Use_Agreement';
   foreach (*R1 in *Quse) {
    *Count = *R1.META_USER_ATTR_NAME;
    if (*Count != "0") {
     *Qcheck = select META_USER_ATTR_VALUE where USER_NAME = '*User' and
META_USER_ATTR_NAME = 'Use_Agreement';
      foreach (*R2 in *Qcheck) {
       *Val = *R2.META_USER_ATTR_VALUE;
```

```
        if (*Val != "RECEIVED") {
          writeLine("stdout","No use agreement for *User");
        }
       }
      }
      else {
       writeLine("stdout","No use agreement for *User");
      }
     }
    }
}
INPUT null
OUTPUT ruleExecOut
```

References

1. Rajasekar, R., M. Wan, R. Moore, W. Schroeder, S.-Y. Chen, L. Gilbert, C.-Y. Hou, C. Lee, R. Marciano, P. Tooby, A. de Torcy, B. Zhu, "iRODS Primer: Integrated Rule-Oriented Data System", Morgan & Claypool, 2010.
2. Ward, J., M. Wan, W. Schroeder, A. Rajasekar, A. de Torcy, T. Russell, H. Xu, R. Moore, "The integrated Rule-Oriented Data System (iRODS 3.0) Micro-service Workbook", DICE Foundation, November 2011, ISBN: 9781466469129, Amazon.com.

Appendix A

EUDAT Policy description and implementation: some examples

by Claudio Cacciari, c.cacciari@cineca.it – Cineca (www.cineca.it) – v. 0.1

The following example intends to show a possible mapping among the high level policy description language defined by the EUDAT project (http://www.eudat.eu/), the template proposed by the RDA practical policy WG and the fine grained iRODS rules. In the appendix are listed the reference models.

The EUDAT schema assumes a certain number of default parameters and conditions to simplify the description.

Policy example based on the EUDAT schema v1.0

Replication of an object, identified by a PID, to a location, identified by iRODS namespace parameters. The replication is triggered by any change applied to the object.

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
    this is a policy template to replicate from site A to B triggering each change
    on the source collection.
-->

<policy name="replication - single copy AB" xmlns="http://eudat.eu/2013/policy"
xmlns:irodsns="http://eudat.eu/2013/iRODS-policy" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 version="1.0" author="Claudio Cacciari" uniqueid="eb9f46e0-0b27-45e8-8732-eceabf70d51d">
  <dataset>
    <collection id="0">
    <persistentIdentifier type="PID"><!--example:11100/6c8ac19e-c982-11e2-b3cb-e41f13eb41b2--></persistentIdentifier>
  </collection>
    <!--
    add here further collections, if needed
    -->
  </dataset>
  <actions>
    <action name="replication onchange">
      <type>replicate</type>
      <trigger>
        <action>modify object</action>
      </trigger>
      <targets>
        <target id="0">
          <location xsi:type="irodsns:coordinates">
            <irodsns:site type="EUDAT"><!--example:CINECA--></irodsns:site>
            <irodsns:path><!--example:/path/to/destination--></irodsns:path>
            <irodsns:resource><!--defaultResc--></irodsns:resource>
          </location>
        </target>
      </targets>
    </action>
  </actions>
</policy>
```

The policy example is described through RDA Practical Policy WG template

Assuming File_ID = persistent identifier.
Grey lines mean elements which are not part of the current Practical Policy WG template, or are just further notes to clarify the example.

On file ingest (File_ID)
Create replica (File_ID, File_replica_location, File_replica_checksum)
Verify size (File_ID, File_checksum)
Verify checksum (File_ID, File_checksum)
On file (File_name = File_replica_location)
Register metadata (Attribute_name = PID,
Attribute_value = 11100/xxxxxxxxxxxx,
Attribute_unit,
Destination_file = File_replica_location
)
Register metadata (Attribute_name = checksum,
Attribute_value = yyyyyyyyyyyyyyyy,
Attribute_unit,
Destination_file = File_replica_location
)
Register metadata (Attribute_name = EUDAT/ROR,
Attribute_value = File_name (File_ID),
Attribute_unit,
Destination_file = File_replica_location
)
Update metadata (Attribute_name = 10230/LOC,
Attribute_value = 11100/xxxxxxxxxxxx,
Attribute_unit,
Destination_file = File_ID
)
On event (Event_name = replication)
Register event_log (Event_ID = xxxxxxxxxxx,
Event_timestamp = 20140908152134,
Event_context = yyyyyyyyyyyyyyyy,
Event_error_message = zzzzzzzzzzzzzzz
)

