integrated Rule-Oriented Data System Reference

Arcot Rajasekar[1]
Michael Wan[2]
Reagan Moore[1]
Wayne Schroeder[2]
Sheau-Yen Chen[2]
Lucas Gilbert[2]
Chien-Yi Hou
Richard Marciano[1]
Paul Tooby[2]
Antoine de Torcy[1]
Bing Zhu [2]

[1] National Center for Data Intensive Cyber Environments, School of Information and Library Science, University or North Carolina at Chapel Hill
[2] Center for Advanced Data Intensive Cyber Environments, Institute for Neural Computation, University of California, San Diego

Acknowledgement:
This research was supported by:

- NSF ITR 0427196, Constraint-based Knowledge Systems for Grids, Digital Libraries, and Persistent Archives. (2004-2007)
- NARA supplement to NSF SCI 0438741, Cyberinfrastructure; From Vision to Reality - Developing Scalable Data Management Infrastructure in a Data Grid-Enabled Digital Library System (2005-2006)
- NARA supplement to NSF SCI 0438741, Cyberinfrastructure; From Vision to Reality – Research Prototype Persistent Archive Extension (2006-2007)
- NSF SDCI 0721400, SDCI Data Improvement: Data Grids for Community Driven Applications, (2007-2010)
- NSF/NARA OCI-0848296, NARA Transcontinental Persistent Archive Prototype, (2009-2012)

Table of Contents

1 Introduction

The integrated Rule Oriented Data System is software middleware that organizes distributed data into a shared collection. When data sets are distributed across multiple types of storage systems, across multiple administrative domains, across multiple institutions, and across multiple countries, data grid technology is needed to enforce uniform management properties on the assembled collection. The specific challenges include:

- Management of interations with storage resources that use different access protocols. The data grid provides mechanisms to map from the actions requested by a client to the protocol required by a specific vendor supplied disk, tape, object ring buffer, or object-relational database.
- Support for authentication and authorization across systems that use different identity management systems. The data grid authenticates all access, and authorizes all operations upon the files registered into the shared collection.
- Support for uniform management policies across institutions that may have differing access requirements such as different Institutional Research Board approval processes. The policies controlling use, distribution, retention, disposition, authenticity, integrity, trustworthiness are enforced by the data grid.
- Support for wide-area-network access. To maintain an interactive response, network optimization is required for moving massive files (through parallel I/O streams), for moving small files (through encapsulation of the file in the initial data transfer request), for moving large numbers of small files (aggregation into containers), and for minimizing the amount of data sent over the network (execution of remote procedures on each storage resource).

In response to these challenges, iRODS is an ongoing research and software development effort to provide middleware solutions that enable collaborative research. The goal of the iRODS team is to develop generic software that can be used to implement all distributed data management applications, through changing the management policies and procedures. This has been realized by creating a highly extensible software infrastructure that can be modified without requiring the development of new software code. This report describes the data grid technology in Section 2, the iRODS architecture in Section 3, the Rule-Oriented Programming model in Section 4, the iRODS Rule system in Section 5, the iRODS attributes in Section 6, the iRODS Micro-services in Section 7, extensions to iRODS in Section 8, example rules in Section 9, and a specific set or rules in Section 10.

2 iRODS

The integrated Rule Oriented Data System (iRODS) is software that organizes distributed data into a Shared Collection, while enforcing Management Policies across the multiple storage locations. The iRODS system is generic software infrastructure that can be tuned to implement any desired data management application, ranging from a Data Grid for sharing data in collaborations, to a digital library for publishing data, to a preservation environment, to a data processing pipeline, to a system for federating real-time sensor data streams.

The iRODS technology is developed by the Data Intensive Cyber Environments (DICE) group which is distributed between the University of North Carolina at Chapel Hill (UNC) and the University of California, San Diego (UCSD). The team at UNC is known as NC-DICE, the National Center for Data Intensive Cyber Environments. The team at UCSD is known as CA-DICE, the Center for Advanced Data Intensive Cyber Environments. NC-DICE is associated with the School of Information and Library Science at UNC. CA-DICE is associated with the Institute for Neural Computation at UCSD.

The ideas for the iRODS project have existed for a number of years, and became more concrete through the NSF-funded project "Constraint-based Knowledge Systems for Grids, Digital Libraries, and Persistent Archives" which started in the fall of 2004. The development of iRODS was driven by the lessons learned in nearly ten years of deployment and use in production of the DICE Storage Resource Broker Data Grid technology (SRB) and through applications of theories and concepts from a wide range of well-known paradigms from computer science fields such as active databases, program verification, transactional systems, logic programming, business Rule systems, constraint-management systems, workflows and service-oriented architecture. The iRODS Data Grid is adaptable middleware, in which management policies and management procedures can be dynamically changed without having to re-write software code.

The iRODS Data Grid expresses management policies as computer actionable Rules and management procedures as sets of remotely executable Micro-services. The iRODS Data Grid manages the information required as input and output from the Micro-services (95 Session Variable Attributes and 116 Persistent State Information Attributes), manages composition of 161 Micro-services into Actions that implement the desired management procedures, and enforces 31 active Rules while managing a Distributed Collection. An additional set of 23 alternate Rules is provided as examples of the tuning of Management Policies to specific institutional requirements. The Rules and Micro-services are targeted towards data management functions needed for a wide variety of data management applications. The open source iRODS Data Grid is extensible, supporting dynamic updates to the Rule Base, the incorporation of new Micro-services, and the addition of new Persistent State Information. With the knowledge provided by this paper, a reader will be able to add new Rules, create new Micro-services, and build a data management environment that enforces their institutional Management Policies and procedures.

2.1 A Quick Data Grid Overview

The DICE SRB Data Grid is software infrastructure for sharing data and metadata distributed across heterogeneous resources using uniform APIs (Application Programming Interfaces) and GUIs (Graphical User Interfaces). To provide this functionality, the SRB abstracts key concepts in data management: data object names, sets of data objects, resources, users and groups, and provides uniform methods for interacting with them. SRB hides the underlying physical infrastructure from users by providing global, logical mappings for the digital entities registered into the shared collection. Hence, the peculiarities of storage systems and their access methods, the locations of data, user authentication and authorization across systems, are hidden from the users. A user can access files from an online file system, near-line tapes, relational databases, sensor data streams and the Web without worrying about where they are located, what protocol to use to connect and access the system, and without establishing a separate account or password/certificate to each of the underlying computer systems to gain access, etc. These Virtualization mechanisms are implemented in the SRB system by maintaining mappings and profile metadata in a permanent database system called the MCAT metadata catalog and by providing integrated data and metadata management which links the various sub-systems in a seamless manner.

A key concept is the use of Logical Name Spaces to provide uniform names to entities located in different administrative domains and possibly stored on different types of storage resources. When we use the term, Logical Name Space, we mean a set of names that are used by the Data Grid to describe entities. An implication is that the Data Grid must maintain a mapping from the logical names to the names understood by the remote storage locations. All operations within the iRODS Data Grid are based on the iRODS Logical Name Spaces. The iRODS system internally performs the mapping to the physical names, and issues operations on behalf of the user at the remote storage location. Figure 1 shows this mapping from the names used by a storage repository to the logical names managed by iRODS.

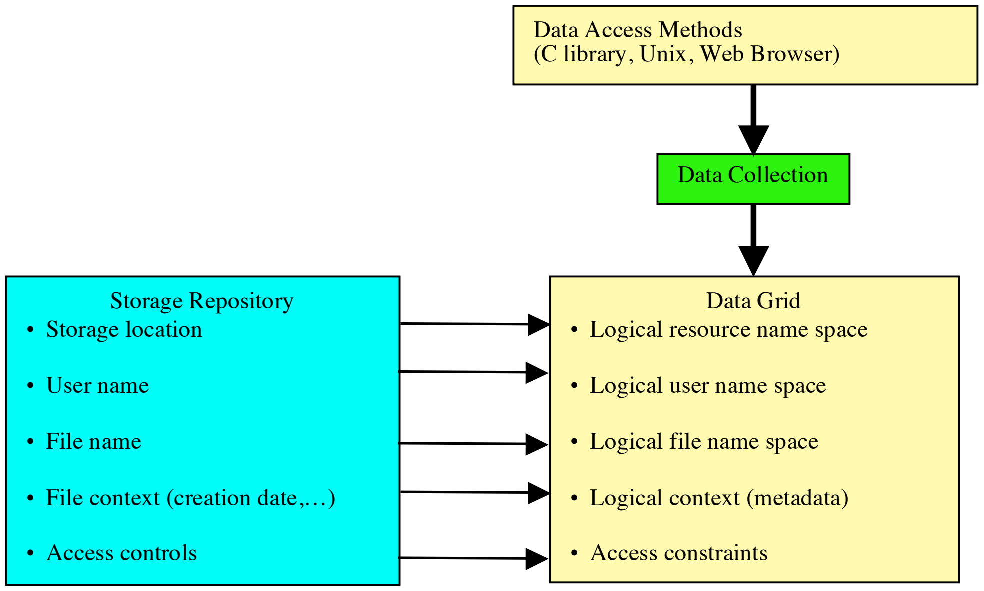
Data Access Methods
(C library, Unix, Web Browser)
Data Collection
Storage Repository
• Storage location
• User name
• File name
• File context (creation date,...)
• Access controls
Data Grid
• Logical resource name space
• Logical user name space
• Logical file name space
• Logical context (metadata)
• Access constraints

Figure 1. Mapping from local names to Logical Name Spaces

Note that the original SRB Data Grid defined three Logical Name Spaces:

1. **Logical names for users.** Each person is known to the Data Grid by a unique name. Each access to the system is authenticated based upon either a public key certificate or a shared secret.
2. **Logical names for files and collections.** The Data Grid supports the logical organization of the distributed files into a hierarchy that can be browsed. A logical collection can be assembled in which files are logically grouped together even though they reside at different locations.
3. **Logical names for storage resources.** The Data Grid can organize resources into groups, and apply operations on the group of resources. An example is load leveling, in which files are distributed uniformly across multiple storage systems. An even more interesting example is the dynamic addition of a new storage resource to a storage group, and the removal of a legacy storage system from the storage group transparently to the users of the system.

Both the SRB and iRODS Data Grids implement Logical Name Spaces for users, files, and storage resources. The best example to start with is the logical names for files and directories in iRODS: the Data Object and Collection names. Each individual file stored in iRODS has both a logical and physical path and name. The logical names are the collection and dataObject names as they appear in iRODS. These are the names that users see when accessing the iRODS data grid.

The iRODS system keeps track of the mapping of these logical names to the physical files (via storage of the mapping in the ICAT Metadata Catalog). Within a single collection, the individual data objects might exist physically on separate file systems and perhaps even on separate host computers. The iRODS system is software middleware that manages information about these files and enables users to access them (if they have the appropriate authorization) regardless of where the files are located.

This is a form of "infrastructure independence" which is essential for managing distributed data. The user or administrator can move the files from one storage file system (Resource) to another, and the logical name the users see remains the same. An old storage system can be replaced by a new one with the physical files migrated to the new storage system. The iRODS system automatically tracks the changes for the users, who continue to reference the files by the persistent Logical Name Space.

The following example illustrates this with the iRODS i-commands (Unix style shell commands that are executed from a command line prompt). The full list of i-commands is given in Appendix A. Comments are added after each shell command as a string in parentheses. The command line prompt is "zuri%" in this example. The commands are shown in *"italics"*. The output is shown in "**bold**"

```
zuri% imkdir t1          (Make a a new sub-collection t1)
zuri% icd t1             (Make t1 the current default working directory)
zuri% iput file1         (Store a file into iRODS into the working directory)
zuri% ils                (Show the files in iRODS, that is the logical names)
/zz/home/rods/t1:
  file1
zuri% ils -l             (Show more detail, including the logical resource)
/zz/home/rods/t1:
  rods        0 demoResc          18351 2008-11-17.12:22 & file1
zuri% ils -L             (Show more detail, including the physical path)
/zz/home/rods/t1:
  rods        0 demoResc          18351 2008-11-17.12:22 & file1
/scratch/slocal/rods/iRODS/Vault/home/rods/t1/file1
```

The first item on the *ils* output line is the name of the owner of the file (in this case "rods"). The second item is the replication number which we further explain below. The third item is the Logical Resource Name. The fourth item is the size of the file in bytes. The fifth item is the date. The sixth item ("&") indicates the file is up to date. If a replica is modified, the "&" flag is removed from the out-of-date copies.

In the above example, the iRODS logical name for the file was "file1" and the file was stored in the logical collection "/zz/home/rods/t1". The original physical file name was also "file1". The logical resource name was "demoResc". When iRODS stored a copy of the file onto the storage resource "demoResc", the copy was made at the location:
"/scratch/slocal/rods/iRODS/Vault/home/rods/t1/file1"

Any storage location at which an iRODS Server has been installed can be used for the repository through the "-R" command line option. Even though the example below stores "file2" on storage resource "demoRescQe2", both "file1" and "file2" are logically organized into the same logical collection "/zz/home/rods/t1".

```
zuri% iput -R demoRescQe2 file2  (Store a file on the "demoRescQe2" vault/host)
zuri% ils
/zz/home/rods/t1:
  file1
  file2
zuri% ils -l
/zz/home/rods/t1:
  rods          0 demoResc              18351 2008-11-17.12:22 & file1
  rods          0 demoRescQe2             64316 2008-11-17.12:29 & file2
zuri% ils -L
/zz/home/rods/t1:
  rods          0 demoResc              18351 2008-11-17.12:22 & file1
        /scratch/slocal/rods/iRODS/Vault/home/rods/t1/file1
  rods          0 demoRescQe2           64316 2008-11-17.12:29 & file2
        /scratch/s1/schroede/qe2/iRODS/Vault/home/rods/t1/file2
```

Other operations can be performed upon files:

- **Registration** is the creation of iRODS metadata that point to the file without making a copy. The *ireg* command is used instead of *iput* to register a file. In the example below, "file3a" is added to the logical collection. Note that its physical location remains the original file system ("/users/u4/schroede/test/file3"), and a copy was not made into the iRODS Data Grid.

```
zuri% ireg /users/u4/schroede/test/file3 /zz/home/rods/t1/file3a
zuri% ils
/zz/home/rods/t1:
  file1
  file2
  file3a
zuri% ils -l
/zz/home/rods/t1:
  rods          0 demoResc              18351 2008-11-17.12:22 & file1
  rods          0 demoRescQe2             64316 2008-11-17.12:29 & file2
  rods          0 demoResc              10627 2008-11-17.12:31 & file3a
zuri% ils -L
/zz/home/rods/t1:
  rods          0 demoResc              18351 2008-11-17.12:22 & file1
        /scratch/slocal/rods/iRODS/Vault/home/rods/t1/file1
  rods          0 demoRescQe2           64316 2008-11-17.12:29 & file2
        /scratch/s1/schroede/qe2/iRODS/Vault/home/rods/t1/file2
  rods          0 demoResc              10627 2008-11-17.12:31 & file3a
        /users/u4/schroede/test/file3
```

- **Replication** is the creation of multiple copies of a file on different physical resources. The "*irepl*" command is used in place of "*ireg*". Note that the replication is done on a file that is already registered or put into an iRODS logical collection.

```
zuri% irepl -R demoRescQe2 file1
zuri% ils
/zz/home/rods/t1:
  file1
  file1
  file2
  file3a
zuri% ils -l
/zz/home/rods/t1:
  rods        0 demoResc           18351 2008-11-17.12:22 & file1
  rods        1 demoRescQe2          18351 2008-11-17.12:33 & file1
  rods        0 demoRescQe2          64316 2008-11-17.12:29 & file2
  rods        0 demoResc           10627 2008-11-17.12:31 & file3a
zuri% ils -L
/zz/home/rods/t1:
  rods        0 demoResc           18351 2008-11-17.12:22 & file1
      /scratch/slocal/rods/iRODS/Vault/home/rods/t1/file1
  rods        1 demoRescQe2          18351 2008-11-17.12:33 & file1
      /scratch/s1/schroede/qe2/iRODS/Vault/home/rods/t1/file1
  rods        0 demoRescQe2          64316 2008-11-17.12:29 & file2
      /scratch/s1/schroede/qe2/iRODS/Vault/home/rods/t1/file2
  rods        0 demoResc           10627 2008-11-17.12:31 & file3a
      /users/u4/schroede/test/file3
```

The replica is indicated by listing the file twice, once for the original vault where the file was stored in the iRODS "demoResc" storage vault, and once for the location where the replica was stored in the "demoRescQe2" storage vault. The replication number is listed after the name of the owner (the second item on the output line). Note that the creation dates of the replicas may be different.

A second critical point is that the operations that were performed to put, register, and replicate files within the iRODS Data Grid, were executed under the control of a Rule Engine. Computer actionable Rules are read from a Rule Base "core.irb" and used to select the procedures that will be executed on each interaction with the system. In the above examples, a default Policy was used to specify how the pathname for each file was defined when the file was written to an iRODS storage resource (vault). The specific default Rule that was used, set the path name under which the file was stored to be the same as the logical path name. This makes it easy to correlate files in storage resources with files in the iRODS logical collection. We explain the syntax of this Rule in section 4.2 on iRODS Rules:

```
acSetVaultPathPolicy||msiSetGraftPathScheme(no,1)|nop
```

When managing large numbers of files, the remote physical storage location may have a maximum number of files that can be effectively stored in a single directory. When too many files are put into a single physical directory, the file system becomes unresponsive. The iRODS Data Grid provides a procedure (Micro-service) that can be used to impose two levels of directories and create a random name for the physical path name to the file. We can replace the default Rule in the "core.irb" Rule Base for controlling definition of path names with the following Rule:

```
acSetVaultPathPolicy||msiSetRandomScheme|nop
```

Once the core.irb file is changed, all subsequent operations will be controlled by the new set of Rules. In the example below, a file is put into the iRODS Data Grid using the new Rule set. We observe that the physical file path is now ".../rods/10/9/file4.1226966101" instead of ".../rods/t1/file4". That is, the new Rule assigns a random number at the end of the physical name and creates and uses two levels of directories ("/10/9/") to keep the number of items in each directory low. In some cases, this will provide improved performance and greater capacity.

```
zuri% iput file4
```

```
zuri% ils
/zz/home/rods/t1:
  file1
  file1
  file2
  file3a
  file3b
  file4
zuri% ils -l file4
  rods          0 demoResc                 27 2008-11-17.15:55 & file4
zuri% ils -L file4
  rods          0 demoResc                 27 2008-11-17.15:55 & file4
        /scratch/slocal/rods/iRODS/Vault/rods/10/9/file4.1226966101
```

This simple example illustrates why the iRODS Data Grid is viewed as a significant advance over the SRB Data Grid technology. The policy for defining how physical files will be named is under the control of the Data Grid administrator. Changes to the policies can be made without having to write new software. The SRB Data Grid was a one-size fits all system. The policies used in managing the data at the server level were hard-coded. Also, if a user wanted to perform complex sets of operations of the files, they had to create a script or program that was run at the client level. If a community wanted to perform a different type of operation (say change the way the access control for files was implemented), they had to change the SRB code with the hope that it did not introduce unintended side-effects on other operations.

Examples for such customizable requirements come from the SRB user community itself. For example, one user wanted a feature in which all files in a particular collection should be disabled from being deleted even by the owner or Data Grid administrator, but other collections should behave as before! This kind of collection-level data management Policy is not easily implemented in the SRB Data Grid without a lot of work. Also the required software changes are hardwired, making it difficult to reapply the particular SRB Data Grid instance in another project that has a different data deletion policy. Another example is based on a request to use additional or alternate checks for access controls on sensitive files. This again required specialized coding to implement the capability in the SRB. A third example occurred when a user wanted to asynchronously replicate (or extract metadata from, or create a lower resolution file from) newly ingested files in a particular collection (or file type). Implementation of this feature required additional coding and asynchronous scheduling mechanisms not easily done in the SRB.

3 iRODS Architecture

The iRODS system belongs to a class of middleware which we term *adaptive middleware*. The Adaptive Middleware Architecture (AMA for short) provides a means for adapting the middleware to meet the needs of the end user community without requiring that they make programming changes. One can view the AMA middleware as a glass box in which users can see how the system works and can tweak the controls to meet their demands. Usually middleware is the equivalent of a black box for which no changes are programmatically possible to adjust the flow of the operations, except pre-determined configuration options that may allow one to set the starting conditions of the middleware.

There are multiple ways to achieve an Adaptive Middleware Architecture. In our approach, we use a particular methodology that we name *Rule Oriented Programming* or ROP for short. The *Rule Oriented Programming* concept is discussed in some detail in section 3.

The iRODS architecture provides a means for encoding customization of data management functionalities in an easy and declarative fashion using the ROP paradigm. This is accomplished by coding the processes that are being performed in the iRODS Data Grid system as Rules (see section 4.2 on Rules) that explicitly control the operations that are being performed when a Rule is invoked by a particular task. These operations are called Micro-services (see section 6 on Micro-services) in iRODS and are C-functions that are called when executing the Rule body. One can modify the flow of tasks when executing the Rules, by interposing new Micro-services (or Rule invocations) in a given Rule or by changing and recompiling the Micro-service code. Moreover, one can add another Rule in the Rule Base for the same task, but with a higher priority so that it is chosen before an existing Rule. This pre-emptive Rule will be executed before the original Rule. If there is a failure in the execution of any part of this new Rule then the original Rule is executed.

The major features of the iRODS architecture include the following:

1) **Data Grid Architecture** based on a client/server model that controls interactions with distributed storage and compute resources.
2) A **Metadata Catalog** managed in a database system for maintaining the attributes of data and state information generated by remote operations.
3) A **Rule System** for enforcing and executing adaptive Rules

The *iRODS Server* software is installed at each physical location where data will be stored. The *iRODS Server* translates operations into the protocol required by the remote storage system. In addition, a Rule Engine is also installed at each storage location. The Rule Engine controls operations performed at that site. Figure 2 illustrates the components of the iRODS system, including a Client for accessing the Data Grid, Data Grid Servers installed at each storage system, a Rule Engine installed at each storage location, the iCAT Metadata Catalog that stores the persistent state information, and a Rule Base that holds the Rules.

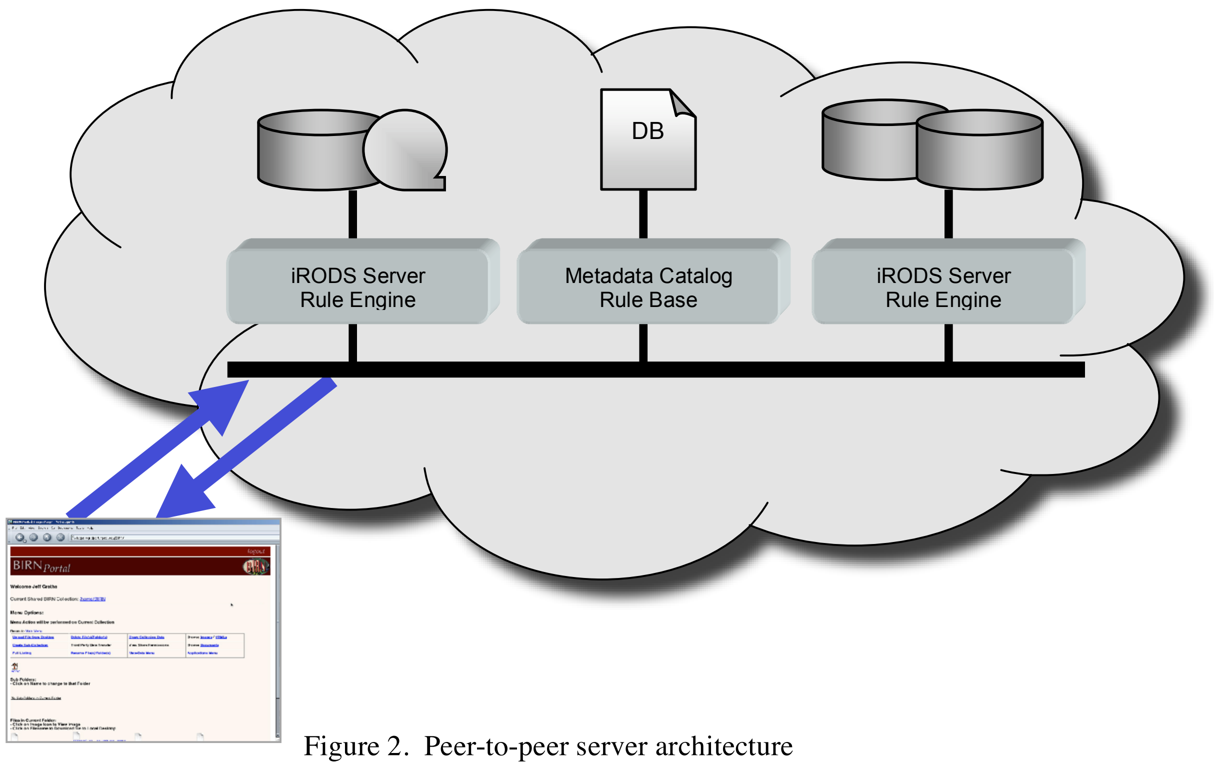

Figure 2. Peer-to-peer server architecture

The Rule Base is replicated to each iRODS server. When the iRODS server is installed at a particular storage location, an iRODS Rule Base is also installed. Future enhancements to the iRODS system will investigate automated updates to the Rule Base depending upon the version that is installed at the coordinating Metadata Catalog site. In the current approach, each site can choose to run a different set of Rules, including Rules that are specific to the type of storage system at the site's location.