The policy example is translated to EUDAT iRODS rules

Replication of an object, defined by an iRODS path. The process implies checks about ownership, size and checksum. Moreover it associates a persistent identifier to the

replica and links together the two handle records, which implements the persistent identifier's attribute sets of the source and of the target of the replication.

```
acPostProcForPut
|-- EUDATTransferSingleFile
```

```
EUDATTransferSingleFile (*path_of_transfered_file, *target_of_transfered_file)
|-- EUDATCatchErrorDataOwner (*path_of_transfered_file, *status_identity)
|-- triggerReplication (*controlfilename, *pid, *path_of_transfered_file, *target_of_transfered_file)
  |-- EUDATGetRorPid (*pid, *ror)
  |-- getEpicApiParameters (*credStoreType, *credStorePath, *epicApi, *serverID, *epicDebug)
  |-- writeFile ("*commandFile", "*pid;*source;*destination;*ror")
    |-- acPostProcForPut
      |-- processReplicationCommandFile ($objPath)
        |-- doReplication (*pid, *source, *destination, *ror, *status)
          |-- msiDataObjRsync (*source, "IRODS_TO_IRODS", "null", *destination, *rsyncStatus)
          |-- triggerCreatePID ("*collectionPath*filepathslash.pid.create", *pid, *destination, *ror)
            |-- writeFile("*commandFile", "create;*pid;*destination;*ror")
              |-- acPostProcForPut
                |-- processPIDCommandFile($objPath)
                  |-- EUDATCreatePID(*parent, *destination, *ror, bool("true"), *new_pid)
                    |-- [...]
                  |-- triggerUpdateParentPID ( "*collectionPath*filepathslash.pid.update",  *parent,
*new_pid )
                    |-- writeFile ("*commandFile",  "update;*pid;*new_pid")
                      |-- acPostProcForPut
                        |-- processPIDCommandFile($objPath)
                          |-- EUDATUpdatePIDWithNewChild ( *parent, *destination)
          |-- updateMonitor ("*collectionPath*filepathslash.pid.update")
            |-- processPIDCommandFile(*file)
              |-- [...]
            |-- EUDATProcessErrorUpdatePID(*file)
|-- EUDATUpdateLogging (*status_transfer_success, *path_of_transfered_file,
*target_of_transfered_file, errormessage)
  |-- EUDATQueue("push", *message, 0)
  |-- EUDATLog(*message, *level)
|-- EUDATCheckError (*path_of_transfered_file, *target_of_transfered_file)
  |-- msiIsData(*target_of_transfered_file, *result, *status)
  |-- EUDATCatchErrorChecksum (*path_of_transfered_file, *target_of_transfered_file)
  |-- EUDATCatchErrorSize (*path_of_transfered_file, *target_of_transfered_file)
  |-- EUDATUpdateLogging (*status_transfer_success, *path_of_transfered_file,
*target_of_transfered_file, *cause)
```

```
EUDATTransferUsingFailLog (*buffer_length)
|-- EUDATQueue ("queuesize", *l, 0)
|-- EUDATQueue ("pop", *messages, *buffer_length)
|-- EUDATTransferSingleFile (*path_of_transfer_file, *target_of_transfer_file)
|-- EUDATQueue ("queuesize", *la, 0)
```

EUDAT schema v1.0

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:irodsns="http://eudat.eu/2013/iRODS-policy" xmlns:tns="http://eudat.eu/2013/policy"
xmlns:xs="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified"
elementFormDefault="qualified" targetNamespace="http://eudat.eu/2013/policy">

 <xs:import namespace="http://eudat.eu/2013/iRODS-policy" schemaLocation="iRODS-policy.xsd"/>

 <xs:element name="policy">
  <xs:annotation>
   <xs:documentation>the policy core consists of the policy target
       (=data set) and the actions (=replication) to apply. A trigger should be
       defined, which could be another action or a time. A set of action
       targets (e.g. the destination of a replication) will be also defined
       The data sets can be identified through a set of  PIDs. The only people
       entitled to apply a policy on that data set should be the owners and the
       administrators of the storage archive which hosts the data set.</xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="dataset">
     <xs:annotation>
      <xs:documentation>the dataset represents a set of collections, which are the target of the policy</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="unbounded" minOccurs="1" name="collection" type="tns:locationPoint">
        <xs:annotation>
         <xs:documentation>the collection represents a set of objects. They can be defined by a single identifier.
                    Or by a set of coordinates, such for example the name of the hosting site and the absolute
path.</xs:documentation>
        </xs:annotation>
       </xs:element>
       <xs:any minOccurs="0"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="actions">
     <xs:annotation>
      <xs:documentation>the element actions represents a set of actions.
            Each action has the dataset as default data source, but each single action can replace it with its
            own data source.</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="unbounded" minOccurs="1" name="action">
        <xs:annotation>
         <xs:documentation>the action represents the task to be performed. it is defined by type and trigger.
                    For example, there could be actions of type "replication", or "integrity check",
                    or "pid assignment". And the trigger to start the action could be a date, a time interval,
                    a change in the dataset or even another action. In fact it is possible to concatenate
                    multiple actions, using the name of a previous action as a trigger.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:sequence>
          <xs:element name="type" type="tns:actionType">
           <xs:annotation>
            <xs:documentation>the type defines the action</xs:documentation>
```

```
      </xs:annotation>
    </xs:element>
    <xs:element name="trigger" type="tns:triggerType">
      <xs:annotation>
        <xs:documentation>the trigger defines the way to activate an action, it could be a date, a time interval,
                    a dataset change or another action.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element minOccurs="0" name="sources">
      <xs:annotation>
        <xs:documentation>the element sources represents a set of locations, which are the data sources for
                    the action.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="source" type="tns:locationPoint">
            <xs:annotation>
              <xs:documentation>a data source for the action.</xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:any minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="targets">
      <xs:annotation>
        <xs:documentation>the element targets represents a set of locations, which are the data destinations
                    for the action.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="target" type="tns:locationPoint">
            <xs:annotation>
              <xs:documentation>a data destination for the action.</xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:any minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:any minOccurs="0"/>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="optional"/>
   <xs:attribute name="id" type="xs:integer" use="optional"/>
  </xs:complexType>
 </xs:element>
 <xs:any minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:attribute name="uniqueid" type="xs:string" use="required"/>
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="version" type="xs:string" use="required"/>
<xs:attribute name="author" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

<xs:complexType name="locationPoint">
 <xs:choice maxOccurs="1" minOccurs="0">
  <xs:element name="persistentIdentifier">
```

```
    <xs:annotation>
      <xs:documentation>the persistentIdentifier represents an identifier, which should be persistent over the time.
          It is usually an alphanumeric string. Further restrictions depend on the type.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="type" type="xs:string" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="location" type="tns:locationType">
    <xs:annotation>
      <xs:documentation>the location is a set of coordinates that defines a position in the space.
        Since it is an abstract type element, there can be many different implementations.</xs:documentation>
    </xs:annotation>
  </xs:element>
 </xs:choice>
 <xs:attribute name="id" type="xs:integer" use="optional"/>
 <xs:attribute name="ref" type="xs:integer" use="optional"/>
</xs:complexType>