In order to create a highly extensible architecture, the iRODS Data Grid implements multiple levels of *virtualization*. As shown in Figure 3, the Actions that are requested by a client are mapped to sets of standard operations, called Micro-services. A single client request may invoke the execution of multiple Micro-services and Rules.

In turn, the Micro-services execute standard operations that are performed at the remote storage location. The standard operations are based upon the Posix I/O functions:

- Create a file
- Open a file
- Close a file
- Read a file
- Write a file
- Unlink a file
- Seek to a location in a file
- Force completion of pending disk write
- Display file status
- List information about files
- Make a directory
- Remove a directory
- Change access permission
- Open a directory
- Close a directory
- Read a directory

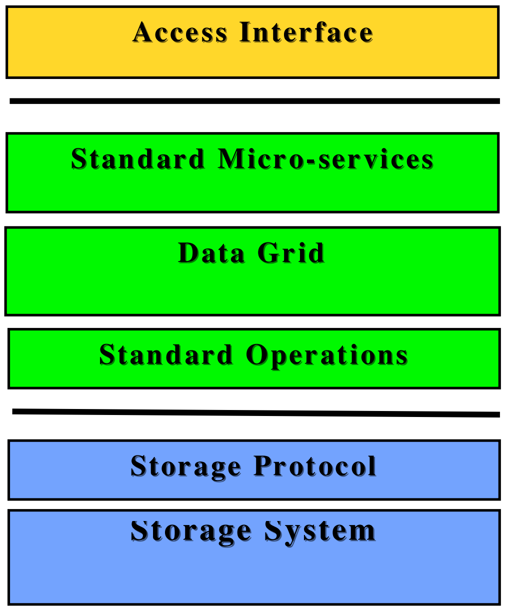

Figure 3. iRODS layered architecture

A given Micro-service can invoke multiple Posix I/O calls. Thus the Micro-service is intended to simplify expression of procedures by providing an intermediate level of functionality that is easier to chain into the desired Action.

The Posix I/O calls are then mapped into the protocol required by the storage system through a driver that is written explicitly for that storage system. The Data Grid Middleware corresponds to the software that maps from the Actions requested by the *client access interface* to the *storage protocol* required by the storage system.

This approach means that new access mechanisms can be added without having to modify the standard operations performed at the storage systems. Also, new types of storage systems can be integrated into the system by writing new drivers without having to modify any of the access clients.

The list of Posix I/O calls includes the ability to do partial I/O upon a file at a storage device. Since not all of the storage systems that may be integrated into the Data Grid have this ability, caching of files on a second storage system may be necessary. This approach was used to support manipulation of files within the NCAR mass storage system, which only allowed complete file input and retrieval. A copy was made on a disk file system, where partial I/O commands were then executed.

The iRODS Data Grid effectively implements a *distributed operating system*. The remote operations generate Structured Information that must then be passed between Micro-services and to the Client.

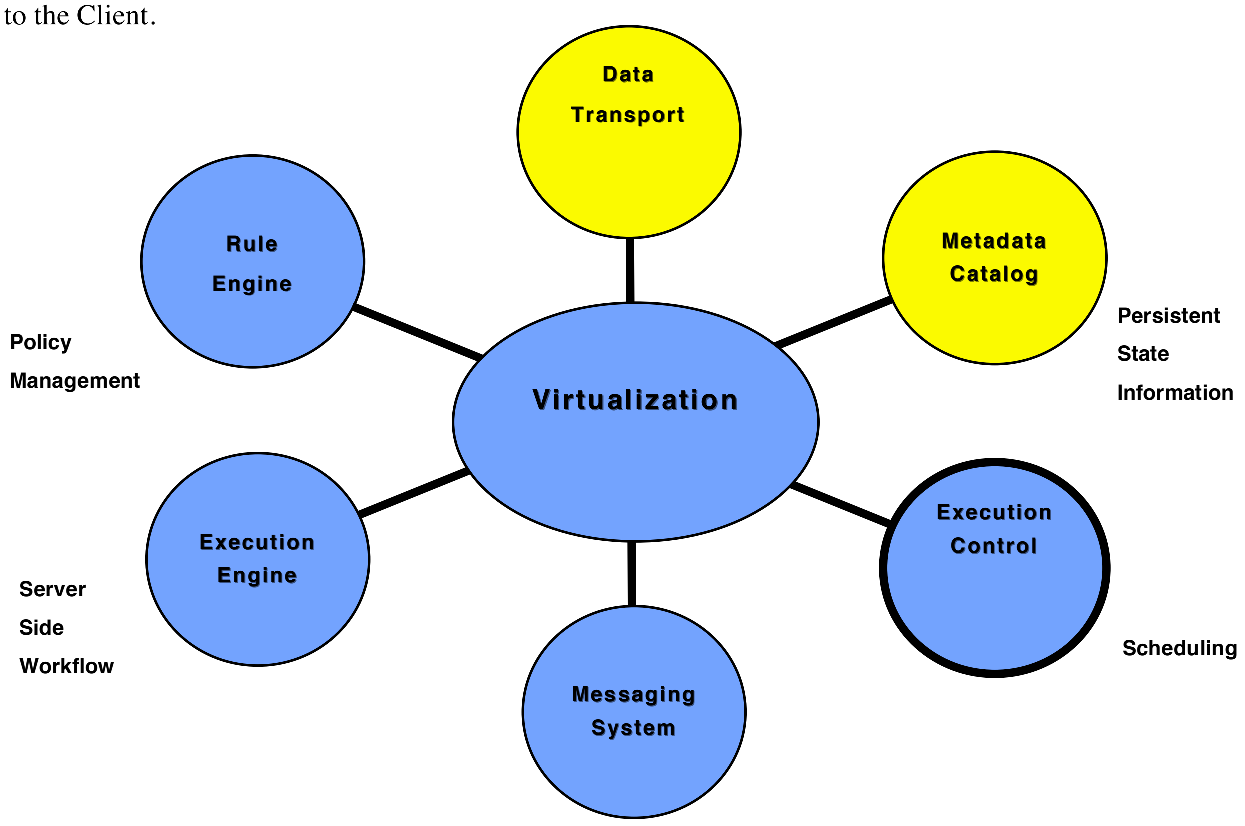

Figure 4. iRODS Distributed Operating System

The iRODS framework implements multiple mechanisms needed to control the exchange of Structured Information, the execution of the remote Micro-services, and the interactions between the Rule Base, Rule Engine, Metadata Catalog, and network. The iRODS framework is illustrated in Figure 4. The components include:

- **Data Transport:** Manages parallel I/O streams for moving very large files (greater than 30 Megabytes in size) over the network. An optimized transport protocol is used that sends the data with the initial transfer request for small files less than 30 Megabytes in size.
- **Metadata Catalog:** Manages interactions with a vendor-specific or open source database to store descriptive metadata and Persistent State Information.
- **Rule Engine:** Manages the computer actionable Rules to control selection of Micro-services.
- **Execution Control:** Manages scheduling of the Micro-services that are selected by the Rule engine. Micro-services may be executed at multiple storage locations, or deferred for execution, or executed periodically.
- **Execution Engine:** Manages execution of a Micro-service. The Micro-services are written in the "C" language, and are compiled for a specific operating system. The execution engine

manages the input of data to the Micro-service, and manages the output of data from the Micro-service.
- **Messaging System:** Manages high-performance message exchange between iRODS Servers. This is required when Structured Information is moved between Micro-services that are executed at different storage locations.
- **Virtualization Framework:** Coordinates interaction between the framework components.

The mechanisms implemented within the iRODS system are sufficiently powerful to control the execution of workflows at each remote storage location. This linking of multiple remote procedures is called a *server-side workflow* to differentiate it from workflows executed at a compute server under the control of a client (*client-side workflows*). This implies that a Rule represents a workflow that will be executed to implement a desired Client Action. The types of workflows that should be executed directly on a storage system have low complexity - a small number of operations compared to the number of bytes in the file. If the complexity is sufficiently small, then the amount of time needed to perform the workflow will be less than the time that would have been required to move the file to a computer server. For workflows that have high complexity, it is faster to move the file to a compute server than it is to perform the operations at the remote storage system. Thus iRODS is expected to control low-complexity workflows that can be most efficiently executed at each storage system. Examples of low-complexity workflows include the extraction of a data subset from a large file, or the parsing of metadata from a file header.

An implication of the restriction to low-complexity workflows is that the iRODS system should also restrict the Rule Set to control a well-defined set of Micro-services. The Rule Base should encompass a small number of Rules that are highly tuned to the specific data management policies for which the shared collection was created.

3.1 Virtualizations in iRODS

iRODS can be thought of as providing a new abstraction for data management processes and policies (using the logical Rule paradigm) in much the same way that the SRB provided abstractions for data objects, collections, resources, users and metadata. The goal is to be able to characterize the Management Policies that are needed to enforce *authenticity, integrity, access restrictions, data placement, and data presentation*, and to automate the application of the Policies for services such as administration, authentication, authorization, auditing and accounting, as well as data management policies for replication, distribution, pre- and post-processing and metadata extraction and assignment. The Management Policies are mapped onto Rules that control the execution of all data management operations. iRODS can be seen as supporting four types of virtualization beyond those supported by a Data Grid such as the SRB.

- **Workflow virtualization.** This is the ability to manage the execution of a distributed workflow independently of the compute resources where the workflow components are executed. This requires the ability to manage the properties of the executing jobs. iRODS implements the concept of workflows through chaining of Micro-services within nested Rule sets and using shared logical variables that control the workflow.
- **Management Policy virtualization.** This is the expression of Management Policies as Rules that can be implemented independently of the remote storage system. We characterize Management Policies in terms of *policy attributes* that control desired outcomes. For each desired outcome, Rules are defined that control the execution of the standard remote operations. For each Rule application, Persistent State Information is maintained to describe

the result of the remote operation. Consistency Rules can be implemented that verify that the remote operation outcomes comply with the Policy Attributes. Rule-based data management infrastructure makes it possible to express Management Policies as Rules and define the outcome of the application of each Management Policy in terms of updates to the Persistent State Information. iRODS applies the concept of transactional Rules using datalog-type Event-Condition-Action Rules working with persistent shared metadata. iRODS implements traditional ACID database properties (Atomicity, Consistency, Isolation, Durability).

- **Service virtualization.** The operations that are performed by Rule-based data management systems can be encapsulated in Micro-services. A Logical Name Space can be constructed for the Micro-services that makes it possible to name, organize, and upgrade Micro-services without having to change the Management Policies. This is one of the key capabilities needed to manage versions of Micro-services, and enable a system to execute correctly while the Micro-services are being upgraded. iRODS Micro-services are constructed on the concepts of well-defined input-output properties, consistency verification, and roll-back properties for error recovery. The iRODS Micro-services provide a compositional framework realized at run-time.
- **Rule virtualization.** This is a Logical Name Space that allows the Rules to be named, organized in sets, and versioned. A Logical Name Space for Rules enables the evolution of the Rules themselves.

3.2 iRODS Components

The iRODS system consists of Servers that are installed at each storage location, a central Metadata Catalog, and Clients. The iRODS Server contains both the driver that issues the local storage resource protocol and a Rule Engine that controls operations performed at the storage location. The components of the iRODS system are shown in Figure 5.

The client interface is typically built on either a C library interface to the iRODS Data Grid or a Java I/O class library, or uses Unix-style shell commands. These interfaces send messages over the network to an iRODS Server. The server interacts with the iRODS iCAT Metadata Catalog to validate the user identity, and authorize the requested operation. The location where the operation will be performed is identified, and the operation request is forwarded to the remote storage location. A Rule Engine at the storage location selects the Rules to invoke (Rule Invoker) from the Rule Base, retrieves current state information as needed from the Configuration files and the Metadata Persistent Repository, stores the current state information in a Session memory, and then invokes the Micro-services specified by the Rules.

An important component that is being developed is the administrator interface. As Rules, Micro-services, Resources, and Metadata are changed, the consistency of the new system must be verified. The design allows for the execution of consistency modules to verify that the new system is compliant with selected properties of the old system. A Rule composer that checks input and output attributes of Micro-services is needed to simplify creation of new Rules. The Data Grid Administrator manages and designs the Rules used by iRODS, and executes administrative functions through the icommand “iadmin”.

Interaction with the storage location is done through a software driver module that translates requests to the protocol of the specific storage device. This makes it possible to store data in a wide variety of types of storage systems.

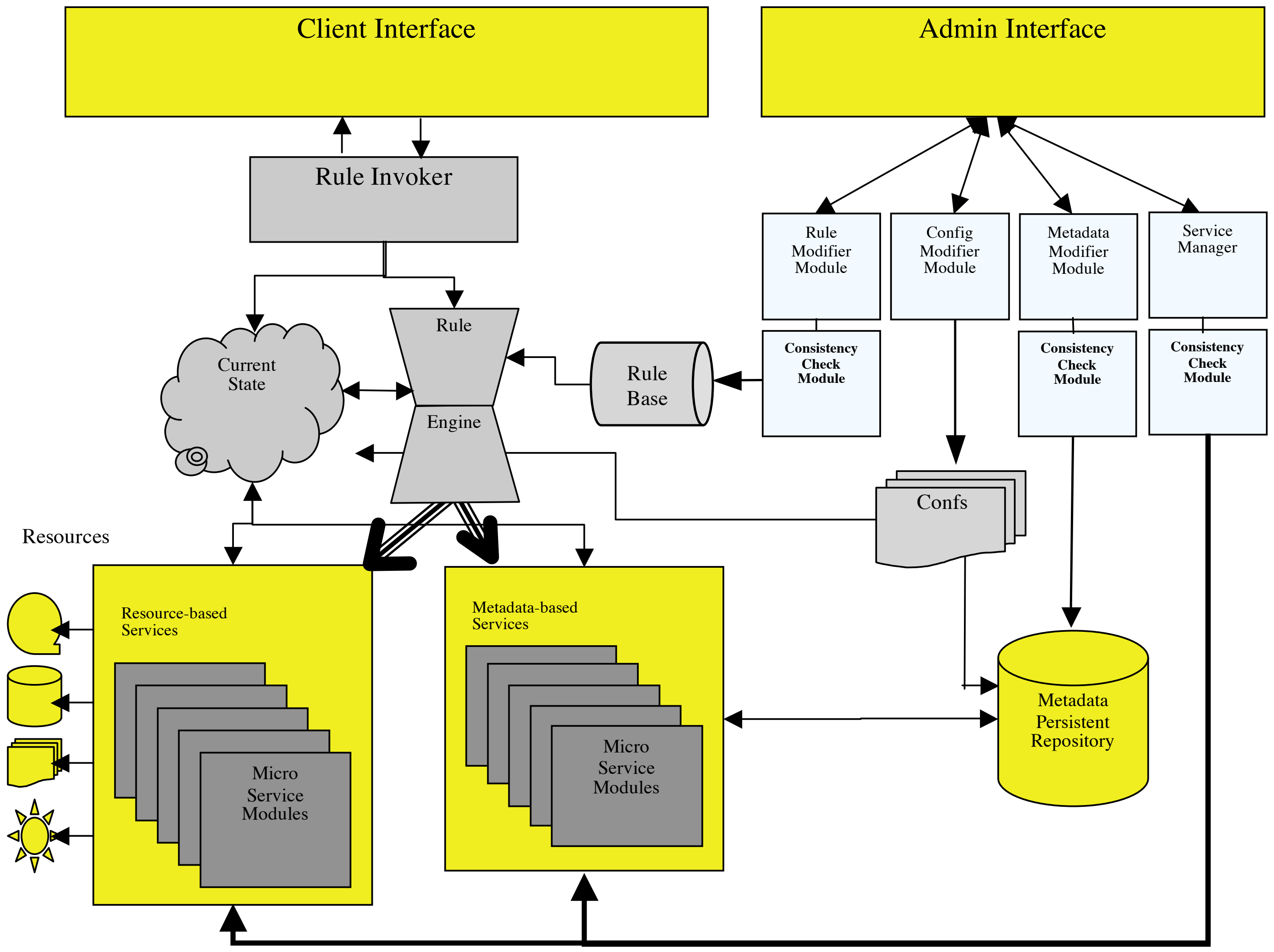

Figure 5. iRODS Architecture Components

4 ROP – Rule-Oriented Programming

Rule-oriented programming (ROP) is a different (though not new) paradigm from normal programming practice. In Rule-Oriented Programming, the power of controlling the functionality rests more with the users than with system and application developers. Hence, any change to a particular process or policy can be easily constructed by the user and tested and deployed without the aid of system and application developers.

ROP can be viewed as lego-block type programming. The building blocks for the ROP are “Micro-services.” Micro-services are small, well-defined procedures/functions that perform a certain task. Micro-services are developed and made available by system programmers and application programmers. Users and administrators can “chain” these Micro-services to implement a larger macro-level functionality that they want to use or provide for others. For example, one of the Micro-services might be to “createCollection”, another one might be to “computeChecksum” and a third to “replicateObject”.

Larger macro-level functionalities are called “Actions.” Since one can perform an Action in more than one way, each Action might have one or more chains of Micro-services associated with it. Hence one can view an Action as a name of a task and the chains of Micro-services as the procedural counterpart for performing the task. Since there may be more than one chain of Micro-services possible for an Action, iRODS provides two mechanisms for finding the best choice of Micro-service to apply in a given situation. The first mechanism is a “condition” that can be attached to any Micro-service chain which will be tested for compliance before executing the chain. These conditions in effect act as guards that check permission for execution of the chain. The triplet <action, condition, chain> is called a “Rule” in the ROP system. (There is another concept called “recovery micro–services chain” that will be introduced later which will make the Rule into a quartet).

The second mechanism that is used for identifying an applicable Rule is a “*priority*” associated with a chain. Priority is an integer associated with a Rule that identifies the order in which it will be tested for applicability: the lower the number, the higher the priority. In our current implementation, the priority is associated with the way the Rules are read from Rule files upon initialization. The earlier the Rule is read and included in the Rule Base, the higher its priority compared to all the Rules for the same Action.

The implementation of iRODS includes another helpful feature. The chain of Micro-services is not limited to just Micro-service procedures and functions but can also include Actions. Hence when executing a chain of Micro-services, if an Action needs to be performed, the system will invoke the Rule application program for the new Action. Hence, an Action can be built using other Actions. Care should be taken so that there are no infinite cycles in any loop formed by recursive calls to the same Action.

In summary, the first three components of a Rule consist of an action name, a testable condition, and a chain of action and a set of Micro-services. Each Rule also has a priority associated with it.

The fourth component of a Rule is a set of recovery Micro-services. An important question that arises is what should be done when a Micro-service fails (returns a failure). A Micro-service failure means that the chain has failed and hence that instance of the Action has failed. But, as mentioned above, there can be more than one way to perform an Action. Therefore, when a failure is encountered, one can try to execute another Rule of a lower priority for that Action. When doing this, a decision must be made about the changes that were made to variables generated by the failing chain of Micro-services. In particular, any side-effects (such as a physical file creation on a disk) that might have happened as a result of successful Micro-service execution before the failure must be handled. The same question applies to any changes made to the metadata stored in the iCAT.

The iRODS architecture is designed so that if one Rule for an Action faila, another applicable Rule of lower priority is attempted. If one of these Rules succeeds then the Action is considered to be successful. To make sure that the failing chain of Micro-services does not leave any changes and side-effects, we provide the following mechanism. For every Micro-service in the chain in a Rule, the Rule designer specifies a “recovery Micro-service or Action” that is listed in the same order as in the chain.

A recovery Micro-service is just like any Micro-service, but with the functionality that it recovers from the task rather than performs a task. A recovery Micro-service should be able to recover from the multiple types of errors that can result from an execution of the corresponding Micro-service. More importantly, a recovery Micro-service should also be able to recover from a successful Micro-service execution! This feature is needed because in the chain of Micro-services, when a downstream Micro-service fails in a chain of Micro-services, one should recover from all changes and side-effects performed, not only those of the failing Micro-service but also those of all the successful Micro-services in the chain performed prior to the failed Micro-service. The recovery mechanism for an Action is of the same type as that of a recovery Micro-service, though one only needs to recover from successful completion of a Rule, when a later Micro-service/Rule fails in the chain. If an Action fails, by definition, any Rule for that Action would have recovered from the effects of the failed Action!

During the recovery process, the recovery Micro-services for all the successful Micro-services will be performed, so that when completed, the effect of the Rule for that Action is completely neutralized. Hence, when an alternate, lower priority Rule is tried for the same Action, it starts with the same initial setting used by the failed Rule. This property of complete recovery from failure is called the “atomicity” property of a Rule. Either a Rule is fully successful with attendant changes and side-effects completed, or the state is unchanged from the time of invocation. If all Rules for a particular Action fail, one can see that the system is left in the same state as if the Rule was not executed. One can view this as a “transactional” feature for Actions. The concepts of atomicity and transactions are adapted from relational databases.

In summary, every Rule has an Action name, a testable condition, a chain of Actions and Micro-services, and a corresponding chain of recovery Actions and Micro-services. Each Rule also has an associated priority.

For example, consider a very simple Rule for data ingestion into iRODS with two Micro-services, “createPhysicalFile” and “registerObject” and no conditions. The Rule creates a copy of the file in an iRODS storage vault and registers the existence of the file into the iCAT Metadata Catalog. The Data Grid administrator can define an alternate Rule of a higher priority, which can also check whether the data type of the file is “DICOM image file” and invoke an additional Micro-service called “extractDICOMMetadata” to populate the iCAT with metadata extracted from the file, after the file has been created on the iRODS Server and registered in the iCAT Metadata Catalog.

We did not implement the idea that an Action or Micro-service should be implicitly tied to a single recovery Action or Micro-service. While this might make it easier to find a recovery service by this implicit link, we recognized that recovery from an Action or Micro-service can be dependent upon where and how it is being invoked. Sometimes, a simpler recovery would do the trick instead of a more complex recovery. For instance, a database rollback might suffice if one knew that the Action started a new iCAT Metadata Catalog database transaction. Otherwise a longer sequence of recovery delete/insert and update SQL statements is needed to recover from multiple SQL statement activities. So we give the Rule designer the ability to tie in the appropriate recovery for each Micro-service or Action as part of the Rule instead of having the system or application designer who develops the Micro-service do this.

The Rules for version 2.0 of iRODS are stored in “iRODS Rule Base” files (files with extension “.irb”). These files are located in the server/config/reConfig directory. One can specify use of more than one irb file which will be read one after the other during initialization. By default the single core.irb file will be read.

Now that we have seen what comprises a Rule, an Action and a Micro-service, we will next look at how a Rule is invoked and what type of mechanisms are available to communicate to the

Micro-services and Actions. As one can see, Micro-services (by this we also mean Actions) do not operate in a vacuum. They need input and produce output and communicate with other Micro-services, make changes to the iCAT database, and have side-effects such as file creation. Hence, the question arises, what do the Rules operate on?

To answer this question, we need the concept of a *session*. A session is a single iRODS Server invocation. The session starts when a client connects to the iRODS Server and ends when the Client disconnects. During the session, there are two distinct “state” mechanisms that are operated upon by Actions and Micro-services.

1. **Persistent State Information** (denoted by #) as defined by the attributes and schema that are stored in the iCAT catalog, and
2. **Session State Information** (denoted by $) is temporary information that is maintained in the memory as a C-structure only during the time period when the Actions are being performed.

The Persistent State Information (denoted by #) is the content that is available across sessions and persists in the iCAT Metadata Catalog after the Action has been performed, provided proper commit operations were performed before the end of the session. The Session State Information (denoted by $) does not persist across sessions but is a mechanism for Micro-services to communicate with each other during the session without going to the database # to retrieve every attribute value. One can view the # as a “persistent blackboard” through which sessions communicate with each other (sessions possibly started by different users). Similarly one can view $ as a “temporary blackboard” that communicates within a session some of the state information needed by Actions and Micro-services. Note that $ persists beyond a single Action, and hence when one is performing multiple Actions during a single session, the memory of the earlier Actions (unless destroyed or over-written) can be utilized by a current Action.