<xs:complexType abstract="true" name="locationType"/>

<xs:simpleType name="actionValueType">
 <xs:annotation>
  <xs:documentation>This is the list of "atomic" actions which is expected to be implemented
   at the policy enforcement level</xs:documentation>
 </xs:annotation>
 <xs:restriction base="xs:string">
  <xs:enumeration value="update PID URL"/>
  <xs:enumeration value="update PID checksum"/>
  <xs:enumeration value="update PID child"/>
  <xs:enumeration value="search PID by URL"/>
  <xs:enumeration value="search URL by PID"/>
  <xs:enumeration value="search ROR by PID"/>
  <xs:enumeration value="create PID"/>
  <xs:enumeration value="get checksum by path"/>
  <xs:enumeration value="replicate"/>
  <xs:enumeration value="check replicas"/>
  <xs:enumeration value="remove object"/>
  <xs:enumeration value="copy object"/>
  <xs:enumeration value="compare objects by checksum"/>
  <xs:enumeration value=""/>
 </xs:restriction>
</xs:simpleType>

<xs:complexType name="actionType">
 <xs:simpleContent>
  <xs:extension base="tns:actionValueType">
   <xs:attribute name="policyID" use="optional">
    <xs:simpleType>
     <xs:annotation>
      <xs:documentation>This is the reference (uuid) to a policy document within the policy manager scope.</xs:documentation>
     </xs:annotation>
     <xs:restriction base="xs:string">
      <xs:pattern value="[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
  </xs:extension>
```