The $ structure is a complex C structure, called the Rule Execution Infrastructure (REI). The REI structure will evolve during the implementation of new iRODS versions as additional functionality is added. In order to hide the physical nature of the $ structure, we adopt a Logical Name Space that can support extensions to the physical table structure. The $ Logical Name Space defines a set of “$variables” which map to a value node (possibly a leaf-node, but need not be one) in the REI structure. This mapping is defined by “data variable mapping” files (files with extensions “.dvm”) which are located in the server/config/reConfig directory. The $variable mapping defines how the system can translate a $variable name to a path name in the REI structure. iRODS provides utility routines to get the run-time address of the denoted value. One can have more than one definition for a $variable name. This is needed because different Micro-services might use the same $variables to denote different paths in the REI structure. The utility routine for getting values from the REI structure will cycle through these definitions to get to the first non-NULL value. We strongly recommend the use of unique definitions for each $variable, and we advocate that designers use multiple definitions only under very controlled circumstances.

The #variables have a Logical Name Space as defined by the attribute set of the iCAT Metadata Catalog. The mappings from #variables to columns in tables in the iCAT schema are defined in the lib/core/include/rodsGenQuery.h. These variables can be queried using the generic query call “iquest” available for accessing data from the iCAT database.

4.1 Session State Variables

The mapping of the $variables that are defined in version 2.0 of the iRODS Data Grid to structures in the Rule Execution Infrastructure (REI) is listed in Appendix B. The meaning of each $variable is listed in Table 1. The $variables can be defined as input to a Rule, can be used to define input to a Micro-service, and can be used to define output from a Micro-service that will be used by subsequent Micro-services. The addition of new variables to the Data Grid requires the re-compilation of the software. Thus, an attempt has been made to provide a complete set of variables needed for the management and access of files within a Data Grid. In practice, a small fraction of these variables is needed for most administrator-modified Rules.

Table 1. Meaning of Session State Variables

otherUser	Pointer to other user structure, useful when giving access
otherUserName	Other user name in the form 'name@domain'
otherUserZone	Name of the iCAT metadata catalog, or Data Grid, or zone. Unique globally
otherUserType	Role of a other user (rodsgroup, rodsadmin, rodsuser, domainadmin, groupadmin, storageadmin, rodscurators)
otherSysUidClient	Internal identifer for the other user
rescName	Logical resource name
objPath	Physical path name of a file on the logical resource
destRescName	Logical resource name of the destination for the operation
backupRescName	Logical resource name
dataType	Data type of the object
dataSize	Size of files in bytes
chksum	Checksum (MD5) of the file
version	Version number of the file
filePath	Logical path name
replNum	Replica number
replStatus	Replica status (0 if up-to-date, 1 if not up-to-date)
dataOwner	Owner of the file
dataOwnerZone	Home Data Grid (zone) of the owner of the file
dataExpiry	Expiration date for the file
dataComments	Comments associated with a file
dataCreate	Creation date of a file
dataModify	Modification date of a file
dataAccess	Access date of a file
dataAccessInx	Data access identifer (internal)
dataId	Internal Identifier for a file
collId	Internal Identifier for a collection
rescGroupName	Resource group name
statusString	String for outputting status informtion
dataMapId	(not used)
userClient	Pointer to the user client structure
userNameClient	User name in the form 'name@domain'
rodsZoneClient	Name of Data Grid
userTypeClient	Role of a user (rodsgroup, rodsadmin, rodsuser, domainadmin, groupadmin, storageadmin, rodscurators)
sysUidClient	Internal identifer of the user

hostClient	IP address of host
authStrClient	Authorization string of proxy client (such as DN string)
userAuthSchemeClient	Authorization scheme such as GSI, password, etc.
userInfoClient	Tagged information: <EMAIL>user@unc.edu</EMAIL><PHONE>5555555555</PHONE>
userCommentClient	Comment on user
userCreateClient	Pointer to create client structure
userModifyClient	Pointer to modify client structure
userProxy	Pointer to the structure of the system user who acts as proxy for the client user
userNameProxy	Proxy user name in the form 'name@domain'
rodsZoneProxy	Data Grid zone name of the proxy user
userTypeProxy	Role of a user (rodsgroup, rodsadmin, rodsuser, domainadmin, groupadmin, storageadmin, rodscurators)
sysUidProxy	Internal identifer of the proxy user
hostProxy	IP address of host of the proxy user
authStrProxy	Authorization string of proxy user (such as DN string)
userAuthSchemeProxy	Type of authentication scheme such as GSI, password
userInfoProxy	Tagged information: <EMAIL>user@unc.edu</EMAIL><PHONE>5555555555</PHONE>
userCommentProxy	Comment about user
userCreateProxy	Pointer to create client structure (not used)
userModifyProxy	Pointer to modify client structure (not used)
collName	Name of collection
collParentName	Name of parent collection
collOwnername	Name of owner of a collection
collExpiry	Expiration date for a collection
collComments	Comment on a collection
collCreate	Creation date for a collection
collModify	Modification date for a collection
collAccess	Access date for a collection
collAccessInx	Internal identifer for access control
collMapId	Not used
collInheritance	Attributes inherited by objects and subcollections: ACL, metadata, pins, locks
zoneName	The name of the iCAT instance. This is globally unique.
rescLoc	IP address of storage resource
rescType	Type of storage resource: hpss, samfs, database, orb
rescTypeInx	Internal identifier for resource type
rescClass	Class of resource: primary, secondary, archival
rescClassInx	Internal identifer for resource class
rescVaultPath	Physical path name used at storage resource
numOpenPorts	Number of ports open on storage resource
paraOpr	Flag for whether parallel operation is supported
rescId	Resource ID
gateWayAddr	IP address of gateway
rescMaxObjSize	Maximum file size allowed on storage resource
freeSpace	Amount of free space on storage resource
freeSpaceTime	Unix time when last free space was computed and registered

freeSpaceTimeStamp	Time stamp information
rescInfo	Information about resource
rescComments	Comments about resource
rescCreate	Creation date for resource
rescModify	Modification date for resource
connectCnt	Poiter to Connection structure
connectSock	Socket number for connection
connectOption	Type of connection
connectStatus	Status of connection
connectApiTnx	API that is being used
connectWindowSize	Data transmission window size for connection
connectReconnFlag	Data transmission reconnection flag
connectReconnSock	Socket number for reconnection
connectReconnPort	Port number for reconnection
connectReconnAddr	IP address for reconnection
ConnectCookie	Shared secret for connection

4.2 iRODS Persistent State Information Attributes (# variables)

The #variables are based on a Logical Name Space as defined by the attribute set of the iCAT Metadata Catalog. The columns in the tables in the iCAT schema are defined in the source file lib/core/include/rodsGenQuery.h. These variables can be queried using the generic query call "iquest" which is available for accessing data from the iCAT database. The #variables defined within release 2 of iRODS are listed in Table 2.

Table 2. Persistent State Variables in iCAT

Persistent state #variable	**ID**	**Explanation**
COL_ZONE_ID	101	Data Grid or zone identifier
COL_ZONE_NAME	102	Data Grid or zone name, name of the iCAT
COL_ZONE_TYPE	103	Type of zone: local/remote/other
COL_ZONE_CONNECTION	104	Connection information in tagged list; <PASSWORD>RPS1</PASSWORD> <GSI>DISTNAME</GSI>
COL_ZONE_COMMENT	105	Comment about the zone
COL_USER_ID	201	User internal identifier
COL_USER_NAME	202	User name
COL_USER_TYPE	203	User role (rodsgroup, rodsadmin, rodsuser, domainadmin, groupadmin, storageadmin, rodscurators)
COL_USER_ZONE	204	Home Data Grid or user
COL_USER_DN	205	Distinguished name in tagged list: <authType>distinguishedName</authType>
COL_USER_INFO	206	Tagged information: <EMAIL>user@unc.edu</EMAIL> <PHONE>5555555555</PHONE>
COL_USER_COMMENT	207	Comment about the user
COL_USER_CREATE_TIME	208	Creation timestamp

COL_USER_MODIFY_TIME	209	Last modification timestamp
COL_R_RESC_ID	301	Internal resource identifier
COL_R_RESC_NAME	302	Logical name of the resource
COL_R_ZONE_NAME	303	Name of the iCAT, unique globally
COL_R_TYPE_NAME	304	Resource type: hpss, samfs, database, orb
COL_R_CLASS_NAME	305	Resource class: primary, secondary, archival
COL_R_LOC	306	Resource IP address
COL_R_VAULT_PATH	307	Resource path for storing files
COL_R_FREE_SPACE	308	Free space available on resource
COL_R_RESC_INFO	309	Tagged information list: <MAX_OBJ_SIZE>2GBB</MAX_OBJ_SIZE> <MIN_LATENCY>1msec</MIIN_LATENCY>
COL_R_RESC_COMMENT	310	Comment about resource
COL_R_CREATE_TIME	311	Creation timestamp of resource
COL_R_MODIFY_TIME	312	Last modification timestamp for resource
COL_D_DATA_ID	401	Data internal identifier. A digital object is identified by (zone, collection, data name, replica, version)
COL_D_COLL_ID	402	Collection internal identifer
COL_DATA_NAME	403	Logical name of the digital object
COL_DATA_REPL_NUM	404	Replica number starting with “1”
COL_DATA_VERSION	405	Version string assigned to the digital object. Older versions of replicas have a negative replica number
COL_DATA_TYPE_NAME	406	Type of data: jpeg image, PDF document
COL_DATA_SIZE	407	Size of the digital object in bytes
COL_D_RESC_GROUP_NAME	408	Name of resource group in which data is stored
COL_D_RESC_NAME	409	Logical name of storage resource
COL_D_DATA_PATH	410	Physical path name for digital object in resource
COL_D_OWNER_NAME	411	User who created the object
COL_D_OWNER_ZONE	412	Home zone of the user who created the object
COL_D_REPL_STATUS	413	Replica status: locked, is-deleted, pinned, hide
COL_D_DATA_STATUS	414	Digital object status: locked, is-deleted, pinned, hide
COL_D_DATA_CHECKSUM	415	Checksum stored as tagged list: <BINHEX>12344</BINHEX> <MD5>22234422</MD5>
COL_D_EXPIRY	416	Expiration date for the digital object
COL_D_MAP_ID	417	
COL_D_COMMENTS	418	Comments about the digital object
COL_D_CREATE_TIME	419	Creation timestamp for the digital object
COL_D_MODIFY_TIME	420	Last modification timestamp for the digital object
COL_DATA_MODE	421	
COL_COLL_ID	500	Collection internal identifier
COL_COLL_NAME	501	Logical collection name
COL_COLL_PARENT_NAME	502	Parent collection name
COL_COLL_OWNER_NAME	503	Collection owner
COL_COLL_OWNER_ZONE	504	Home zone of the collection owner
COL_COLL_MAP_ID	505	

COL_COLL_INHERITANCE	506	Attributes inherited by subcollections: ACL, metadata, pins, locks
COL_COLL_COMMENTS	507	Comments about the collection
COL_COLL_CREATE_TIME	508	Collection creation timestamp
COL_COLL_MODIFY_TIME	509	Last modification timestamp for collection
COL_COLL_TYPE	510	
COL_COLL_INFO1	511	Information about collection
COL_COLL_INFO2	512	Information about collection
COL_META_DATA_ATTR_NAME	600	Metadata attribute name for digital object
COL_META_DATA_ATTR_VALUE	601	Metadata attribute value for digital object
COL_META_DATA_ATTR_UNITS	602	Metadata attribute units for digital object
COL_META_DATA_ATTR_ID	603	Internal identifier for metadata attribute for digital object
COL_META_COLL_ATTR_NAME	610	Metadata attribute name for collection
COL_META_COLL_ATTR_VALUE	611	Metadata attribute value for collection
COL_META_COLL_ATTR_UNITS	612	Metadata attribute units for collection
COL_META_COLL_ATTR_ID	613	Internal identifer for metadata attribute for collection
COL_META_NAMESPACE_COLL	620	Namespace of collection AVU-triplet attribute
COL_META_NAMESPACE_DATA	621	Namespace of digital object AVU-triplet attribute
COL_META_NAMESPACE_RESC	622	Namespace of resource AVU-triplet attribute
COL_META_NAMESPACE_USER	623	Namespace of user AVU-triplet attribute
COL_META_RESC_ATTR_NAME	630	Metadata attribute name for resource
COL_META_RESC_ATTR_VALUE	631	Metadata attribute value for resource
COL_META_RESC_ATTR_UNITS	632	Metadata attribute units for resource
COL_META_RESC_ATTR_ID	633	Internal identifer for metadata attribute for resource
COL_META_USER_ATTR_NAME	640	Metadata attribute name for user
COL_META_USER_ATTR_VALUE	641	Metadata attribute value for user
COL_META_USER_ATTR_UNITS	642	Metadata attribute units for user
COL_META_USER_ATTR_ID	643	Internal identifier for metadata attribute for user
COL_DATA_ACCESS_TYPE	700	Access allowed for the digital object; r, w, x
COL_DATA_ACCESS_NAME	701	
COL_DATA_TOKEN_NAMESPACE	702	Namespace of the data token: e.g. data type
COL_DATA_ACCESS_USER_ID	703	User or group for which the access is defined on digital object
COL_DATA_ACCESS_DATA_ID	704	Internal identifer of the digital object for which access is defined
COL_COLL_ACCESS_TYPE	710	Access allowed for the collection; r, w, x
COL_COLL_ACCESS_NAME	711	
COL_COLL_TOKEN_NAMESPACE	712	Namespace of the collection token: e.g. collection type
COL_COLL_ACCESS_USER_ID	713	User or group for which the access is defined on the collection
COL_COLL_ACCESS_COLL_ID	714	Internal identifer of the collection for which access is defined
COL_RESC_GROUP_RESC_ID	800	Internal identifier for the resource group
COL_RESC_GROUP_NAME	801	Logical name of the resource group
COL_USER_GROUP_ID	900	Internal identifer for the user group

COL_USER_GROUP_NAME	901	Logical name for the user group
COL_RULE_EXEC_ID	1000	Internal identifer for a delayed Rule execution request
COL_RULE_EXEC_NAME	1001	Logical name for a delayed Rule execution request
COL_RULE_EXEC_REI_FILE_PATH	1002	Path of the file where the context (REI) of the delayed rule is stored
COL_RULE_EXEC_USER_NAME	1003	User requesting a delayed Rule execution
COL_RULE_EXEC_ADDRESS	1004	Host name where the delayed Rule will be executed
COL_RULE_EXEC_TIME	1005	Time when the delayed rule will be executed
COL_RULE_EXEC_FREQUENCY	1006	Delayed Rule execution frequency
COL_RULE_EXEC_PRIORITY	1007	Delayed Rule execution priority
COL_RULE_EXEC_ESTIMATED_EXE_TIME	1008	Estimated execution time for the delayed Rule
COL_RULE_EXEC_NOTIFICATION_ADDR	1009	Notification address for delayed Rule completion
COL_RULE_EXEC_LAST_EXE_TIME	1010	Previous execution time for the delayed Rule
COL_RULE_EXEC_STATUS	1011	Current status of the delayed Rule
COL_TOKEN_NAMESPACE	1100	Namespce for tokens; e.g. data type
COL_TOKEN_ID	1101	Internal identifier for token name
COL_TOKEN_NAME	1102	A value in the token namespace; e.g. "gif image"
COL_TOKEN_VALUE	1103	Additional token information string
COL_TOKEN_VALUE2	1104	Additional token information string
COL_TOKEN_VALUE3	1105	Additional token information string
COL_TOKEN_COMMENT	1106	Comment on token
COL_AUDIT_OBJ_ID	1200	Identifer that starts the range of audit types
COL_AUDIT_USER_ID	1201	User identifier of person requesting operation
COL_AUDIT_ACTION_ID	1202	Internal identifer for type of action that is audited
COL_AUDIT_COMMENT	1203	Comment on audit trail
COL_AUDIT_CREATE_TIME	1204	Creation timestamp for audit trail
COL_AUDIT_MODIFY_TIME	1205	Modification timestamp for audit trail
COL_AUDIT_RANGE_START	1200	Identifer that starts the range of audit tpes
COL_AUDIT_RANGE_END	1299	Identifer that ends the range of audit types
COL_COLL_USER_NAME	1300	Internal identifier for user creating collection
COL_COLL_USER_ZONE	1301	Zone in which collection is created

4.3 User Environment Variables

Information that defines the preferred user environment is maintained in enviroment variables that are stored on the user's computer. The environment variables specify the default data grid that will be accessed, and properties about the user's default collection.

Four environment variables are needed by the icommands to function:

1. irodsUserName - User name in the iRODS data grid
2. irodsHost - Network address of a data grid server
3. irodsPort - Port number for the data grid metadata catalog (iCAT)
4. irodsZone - Unique identifier for the data grid zone

Each iRODS Data Grid requires a metadata catalog (iCAT) that is managed as an instance within a database. Since databases can manage multiple instances, we assign a unique port number to each instance. The iRODS Data Grid is therefore specified completely by:

irodsZone : irodsHost : irodsPort

The complete set of icommands environment variables are:

irodsUserName	- Your irods user name
irodsHost	- An iRODS Server host to connect to
irodsPort	- The network port (TCP) the server is listening on
irodsHome	- Your iRODS Home collection
irodsCwd	- Your iRODS current working directory
irodsAuthScheme	- Set to GSI for GSI authentication, default is password.
irodsDefResource	- A default resource to use when storing (rules may also apply)
irodsZone	- Your iRODS Zone (the name of your home iRODS Data Grid)
irodsServerDn	- (for future use with GSI)
irodsLogLevel	- The level of messages to log or display
irodsAuthFileName	- The file storing your scrambled password (for authentication)
irodsEnvFile	- Name of the file storing the rest of the environment variables

If you do not provide the irodsHome, it will be set based on the irodsZone and irodsUserName. If you do not specify the irodsCwd, it will be assumed to be irodsHome.

Each of these can be set via a unix environment variable or as a line in a text file, your iRODS environment file. The environment variables, if set, override the lines in the environment file. By default, your iRODS environment file is ~/.irods/.irodsEnv. The install script creates a ~/.irods/.irodsEnv file for the admin account, for example:

```
irodsHost 'zuri.sdsc.edu'
irodsPort 1378
irodsDefResource=demoResc
irodsHome=/tempZone/home/rods
irodsCwd=/tempZone/home/rods
irodsUserName 'rods'
irodsZone 'tempZone'
```

You can change the ~/.irods/.irodsEnv to some other file by setting the irodsEnvFile environment variable. When you do this, a child process will share that environment (and cwd - the current working directory (collection)) which is useful for scripts. By default, without irodsEnvFile set, the cwd will be read by children processes but not by 'grand-children' and beyond; this is so that separate sessions are possible at the same time on the same computer.

5 The iRODS Rule System

iRODS Rules can generally be classified into two Rule Classes. These are:

1. **System Level Rules:** These are Rules that are invoked on the iRODS Servers internally to enforce/execute Management Policies for the system. Examples of policies include data management policies such as enforcement of authenticity, integrity, access restrictions, data placement, data presentation, replication, distribution, pre- and post-processing, and metadata extraction and assignment. Another example is the automation

of services such as administration, authentication, authorization, auditing, and accounting.

2. **User Level Rules**: The iRODS Rule Engine can also be invoked externally by clients through the irule command or the rcExecMyRule API. Typically, these are work-flow type Rules which allow users to request that the iRODS Servers perform a sequence of operations (Micro-services) on behalf of the user. In addition to providing useful services to users, this type of operation can be very efficient because the operations are done on the servers where the data are located.

Some Rules require immediate execution while others may be executed at a later time in the background (depending upon the Rule Execution mode). The *Delayed Execution Service* allows Rules/Micro-services to be queued and executed at a later time by the Rule Execution Server. Examples of Micro-services that are suitable for delayed execution are post-processing operations such as checksuming, replication and metadata extraction. For example, the post-processing Micro-service msiExtractNaraMetadata was designed specifically to extract and register metadata from NARA Archival Information Locator data objects (NAIL files) that are uploaded into a NARA collection.

5.1 The iRODS Rule Architecture

At the core of the iRODS Rule System is the iRODS Rule Engine which runs on all iRODS Servers. The Rule Engine can invoke a number of predefined Micro-services based on the interpretation of the Rule being executed.

The underlying operations that need to be performed are based on C functions that operate on internal C structures. The external view of the execution architecture is based on Actions (typically called tasks) that need to be performed, and external input parameters (called Attributes) that are used to guide and perform these Actions. The C functions themselves are abstracted externally by giving them logical names (we call the functions "internal Micro-services" and the abstractions "external Micro-services"). To make the links between the external world and the internal C apparatus transparent, we define mappings from client libraries to Rules. Moreover, since the operations that are performed by iRODS need to change the Persistent State Information in the ICAT Metadata Catalog, the attributes are mapped to a persistent Logical Name Space for metadata names that are used in the ICAT.

The foundation for the iRODS architecture is based on the following key concepts, partially discussed in section 2.1 on ROP:

1. A **Persistent Database** [#] that shares data (facts) across time and users. Information that is extracted from the persistent database is labeled in the following with the symbol “#”.
2. A Transient Memory [$] that holds data during a session. **Session Information** that resides in the transient memory is labeled in the following with the symbol “$”.
3. A set of **Actions** [T] that name and define the tasks that need to be performed.
4. A set of internal well-defined callable **Micro-services** [P] made up of procedures and functions that provide the methods for executing the sub-tasks that need to be performed,
5. A set of **external Attributes** [A] that is used as a Logical Name Space to externally refer to data and metadata.
6. A set of **external Micro-services** [M] (or methods) that is used as a **Logical Name Space** to externally refer to methods in the Rules.

7. A set of **data variable mappings** [DVM] that define a relationship from external Attributes in A to internal elements in # and $.
8. A set of **Micro-service name mappings** [FNM] that define a relationship from external Micro-services in M and Actions T to procedures and functions in P and other Action names in T. In a sense FNM can be seen as providing aliases for Micro-services. One use will be to map different versions of the functions/procedures in P at run time to the actual execution process.
9. A set of **Rules** [R] which defines what needs to be done for each Action [T] and is based on A and M.

5.2 Rules

A Rule consists of a name, condition, workflow-chain, and recovery-chain. A Rule is specified with a line of text that contains these four parts separated by the '|' separator:

actionDef | condition | workflow-chain |recovery-chain

'actionDef' is the name of the Rule. It is an identifier that can be used by other Rules or external functions to invoke the Rule.

'condition' is the condition under which this Rule may be executed. I.e., this Rule will apply only if the condition is satisfied. Typically, one or more of the session Attributes are used to compose a condition. For example:

$rescName == demoResc8

is a condition on the storage resource name. If the storage resource is "demoResc8", then the Rule that uses this condition will be applied. Another example:

$objPath like /x/y/z/*

is a condition on the data object or collection path name. The Rule will only be fired if the file in the iRODS Data Grid is underneath the collection x/y/z.

'workflow-chain' is the sequence of Micro-services/Rules to be executed by this Rule. The Micro-services/Rules in the sequence are separated by the '##' separator. Each may contain a number of input/output parameters. Note that a Rule can invoke another Rule, or even itself (recursion). If a Rule invokes itself, the designer of the Rule must ensure that the recursion will terminate.

'recovery-chain' is the set of Micro-services/Rules to be called when execution of any one of the Micro-services/Rules in the workflow-chain fails. The number of Micro-services/Rules in the recovery-chain should be equal to the number in the workflow-chain. If no recovery Action is needed for a given Micro-service/Rule, a 'nop' action should be specified. When a Micro-service fails, all of the Micro-services in the 'recovery-chain' are executed. Thus all components within a Rule need to succeed.

5.3 Rule Grammar

The detailed syntactic structure for Rule specification is given below (alternate definitions are given below each other).

Table 3. Rule Grammar

Rule	::= actionDef | condition | workflow-chain |recovery-chain	
actionDef	::= actionName	
	::= actionName(param1, ..., paramn)	
action	::= actionName	
	::= actionName(arg1, ..., argn)	
actionName	::= alpha-numeric string	
Micro-service	::= msName	
	::= msName(arg1, ..., argn)	
msName	::= alpha-numeric string	/* pre-defined and compiled */
condition	::=	/* can be empty */
	log-expr	
	(log-expr)	/* parentheses to impose order */
	condition && condition	/* and condition */
	condition !! condition	/* or condition */
log-expr	::= 1	/* true */
	::= 0	/* false */
	expr == expr	
	expr > expr	
	expr < expr	
	expr >= expr	
	expr <= expr	
	expr != expr	/* not equal */
	expr like reg-expr	
	expr not like reg-expr	
expr	::= string	
	number	
	$-variable	/* session variable */
	*-variable	/* state variable */
	concatenation of string, $-variables and/or *-variables	
reg-expr	::= regular-expression-string	
argx	::= expr	
paramx	::= *-variable	
	number	
	string	
workflow-chain	::= Micro-service	
	action	
	workflow-chain ## workflow-chain	
recovery-chain	::= workflow-chain	

The above syntax defines how Rules can be composed from conditions, state information, session information, and Micro-services. Some sample Rules are:

acCreateUser||msiCreateUser##acCreateDefaultCollections##msiCommit|msiRollback##msiRollback##nop

This Rule invokes the following chained Micro-service and Rule:

- msiCreateUser /* execute a Micro-service to create a new user */
- acCreateDefaultCollections /* execute another Rule to create default collections */
- msiCommit /* register the new state information into the iCAT Metadata Catalog */

The corresponding recovery Micro-services are:

- msiRollback /* delete user information from the iCAT Metdata Catalog */
- msiRollback /* delete collection information from the iCAT Metadata Catalog */
- nop /* no operation required */

A second sample Rule is:

acSetRescSchemeForCreate||msiSetDefaultResc(demoResc,noForce)##msiSetRescSortScheme(random)##msiSetRescSortScheme(byRescType)|nop##nop##nop

This Rule invokes the following chained Micro-services:

- msiSetDefaultResc /* set the default resource if it is available */
- msiSetRescSortScheme /* set random selection of vaults within a storage resource group */
- msiSetRescSortScheme /* set select selection of vaults based upon storage type */

The Rule sets three different mechanisms for selecting the location where a file will be created. The corresponding recovery Micro-services are all nop's, meaning no recovery operation is required.

A third example is:

acRegisterData|$objPath like /home/collections.nvo/2mass/*|acGetResource##msiRegisterData##msiAddACLForDataToUser(2massusers.nvo,write)|nop##recover_msiRegisterData##recover_msiAddACLForDataToUser(2massusers.nvo,write)

This Rule specifies a condition "$objPath like /home/collections.nov/2mass/*". If the iRODS collection under which the file will be registered includes the path name "/home/collections.nov/2mass/*", then the Rule will be executed. The Rule invokes the following Rule and chained Micro-services:

- acGetResource /* find an acceptable storage location based on the resource selection scheme */
- msiRegisterData /* register a file into the iRODS Data Grid */
- msiAddACLForDataToUser /* give write permission on the file to the user group "2massusers" in the "nvo" project. */

The recovery operations are:

- recover_msiRegisterData /* this Micro-service will delete registration of the file */
- recover_msiAddACLForDataToUser /* this Micro-service will delete the ACL for the file */

5.4 RuleGen Language

To make it easier to construct rules, a rulegen parser has been developed which translates from a nicer rule language to the grammar spedified in Section 5.3. The naming convention for the input files to rulegen is that they should have a ".r" extension. The output files created by rulegen should have a ".ir" extension. The grammar for the langauge of the input files is given in section 5.4.1

5.4.1 Using the RuleGen Parser

The rulegen parser is creatd by executing

```
make
```

in the ~/irods/clients/icommands/rulegen directory. A binary file 'rulegen' is created in the ~irods/clients/icommands/bin directory.

The rulegen parser will convert a rulegen input file (".r" extension) into a rule file (".ir" extension). This is done by running the rulegen as shown below:

```
 ~/irods/clients/icommands/bin/rulegen -s test1.r > test1.ir
```

The output file, test1.ir, can be used as an input file for the irule command.

5.4.2 Grammar of the rulegen language

The reserved words used by the rulegen program are listed below:

```
program
      : program rule_list inputs outputs
      ;
inputs
      : INPUT inp_expr_list
      ;
outputs
      : OUTPUT out_expr_list
      ;
rule_list
      : rule
      | rule rule_list
      ;
rule
      : action_def '{' first_statement '}'
      | action_def '{' first_statement statement_list '}'
      ;
action_def
      : action_name
      | action_name '(' arg_list ')'
      ;
microserve
      : action_name
      | action_name '(' arg_list ')'
      ;
action_name
      : identifier
```

```
        ;
arg_list
        : arg_val
        | arg_val ',' arg_list
        ;
arg_val
        : STR_LIT
        | Q_STR_LIT
        | NUM_LIT
        ;
first_statement
        :  selection_statement
        |
        ;
compound_statement
        : '{' '}'
        | '{' statement_list '}'
        ;
statement_list
        : statement
        | statement_list statement
        ;
statement
        : selection_statement
        | iteration_statement
        | compound_statement
        | action_statement ';'
        | ass_expr ';'
        | execution_statement
        ;
selection_statement
        : ON '(' cond_expr ')' statement
        | ON '(' cond_expr ')' statement or_list_statement_list
        ;
iteration_statement
        : WHILE '(' cond_expr ')' statement
        | FOR '(' ass_expr_list ';' cond_expr ';'  ass_expr_list ')' statement
        | IF '(' cond_expr ')' THEN statement
        | IF '(' cond_expr ')' THEN statement ELSE statement
        | 'break'
        ;
or_list_statement_list
        : ORON '(' cond_expr ')' statement
        | OR  statement
        | ORON '(' cond_expr ')' statement or_list_statement_list
        | OR statement or_list_statement_list
        ;
action_statement
        : microserve ACRAC_SEP microserve
        | microserve
        ;
```

```
execution_statement
        : DELAY '(' cond_expr ')' statement
        | REMOTE '(' identifier ',' cond_expr ')' statement
        | PARALLEL '(' cond_expr ')' statement
        | ONEOF    statement
        | SOMEOF  '(' identifier ')'   statement
        | FOREACH  '(' identifier ')'   statement
        ;
```

5.4.3 Expressions used by Rulegen Parser:

```
inp_expr
        : identifier  '=' cond_expr
        ;
inp_expr_list
        : inp_expr
        | inp_expr ',' inp_expr_list
        ;
out_expr
        : arg_val
out_expr_list
        : out_expr
        | out_expr ',' out_expr_list
        ;
ass_expr
        : identifier  '=' cond_expr
ass_expr_list
        : ass_expr
        | ass_expr ',' ass_expr_list
        ;
cond_expr
        : logical_expr
        | '(' logical_expr ')'
        | cond_expr AND_OP cond_expr
        | cond_expr OR_OP cond_expr
        | cond_expr '+'  cond_expr
        | cond_expr '-'  cond_expr
        ;
logical_expr
        : TRUE
        | FALSE
        | relational_expr
        | logical_expr EQ_OP logical_expr
        | logical_expr NE_OP logical_expr
        | logical_expr '<' logical_expr
        | logical_expr '>' logical_expr
        | logical_expr LE_OP logical_expr
        | logical_expr GE_OP logical_expr
        | logical_expr LIKE logical_expr
        | logical_expr NOT LIKE logical_expr
        ;
```

```
relational_expr
      :  STR_LIT
      | NUM_LIT
      | Q_STR_LIT
      ;
identifier
      :  STR_LIT
      | Q_STR_LIT
      | NUM_LIT
      ;

STR_LIT    : string of characters
Q_STR_LIT  : quoted (") string of characters
NUM_LIT    : number-string
AND_OP     : "&&"
OR_OP      :"||"
LE_O     :"<="
GE_OP      :">="
EQ_OP      :"=="
NE_OP      :"!="
ACRAC_SE     :":::"
```

5.4.4 Example Rule Build

Using the rulegen language, a rule can be defined in a syntax similar to the C language. The body of the rule is specified in the brackets after the rule name “myTestRule”. The input parameters are listed as a comma separated list. The output parameters are listed as a second comma separated list. An example from the file ~/irods/clients/icommands/rulegen/test2.r is listed below:

```
myTestRule
{
      acGetIcatResults(*Action,*Condition,*B);
      foreach ( *B )
      {
            remote ("andal.sdsc.edu" , "null")
            {
                  msiDataObjChksum(*B,*Operation,*C);
            }
            msiGetValByKey(*B,DATA_NAME,*D);
            msiGetValByKey(*B,COLL_NAME,*E);
            writeLine(stdout,"CheckSum of *E/*D is *C");
      }
}

INPUT *Action=chksum,*Condition="COLL_NAME =
'/tempZone/home/rods/loopTest'",*Operation=ChksumAll
OUTPUT *Action,*Condition,*Operation,*C,ruleExecOut
```

This rule queries the iCAT metadata catalog, loops over the result set which is stored in variable “B”, for each item in the result set it calculates a checksum on a remote resource, gets the file name and the collection name from the iCAT catalog, and writes an output line. After running

the rulegen program, an iRODS rule file is generated (listed in
~/irods/clients/icommands/rulegen/test2.ir

```
myTestRule||acGetIcatResults(*Action,*Condition,*B)##forEachExec(*B,remoteExec(a
ndal.sdsc.edu,null,msiDataObjChksum(*B,*Operation,*C),nop)##msiGetValByKey(*B,
DATA_NAME,*D)##
msiGetValByKey(*B,COLL_NAME,*E)##writeLine(stdout,CheckSum of *E/*D is
*C),nop##nop##nop##nop)|nop##nop##nop
*Action=chksum%*Condition=COLL_NAME =
'/tempZone/home/rods/loopTest'%*Operation=ChksumAll
*Action%*Condition%*Operation%*C%ruleExecOut
```

The rule is written on a single very-long line. The input parameters are listed at the end of the rule (no blank lines), followed by the output parameters. This rule can be executed by typing

```
Irule –vF test2.ir
```

5.5 iRODS Rule Engine

The Rule Engine is the interpreter of Rules in the iRODS system. The Rule Engine can be invoked by any server-side procedure call using the applyRule API.

```
int applyRule(char *inAction, msParamArray_t *inMsParamArray,
      RuleExecInfo_t *rei, int reiSaveFlag)
```

The Rule Engine reads the Rule Base to decide which Rules will apply. First, the Rule Engine selects all the Rules whose Action names are the same as given by the inAction string. These Rules are prioritized based on how they were read into the Rule Base of the Rule Engine when the system was started. The first Rule in the list is checked for validation of its condition. If the condition fails, then the next Rule is tried. If no more Rules are available then the Action fails and a failure status (negative number) is returned to the calling routine. The Micro-services can use the REI strucure to pass other failure status and messages. If the condition succeeds, then the Micro-services and Action chain in the Rule are executed one after the other in a left-to-right order. If all of the Micro-services succeed, then the Action is considered a success and a success status (0) is sent to the calling routine. After successful completion, the inMsParamArray will hold any output values returned by the Rule execution and the structure REI will reflect any modifications that were made by the Rule execution.

While executing the chain of Micro-services/Actions, if one of them fails, then the Rule Engine starts a recovery procedure. It applies the corresponding recovery Micro-service or Action defined in the Rule. The recovery for the failed Micro-service/Action is first performed, followed by the recovery of all the previously successful Micro-services/Actions in reverse order. By the time the recovery is completed, the status of the black-board($) and the persistent database (#) and any side-effects should have been rolled back by these recovery procedures. If the calling procedure wants the Rule Engine to recover changes in the REI structure, it can do so by setting the reiSaveFlag. In this case, the Rule Engine will save the REI structure before invoking the first Micro-service/Action in a Rule and will recover back by resetting the REI structure in case of any Rule failure before invoking any alternate Rules.

Most of the values in the REI structure have a logical name defined for them. We call these names “$variable” names. The mappings from the REI structure to the $variable names can be found in the file “server/config/reConfigs/core.dvm”.

For example, the string $objPath provides the value in the structure rei->doi->objPath. The "assign" Micro-service can change the value of the $variable:

```
                    assign($objPath, $objPath.Ver0)
```

will add ".Ver0" to the dataName.

5.6 Default iRODS Rules

The **core.irb file** contains the Rules that are applied by default when an iRODS Data Grid is created. These Rules are typically modified by the Data Grid Administrator to impose the data Management Policies for the shared collection. For example, the modifications can be specific to a data collection, or to a data type, or to a storage resource, or to a user group.

These Rules can be thought of as policy hooks into the operation of the iRODS Data Grid that enable different policies to be enforced at the discretion of the Data Grid Administrator. Multiple versions of each Rule can be placed in the core.irb file. The Rule listed closest to the top of the core.irb file will be executed first. If the Rule does not meet the required condition or fails, the next version of the Rule will be tried. A generic version of the Rule should be included that will apply if all of the higher priority Rules fail. Note that most of the Rules in the default core.irb file are place holders that do not execute any Micro-services. The core.irb file (located in the release directory server/config/reConfigs) contains multiple examples of each Rule that have been commented out by inserting a “#” symbol at the beginning of the line.

Table 2. Default iRODS Rules in the core.irb File

Default Rules in core.irb	**Input Parameters**	**Meaning**
acCreateUser		Create a new user
acVacuum	*arg1	Optimize the Postgresql database after waiting “arg1” specified time. See delayExec Micro-service
acCreateDefaultCollections		Create default collections (home,trash)
acCreateUserZoneCollections		Create collections in Data Grid zone
acCreateCollByAdmin	*parColl, *childColl	Create a new collection with name “childColl” under the parent collection “parColl”
acDeleteUser		Delete user
acDeleteDefaultCollections		Delete home collection
acDeleteUserZoneCollections		Delete collections in a Data Grid zone
acDeleteCollByAdmin	*parColl, *childColl	Delete the child collection “childColl” under the parent collection “parColl”
acRenameLocalZone	*oldZone, *newZone	Rename the Data Grid zone from the name “oldZone” to the name “newZone”
acSetRescSchemeForCreate		Define selection scheme for default resource
acPreprocForDataObjOpen		Select which copy of file to open
acSetMultiReplPerResc		Specify number of copies per resource

acPostProcForPut		Apply processing to file on put
acPostProcForCopy		Apply processing to file on copy
acSetNumThreads		Set the default number of threads for data transfers
acDataDeletePolicy		Set policy for data deletion
acNoChkFilePathPerm		Set policy for checking permissions on registering a file
acTrashPolicy		Set policy for using trash can
acSetPublicUserPolicy		Set policy for allowed operations by public
acChkHostAccessControl		Set policy for host access control
acSetVaultPathPolicy#		Set policy for assigning physical path name
acDataObjCreate		Test Rule for object creation
acSetCreateConditions		Test Rule for object descriptor
acDOC		Test Rule for registering object
acSetResourceList		Test Rule for setting resources
acSetCopyNumber		Test Rule for setting copy number
acRegisterData		Test Rule for registering data
acGetIcatResults	*Action, *Condition, *GenQOut	Apply the “action to the list of files that meet the specified condition
acPurgeFiles	*Condition	Purge files satisfying condition on expiration time

6 iRODS Attributes

The system state information that is generated by application of Micro-services is stored in the iCAT Metadata Catalog. The iCAT catalog can be queried. The source file
lib/core/include/rodsGenQuery.h
defines the columns available via the General Query interface. Each of the names for a column (metadata attribute or item of state information) begins with 'COL_' (column) for easy identification throughout the source code. The “iquest” client program also uses these field names but without the COL_ prefix.

6.1 First-Class Objects in iRODS

Table 3 defines the list of **first class objects** (FCOs) about which information is stored in the iRODS iCAT Metadata Catalog . These first class objects have unique identifiers and associated metadata information, also listed in Table 3. This file defines derived objects based on the first class objects and ontologies maintained by the ICAT system. We attempt to keep the list of metadata attributes to the minimum.

The first class objects are organized into the following groups:

1. Zone (Data Grid) parameters
2. User parameters
3. Resource parameters
4. Collection parameters
5. Digital object (data) parameters
6. Attribute (metadata) parameters

7. Rule parameters
8. Token parameters

Derived properties are also organized into classes:

1. Collection hierarchy parameters
2. Data collection mapping parameters
3. User group mapping parameters
4. Resource group mapping parameters
5. Metadata mapping parameters
6. Accesss control list parameters
7. Deny access control list parameters
8. Audit information parameters
9. User session information parameters

Table 4. First Class Objects in iRODS

First Class Objects:	
Note: All first class objects have internal identifiers (id numbers).	These identifiers are all generated from the same sequence numbering scheme.
FC01 : Zone	
R_ZONE_ID	Internal identifier.
R_ZONE_NAME	The name of the ICAT. Unique globally.
R_ZONE_TYPE	Type of zone - local|remote|other.
R_ZONE_CONN_STRING	Connection information to (remote) zone.
	This a tagged list with elements of the form: <PASSWORD>RPS1<PASSWORD><GSI>DISTNAME</GSI> <PASSWORD>system</PASSWORD>...
R_ZONE_COMMENT	Information about the zone.
R_ZONE_CRT_TIME	Creation timestamp.
R_ZONE_MOD_TIME	Last modification timestamp.
FC02 : User	
R_USER_ID	Internal identifier.
R_USER_NAME	A string of the form 'name@domain'.
R_USER_ZONE	Native zone of user (from R_ZONE_NAME). An ICAT can register users from foreign zones.
R_USER_TYPE	User-type: rodsadmin, normal, group, public, ...
R_USER_PASS	Password which is probably scrambled.
R_USER_DISTIN_NAME	Any external name of the user. This is a tagged list with elements of the form: <authtype>distinguishedname</authtype>.
R_USER_INFO	Information. This a tagged list with elements of the form: <email>user@sdsc.edu</email> <phone>5555555555</phone>..
R_USER_CRT_TIME	Creation timestamp.
R_USER_MOD_TIME	Last modification timestamp.
FC03 : Resource	
R_RSRC_ID	Internal identifier.
R_RSRC_NAME	The name of the resource.
R_RSRC_ZONE	Native zone of resource (from R_ZONE_NAME). An ICAT can register resources from foreign zones.
R_RSRC_TYPE	Resource type: hpss, samfs, database, orb,...
R_RSRC_CLASS	Resource class: primary, secondary, archival, ...
R_RSRC_NET	Internet address of the resource host.
R_RSRC_DEF_PATH	Default path used by the resouce.
R_FREESPACE	Free space available on resource.
R_RSRC_INFO	Information. This is a tagged list with elements of the form: <max_obj_size>2gb</max_obj_size> <min_latency>1msec</min_latency>...
R_RSRC_CRT_TIME	Creation timestamp.

R_RSRC_MOD_TIME	Last modification timestamp.
FC04 : Collection	
R_COLL_ID	Internal identifier.
R_COLL_NAME	Data objects are clustered into collections. Each collection name is unique in the icat. It also has the format /r_zone_name/....
R_INHERITANCE	Information about what is inherited by the objects and sub-collections: ACL, metadata, pins, locks,
R_COLL_OWNER	Collection owner.
R_COLL_COMMENTS	Comments about the collection.
R_COLL_CRT_TIME	Creation timestamp.
R_COLL_MOD_TIME	Last modification timestamp.
FCO5 : Data - Accessible Digital Objects stored in iRODS	
R_DATA_ID	Internal identifier.
R_DATA_NAME	Logical name of the digital object (see also DCO2) .
R_REPLICA	A replica number given to the physical instance of the digital object. This is an integer. Replica numbers start at 1 and increase over time. For a given digital object, the replica number is not reused.
R_VERSION	A version string given to the physical instance of the digital object. A replica can have more than one version. Older versions of a replica will have a negative r_replica number. For access, when no version is specified, a positive r_replica is selected.
R_PDATA_ID	Internal identifier. The combination: (r_zone_name, r_coll_name, r_data_name, r_replica, r_version) uniquely identifies one physical copy of the digital object. We refer to this combination as pdata.
R_PDATA_TYPE	Data-type: jpeg image, PDF doc,....
R_PDATA_SIZE	Size of the digital object in bytes.
R_PDATA_RSRC	Resource-name where the copy of the digital object is stored.
R_PDATA_PATH	Access path for the object in the resource.
R_PDATA_OWNER	User who created the object.
R_PDATA_IS_DIRTY	Dirty status of the object - used for synch.
R_PDATA_STATUS	Status: locked, is-deleted,pinned,hide,...
R_PDATA_CHECKSUM	Checksum. This is a tagged list with elements of the form: <binhex>12344</binhex><md5>22234422</md5>...
R_PDATA_EXPIRY	Sunset date for the object.
R_PDATA_COMMENTS	Comments about the physical object.
R_PDATA_CRT_TIME	Creation timestamp.
R_PDATA_MOD_TIME	Last modification timestamp.
FC06 : Attribute-Value Metadata	
R_META_ID	Internal identifier.
R_META_NAMESPACE	Namespace in which attribute is located.
R_META_ATTR	Attribute name of the metadata.

R_META_VALUE	Attribute value of the metadata.
R_META_UNIT	Unit of the value: cms, mph, deg. Cel.,...
R_META_OWNER	User who created the metadata.
R_META_COMMENTS	Any comments about the metadata.
R_META_CRT_TIME	Creation timestamp.
R_META_MOD_TIME	Last modification timestamp.
FC07: Rules	
R_RULE_ID	Internal identifier.
R_RULE_NAME	Name of the Rule - normally the head predicate.
R_RULE_SEQ	Sequence number of Rule (an enumeration that is used to set priority of execution) .
R_RULE_OBJ_TYP	First class object type to which the Rule applies: collection, data, zone,...
R_RULE_EVENT	Event on which to trigger the Rule.
R_RULE_CONDITION	Conditions to be checked to trigger the Rule. This check should not have any side effects.
R_RULE_BODY	Body of the Rule. Execution may have side effects and hence if the Rule fails somewhere in the middle, there should be recovery to the state before the Rule body execution (transaction: all or none) .
R_RULE_EXEC_TYPE	When the Rule-body needs to be executed: immediate, delayed, background,...
R_RULE_OWNER	User who created the metadata.
R_RULE_COMMENTS	Any comments about the Rule.
R_RULE_CRT_TIME	Creation timestamp.
R_RULE_MOD_TIME	Last modification timestamp.
FC0D87: Tokens	
R_TOKEN_NAMESPACE	Namespace of the token, eg. Data_type.
R_TOKEN_NAME	A value in the name space. Eg. 'Gif image'.
R_TOKEN_ID	Exposed internal identifier for the token_name.
R_TOKEN_VALUE	(Possibly) null string for other purposes.
Derived Class Objects:	
DC01 : Collection hierarchy	Child can be in more than one parent.
R_PARENT_COLL_ID	Internal identifier for parent collection.
R_CHILD_COLL_ID	Internal identifier for child collection.
R_COLL_COLL_MAP	Type of parentage: direct, soft link.
DC02: Data collection mapping - data can be in more than one coll.	
R_COLL_ID	Internal identifier for collection.
R_DATA_ID	Internal identifier for data in collection.

R_DATA_COLL_MAP	Type of mapping: direct, soft link.
DCO3 : User Group	
R_GROUP_USER_ID	Internal identifier of user-group.
R_MEMBER_USER_ID	Internal identifier of user.
DC04: Resource Group mapping	
R_RSRC_GROUP_NAME	Name of logical resource.
R_RSRC_ID	Internal identifier for physical resource.
DC05 : Metadata mapping	
R_OBJ_ID	Internal identifier. Note that all internal identifiers are generated from same sequence. Metadata can be attached to any first class object including other metadata! The obj_id is used internally. But the client will access using the first class object's name.
R_META_ID	Internal identifier of the metadata.
DC06 : Access Control List	
R_OBJ_ID	Internal identifier. Note that all internal identifiers are generated from same sequence. Access control can be attached to any first class object. The obj_id is used internally. But the client will access using the first class object's name.
R_USER_ID	User or group-user identifier.
R_ACCESS_TYPE	Access allowed for the user: r,w,x,....
DC07 : Deny Access Control List	
R_OBJ_ID	Internal identifier. Note that all internal identifiers are generated from same sequence. Access control can be attached to any first class object. The obj_id is used internally. But the client will access using the first class object's name.
R_USER_ID	User or group-user identifier.
R_ACCESS_TYPE	Access allowed for the user: r,w,x,....
DC08 : Audit Information	
R_OBJ_ID	Internal identifier. Note that all internal identifiers are generated from same sequence. The obj_id is used internally. But the client access using the first class object's name.
R_USER_ID	User identifier.
R_ACTION_TYPE	Action being performed.
R_COMMENT	Information about the action.
R_AUDIT_CRT_TIME	Creation timestamp.
DC09 : User Session Info	

R_USER_NAME	User_name.
R_SESSION_KEY	Session key generated for the session.
R_SESSION_INFO	Information about host:ppid - can be any string that uniquely identifies the session for that user. This string is sent by client.
R_SESSION_EXPIRY	Time at which the session expires.