```
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="triggerType">
  <xs:annotation>
    <xs:documentation>This is the main trigger type for the action.
          It can be another action, a date, or a periodic time interval.
          The repeat-counter element represents the number of times, the action has to be repeated.
          While its attribute interval-minutes is the interval in minutes between the repetitions.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:choice>
      <xs:element name="action" type="tns:triggerActionType"/>
      <xs:element name="time" type="tns:triggerTimeType"/>
    </xs:choice>
    <xs:element maxOccurs="1" minOccurs="0" name="repeat-counter">
      <xs:complexType>
        <xs:simpleContent>
          <xs:extension base="xs:integer">
            <xs:attribute name="interval-minutes" type="xs:integer" use="required"/>
          </xs:extension>
        </xs:simpleContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="triggerActionValueType">
  <xs:annotation>
    <xs:documentation>This is the list of "atomic" events which are expected to be triggered
          at the policy enforcement level</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="create object"/>
    <xs:enumeration value="modify object"/>
    <xs:enumeration value="remove object"/>
    <xs:enumeration value="replicate"/>
    <xs:enumeration value="check replicas"/>
    <xs:enumeration value=""/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="triggerActionType">
  <xs:simpleContent>
    <xs:extension base="tns:triggerActionValueType">
      <xs:attribute name="ref" type="xs:integer" use="optional"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:simpleType name="triggerTimeType">
  <xs:annotation>
    <xs:documentation>This represents a cron-like expression to specify a date or a period:
       SECONDS / MINUTES / HOURS / DAY OF MONTH / MONTH / DAY OF WEEK / YEAR (OPTIONAL)
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:pattern value="((([0-9]|[0-5][0-9])(-([0-9]|[0-5][0-9]))?,)*([0-9]|[0-5][0-9])(-([0-9]|[0-5][0-9]))?)|(([\*]|[0-9]|[0-5][0-9])/([0-
9]|[0-5][0-9]))|([\?])|([\*]))[\s]((([0-9]|[0-5][0-9])(-([0-9]|[0-5][0-9]))?,)*([0-9]|[0-5][0-9])(-([0-9]|[0-5][0-9]))?)|(([\*]|[0-9]|[0-5][0-
9])/([0-9]|[0-5][0-9]))|([\?])|([\*]))[\s]((([0-9]|[0-1][0-9]|[2][0-3])(-([0-9]|[0-1][0-9]|[2][0-3]))?,)*([0-9]|[0-1][0-9]|[2][0-3])(-([0-9]|[0-
1][0-9]|[2][0-3]))?)|(([\*]|[0-9]|[0-1][0-9]|[2][0-3])/([0-9]|[0-1][0-9]|[2][0-3]))|([\?])|([\*]))[\s]((([1-9]|[0][1-9]|[1-2][0-9]|[3][0-1])(-([1-
9]|[0][1-9]|[1-2][0-9]|[3][0-1]))?,)*([1-9]|[0][1-9]|[1-2][0-9]|[3][0-1])(-([1-9]|[0][1-9]|[1-2][0-9]|[3][0-1]))?(C)?)|(([1-9]|[0][1-9]|[1-
```

2][0-9]|[3][0-1])/([1-9]|[0][1-9]|[1-2][0-9]|[3][0-1])(C)?)|(L(-[0-9])?)|(L(-[1-2][0-9])?)|(L(-[3][0-1])?)|(LW)|([1-9]W)|([1-3][0-9]W)|([\?])|([*]))[\s](((([1-9]|0[1-9]|1[0-2])(-([1-9]|0[1-9]|1[0-2]))?,)*([1-9]|0[1-9]|1[0-2])(-([1-9]|0[1-9]|1[0-2]))?)|(([1-9]|0[1-9]|1[0-2])/([1-9]|0[1-9]|1[0-2]))|(((JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC)(-(JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC))?,)*(JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC)(-(JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC))?)|((JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC)/(JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC))|([\?])|([*]))[\s]((([1-7](-([1-7]))?,)*([1-7])(-([1-7]))?)|([1-7]/([1-7]))|(((MON|TUE|WED|THU|FRI|SAT|SUN)(-(MON|TUE|WED|THU|FRI|SAT|SUN))?,)*(MON|TUE|WED|THU|FRI|SAT|SUN)(-(MON|TUE|WED|THU|FRI|SAT|SUN))?(C)?)|((MON|TUE|WED|THU|FRI|SAT|SUN)/(MON|TUE|WED|THU|FRI|SAT|SUN)(C)?)|(([1-7]|(MON|TUE|WED|THU|FRI|SAT|SUN))?(L|LW)?)|(([1-7]|MON|TUE|WED|THU|FRI|SAT|SUN)#([1-7])?)|([\?])|([*]))([\s]?(([*])?|(19[7-9][0-9])|(20[0-9][0-9]))?| (((19[7-9][0-9])|(20[0-9][0-9]))/((19[7-9][0-9])|(20[0-9][0-9])))?|((((19[7-9][0-9])|(20[0-9][0-9]))(-((19[7-9][0-9])|(20[0-9][0-9])))?,)*((19[7-9][0-9])|(20[0-9][0-9]))(-((19[7-9][0-9])|(20[0-9][0-9])))?)?)"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
targetNamespace="http://eudat.eu/2013/iRODS-policy" xmlns:tns="http://eudat.eu/2013/iRODS-policy"
xmlns:polns="http://eudat.eu/2013/policy" xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:import namespace="http://eudat.eu/2013/policy" schemaLocation="policy.template.xsd"/>
<!--
It implements the abstract type locationType for iRODS.
-->
  <xs:complexType name="coordinates">
    <xs:complexContent>
      <xs:extension base="polns:locationType">
        <xs:sequence>
                <xs:element name="site">
                  <xs:annotation>
            <xs:documentation>
              the site is a unique name associated to an iRODS instance.
            </xs:documentation>
          </xs:annotation>
                <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute use="required" type="xs:string" name="type"/>
                          </xs:extension>
                        </xs:simpleContent>
                </xs:complexType>
                </xs:element>
                <xs:element name="path" type="xs:string">
                  <xs:annotation>
            <xs:documentation>
              the path is an absolute object path associated to an iRODS object and related to a specific iRODS instance
            </xs:documentation>
          </xs:annotation>
                </xs:element>
        <xs:element name="resource" minOccurs="0" type="xs:string">
                  <xs:annotation>
            <xs:documentation>
              the resource is a name, unique within a site, associated to an iRODS resource.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
                <xs:any minOccurs="0"/>
              </xs:sequence>
```

```
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

EUDAT iRODS rules

https://github.com/EUDAT-B2SAFE/B2SAFE-core/tree/latest/docs/administrator.guide.pdf