7 iRODS Micro-services

Micro-services are small, well-defined procedures/functions that perform a certain task. Micro-services are developed and made available by system programmers and application programmers and compiled into the iRODS Server code. Users and administrators can chain these Micro-services to implement a larger macro-level functionality that they want to use or provide for others. In this manner, the users/administrators can have full control over what happens when one performs a macro-level functionality. These macro-level functionalities are called Actions. By having more than one chain of Micro-services for an Action, a system can have multiple ways of performing the Action. Using priorities and validation conditions at run-time, the system chooses the 'best' Micro-service chain to be executed. There are other caveats to this execution paradigm which were discussed in section 3 on Rule Oriented Programming.

The task performed by a Micro-service can be quite small or very involved. We leave it to the Micro-service developer to choose the proper level of granularity for their task differentiation. A good Rule of thumb is to divide a large task into sub-tasks with well-defined interfaces and make each into a Micro-service. If two such sub-tasks are always done together, it would be a good idea to group them together into one Micro-service. Since the user/administrator chains the Micro-services into Actions, having too fine grained a differentiation will make the splicing cumbersome and difficult. On the other hand, making a large task into a single Micro-service takes away the control that is given to the end user/administrator who might want to choose not to do some parts of the task. We recommend that normal coding practices and good design principles used in module and method generation be applied in deciding the granularity for each Micro-service task.

The Micro-services are organized into the following categories:

- Core Micro-services
  - Rule Engine Micro-services
  - Workflow Micro-services
  - Data Object Low-level Micro-services
  - Data Object Micro-services
  - Collection Micro-services
  - Proxy Command Micro-services
- iCAT Services
  - iCAT System Services:
  - iCAT Micro-services:
- Framework Services:
  - Rule-oriented Database Access Micro-services
  - XMessaging System Micro-services
  - Email Micro-services
  - Key-Value (Attr-Value) Micro-services
  - User Micro-services
  - System Micro-services

- Module Micro-services
    - ERA - Electronic Records Archives Program
    - XML
    - HDF
    - Properties
    - Web Services
    - Guinot

Within each category, multiple Micro-services may be defined. The list will grow over time as more functionality is added to the data grid.

Core Micro-services

Rule Engine Micro-services

* msiAdmChangeCoreIRB	- Changes the core.irb file from the client
* msiAdmAppendToTopOfCoreIRB	- Prepends another irb file to the core.irb file
* msiAdmAddAppRuleStruct	- Adds application level IRB Rules and DVM and FNM mappings to the Rule engine.
* msiAdmClearAppRuleStruct	- Clears application level IRB Rules and DVM and FNM mappings that were loaded into the Rule engine.
* msiAdmShowIRB	- Displays the currently loaded Rules
* msiAdmShowDVM	- Displays the currently loaded variable name mappings
* msiAdmShowFNM	- Displays the currently loaded microServices/Actions name mappings

Workflow Micro-services

* nop, null	- No action
* cut	- Not to retry any other applicable Rules for this action
* succeed	- Succeed immediately
* fail	- Fail immediately - recovery and retries are possible
* msiGoodFailure	- Useful when you want to fail but no recovery initiated.
* msiSleep	- Sleep
* whileExec	- While loop
* forExec	- For loop with initial,step and end condition
* forEachExec	- For loop iterating over a row of tables or a list
* break	- Breaks out of while, for and forEach loops
* writeString	- Writes a string to stdout buffer
* writeLine	- Writes a line (with end-of-line) to stdout buffer
* assign	- assigna a value to a parameter
* ifExec	- If-then-else conditional branch
* delayExec	- Delays an execution of Micro-services or Rules
* remoteExec	- Remote execution of Micro-services or Rules
* applyAllRules	-Apply all applicable Rules when executing a given Rule

Data Object Low-level Micro-services (Can be called by client through irule.)

* msiDataObjCreate	- Create a data object
* msiDataObjOpen	- Open a data object
* msiDataObjClose	- Close an opened data object
* msiDataObjLseek	- Lseek
* msiDataObjRead	- Read an opened data object
* msiDataObjWrite	- Write

Data Object Micro-services (Can be called by client through irule.)

* msiDataObjUnlink	- Delete
* msiDataObjRepl	- Replicate
* msiDataObjCopy	- Copy
* msiDataObjGet	- Get
* msiDataObjPut	- Put
* msiDataObjPutWithOptions	- Put with options
* msiDataObjChksum	- Checksum a data object
* msiDataObjPhymv	- Move a data object from one resource to another
* msiDataObjRename	- Rename a data object
* msiDataObjTrim	- Trim the replica
* msiPhyPathReg	- Register a physical file into iRODS
* msiObjStat	- Stat an object
* msiDataObjRsync	- Rsync a data between iRODS and local file
* msiGetObjType	- Finds if a given value is a data,coll,resc,...
* msiCheckPermission	- Check whether permission is granted
* msiCheckOwner	- Check whether user is owner

Collection Micro-services

* msiCollCreate	- Create a collection
* msiCollRepl	- Replicate all files in a collection
* msiRmColl	- Delete a collection

Proxy Command Micro-services

* msiExecCmd	- Remote execute a command

iCAT Services
iCAT System Services:

* msiVacuum	- Postgres vacumm - done periodically

iCAT Micro-services:

* msiCommit	- Commit the database transaction
* msiRollback	- Roll back the database transaction
* msiCreateUser	- Create a new user
* msiDeleteUser	- Delete a user
* msiAddUserToGroup	- Adds a user to a group
* msiCreateCollByAdmin	- Create a collection by administrator
* msiDeleteCollByAdmin	- Delete a collection by administrator
* msiRenameLocalZone	- Renames the local zone by updating various tables
* msiRenameCollection	- Renames a collection; used via a Rule with the above msiRenameLocalZone
* msiExecStrCondQuery	- Given a condition string creates an iCAT query, executes it and returns the values
* msiExecGenQuery	- Executes a given general query structure and returns results
* msiMakeQuery	- Given a select list and a condition list, creates a pseudo-SQL query
*msiGetMoreRows	- Continues an unfinished query, calls msiExecStrCOndQuery and returns results
* msiMakeGenQuery	- Combines msiMakeQuery and msiExecGenQuery and

returns the results of the execution

Rule-oriented Database Access Micro-services

* msiRdaToStdout	- Calls new RDA functions to interface to an arbitrary database returning results in standard-out.
* msiRdaToDataObj	- As above but stores results in an iRODS DataObject.
* msiRdaNoResults	- As above, performs a SQL operation but without resulting output.
* msiRdaCommit	- Commit changes to the database.
* msiRdaRollback	- Rollback (don't commit) changes to the database.

XMessaging System Micro-services

* msiXmsgServerConnect	- Connects to the XMessage Server designated by an iRODS Environment file variable
* msiXmsgCreateStream	- Creats a new Message Stream
* msiCreateXmsgInp	- Creates an Xmsg packet, given required information
* msiSendXmsg	- Sends an Xmsg packet
* msiRcvXmsg	- Receives an Xmsg packet
* msiXmsgServerDisConnect	- Disconnects from the XMessage Server

Email Micro-services

* msiSendMail	- Sends email!
* sendStdoutAsEmail	- Sends rei's stdout as email

Key-Value (Attr-Value) Micro-services

* writeKeyValPairs	-Write keyword value pairs to stdout or stderr, using the given separator
* msiPrintKeyValPair	- Print key-value pairs to rei's stdout buffer
* msiGetValByKey	- Given a key and a keyValPair struct, extract the corresponding value
* msiString2KeyValPair	- Convert a %-separated key=value pair strings into keyValPair Structure
* msiStrArray2String	- Array of Strings converted to a string separated by %-signs
* msiAssociateKeyValuePairsToObj	- Ingest object metadata into iCAT from a AVU structure
• msiRemoveKeyValuePairsFromObj	-Remove object metadata from iCAT using a AVU structure

User Micro-services

* msiExtractNaraMetadata	- Extracts NARA-style metadata from AVU triplets
* msiLoadMetadataFromFile	- Loads AVU metadata from a file
* msiApplyDCMetadataTemplate	- Adds Dublin Core Metadata fields to an object or collection
* writeBytesBuf	- Writes the buffer in an inOutStruct to stdout or stderr
* msiFreeBuffer	- Frees a buffer in an inOutStruct
* writePosInt	- Writes an integer to stdout or stderr
* msiGetDiffTime	- Returns the difference between two system timestamps, given in Unix format (stored in a string)
* msiGetSystemTime	- Returns the local system time of an iRODS Serer
* msiHumanToSystemTime	- Converts a human readable date to a system timestamp

* msiGetIcatTime	- Returns the system time for the iCAT Server
* msiGetTaggedValueFromString	- Given a Tag-Name gets the value from a file in tagged-format (pseudo-XML)
* msiExtractTemplateMDFromBuf	- Extract AVU information using a template
* msiReadMDTemplateIntoTagStruct	-Load template file contents into Tag structure

System Micro-services (Can only be called by the server process)

* msiSetDefaultResc	- Set the default resource
* msiSetNoDirectRescInp	- Sets a list of resources that cannot be used by a normal user directly.
* msiSetRescSortScheme	- Set the scheme for selecting the best resource to use
* msiSetMultiReplPerResc	- Sets the number of copies per resource to unlimited
* msiSetDataObjPreferredResc	- If the data has multiple copies, specify thepreferred copy to use
* msiSetDataObjAvoidResc	- Specify the copy to avoid
* msiSetGraftPathScheme	- Set the scheme for composing the physical path in the vault to GRAFT_PATH.
* msiSetRandomScheme	- Set the the scheme for composing the physical path in the vault to RANDOM.
* msiSetResource	- Sets the resource from default
* msiSortDataObj	- Sort the replica randomly when choosing which copy to use
* msiSetNumThreads	- Specify the parameters for determining the number of threads to use for data transfer.
* msiSysChksumDataObj	- Checksum a data object.
* msiSysReplDataObj	- Replicate a data object.
* msiStageDataObj	- Stage the data object to the specified resource before operation.
* msiNoChkFilePathPerm	- Do not check file path permission when registering
* msiNoTrashCan	- Set the policy to no trash can.
* msiSetPublicUserOpr	- Sets a list of operations that can be performed by the user "public".
* msiCheckHostAccessControl	- Set the access control policy.
* msiDeleteDisallowed	- Set the policy for determining certain data cannot be deleted.
* msiSetDataTypeFromExt	- Get data type based on file name extension

Module Micro-services
ERA - Electronic Records Archives Program

* msiRecursiveCollCopy	- Recursively copies a collection and its contents including metadata
* msiGetDataObjACL	- Gets ACL (Access Control List) for a data object in “|” separated format
* msiGetCollectionACL	- Gets ACL (Access Control List) for a collection in “|” separated format
* msiGetDataObjAVUs	- Retrieves metadata AVU triplets for a data object and returns them as an XML file
* msiGetDataObjPSmeta	- Retrieves metadata AVU triplets for a data object in “|” separated format
* msiGetCollectionPSmeta	- Retrieves metadata AVU triplets for a collection in “|” separated format

* msiGetDataObjAIP	- Gets the Archival Information Package of a data object in XML format
* msiLoadMetadataFromDataObj	- Parses an iRODS object for new metadata AVUs
* msiExportRecursiveCollMeta	- Export metadata AVU triplets for a collection and its contents in a “|” separated format
* msiCopyAVUMetadata	- Copies metadata triplets from an iRODS object to another iRODS object
* msiGetUserInfo	- Gets information about user
* msiGetUserACL	- Gets user ACL for all objects and collections
* msiCreateUserAccountsFromDataObj	– Creates new user from information in a iRODS data object
* msiLoadUserModsFromDataObj	- Modify user information from information in a iRoDS data object
* msiDeleteUsersFromDataObj	- Delete user based on information in a iRODS data object
* msiLoadACLFromDataObj	- Loads ACL from information in a iRODS data object
* msiGetAuditTrailInfoByUserID	- Retrieves Audit Trail information for a user ID
* msiGetAuditTrailInfoByObjectID	- Retrieves Audit Trail information for an object ID
* msiGetAuditTrailInfoByActionID	- Retrieves Audit Trail information for a given action ID
* msiGetAuditTrailInfoByKeywords	- Retrieves Audit Trail information by keywords in the comment field
* msiGetAuditTrailInfoByTimeStamp	– Retrieves Audit Trail information by time stamp period
* msiSetDataType	- Sets data type for an object
* msiGuessDataType	- Guesses the data type of an object based on its file extension
* msiGetCollectionContentsReport	- Returns the object count and total disk usage of a collection

XML

* msiXsltApply	- Given an XML object and an XSLT object, returns the XML object after applying the XSLT transformation

HDF

* msiH5File_open	- Open an HDF file
* msiH5File_close	- Close an HDF file
* msiH5Dataset_read	- Read data from and HDF file
* msiH5Dataset_read_attribute	- Read data attribute from an HDF file
* msiH5Group_read_attribute	- Read attributes of a group in an HDF file

Properties

* msiPropertiesNew	- Create a new empty property list
* msiPropertiesClear	- Clear a property list
* msiPropertiesClone	- Clone a property list, returning a new property list
* msiPropertiesAdd	- Add a property and value to a property list. If the property is already in the list, its value is changed. Otherwise the property is added.
* msiPropertiesRemove	- Remove a property from the list
* msiPropertiesGet	- Get the value of a property in a property list. The property list is left unmodified
* msiPropertiesSet	- Set the value of a property in a property list. If the

	property is already in the list, its value is changed. Otherwise the proprety is added.
* msiPropertiesExists	- Return true (integer 1) if the keyword has a property value in the property list, and false (integer 0) otherwise. The property list is unmodified.
* msiPropertiesToString	- Convert a property list into a string buffer. The property list is left unmodified.
* msiPropertiesFromString	- Parse a string into a new property list. The existing property list, if any, is deleted

Web Services

* msiGetQuote	- Returns a stock quotation using web service provided by http://www.wegserviceX.NE
* msiIp2location	- Returns host name and details given an IP address, using the web service provided by http://ws.fraudlabs.com/
* msiConvertCurrency	- Returns conversion reate for currencies from one country to another, using web service provided by http://www.webserviceX.NET
* msiObjByName	- Returns position and type of an astronomical object given a name using the NASA/IPAC Extragalactic Database (NED) web service at http://voservices.net/ws_v2__0/NED.asmx
* msiSdssImgCutout_GetJpeg	- Returns an image buffer given a position and cutout size using the SDSS Image Cut Out web service at http://skyserver.sdss.org

Guinot

* msiGetFormattedSystemTime	- Returns the local system time

Though Micro-services can be any normal C procedure, there is a standard interfacing mechanism that needs to be adopted when making a C procedure into a Micro-service. A C procedure that you want to turn into a Micro-service can have any number of arguments and any type or structure (with some caveats, see discussion on the parameter structure) . When a Rule Engine interacts with a Micro-service, it interacts with a published (standardized within iRODS) parametric structure of the type called msParam_t. Hence, glue code is needed that converts from msParam_t to the actual argument type of the underlying Micro-service. We call this glue code a Micro-service interface (msi for short). The msi routine will map the msParam_t structure to the call arguments and convert back any output parameters to the msParam_t structure. We illustrate with an example below.

We recommend that the Micro-service interface procedures be pre-fixed with the three-letter acronym msi. Hence, a procedure called createCollection can have an interface routine called msiCreateCollection. The Rule Engine will invoke msiCreateCollection which in turn will invoke createCollection.

Each of the msi procedure calls is registered in the Rule Engine. Only these registered Micro-services can be invoked by the Rule Engine. The registration is done by adding the name of the msi procedure call to a C structure table maintained for this purpose. The table is called ActionTable[] and is found in the server/include/reAction.h file.

7.1 Micro-service Input/Output Arguments

One can pass arguments to a Rule, Micro-service or Action through explicit arguments, as done in the case of C function or procedure calls. The input parameters can take two forms:

- **Literal:** If an argument does not begin with a special character (#, $ or *), it is treated as a character string input. For example, in the Micro-service msiSortRescSortScheme(random), the character string "random" will be passed in as input. Literal can only be used as input parameters and not output parameters.

- **Variable:** If an argument begins with the * character, it is treated as a variable argument. Variable arguments can be used both as input and output parameters. The output parameter from one Micro-service can be explicitly specified as the input parameter of another Micro-service. This powerful capability allows very complex workflow-like rules to be constructed. For example, in the following workflow chain:

  ```
  msiDataObjOpen(/x/y/z,*FD)##msiDataObjRead(*FD,10000,*BUF)
  ```

  msiDataObjOpen opens a data object with the input path /x/y/z and the output file descriptor is placed in the variable parameter *FD. *FD is then used by msiDataObjRead as an input parameter for the read.

User-level workflow-like rules can be invoked through the irule command or the rcExecMyRule API. Internally, the Rule system uses the msParam_t struct to store the content of Variable arguments. The definition of the structure can be found in the file clientLib/include/api/dataObjInpOut.h.

```
typedef struct MsParam {
   char *label;
   char *type;        /* this is the name of the packing instruction in
                       * rodsPackTable.h */
   void *inOutStruct;
   bytesBuf_t *inpOutBuf;
} msParam_t;
```

This data structure is universal in the sense that it can be used to represent all parameter types including very complex data structures. The field label is the identifier used in the actual Rule. e.g., if a Rule calls a Micro-service msiDataObjCreate(*A,*S_FD), the strings "*A" and "*S_FD" are the labels of their respective structures. The type field identifies the type of data that is stored in the inOutStruct. The data-types suppported, though fairly extensive, are restricted to the ones that are given in the file clientLib/include/api/rodsDef.h. The set of supported data types and structures is listed in Table 4. The value of the input/output is given in the inOutStruct field. The inpOutBuf is a buffer that can be used to pass binary data as part of the parameter. The parameters are passed as an array as defined in the following type definition.

```
typedef struct MsParamArray {
  int len;
  int oprType;
  msParam_t **msParam;
} msParamArray_t;
```

The msParam_t structure provides a uniform type definition for the Rule Engine to handle and operate. msParam_t has the following type definition:

```
typedef struct MsParam {
  char *label;
  char *type;        /* this is the name of the packing instruction in
                  * rodsPackTable.h */
  void *inOutStruct;
  bytesBuf_t *inpOutBuf;
} msParam_t;
```

Here "label" is the name given to the argument in the call; "type" is the C structure type that is supported by the iRODS system. The set of supported types is listed in Table 4. The data structures include support for passing parameters for rule invocation using the msParam structure and for passing values between client-server and server-server interactions. The values can be found in the file clientLib/include/rodsPackTable.h.

Table 4. Data Structures Supported by iRODS

STR_PI
INT_PI
StartupPack_PI
Version_PI
RErrMsg_PI
RError_PI
RHostAddr_PI
RODS_STAT_T_PI
RODS_DIRENT_T_PI
KeyValPair_PI
InxIvalPair_PI
InxValPair_PI
PortList_PI
PortalOprOut_PI
PortList_PI
DataOprInp_PI
GenQueryInp_PI
SqlResult_PI
GenQueryOut_PI
DataObjInfo_PI
TransStat_PI
RescGrpInfo_PI
AuthInfo_PI
UserOtherInfo_PI
UserInfo_PI
CollInfo_PI
Rei_PI
ReArg_PI
ReiAndArg_PI
BytesBuf_PI
MsParam_PI
MsParamArray_PI

The inOutStruct is a pointer to the value of the input structure being passed. It can be NULL. The inpOutBuf is used to pass any binary buffers that need to be passed as part of the argument.

7.2 Naming Conventions

When adding files and functions, we recommend some naming conventions for ease of maintenance. Naming conventions are useful for helping maintain the programs and functions that are created under iRODS. Even though we do not force these conventions on developers, we recommend their usage for maintaining good programming practice.

7.2.1 Variable Naming Conventions

We recommend that:

- Variable names use multiple descriptive words.
  Example: myRodsArgs
- Variable names use camel-case to distinguish words:
  Example: genQueryInp

7.2.2 Constant Naming Conventions

We recommend using one of the two following conventions:

- Constant string names use multiple descriptive words and start with an upper-case letter.
  Example: Msg_Header_PI
- Constant string names use upper-case letters separated with an under-score.
  Example: NAME_LEN

7.2.3 Function Naming Conventions

All C functions in iRODS occupy the same namespace. To avoid function name collisions, we recommend that:

- Function names use multiple descriptive words.
  Example: getMsParamByLabel
- Function names use camel-case to distinguish words:
  Example: printMsParam
- Micro-service function names start with "msi":
  Example: msiDataObjGet
- Micro-service helper function names start with "mh":
- Server function names start with "rs".
- Client function names start with "rc".

7.2.4 File Naming Conventions

General file purpose may be inferred by the location of the file in the iRODS directory tree. For instance, those in the server/re/src directory are part of the Rule engine, while those in the clients/icommands/src directory are command-line tools. Beyond this, we recommend that:

- File names use multiple descriptive words.
  Example: rodsServer.c contains the iRODS Server main program.
- File names reflect the names of functions in the file.
  Example: msParam.c contains utility functions that work with the msParam struct.
- File names use camel-case to distinguish words:
  Example: irodsReServer.c

No two files in the same directory have names that differ only by case. This causes problems with Windows and old Mac file systems.

8 Extending iRODS

The iRODS Data Grid is highly extensible. New Micro-services can be added, new Rules can be created, and new state information can be saved. This means the iRODS Data Grid can evolve. A new collection can be created that is governed by the new Rules and Micro-services and the associated state information. The new collection can be run in parallel with an original collection that is still governed by the original Rules, Micro-services, and state information. The user of the data grid can then migrate files from the collection managed by the old policies to the collection managed by the new policies. Detailed instructions are available on how to apply these extensions to an iRODS Data Grid.

8.1 Changing the IP Address

The iRODS Data Grid assumes that a fixed IP address is used for the iRODS Servers and the iCAT metadata catalog. If a system is moved to a new IP address, multiple parameters should be reset to enable communication to be established between the servers.

On installation, it is possible to have the system use LOCALHOST for the IP address. This makes it possible to set up systems that are self-containted, and suitable for giving demonstrations. The use of LOCALHOST can be set up during installation by setting the environment variable USE_LOCALHOST before running irodssetup. For example, if you are using the bash shell, you would type:

```
USE_LOCALHOST=1
export USE_LOCALHOST
```

Instead of setting USE_LOCALHOST, another way to do this is to disconnect from the network before installing the software. The installation procedure will automatically use LOCALHOST for the IP address.

To modify the IP address of an existing installation, use the following steps:

- Stop iRODS and PostgreSQL:
  ```
  irodsctl stop
  ```
- Edit each of the following files, changing the host name to the new address:
  .odbc.ini in the home directory
  .irods/.irodsEnv in home directory
  pgsql/etc/odbc.ini in the Postgres directory
  server/config/server.config in IRODS directory
- Then start iRODS again:
  ```
  irodsctl start
  ```
- And modify the address of the local resource:
  ```
  iadmin modresc demoResc host localhost
  ```

8.2 How to interface a Micro-service

Micro-services can be added either as a system Micro-service in the server area, or as a module Micro-service that is compiled only as necessary. We discuss both options below.

8.2.1 How to interface a Micro-services as a module

This requires a two-step process:

1. Create a module, which is done once for each module.
2. Add a Micro-service to the module, which is done for every new Micro-service that is added to the module.

8.2.1.1 Creating a Module

A "module" is a bundle of optional software components for iRODS. Typically, a module provides specialized Micro-services. Modules also may provide new rules, library functions, commands, and even application servers. Once you have developed the software to perform a new iRODS function, you can add your software as a new iRODS module via the following steps:

1. Create a directory named for the module:
   ```
   mkdir modules/MODNAME
   ```
2. Move into that directory:
   ```
   cd modules/MODNAME
   ```
3. Create one or more subdirectories for components being added to iRODS:
   ```
   mkdir microservices
   mkdir rules
   mkdir lib
   mkdir clients
   mkdir servers
   ```

For the rest of these instructinos, we'll assume you're adding Micro-services, but similar instructions apply for other additions.

4. Create source, include, and object subdirectories:
   ```
   mkdir microservices/src
   mkdir microservices/include
   mkdir microservices/obj
   ```
5. Add source and include files to the "src" and "include" directories.
6. Create a Makefile by copying one from an existing module, such as "properties":
   ```
   cp ../properties/Makefile .
   ```
7. Edit the Makefile to list your source files and add any special compile flags or libraries you may need. The Makefile must respond to a set of standard targets:

all	build everything
microservices	build new microservices
client	build new clients
server	build new servers
rules	build new rules
clean	remove built objects, etc.
client_cflags	compile flags for building clients
client_ldflags	link flags for building clients
server_cflags	compile flags for building servers
server_ldflags	link flags for building servers

The Micro-services, client, and server targets should compile your code. The client and server targets should link your custom clients and servers. If your module doesn't have one or more of these, the target should exist but do nothing.

The client and server flag targets should echo to stdout the compiler or linker flags needed on *other* clients and servers that use the module. The "cflags" echos should list -I include paths and specific include files. The "ldflags" echos should list -L link paths, -l library names, and specific library or object files.

8. Create an info.txt file by copying one from an existing module:
   ```
   cp ../properties/info.txt .
   ```
9. Edit the info.txt file to include information about your module. The file must contain:

Name:	the name of the module
Brief:	a short description of the module
Description:	a longer description of the module
Dependencies:	a list of modules this module needs
Enabled:	whether the module is enabled by default

Each of these must be on a single (possibly long) line. For dependencies, list module names separated by white-space. Module names must match exactly the directory name of other modules. Case matters.

8.2.1.2 How to use module's info.txt

The "info.txt" file in a module's top-level directory describes the module. It is intended for use by the iRODS Makefiles and configuration scripts.

The file is a list of keyword-value pairs, one per line. For instance:

```
Name:            sample
Brief:           A sample microservice module
Description:     This is a sample module description.
Dependencies:    example
Enabled:         yes
Creator:         University of California, San Diego
Created:         Sept 2007
License:         BSD
```

Name: The name of the module. The name should match the module directory name. (currently unused)

Brief: A brief half-line description of the module. The iRODS configure script uses this value when printing help information about available modules.

Description: A longer description of the module. While the value must be on a single line, it can be several sentences long. (currently unused)

Dependencies: A list of module names upon which the module depends. Names should be space-separated and must match module directory names. The iRODS configure script uses this to insure that all modules that must be enabled together are enabled together.

Enabled: The value "yes" or "no" to indicate if the module should be enabled by default. The iRODS configure script uses this to set defaults on configuring iRODS.

Creator: The name of the principal individual or organization responsible for creating the module. (currently unused)

Created: The approximate creation date of the module. (currently unused)

License: The license covering the module's source code. Additional information may be in the source files or in module documentation. This value is only used as a general indicator. (currently unused)

8.2.1.3 Adding a Micro-service to a Module

All Micro-service functions are discovered by the iRODS Server by reading a master "action" table compiled into the Server. The action table is split into three files:

1. server/re/include/reAction.header
2. server/re/include/reAction.table
3. server/re/include/reAction.footer

The iRODS Makefiles assemble these files, and those provided by modules, to build the file server/re/include/reAction.h. This file contains:

- reAction.header - The header for each Micro-service module.
- reAction.table - The table entries for each Micro-service module.
- reAction.footer

There are two ways to add a Micro-service:

For system Micro-services:

- Edit reAction.header to add function prototypes
- Edit reAction.table to add table initializations

For module Micro-services:

1. Create MODNAME/microservices/include/microservices.header. Edit this to add function prototypes.
2. Create MODNAME/microservices/include/microservices.table. Edit this to add table initializations.

Function prototypes declare the C Micro-service function. While these can be added to the above files directly, authors are encouraged to use a separate include file and just add a #include of that file. For instance, here's a typical Micro-services.header file for a module:

```
// Sample module microservices
#include "sampleMS.h"
```

The reAction.table and each module's microservices.table file contains a C array initialization listing all available Micro-services. Each line in the initialization looks like this:

```
// Sample module microservices
{ "sample", 2, (funcPtr) msiSample },
```

There are three values, in order:

1. The service name is the user-visible name of the Micro-service. It is a string using letters, numbers, and underbar. It should be descriptive and need not match the Micro-service function name.
2. The argument count is the number of msParam_t arguments for the function. It does not include the ruleExecInfo_t argument.

3. The function name is a pointer to the C function for the Micro-service.

8.2.1.4 How to rebuild reAction.h

The server's reAction.h action table is rebuilt automatically by the iRODS root Makefile if any include file changes in server/re/include or modules/*/microservices/include. To force reAction.h to be rebuilt, delete the existing file and run the "reaction" Makefile target:

```
rm server/re/include/reAction.h
make reaction
```

8.2.1.5 How to interface a system (server) Micro-service

1. Create the Micro-service function as needed.

```
int myPetProc(char *in1, int in2, char *out1, int *out2)
{
  ...  my favorite code ...
}
```

2. Create the Micro-service interface (msi) glue procedure.

```
int msiMyPetProc(msParam_t *mPin1, msParam_t *mPin2,
           msParam_t *mPout1, msParam_t *mPout2,
           RuleExecInfo_t *rei)
{
  char *in1,  out1;
  int i, in2, out2;

  RE_TEST_MACRO ("    Calling myPetProc")
  /* the above line is needed for loop back testing using the irule -i option */

  in1  = (char *) mPin1->inOutStruct;
  in2  = (int)    mPin2->inOutStruct;
  out1 = (char *) mPout1->inOutStruct;
  out2 = (int)    mPout2->inOutStruct;

  i = myPetProc(in1, in2, out1, &out2);
  mPout2->inOutStruct = (int) out2;

  return(i);
}
```

3. Define the msi call in the file server/re/include/reAction.table by adding the function signature in the area where all function signatures are defined.

```
int msiMyPetProc(msParam_t *mPin1, msParam_t *mPin2, msParam_t *mPout1,
msParam_t *mPout2, RuleExecInfo_t *rei);
```

4. Register the Micro-service by making an entry in the file server/re/include/reAction.table. The first item in the entry is the external name of the Micro-service, the second is the number of user-defined arguments for the msi procedure call (excluding the RuleExecInfo_t *rei), and the third argument is the name of the msi procedure. Note that the names are the same in the following example for the first and third values in the entry. We recommend this format for clarity purposes:

```
{"msiMyPetProc", 4, (funcPtr) msiMyPetProc}
```

5. If there are any 'include' files that are needed, they can be added to server/re/include/reAction.header.

6. Define the called procedure in an appropriate include file (for the present reFuncDefs.h file would be a fit place for this, since this will require no change in any Makefile) by adding the signature.
   ```
   int myPetProc(char *in1, int in2, char *out1, int *out2);
   ```

The Micro-service is now ready for compilation and use!

8.3 Web Services as Micro-services

Web services can be turned into Micro-services. The iRODS Data Grid already has a few Micro-services that invoke Web services as part of the release. Interfacing a web service is a slightly involved process. There are two steps for encapsulating web-servivces in Micro-services.

1. The first step is to build a common library that can be used by all Micro-services that connect to web services. This is done ONLY ONCE.
2. The second step is done for each Micro-service that is built.

8.3.1 First Step (Done Only Once)

1. Acquire the gsoap distribution. This can be found at: http://sourceforge.net/projects/gsoap2 or other mirrored sites.
2. Put the files in the webservices/gsoap directory.
3. Build the gsoap libraries and copy them to appropriate directories (See README.txt in the gsoap distribution for more information on building.)
   ```
    cd gsoap
   ./configure
   ./make
   cd soapcpp2
   cp libgsoap++.a   libgsoap.a  libgsoapck++.a libgsoapck.a  libgsoapssl++.a libgsoapssl.a
   ../../microservices/lib
   cp stdsoap2.h ../../microservices/include
   cp stdsoap2.c stdsoap2.h ../../microservices/src/common
   cd ../../microservices/src/common
   rm *
   touch env.h
   ../../../gsoap/soapcpp2/src/soapcpp2 -c -penv env.h
   gcc -c -DWITH_NONAMESPACES envC.c
   gcc -c -DWITH_NONAMESPACES stdsoap2.c
   cp  envC.o  stdsoap2.o ../../obj
   ```
4. Add a file called info.txt in the webservices directory if it is not already there. The content of this file is similar to that of the info.txt file in the modules/properties directory.
5. Make sure that the value for “Enabled” in the info.txt file is “yes” (instead of “no”).
6. Add the word ‘web services’ (without the quotes) to the MODULES option in the ~/iRODS/config/mk/config file. Make sure that the line is not commented out

   ```
   MODULES- webservices properties
   ```

8.3.2 Second Step (Done for Each Web Service)

Here we show an example Micro-service being built for getting a stockQuotation. Building a Micro-service for a web-service is a multi-step process. If not already enabled, enable the Micro-services for web-services by changing the enabling flag to 'yes' in the file modules/webservices/info.txt

1. mkdir webservices/microservices/src/stockQuote
2. Place stockQuote.wsdl in the directory webservices/microservices/src/stockQuote. The wsdl file is obtained from the web services site.
3. cd webservices/microservices/src/stockQuote
4. setenv GSOAPDIR ../../../gsoap/soapcpp2
5. $GSOAPDIR/wsdl/wsdl2h -c -I$GSOAPDIR -o stockQuoteMS_wsdl.h stockQuote.wsdl

   This creates a file called stockQuoteMS_wsdl.h
6. $GSOAPDIR/src/soapcpp2 -c -w -x -C -n -pstockQuote stockQuoteMS_wsdl.h

   This creates files: stockQuote.nsmap, stockQuoteC.c, stockQuoteClient.c, stockQuoteClientLib.c, stockQuoteH.h and stockQuoteStub.h
7. Create the Micro-service in a file. In this case, file stockQuoteMS.c is created in webservices/microservices/src/stockQuote. The structures and the call can be found at stockQuoteStub.h.

```
/*** Copyright (c), The Regents of the University of California          ***
 *** For more information please refer to files in the COPYRIGHT directory ***/
/**
 * @file    stockQuoteMS.c
 *
 * @brief  Acces to stock quotation web services
 *
 * This Micro-service handles communication with http://www.webserviceX.NET
 * and provides stock quotation - delayed by the web server.
 *
 *
 * @author          Arcot Rajasekar / University of California, San Diego
 */
#include "rsApiHandler.h"
#include "stockQuoteMS.h"
#include "stockQuoteH.h"
#include "stockQuote.nsmap"

int
msiGetQuote(msParam_t* inSymbolParam, msParam_t* outQuoteParam,
            ruleExecInfo_t* rei )
{
  struct soap *soap = soap_new();
  struct _ns1__GetQuote sym;
  struct _ns1__GetQuoteResponse quote;
  char response[10000];
  RE_TEST_MACRO( "    Calling msiGetQuote" );
  sym.symbol = (char *) inSymbolParam->inOutStruct;
  soap_init(soap);
  soap_set_namespaces(soap, stockQuote_namespaces);
  if (soap_call___ns1__GetQuote(soap, NULL, NULL, &sym, &quote) ==
        SOAP_OK)
    {
        fillMsParam( outQuoteParam, NULL, STR_MS_T, quote.GetQuoteResult,
                NULL );
        free (quote.GetQuoteResult);
        return(0);
    }
   else
    {
       sprintf(response,"Error in execution of GetQuote::%s\n",soap->buf);
       fillMsParam( outQuoteParam, NULL, STR_MS_T, response, NULL );
       return(0);
    }
  }
```

8. Create the header file for the Micro-service. In this case, file stockQuoteMS.h is created in webservices/microservice/include

```
/*** Copyright (c), The Regents of the University of California          ***
 *** For more information please refer to files in the COPYRIGHT directory
***/
/**
 * @file stockQuoteMS.h
 *
 * @brief         Declarations for the msiStockQuote* microservices.
 */

#ifndef STOCKQUOTEMS_H /* so that it is not included multiple times by
                                          mistake */
#define STOCKQUOTEMS_H
#include "rods.h"
#include "reGlobalsExtern.h"
#include "rsGlobalExtern.h"
#include "rcGlobalExtern.h"
int msiGetQuote( msParam_t* inSymbolParam, msParam_t* outQuoteParam,
                    ruleExecInfo_t* rei );
#endif   /* STOCKQUOTEMS_H */
```

9. Make sure that the header file is included in the build process. Add the following line in the file microservices.header located in webservices/microservice/include

```
#include "stockQuoteMS.h"
```

10. Add the Micro-service to the list of Micro-services that can be called by the Rule Engine. Add the following line in the file microservices.table located in webservices/microservice/include

```
{ "msiGetQuote",      2,          (funcPtr) msiGetQuote },
```

11. The next step is to change the Makefile in the webservices directory so that the new Micro-service gets compiled and linked during the build process. Add the following lines at the appropriate places in the Makefile located in the modules/webservices directory

```
stockQuoteSrcDir= $(MSSrcDir)/stockQuote
STOCKQUOTE_WS_OBJS = $(MSObjDir)/stockQuoteMS.o
        $(MSObjDir)/stockQuoteClientLib.o
OBJECTS +=    $(STOCKQUOTE_WS_OBJS)
$(STOCKQUOTE_WS_OBJS): $(MSObjDir)/%.o: $(stockQuoteSrcDir)/%.c
        $(DEPEND) $(OTHER_OBJS)
@echo "Compile webservices-stockQuote module `basename $@`..."
@$(CC) -c $(CFLAGS) -o $@ $<
```

12. Compile and run: gmake clean and gmake at the iRODS top level

8.4 iRODS FUSE User Level File System

FUSE (Filesystem in Userspace) is a free Unix kernel module that allows non-privileged users to create their own file systems without editing the kernel code. This is achieved by running the file system code in user space, while the FUSE module only provides a "bridge" to the actual kernel interfaces.

The iRODS FUSE implementation allows normal users to access data stored in iRODS using standard UNIX commands (ls, cp, etc) and system calls (open, read, write, etc).

Building iRODS FUSE:

a) Edit the config/config.mk file:
   1. Uncomment the line:
      ```
      # IRODS_FS = 1
      ```
   2. Set fuseHomeDir to the directory where the fuse library and include files are installed. e.g.,
      ```
      fuseHomeDir=/usr/local/fuse
      ```

b) Making iRODS Fuse:
   Type in:
   ```
   cd clients/fuse
   gmake
   ```

Running irods Fuse:

1. cd clients/fuse/bin
2. make a local directory for mounting. e.g.,
   ```
   mkdir /usr/tmp/fmount
   ```
3. Set up the iRODS client env (~/irods/.irodsEnv) so that iCommands will work. Type in:
   ```
   iinit
   ```
   and do the normal login.
4. Mount the home collection to the local directory by typing in:
   ```
   ./irodsFs/usr/tmp/fmount
   ```
5. The user's home collection is now mounted. The iRODS files and sub-collections in the user's home collection should be accessible with normal UNIX commands through the /usr/tmp/fmount directory.

8.5 Mounted iRODS Collection

The -m option of the imcoll command can be used to associate (mount) an iRODS collection with a a physical directory (e.g.,a UNIX directory) or a structured file. If the mountType is 'f' or 'filesystem', the first argument is the UNIX directory path to be mounted. Only the top level collection/directory will be registered. The entire content of the registered directory can then be accessed using iRods commands such as iput, iget, ils and the client APIs. This is simlilar to mounting a UNIX file system except that a UNIX directory is mounted to an iRODS collection. For example, the following command mounts the /temp/myDir UNIX directory to the /tempZone/home/myUser/mymount collection:

```
imcoll -m filesystem /temp/myDir /tempZone/home/myUser/mymount
```

An admin user will be able to mount any Unix directory. But for normal user, he/she needs to have a UNIX account on the iRODS Server with the same name as his/her iRODS user account. Only a UNIX directory created with this account can be mounted by the user. Access control to the mounted data will be based on the access permission of the mounted collection.

If the mountType is 't' or 'tar', the first argument is the iRODS logical path of a tar file which will be used as the 'structured file' for the mounted collection. The [-R resource] option is used to specify the resource to create this tar structured file in case it does not already exist. Once the tar structured file is mounted, the content of the tar file can be accessed using iRODS commands such as iput ils, iget, and the client APIs. For example, the following command mounts the iRODS tar file /tZone/home/myUser/tar/foo.tar to the /tZone/home/myUser/tarcoll collection:

```
imcoll -m tar /tZone/home/myUser/tar/foo.tar /tZone/home/myUser/tardir
```

The tar structured file implementation uses a cache on the server to cache the mounted tar file. i.e., the tar file is untarred to a cache on the server before any iRODS operation. The 'ils -L' command can be use to list the properties of a mounted collection and the status of the associated cache. For example, the following is the output of the ils command:

```
 C- /tZone/home/myUser/tardir  tarStructFile  /tZone/home/myUser/tar/foo.tar
/data/Vault8/rods/tar/foo.tar.cacheDir0;;;demoResc;;;1
```

The output shows that /tZone/home/myUser/tardir is a tar structured file mounted collection. The iRODS path of the tar file is in /tZone/home/myUser/tar/foo.tar. The last item actually contains 3 items separated the string ;;;. It showed that the tar file is untarred into the /data/Vault8/rods/tar/foo.tar.cacheDir0 directory in the demoResc resource. The value of '1' for the last item showed that the cache content has been changed (dirty) and the original tar file needs be synchronized with the changes. The -s option can be used to synchronize the tar file with the cache. For example:

```
imcoll -s /tZone/home/myUser/tardir
```

The -p option can be used to purge the cache. For example:

```
imcoll -p /tZone/home/myUser/tardir
```

The -s and -p can be used together to synchronize the tar file and then purge the cache. If the mountType is 'h' or 'haaw', the first argument is the logical path of a haaw type structured file developed by UK eScience.

NOTE: the haaw type structured file has NOT yet been implemented in iRODS.

The -U option can be used to unmount an iRODS collection. For example, the following command unmounts the /tempZone/home/myUser/mymount collection:

```
imcoll -U /tempZone/home/myUser/mymount
```

8.5.1 Building libtar and linking the iRODS servers with libtar

The tar structured file implementation requires linking the servers with the libtar library and the procedures are given below.

1. Download the Free BSD libtar software from:
   http://www.feep.net/libtar/
   http://www.freebsdsoftware.org/devel/libtar.html

2. Make the libtar software
   1. Edit the config/config.mk files by uncommenting the line:
      TAR_STRUCT_FILE=1
   2. Set the parameter tarDir to your libtar path. e.g.,
      tarDir=/data/libtar-1.2.11
   3. cd to the top iRods directory and type in "make clean; make".

It is recommended that the libtar software be installed in the same parent directory as iRODS and PostgreSQL installation.

Also note that the current version of libtar 1.2.11 does not support tar file sizes larger than 2 GBytes. We have made a mod to libtar 1.2.11 so that it can handle files larger than 2 GBytes. This mod is only needed for building the irods server software. Please contact all@diceresearch.org for this mod.

8.6 Developer's Corner

Since Micro-services are being built by partners, it would be helpful to know what is available and what is in the works. Also, if one has a wish list of Micro-services, then they may be coded by someone else who has a similar interest. To help in this regard, we have created a wiki page with these details. One can find this page at the Developers Corner on the iRODS wiki at http://irods.diceresearch.org

9 Example Rules

Rules that have been written into a file can be executed through the irule command:

```
irule –vF Rulename.ir
```

The irule command has the following input parameters as listed by the help package:

```
Usage : irule [--test] [-F inputFile] [ruleBody inputParam outParamDesc]
```

Submit a user defined rule to be executed by an irods server. The command requires 3 inputs:
1. ruleBody - This is the body of the rule to be executed.

2. inputParam - The input parameters. The input values for the rule are specified here. If there is no input, a string containing "null" must be specified.
3. outParamDesc - Description of the set of output parmeters to be returned. If there is no output, a string containing "null" must be specified.

The format of the ruleBody follows the specification given section 4.3. The workflow-chain which is the third part of the rule body, is a sequence of Micro-services/rules to be executed by this rule. The Micro-services/rules in the sequence are separated by the '##' separator.

The input can be specified through the command line or in a file using the -F option. The inputFile should contain 3 lines, one for each input. An example of the input is given in the file:

clients/icommands/test/ruleInp1

Options are:

-test enable test mode so that the Micro-services are not executed, instead a loopback is performed
-F inputFile - read the file for the input
-v verbose
-h this help

If an inputParam is preceded by the symbol “$”, the irule command prompts for a new value for the attribute value.

Many types of Rules can be created that automate an administrative task, or that validate an assessment criteria, or that enforce an administrative policy. For each example Rule, we describe the construction of the Rule, the Micro-services that are used to compose the Rule, and the results from application of the Rule. We provide text that shows how the Rule can be invoked from the command line using the irule unix command.

We organize the example Rules into six classes:

1. File manipulation
2. System testing
3. User interaction
4. Rule manipulation
5. Resource setting
6. French National Library example

In rest of this section, the Rules are pretty-printed as in the Rulegen language (*.r). The original Rules can be found in their respective *.ir files in the clients/icommands/test directory in the iRODS release. Since none of the Micro-services have recovery equivalents, they are ignored in the pretty printing.

9.1 File Manipulation Rules

9.1.1 Rule 1: listColl.ir - Lists All Files in a Collection.

The "listColl.ir" Rule queries the iCAT Metadata Catalog and retrieves a list of files that satisfy a specified "Condition". An "ActionName" specifies the operation that is performed upon the files as they are added to the list. Each file that is manipulated is printed to "stdout", followed by the printing of a separator line.

The file called listColl.ir contains the body of the Rule and a specification of the input parameters:

```
myTestRule (*ActionName, *Condition) {
   acGetIcatResults(*ActionName, *Condition, *B)
   forEach   (*B)   {
        msiPrintKeyValPair(stdout,*B)
        writeLine(stdout,*K)
   }
}
*ActionName=$list
*Condition=$COLL_NAME = '/tempZone/home/rods/loopTest'
*K=---------------------
```

The "listColl.ir" Rule invokes the Micro-services:

1. acGetIcatResults - this is a Rule which, given an "ActionName" and an SQL "Condition" returns a table of values. In this case:

   "ActionName": is the "list" command
   "Condition": limits application to a specific Collection name,
   COLL_NAME = '/tempZone/home/rods/loopTest'
   (other conditions can be specified)
2. msiPrintKeyValPair – is a Micro-Service that prints a row in a table as a set of key-value pairs to 'stdout'.
3. writeLine – is a Micro-Service that writes a given string buffer to 'stdout'. In this Rule the "writeLine" Micro-Service prints a separator line (made of dashes).
4. forEachExec – is a Micro-Service that takes a table (or list of strings, or %-separated string list), and for each item in the list, executes the corresponding body of the for-loop. The first parameter specifies the variable that has the list (the same variable name is used in the body of the loop to denote an item of the list!). The second parameter is the body given as a sequence of Micro-Services, and the third parameter is the recoveryBody for recovery from failures.

The two Micro-Services are executed in a loop "forEachExec" which iterates over the values in the list, which is returned in the variable "*B". For every row in the table returned by acGetIcatResults, the two Micro-Services are executed.

The listColl.ir Rule prints out the ActionName and Condition values to stdout.

9.1.2 Rule 2: showicatchksumColl.ir - Lists the Checksum of All Files in a Collection

The Rule chains one Rule for accessing the list of files and one Micro-Service, to obtain the checksum of the file from iCAT, and three other Micro-Services to pretty-print the results.

1. acGetIcatResults - is a Rule which, given an "ActionName" and an SQL "Condition" returns a table of values. In this case:
   "ActionName": is the "chksum" command
   "Condition": COLL_NAME = '/tempZone/home/rods/loopTest'
   (this can be any other condition)
2. msiDataObjChksum – is a Micro-Service which calculates the checksum for a file and stores it in iCAT when the Operation parameter is set to ChksumAll.
3. msiGetValByKey – is a Micro-Service which, given a 'row' in a table and an attribute-name, gets the value for that attribute. It is called twice, first to get the DATA_NAME and then to get COLL_NAME.
4. writeLine – is a Micro-Service that can write a given string buffer to 'stdout'. In this Rule, this Micro-Service is used to print the checksum of the file in sentence form.

The Micro-Services are executed in a loop "forEachExec", such that they are executed for every row in the table returned by acGetIcatResults.

```
myTestRule {
   acGetIcatResults(*ActionName, *Condition, *B)
   foreach  (*B)  {
       msiDataObjChksum(*B,*Operation,*C)
       msiGetValByKey(*B,DATA_NAME,*D)
       msiGetValByKey(*B,COLL_NAME,*E)
       writeLine(stdout,CheckSum of *E/*D is *C)
  }
}

*ActionName=chksum
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
*Operation= ChksumAll
```

The Rule prints out the ActionName, Condition, and Operation values, as well as the stdout.

9.1.3 Rule 3: verifychksumColl.ir- Verifies the Checksum of All Files in a Collection

Verification check to make sure that the file has not been corrupted since the last checksum was computed (similar to showicatchksumColl.ir).

The Rule chains one Rule for accessing the list of files and one Micro-service to check whether the file's checksum is valid, and three other Micro-services to pretty-print the results.

1. acGetIcatResults - is a Rule which, given a "ActionName" and a SQL "Condition" returns a table of values. In this case:
   "ActionName": chksum
   "Condition": COLL_NAME = '/tempZone/home/rods/loopTest'
   (this can be any other condition)
2. msiDataObjChksum – is a Micro-service which verifies the checksum of the physical file when the Operation parameter is set to verifyChksum.
3. msiGetValByKey – is a Micro-service that, given a 'row' in a table and an attribute-name, gets the value for that attribute. It is called twice, first to get DATA_NAME and then to get COLL_NAME.
4. writeLine – is a Micro-service writes a given string buffer to 'stdout'. In this Rule this Micro-service is used to print the checksum of the file in sentence form.

The Micro-services are executed in a loop "forEachExec", such that they are executed for every row in the table returned by acGetIcatResults.

```
myTestRule  {
   acGetIcatResults(*ActionName, *Condition, *B)
   foreach   (*B)   {
       msiDataObjChksum(*B,*Operation,*C)
       msiGetValByKey(*B,DATA_NAME,*D)
       msiGetValByKey(*B,COLL_NAME,*E)
       writeLine(stdout,CheckSum of *E/*D is *C)
   }
}

*ActionName=chksum
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
*Operation= verifyChksum
```

The Rule prints out the ActionName, Condition, and Operation values, as well as the stdout.

9.1.4 Rule 4: forcchksumColl.ir - Recompute the Checksum of All Files in a Collection

Used to reset checksums. Recomputes the checksum of all files in a given Collection, and registers them in the iCAT Metadata Catalog (similar to showicatchksumColl.ir). The Rule chains one Rule for accessing the list of files and one Micro-service to compute a valid checksum and register it in the iCAT, and three other Micro-services to pretty-print the results.

1. acGetIcatResults - is a Rule which, given an a "ActionName" and a SQL "Condition" returns a table of values. In this case:
   "ActionName": chksum
   "Condition": COLL_NAME = '/tempZone/home/rods/loopTest'
   (this can be any other condition)
2. msiDataObjChksum – is a Micro-service which computes the checksum of the physical file when the Operation parameter is set to forceChksum.
3. msiGetValByKey – is a Micro-service that, given a 'row' in a table and an attribute-name, gets the value for that attribute. It is called twice, first to get DATA_NAME and then to get COLL_NAME.
4. writeLine – is a Micro-service that can write a given string buffer to 'stdout'. In this Rule the Micro-service is used to print the checksum of the file in sentence form.

The Micro-services are executed in a loop "forEachExec", such that they are executed for every row in the table returned by acGetIcatResults.

```
myTestRule {
   acGetIcatResults(*ActionName, *Condition, *B)
   foreach  (*B)  {
       msiDataObjChksum(*B,*Operation,*C)
       msiGetValByKey(*B,DATA_NAME,*D)
       msiGetValByKey(*B,COLL_NAME,*E)
       writeLine(stdout,CheckSum of *E/*D is *C)
   }
}

*ActionName=chksum
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
*Operation= forceChksum
```

The Rule prints out the ActionName, Condition, and Operation values, as well as the stdout. As a side-effect, the checksum is computed for each file and registered in the iCAT Metadata Catalog.

9.1.5 Rule 5: copyColl.ir – Copies Files from Source to Destination Collections

Makes a copy of each file from a source collection to a destination collection. The new physical copy is stored in a specified storage resource. The Rule chains one Rule for accessing the list of files and one Micro-service to make the copy, and other Micro-services to pretty-print the results.

1. acGetIcatResults - is a Rule which, given a "ActionName" and a SQL "Condition" returns a table of values. In this case:
   "ActionName": copy
   "Condition": COLL_NAME = '/tempZone/home/rods/loopTest'
   (this can be any other condition)
2. msiDataObjCopy – is a Micro-service which copies a file from one logical (source) collection to another logical (destination) collection that is physically located in the input *Resource. *CC is the status of the copy operation.
3. msiGetValByKey – is a Micro-service that, given a 'row' in a table and an attribute-name, gets the value for that attribute. It is called twice, first to get DATA_NAME and then to get COLL_NAME.
4. writeLine – is a Micro-service that can write a given string buffer to 'stdout'. In this Rule the Micro-service is used to print the checksum of the file in sentence form.

The Micro-services are executed in a loop "forEachExec", such that they are executed for every row in the table returned by acGetIcatResults.

```
myTestRule  {
   acGetIcatResults(*ActionName, *Condition, *B)
   foreach   (*B)   {
       msiGetValByKey(*B,DATA_NAME,*D)
       msiDataObjCopy(*B, *DestColl/*D, *Resource, *CC)
        msiGetValByKey(*B,COLL_NAME,*E)
       writeLine(stdout,CheckSum of *E/*D is *C)
   }
}

*ActionName=$repl
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
```

The Rule prints out the ActionName and Condition values, as well as the stdout.
As a side-effect, the files are copied into a new collection with separate physical copies.
The iCAT Metadata Catalog is modified accordingly.

9.1.6 Rule 6: replColl.ir - Make a Replica of Each File in a Collection.

Makes a replica of each file in the named collection. The physical copy is stored in a storage resource. The Rule chains one Rule for accessing the list of files and one Micro-service to make the copy, which is executed in a loop (one for each file in the list).

1. acGetIcatResults - is a Rule which, given a "ActionName" and a SQL "Condition" returns a table of values. In this case:
   "ActionName": replicate
   "Condition": COLL_NAME = '/tempZone/home/rods/loopTest'
   (this can be any other condition)
2. msiDataObjRepl – is a Micro-service which replicates a file in a Collection (it assigns a different replica number to the new copy in the iCAT Metadata Catalog). The replica is physically stored in the 'tgReplResc' Resource. *Junk contains the status of the operation. In the Rule, the resource is provided as part of the call instead of as an input through a *parameter.

The Micro-Services are executed in a loop "forEachExec", such that they are executed for every row in the table returned by acGetIcatResults.

```
myTestRule {
   acGetIcatResults(*ActionName, *Condition, *B)
   foreach  (*B)  {
       msiDataObjRepl(*B, tgReplResc, *Junk)
   }
}

*ActionName=repl
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
```

The Rule prints out the ActionName and Condition values, as well as the stdout.
As a side-effect, every file is replicated in the same Collection, with a separate physical copy. The iCAT Metadata Catalog is updated accordingly.

9.1.7 Rule 7: trim Coll.ir - Trims the Number of Replicas of a File

Used to delete replicas of a file. The Rule will do nothing if the number of replicas is less than or equal to a specified number given by 'numCopies'. One can specify which replica is preferable for deletion (by defining a 'replNum') and also specify a given resource whose copy is preferred for deletion. If a resource is specified, only copies on that resource, if any, are deleted.

The Rule chains one Rule for accessing the list of files and one Micro-service to make the copy which is executed in a loop (once for each file in the list).

1. acGetIcatResults - is a Rule which, given an"ActionName" and an SQL "Condition" returns a table of values. In this case:
   "ActionName": replicate
   "Condition": COLL_NAME = '/tempZone/home/rods/loopTest'
   (this can be any other condition)
2. msiDataObjTrim– is a Micro-service that trims a file replica. The file is specified by the first parameter. The replica to be deleted is specified by the resource and replNumber parameters. In the call below, the preferred resource is given as 'tgReplResc'. The second parameter gives the preferred resource from which to delete the replica. The third parameter in the Micro-service defines the preferred replica to be deleted. The fourth parameter specifies the minimum number of copies to be retained. Here it is '1', so that at least one copy remains after the trim operation, even if it is a preferred replica number or is located in a preferred resource for deletion. The fifth parameter is useful when performed by an iRODS administrator, and the final parameter is the operation status return.

The Micro-Services are executed in a loop "forEachExec", such that they are executed for every row in the table returned by acGetIcatResults.

```
myTestRule {
   acGetIcatResults(*ActionName, *Condition, *B)
   foreach (*B) {
        msiDataObjTrim(*B,tgReplResc,null,1,null,*C)
   }
}

*ActionName=trim
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
```

The Rule prints out the ActionName and Condition values, as well as the stdout.
As a side-effect, the specified file replicas are deleted. The iCAT Metadata Catalog is modified accordingly.

9.2 User Interaction Rules

9.2.1 Rule 8: sendMailColl.ir - Send e-mail to a Specified e-mail Address.

The Rule is written as a variation of showicatchksumColl.ir, which prints out the checksum of all files in a given Collection. In sendMailColl.ir the results are also sent as an e-mail. The Rule gets a list of files using acGetIcatResults, and then calculates the checksum of each file using the msiDataObjChksum Micro-service, and pretty-prints it to stdout using writeLine. The checksum access and pretty-printing are done in a for-loop for each file in the list. After this loop is completed, the sendStdoutAsEmail Micro-service is invoked to send the e-mail.

1. sendStdoutAsEmail – is a Micro-service which given a sendTo parameter (an e-mail address) and a subjectLine parameter, sends out the stdout buffer as the body of the e-mail. In this case the subject Line is 'Checksum Results'.

The other Micro-Services are defined and used as in the showicatchksumColl.ir Rule; see section 9.1.2.

```
myTestRule {
   acGetIcatResults(*ActionName, *Condition, *B)
   foreach  (*B)   {
       msiDataObjChksum(*B,*Operation,*C)
       msiGetValByKey(*B,DATA_NAME,*D)
       msiGetValByKey(*B,COLL_NAME,*E)
       writeLine(stdout,CheckSum of *E/*D is *C)
  }
   sendStdoutAsEmail(*MailTo,Checksum Results)
}

*ActionName=chksum
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
*Operation= ChksumAll
*MailTo=sekar@sdsc.edu
```

The Rule prints out the ActionName, Condition, and Operation values, as well as the stdout. As a side-effect, an e-mail is sent.

9.2.2 Rule 9: periodicChksumCollColl.ir - Periodically Verify Checksum of Files

Verifies that files have not been corrupted since the last checksum was computed, and sends the results as an e-mail.

The Rule is written as a variation of the verifychksumColl.ir, and sendMailColl.ir Rules; verifychksumColl.ir verifies the checksum of all files in a given Collection and sendMailColl.ir sends the results as an e-mail. The main modification is that the sendMailColl.ir rule-body (with verify instead of show checksum) is executed inside another system Micro-service called delayExec. delayExec queues a given sequence of Micro-services into the queue of the iRODS batch-server, which periodically checks the time and fires the Rule when appropriate. The parameter for delay can be set such that the execution can be done periodically at set intervals.

1. delayExec – is a Micro-service that takes the delayCondition as the first parameter, the Micro-service/rule chain that needs to be executed as the second parameter, and the recovery-Micro-service chain as the third parameter. The delayCondition is given as a tagged condition. In this case, there are two conditions that are specified.

<PLUSET>1m</PLUSET> :	execute the first time after 1 minute.
<EF>5m</EF> :	repeating frequency is every five minutes.

The other Micro-services are defined and used as in the verifychksumColl.ir, and sendMailColl.ir Rules.

```
myTestRule  {
  delayExec(<PLUSET>1m</PLUSET><EF>5m</EF>) {
     acGetIcatResults(*ActionName, *Condition, *B)
     foreach   (*B)   {
         msiDataObjChksum(*B,*Operation,*C)
         msiGetValByKey(*B,DATA_NAME,*D)
         msiGetValByKey(*B,COLL_NAME,*E)
         writeLine(stdout,CheckSum of *E/*D is *C)
     }
     sendStdoutAsEmail(*MailTo,Checksum Results)
  }
}

*ActionName=chksum
*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
*Operation= verifyChksum
*MailTo=sekar@sdsc.edu
```

The Rule prints out the ActionName, Condition, and Operation values, as well as the stdout. As a side-effect, an e-mail is sent at specified periodic intervals.

9.2.3 Rule 10: purgeCollAndEmail.ir – Remove Expired Files

Removes files that have expired (current time greater than the time defined by the DATA_EXPIRY parameter in the iCAT Metadata Catalog) and sends the results as an e-mail. The Rule can be easily converted into a periodic Rule. The Rule makes a call to another Rule named acPurgeFiles, which purges all files whose expiration Time is less than the current time, and whose condition matches the given condition. The call also writes to the stdout buffer the names of the files that have been purged. The sendStdoutAsEmail Micro-service call sends the stdout as an e-mail.

1. acPurgeFiles – is a Rule that takes a Condition as a parameter. All files matching that condition and whose expiration time (as given by the iCAT attribute DATA_EXPIRY) is before the current clock time, are deleted. The Rule is very similar to the Rule in copyColl.ir, but makes use of the msiDataObjUnlink Micro-service to perform the deletion and write a message to stdout.
2. sendStdoutAsEmail – is a Micro-service which given a sendTo parameter (an e-mail address) and a subjectLine parameter, sends the stdout buffer as the body of the e-mail. In this case the subject Line is ‘Purge Results’.

```
myTestRule {
   acPurgeFiles(*Condition)
   sendStdoutAsEmail(*MailTo, Purge Results)
}

*Condition=$ COLL_NAME = '/tempZone/home/rods/loopTest'
*MailTo=sekar@sdsc.edu
```

The Rule prints out the Condition values as well as the stdout.
As a side-effect files are purged and an e-mail is sent. iCAT is modified accordingly.

9.3 Rule Manipulation

9.3.1 Rule 11: showCore.ir - Print the Rules Being Used by the Rule Engine.

The Rule invokes a Micro-Service to pretty-print the Rules being used by the data grid:

1. msiAdmShowIRB - is a Micro-service that reads the data structure in the Rule Engine, which holds the current set of Rules, and pretty-prints that structure to the stdout buffer. The Micro-service has a dummy parameter!

```
myTestRule {
   msiAdmShowIRB(*A)
}
```

No input necessary.

9.3.2 Rule 12: chgCoreToCore1.ir - Change the Rules in the core.irb File

Changes the Rules in the core.irb file using the content of a given input file name. The input file should be in the server/config/reConfigs directory. The Rule invokes a Micro-service to perform this change.

1. msiAdmAppendToTopOfCoreIRB - is a Micro-service that copies the given file in the configuration directory 'server/config/reConfigs' onto the core.irb file in the same directory. The next time a new client-server session is started the newly copied set of Rules will take effect. Note that the core.irb is overwritten and all previous content is lost.

```
myTestRule {
   msiAdmChangeCoreIRB(*A)
}

*A=core.irb.1
```

The value of *A is printed.

NOTE: The following two Rules:

```
chgCoreToCore2.ir has *A=core.irb.2
chgCoreToCore3.ir has *A=core.irb.3
```

and are similar to chgCoreToCore1.ir, except for the input parameter value.

9.3.3 Rule 13: chgCoreToOrig.ir – Change to the Rules in the core.irb.orig File

Changes the core Rules in the core.irb file in the directory server/config/reConfigs back to that of the core.irb.orig file in the same directory. The Rule invokes a Micro-service to perform this change.

1. msiAdmChangeCoreIRB - is a Micro-service that copies the specified file in the configuration directory 'server/config/reConfigs' onto the core.irb file in the same directory. The result is that the next time a new client-server session is started, the new set of Rules will take effect. Note that the core.irb is overwritten and all previous content is lost.

```
myTestRule {
   msiAdmChangeCoreIRB(*A)
}

*A=core.irb.orig
```

The value of *A is printed.

9.3.4 Rule 25: ruleTest17.ir – Prepend Rules and Logical Name Mappings

From files in the server/config/reConfigs directory, additional Rules, function map names, and variable map names are pre-pended to the already existing structures in the Rule Engine. The Rule invokes a Micro-service to perform this addition.

1. msiAdmAddAppRuleStruct - is a Micro-service that reads the given file in the configuration directory 'server/config/reConfigs' and adds them to the Rule list being used by the Rule Engine. These Rules are loaded at the beginning of the core.irb file, and hence can be used to override the core Rules from the core.irb file.

   This Micro-service is different from the Micro-service msiAdmChangeCoreIRB seen in Rule 12, chgCoreToCore1.ir. There are two reasons: first, this Micro-service can be used to pre-pend a new Micro-service name mapping file (*.fnm, defined in the second argument) and a new variable name mapping file (*.dvm, defined in the third argument) to the top of the mapping, thus effectively overriding some of them. The second difference is that the changes made are not permanent. They exist until another change at the end of the client-server session. In contrast, the chgCoreToCore1.ir Micro-Service changes the content of the core.irb file, and hence is permanent and is loaded when the next client-server session is started.

   In this Rule, the core file name is 'core1' (the .irb extension is not needed), which is given as the first parameter of the msiAdmAddAppRuleStruct Micro-service. The other two parameters of the Micro-service are left as null strings in this case. The Micro-service msiAdmShowIRB is used to show the content of the Rule structure. Remember that msiAdmShowIRB uses a dummy argument. Hence *A and *C do not need to be set.

```
myTestRule {
    msiAdmShowIRB(*A)
    msiAdmAddAppRuleStruct(*B,,)
   msiAdmShowIRB(*C)
}

*B=core.irb.1
```

The rules in the core.irb file are printed before and after the update.

9.3.5 Rule 26: ruleTest18.ir – Pre-pend Rules and Logical Name Mappings

This Rule pre-pends **(not changes!)** Rules, function map names, and variable map names from files in the server/config/reConfigs directory to the already existing structures in the Rule Engine. See section 9.24 for more information on the msiAdmAddAppRuleStruct Micro-Service that is used.

This Rule invokes one Micro-service to perform the addition and another to print the data-value-mapping, before and after the addition. The data-value-mapping is used to provide logical names to the values in the whiteboard (REI structure). For example, when one uses userNameClient (e.g. in ruleTest16.ir), it points to a particular leaf value in the complex REI structure: rei->uoic->userName.

1. msiAdmShowDVM is a Micro-service that reads the data-value-mapping data structure in the Rule Engine and pretty-prints that structure to the stdout buffer. The Micro-Service uses a dummy parameter!

.

In this Rule, all three structures – Rules, data-value-mappings, and function-name mappings – are changed.

```
myTestRule {
   msiAdmShowDVM(*A)
   msiAdmAddAppRuleStruct(*B,*B,*B)
  msiAdmShowDVM(*C)
}

*B=core.irb.1
```

The values of *A, *B and *C and stdout are printed. The files that are added are core.irb.1, core.dvm, and core.fnm.

9.3.6 Rule 27: ruleTest19.ir – Appends Rules and logical name mappings.

This Rule appends **(not changes!)** the Rules, function map names, and variable map names from files in the server/config/reConfigs directory to the already existing structures in the Rule Engine. See ruleTest17.ir for more information on the msiAdmAddAppRuleStruct Micro-service that is used.

The Rule invokes a Micro-service to perform this addition and another to print the function-name-mapping before and after the addition. The function-name-mapping is used to map from logical names of Micro-services to internal function names that are compiled in the server code. For example, in core.fnm there is a mapping from the name openObj to the name msiDataObjOpen. Hence, if one writes a Rule using openObj (using the same parametric sequence), then internally the C function msiDataObjOpen would be invoked. This way one can use a logical name for a Micro-service and lazily map it at run time to a physical function name. The msiAdmAddAppRuleStruct provides a means for doing this on the fly.

1. msiAdmShowFNM is a Micro-service that reads the function-name-mapping data structure in the Rule Engine and pretty-prints that structure to the stdout buffer. The Micro-Service has a dummy parameter!

.

In this Rule, all three structures – Rules, data-value-mappings, and function-name mappings – are changed.

```
myTestRule  {
    msiAdmShowFNM(*A)
    msiAdmAddAppRuleStruct(*B,*B,*B)
   msiAdmShowFNM(*C)
}

*B=core.irb.1
```

The values of *A, *B and *C and stdout are printed. The entries in the function name mapping structure are printed before and after the update.

9.3.7 Rule 14: replCollDelayed.ir - Make a Replica of Each File in a Collection

Makes a replica of each file into the same Collection, but at a later point in time. The body of the Rule is quite different from that of replColl.ir, which uses a forEachExec Micro-service to create individual file replicas. In this case, a new Micro-service is used to perform Collection-level replication.

1. msiReplColl – is a Micro-service that replicates a Collection (giving a different replica number to each newly replicated file). In this case the replica is physically stored in the 'demoResc2' Resource, which is given as the second parameter of the Micro-service. The third parameter is a string that provides information about the type of replication being made; the value for this parameter can be an empty string, in which case all files are replicated, or it can be 'backupMode' in which case, if a good copy already exists in the destination resource, the Rule will not perform a replication. The fourth parameter outputs the status of the operation.

The Micro-service is executed in a delayExec mode with a 1 minute delay. See Rule 9, periodicChksumCollColl.ir for more information on delayExec Micro-Service.

```
myTestRule  {
   delayExec(<PLUSET>1m</PLUSET>) {
        msiReplColl(*desc_coll,*desc_resc, backupMode,  *outbuf)
   }
}

*desc_coll=/tempZone/home/rods/repl_test
*desc_resc=demoResc2
```

The Rule prints out the stdout. As a side-effect, the files are replicated in the same Collection with separate physical copies. iCAT is modified accordingly.

9.4 System Testing

9.4.1 Rule 16: ruleTest1.ir, ruleTest2.ir, ruleTest3.ir – Tests Parametric Variable

Three versions of the rule exist, labeled ruleTest1.ir, ruleTest2.ir, and ruleTest3.ir. The Rule assigns the result of evaluating a conditional expression to a parametric variable (*-variable).

1. **Assign** - is a system Micro-service. The value of the second parameter is assigned to the first parameter, after an evaluation is performed, if needed.

The conditional expression in this Rule checks whether the iRODS client user name is equivalent to the string expression r*s (* is a wild card string character). So, for example, if the user is rods, it will evaluate to 1. If there is no string expression match, then 0 is assigned.

```
myTestRule {
   assign(*A, $userNameClient like r*s)
}
```

The Rule prints out the value of *A.

NOTE: The following two Rules show variations of assignment testing:

ruleTest2.ir: assigns *A to the result of the conditional expression
"$userNameClient like r*w"

```
myTestRule {
   assign(*A, $userNameClient like r*w)
}
```

ruleTest3.ir: assigns *A to the result of adding two numbers, 200 and 300:

```
myTestRule {
   assign(*A, 200 + 300 )
}
```

9.4.2 Rule 17: ruleTest4.ir, ruleTest5.ir, ruleTest6.ir, ruleTest7.ir -- Tests $-variable

Assign values to a Whiteboard Variable (REI or Rule Execution Infrastructure variable), or a $-variable.

The Rule shows assignment to whiteboard (REI) variables ($-variable).

1. Assign - is a system Micro-service. The value of the second parameter is assigned to the first parameter, after any evaluation is performed, if needed.

ruleTest4.ir: tests assignment to string $-variable
This Rule first assigns *A to the value of $rodsZoneClient (client's Zone name), then assigns $rodsZoneClient to $userNameClient (client's user name), and lastly assigns *B to the current value in $rodsZoneClient. The test will be correct if *A prints the client's zone name, and *B prints the client's user name.

```
myTestRule {
     assign(*A,$rodsZoneClient)
     assign($rodsZoneClient,$userNameClient)
     assign(*B,$rodsZoneClient)
}
```

ruleTest5.ir: tests assignment to numeric $-variables.
This Rule first assigns *A to the value of $sysUidClient (client's iRODS id), then assigns the valuation of 200+300 to $sysUidClient, and lastly assigns *B to the current value in $sysUidClient. The test will be correct if *A is 0, *C and *B are 500.

```
myTestRule {
     assign(*A,$sysUidClient)
     assign(*C, 200 + 300 )
     assign($sysUidClient, 200 + 300 )
     assign(*B,$sysUidClient)
}
```

ruleTest6.ir: tests assignment to numeric $-variables.
This Rule is the same as ruleTest5.ir, but takes the values of assignment from a parametric variable instead of a string.

```
myTestRule {
     assign(*A,$sysUidClient)
     assign(*C, *D )
     assign($sysUidClient, *D )
     assign(*B,$sysUidClient)
}
```

ruleTest7.ir: tests assignment to both parametric and whiteboard variables.

9.4.3 Rule 18: ruleTest8.ir -- Tests "while" Loop Execution.

The Rule initializes a value to a while loop variable, and then executes a while Micro-service.

1. whileExec – is a Micro-service that executes a while loop. The first argument is a condition that will be checked on each loop iteration. The second argument is the body of the while loop, given as a sequence of Micro-services, and the third argument is the recoveryBody for recovery from failures.

The initial assignment of 0 is made to the loop variable *A and is incremented by 4 every time the loop is executed. In this example the loop terminates when *A is greater than or equal to 20.

The Rule executes correctly if the value of *A is 20 when printed.

```
myTestRule  {
   assign(*A,0)
   while   (*A < 20)    {
       assign(*A, *A + 4)
   }
}
```

No input is needed, even though some are given in this example.

The Rule prints out the *A value as well as the stdout (among others).

9.4.4 Rule 19: ruleTest9.ir -- Tests "for" Loop Execution.

The Rule executes a "for loop" using the forExec Micro-service and prints a sequence.

1. forExec – is a Micro-service that executes a for loop. The first argument is an assignment to a loop-variable. The second argument is a condition check before executing the "for loop", and the third argument is an assignment statement that increments (or decrements) the loop variable. The loop variable can be a string with string conditional checking. The fourth argument is the body of the "for loop", given as a sequence of Micro-services, and the fifth argument is the recoveryBody for recovery from failures.

The initial assignment of 0 is made to the loop variable *A and is incremented by 4 every time the loop is executed. The loop prints to stdout the value of *A, followed by a line break. The loop terminates when *A is greater that or equal to *D which is an input parameter set to (199 * 2) + 200.

The Rule executes correctly if the value of stdout prints a sequence 4, 8, 12,..., 596, each number in a separate line.

```
myTestRule  {
      for   (assign(*A,0), *A < *D , assign(*A,*A + 4) )    {
            writeLine(stdout,*A)
      }
}

*A=1000
*D= (199 * 2) + 200
```

The Rule prints out the *A and *D values as well as the stdout (among others).

9.4.5 Rule 20: ruleTest10.ir -- Tests “if-then-else” Execution.

The Rule executes an “if-then-else” conditional test using the ifExec Micro-service.

1. ifExec – is a Micro-service that executes an if-then-else statement. The first argument is a conditional check. If the check is successful (TRUE), the Micro-service sequence in the second argument will be executed. If the check fails, then the Micro-service sequence in the fourth argument will be executed. The third argument is the recoveryBody for recovery from failures for the then-part, and the sixth argument is the recoveryBody for recovery from failures for the else-part.

The Rule prints sets both *A and *D to the lower of the two values.

```
myTestRule  {
       if ( *A < *D )
         then  assign(*A,*D)
         else assign (*D,*A)
}

*A=1000
*D= (199 * 2) + 200
```

The Rule prints out the *A and *D values as well as stdout (among others).

9.4.6 Rule 21: ruleTest11.ir, ruleTest12.ir -- Tests Writing to stdout and stderr Buffers.

The Rule executes a writeString Micro-service. In the white board (REI structure), there is a structure (called ruleExecOut that is part of the msParamArray) for emulating writing to stdout and stderr buffers. These buffers are part of REI and are not actually written to the screen or console immediately. This structure provides a means to buffer output string messages from the Micro-services. When irule completes execution of the Rule, the buffers can be printed out to the screen. This printing is accomplished by adding ruleExecOut as an output argument to the ruleTest11.ir file. The ruleExecOut structure is passed along for every Micro-Service execution (including remote and delayed executions), and hence can provide serial capture of messages across multiple Micro-Service invocations.

1. writeString – is a Micro-service that writes to a stderr or stdout buffer in the ruleExecOut structure. The first argument is the buffer name (stderr and stdout are the two buffers currently supported). The second argument is the string to be written to the buffer.

```
myTestRule {
        writeString(stdout,alpha beta gamma)
        writeString(stdout,alpha beta gamma)
        writeString(stderr,Error:blah)
}
```

The Rule writes the same string twice to stdout and another string to stderr.
These will be printed out to screen.
There is NO need for any input.
The Rule prints out the stdout and stderr values (among others).

NOTE: ruleTest12.ir is similar to ruleTest11.ir, but uses the writeLine Micro-service.

```
myTestRule {
        writeLine(stdout,alpha beta gamma)
        writeLine(stdout,alpha beta gamma)
        writeLine(stderr,Error:blah)
}
```

The Rule writes the same string twice to stdout and another string to stderr.
These will be printed out to screen.
There is NO need for any input.
The Rule prints out the stdout and stderr values (among others).

9.4.7 Rule 22: ruleTest13.ir -- Test Sending e-mail.

The Rule executes the msiSendMail Micro-Service to send e-mail.

1. msiSendMail – is a Micro-service which sends e-mail using the mail command in the unix system. The first argument is the e-mail address of the receiver. The second argument is the subject string and the third argument is the body of the e-mail No attachments are supported. The sender of the e-mail is the unix uxr-id running the irodsServer.

```
myTestRule  {
      msiSendMail(sekar@sdsc.edu,irods test,mail sent by an msi.did you get this)
}
```

There is NO need for any input or output.
The side effect of the Rule is that an e-mail sent to the specified recipient.

9.4.8 Rule 23: ruleTest14.ir -- Tests "for each" Loop for Comma-separated List

The Rule executes a loop using the forEachExec Micro-service, based on a list of items given to the Micro-service.

1. forEachExec – is a Micro-service that executes a loop for every item in a list given as the first argument. The list can be a comma-separated string (STR_MS_T), array of strings (StrArray_MS_T), array of integers (IntArray_MS_T), or iCAT query result (GenQueryOut_MS_T). The second argument is the body of the forEach loop, given as a sequence of Micro-services, and the third argument is the recoveryBody for recovery from failures.

The Rule takes a comma-separated string and prints every item in that list.

```
myTestRule  {
       forEach   ( *A )   {
            writeLine(stdout,*A)
      }
}

*A= 123,345,567,aa,bb,678
```

The Rule prints out the stdout (among others).

9.4.9 Rule 24: ruleTest15-16.ir -- Tests “for each” Loop Execution on a Query Result

The Rule executes a loop using the forEachExec Micro-service based on a table of rows.

1. forEachExec – is a Micro-service that executes a loop for very item in a list given as the first argument. The list can be a comma-separated string (STR_MS_T), array of strings (StrArray_MS_T), array of integers (IntArray_MS_T), or iCAT query result (GenQueryOut_MS_T). The second argument is the body of the forEach loop, given as a sequence of Micro-services, and the third argument is the recoveryBody for recovery from failures.
2. msiExecStrCondQuery – is a Micro-service which, given an iCAT query, executes it and returns the list in a tabular row structure (GenQueryOut_MS_T).
3. msiPrintKeyValPair – is a Micro-service that takes a row-structure from GenQueryOut_MS_T and prints it as a ColumnName=Value pair.

The Rule uses the result (tabular) from execution of a iCAT query. The Micro-service msiExecStrCondQuery is used to run the query:

```
SELECT DATA_NAME, DATA_REPL_NUM, DATA_CHECKSUM WHERE
DATA_NAME
LIKE ‘foo%’.
```

The result is printed using the msiPrintKeyValPair Micro-service, which prints each row as an attribute-value pair. A separator line is printed after each row.

```
myTestRule {
      |msiExecStrCondQuery(*A 'foo%' ,*B)
       forEach  ( *B )  {
           msiPrintKeyValPair(stdout,*B)
           writeLine(stdout,*K)
      }
}
*A=SELECT DATA_NAME , DATA_REPL_NUM, DATA_CHECKSUM WHERE
       DATA_NAME LIKE
*K=----------------------
```

The Rule prints out the query and the stdout (among others).

NOTE: ruleTest16.ir is similar, but generates the table using the acGetIcatResults Micro-service.

```
myTestRule {
      | acGetIcatResults(*Action,*Condition,*B)
       forEach  ( *B )  {
           msiPrintKeyValPair(stdout,*B)
           writeLine(stdout,*K)
      }
}
*Action=trim
*Condition= COLL_NAME = '/tempZone/home/rods'
*K=---------------------
```

9.4.10 Rule 28: ruleTest20.ir -- Tests Remote Execution of Micro-service Writes.

The Rule invokes the remoteExec to execute a given sequence of Micro-services remotely.

1. remoteExec - is a Micro-service that executes a Micro-service chain remotely on another iRODS Server. The first argument is the remote server's network id; the second argument is the sequence of Micro-services to be remotely executed; and the third argument is the recoveryBody for recovery from failures.

In this Rule, the original Rule is invoked on srbbrick14.sdsc.edu, which in turn calls remote executions at another server (andal.sdsc.edu) and remote calls to itself.

```
myTestRule  {
     writeLine(stdout,begin)
     writeLine(stdout,just write in srbbrick1)
     remoteExec(andal.sdsc.edu,null) {
            writeLine(stdout,remote write in andal)
     }
     remoteExec(andal.sdsc.edu,null) {
           writeLine(stdout,remote write again in andal)
     }
     remoteExec(srbbrick14.sdsc.edu,null) {
          writeLine(stdout,remote write in srbbrick1)
     }
     remoteExec(andal.sdsc.edu,null) {
          writeLine(stdout,remote write again and again in andal)
     }
     remoteExec(srbbrick14.sdsc.edu,null) {
           writeLine(stdout,again remote write in srbbrick1)
     }
     remoteExec(andal.sdsc.edu,null) {
           writeLine(stdout,remote write third in andal)
     }
     remoteExec(srbbrick14.sdsc.edu,null) {
           writeLine(stdout,second remote write in srbbrick1)
     }
     remoteExec(srbbrick14.sdsc.edu,null) {
            writeLine(stdout,third remote write in srbbrick1)
     }
    writeLine(stdout,again just write in srbbrick1)
    writeLine(stdout,end)
}
```

No input is needed
The stdout is printed.

9.4.11 Rule 29: ruleTest21.ir -- Tests Remote Execution of Delayed Writes.

The Rule invokes the remoteExec Micro-service to execute a given sequence of Micro-Services remotely.

1. remoteExec - is a Micro-service that executes a Micro-service chain remotely on another iRODS server. The first argument is the remote server's network id; the second argument is the sequence of Micro-services to be remotely executed; and the third argument is the recoveryBody for recovery from failures.

In this Rule the original Rule is invoked on srbbrick14.sdsc.edu and a line is printed to that effect; then, a remote execution is invoked on a server called andal.sdsc.edu, which after sleeping for 10 seconds and writing a message, calls a remote execution back on srbbrick14.sdsc.edu, which, after sleeping for 10 seconds and writing a message, returns back to andal.sdsc.edu. At andal.sdsc.edu the execution again calls for a remote execution at srbbrick14.sdsc.edu, which immediately executes a remote execution at andal.sdsc.edu, which sleeps and writes a message. Control then reverts back to srbbrick14.sdsc.edu, which in turn gives it to andal.sdsc.edu, which sleeps and writes one more message before returning back to the original invocation at srbbrick14.sdsc.edu, which prints an end message.

```
myTestRule  {
   writeLine(stdout,begin)
   remoteExec(andal.sdsc.edu,null) {
        msiSleep(10,0)
        writeLine(stdout,open remote write in andal)
         remoteExec(srbbrick14.sdsc.edu,null) {
                 msiSleep(10,0)
                writeLine(stdout,remote of a remote write in srbbrick1)
                }
        remoteExec(srbbrick14.sdsc.edu,null) {
                remoteExec(andal.sdsc.edu,null) {
                        msiSleep(10,0)
                        writeLine(stdout,remote of a remote of a remote write in andal)
                        }
                }
        msiSleep(10,0)
        writeLine(stdout,close remote write in andal)
        }
   writeLine(stdout,end)
    }
```

No input is needed
The stdout is printed.

9.5 Resource Selection Example

In the server/config/reConfigs/core.irb file there is a Rule called "acSetRescSchemeForCreate" which is used for setting the resource preferences. By default, this Rule is set to:

```
acSetRescSchemeForCreate||msiSetDefaultResc(demoResc,null)|nop
```

which basically sets 'demoResc' as a default resource if no resource is specified.
This Rule can be modified to randomly select a storage resource from a group of resources as follows:

```
acSetRescSchemeForCreate||msiSetDefaultResc(demoResc,null)##
            msiSetRescSortScheme(random)
           |nop##nop
```

(all of above in one line - no line breaks)

Adding the "msiSetRescSortScheme" to the Rule executes a random pick of one of the resource. If you want everyone to use your resource by force, you can change the first Micro-service in the Rule to:

```
msiSetDefaultResc(my_group,forced)
```

This will over-ride the resource given by the client.

The good thing abut Micro-services is that if you don't like the random sorting given by the default Micro-service, you can write your own and use that in the "acSetRescSchemeForCreate" Rule and achieve your goal.

As you can see there are no conditions being checked; "||" is used in the above Rule for the condition. You can add Rules to the core.irb file with different conditions (you need at least one catch all Rule, as the default in case no conditions are satisfired) which might use different resource sets and different selection criteria as you prefer with conditions based on collection basis or on user/group basis or both!!

This level of customization provides an in depth control of resource management that can make life easier or harder for the data manager

9.6 French National Library Rule Base

Three rules were constructed to control ingestion of documents, retrieving a file, and auditing properties of the digital library.

9.6.1 PUT Use Case

This Rule imports an input document into iRODS, adds import date and checksum as AVU (Attribute-Value-Unit triplet) metadata, and replicates it to other resources, the list of which should be stored as a comma-separated list as resource metadata, named replicaResources.

```
myiput||assign(*rodsPath,/$rodsZoneClient/home/$userNameClient/*rodsName)
##acObjPutWithDateAndChksumAsAVUs(*rodsPath,*mainResource,
        *localFilePath,*inputChecksum,*outstatus)
##acGetValueForResourceMetaAttribute(*replicasAttributeCondition,*replList)
##ifExec(*replList != none,
        delayExec(<PLUSET>1m</PLUSET>,
        forEachExec(*replList,writeLine(stdout,"replicating to *replList ...")
                ##msiDataObjRepl(*rodsPath,*replList,*replStatus),nop),
        nop),nop,nop,nop)
|nop##nop##nop

*localFilePath=$1%*mainResource=$2%*rodsName=$3%*inputChecksum=$4%*replicasAttrib
uteCondition= RESC_NAME = '*mainResource' AND META_RESC_ATTR_NAME =
'replicaResources'
RuleExecOut
```

Import document Rule. This Rule imports an input document into iRODS, adds import date and checksum as AVU metadata, and replicates it to other resources, the list of which should be stored as a comma-separated list:
    resource metadata, named replicaResources.

```
Input parameters :
        localFilePath    - String - Import file's current location (outside iRODS)
        mainResource     - String - iRODS resource name to write to for first put
                                    (probably a cache resource)
        rodsName         - String - iRODS content name (path will be derived from
                                    context for zone & username)
        inputChecksum - String - input checksum of imported content

Invocation example
        iRule -F sparPut.ir foo/titi3 oscresc 9c24fde7b0a0c6e5f3b5490cb9841597
```

9.6.2 GET use case

This Rule locates a copy of the record. If its physical checksum (computed externally using msiExecCmd and the OS's md5sum utility) corresponds to the stored checksum (AVU), said copy is returned. If it doesn't, asynchronous recovery (delete the replica, and copy a good one over it, haven't tried rsync microservices yet) is scheduled using delayExec, and the next copy is checked until a good one is located and staged onto a local directory.

```
sparGet||assign(*goodReplicaEncountered,0)
##assign(*rodsPath,/$rodsZoneClient/home/$userNameClient/*rodsName)
##acGetValueForDataObjMetaAttribute(*storedChecksumCondition,*objStoredChksum)
##acGetDataObjLocations(*locationsCondition,*matchingObjects)
##forEachExec(*matchingObjects,
        ifExec(*goodReplicaEncountered == 0,
        msiGetValByKey(*matchingObjects,RESC_LOC,
        *objReplicaHost)
##msiGetValByKey(*matchingObjects,DATA_PATH,*objPhysicalPath)
##msiGetValByKey(*matchingObjects,RESC_NAME,*currRescName)
##writeLine(stdout,"getting PHY CHK for *objReplicaHost ,*objPhysicalPath")
##remoteExec(*objReplicaHost,null,
                acGetPhysicalDataObjMD5SUM(*objPhysicalPath,
                        *objReplicaHost ,
                        *objPhysicalMD5),nop)
##writeLine(stdout,"Checksum of *rodsPath at *objReplicaHost on *currRescName
(*objPhysicalPath) is *objPhysicalMD5")
##ifExec(*objStoredChksum == *objPhysicalMD5,
        writeLine(stdout,"input and computed MD5 checksums match")
##assign(*goodReplicaEncountered,1)
##writeLine(stdout,"getting *rodsName to  *stagePath/*rodsName")
##msiDataObjGet(*rodsPath,*stagePath/*rodsName,*getStatus),
        writeLine(stdout,"if cond failed"),
        writeLine(stdout,"replace policy is : *replacePolicy ")
##acPolicyBasedReplicaReplacement(*rodsPath,*currRescName,*replacePolicy),nop),
nop,nop,nop),nop)
|nop##nop
*rodsName=$1%*stagePath=$2%*replacePolicy='lazy'%*locationsCondition=DATA_NAME =
'*rodsName'%*storedChecksumCondition=*locationsCondition AND
META_DATA_ATTR_NAME = 'MD5SUM'
RuleExecOut
```

Get data object. This Rule locates a copy of the record. If its physical checksum corresponds to the stored checksum, said copy is returned. If it doesn't, asynchronous recovery (delete the replica, and copy a good one over it) is scheduled, and the next copy is checked until a good one is located and returned.

Input parameters :

rodsName	- String - iRODS content name (path will be derived from context for zone & username)
stagePath	- String - FS directory where a clean copy will be transferred

Invocation example

```
        rm /tmp/stage/* ; iRule -F sparGet.ir titi2 /tmp/stage
```

9.6.3 AUDIT use case

Audit data object. This Rule locates all replicas of a data object, computes a physical checksum using system's md5sum, compares the result to the checksum stored in user metadata. All stale copies are trimmed (i.e. removed), and then replicated from another good copy. When all copies are audited and/or repaired, a clean copy will be staged onto a specified FS directory

That way, we are sure to get a good copy in return, and schedule reparation of the others either synchronously or asynchronously (for now, we need to have at least one good copy for this to work).

```
sparAudit||assign(*rodsPath,/$rodsZoneClient/home/$userNameClient/*rodsName)
##acGetValueForDataObjMetaAttribute(*storedChecksumCondition,*objStoredChksum)
##acGetDataObjLocations(*locationsCondition,*matchingObjects)
##forEachExec(*matchingObjects,msiGetValByKey(*matchingObjects,RESC_LOC,
        *objReplicaHost)
##msiGetValByKey(*matchingObjects,DATA_PATH,*objPhysicalPath)
##msiGetValByKey(*matchingObjects,RESC_NAME,*currRescName)
##remoteExec(*objReplicaHost,null,acGetPhysicalDataObjMD5SUM(*objPhysicalPath,
        *objReplicaHost ,*objPhysicalMD5),nop)
##writeLine(stdout,"Checksum of *rodsPath at *objReplicaHost on *currRescName
(*objPhysicalPath) is *objPhysicalMD5")
##ifExec(*objStoredChksum == *objPhysicalMD5,
        writeLine(stdout,"input and computed MD5 checksums match" ),
        writeLine(stdout,"if recov - actual comparison failed :("),
        writeLine(stdout,"replace policy is : *replacePolicy ")
##acPolicyBasedReplicaReplacement(*rodsPath,*currRescName,*replacePolicy),
        writeLine(stdout,"repair schedule placeholder - recovery")),nop)
##writeLine(stdout,"getting *rodsName to  *stagePath/*rodsName")
##msiDataObjGet(*rodsPath,*stagePath/*rodsName,*getStatus)|nop##nop
*rodsName=$1%*stagePath=$2%*replacePolicy='eager'%*locationsCondition=DATA_NAME
= '*rodsName'%*storedChecksumCondition=*locationsCondition AND
META_DATA_ATTR_NAME = 'MD5SUM'
RuleExecOut
```

Audit data object. This Rule locates all replicas of a data object, computes a physical checksum using system's md5sum, compares the result to the checksum stored in user metadata. All stale copies are trimmed (i.e. removed), and then replicated from another good copy. When all copies are audited and/or repaired, a clean copy will be staged onto a specified FS directory

resource metadata, named replicaResources.

Input parameters :

rodsName	- String - iRODS content name (path will be derived from context for zone & username)
stagePath	- String - FS directory where a clean copy will be transferred

Invocation example

```
	rm /tmp/stage/titi3 ; iRule -F sparAudit.ir titi3 /tmp/stage
```

9.6.4 Utilities

The Rules above rely on a "utility" spar.irb Rule definitions:

This file is a library of utility Micro-services. Each one is a combination of existing Micro-services. These utilities are useful to increase code reuse, and reduce the size of the actual Rules using them. It must be linked or copied into the server/config/reConfigs directory, and referenced in the server/config/server.config file, like this :

```
	reRuleSet   core,spar
```

acAddMetadataFromString : Adds metadata to an iRODS Object, from a keyval string (i.e. : KEY=VALUE)

Input parameters :

rodsPath	- String - iRODS content path
KVString	- String - keyval string
objType	- String - object type, can be -d for data object, -R for resource, -C for collection, or -u for user

```
acAddMetadataFromString(*rodsPath,*KVString,*objType)||
	msiString2KeyValPair(*KVString,*KVPair)
	##msiAssociateKeyValuePairsToObj(*KVPair,*rodsPath,*objType)
```

acObjPutWithDateAndChksumAsAVUs : imports (puts) an object into the iRODS repository, computes MD5 checksum and validates it against the supplied one. Once validated, adds MD5SUM and import date as metadata. If invalid, content is removed from iRODS.

Input parameters :

rodsPath	- String - iRODS content path
resource	- String - resource in which content must be added
localFilePath	- String - input file path on filesystem
inputChecksum	- String - input checksum, for now MD5

```
acObjPutWithDateAndChksumAsAVUs(*rodsPath,*resource,*localFilePath,
	*inputChecksum, *outstatus)||
msiDataObjPut(*rodsPath,*resource,*localFilePath,*outstatus)
##msiDataObjChksum(*rodsPath,null,*objChecksum)
##writeLine(stdout,"Input Checksum is *inputChecksum")
##writeLine(stdout,"Computed Checksum is  *objChecksum")
##ifExec(*objChecksum == *inputChecksum,
	writeLine(stdout,input and computed checksums match )
##msiGetSystemTime(*humanDate,human)
##acAddMetadataFromString(*rodsPath,MD5SUM=*objChecksum,-d)
##acAddMetadataFromString(*rodsPath,importDate=*humanDate,-d),
        writeLine(stdout,"if effed up"),
        writeLine(stdout,"integrity failure:checksums do not match. removing content...")
##msiDataObjUnlink(*rodsPath,*deleteStatus)
##writeLine(stdout,"content cleaned up"),
        writeLine(stdout,"could not delete content after integrity failure"))|nop
```

acGetValueForObjectAttribute : returns the value of an iRODS object metadata attribute

Input parameters :

rodsAttribute	- String - attribute which value should be retrieved (@see /lib/core/include/rodsGenQueryNames.h)
attributeCondition	- String - semi-SQL WHERE statement (such as RESC_NAME = foo AND META_RESC_ATTR_NAME = 'bar'}

Output parameters :

attributeValue	- String - the attribute value

```
acGetValueForObjectAttribute(*rodsAttribute,*attributeCondition,*attributeValue)||
msiMakeQuery(*rodsAttribute,*attributeCondition,*attributeQuery)
##msiExecStrCondQuery(*attributeQuery,*queryResults)
##forEachExec(*queryResults,
        msiGetValByKey(*queryResults,*rodsAttribute,*attributeValue),nop)
```

acGetValueForResourceMetaAttribute : wrapper around acGetValueForObjectAttribute prepared to retrieve a metadata attribute for a resource

Input parameters :

attributeCondition	- String - semi-SQL WHERE statement (such as RESC_NAME = foo AND META_RESC_ATTR_NAME = 'bar'

Output parameters :

attributeValue	- String - the attribute value

```
acGetValueForResourceMetaAttribute(*attributeCondition,*attributeValue)||
acGetValueForObjectAttribute("META_RESC_ATTR_VALUE",
        *attributeCondition,*attributeValue)
```

acGetValueForDataObjMetaAttribute : wrapper around acGetValueForObjectAttribute prepared to retrieve

Input parameters :

attributeCondition	- String - semi-SQL WHERE statement (such as RESC_NAME = foo AND META_RESC_ATTR_NAME = 'bar'

Output parameters :

attributeValue	- String - the attribute value

```
acGetValueForDataObjMetaAttribute(*attributeCondition,*attributeValue)||
acGetValueForObjectAttribute("META_DATA_ATTR_VALUE",
        *attributeCondition,*attributeValue)
```

acGetResourceZoneName : get the input resource's belonging zone name
Input parameters :
    resourceCondition   - String - semi-SQL WHERE statement to identify the resource (such as RESC_NAME = 'foo')
Output parameters :
    zoneName   - String - the zone name

```
acGetResourceZoneName(*resourceCondition,*zoneName)||
acGetValueForObjectAttribute("RESC_ZONE_NAME",*resourceCondition,*zoneName)
```

acGetDataObjLocations : get ICAT results regarding location info for a record (name, physical path,collection, resource, host)
Input parameters :
    locCondition   - String - semi-SQL WHERE statement to identify the object (such as DATA_NAME = 'foo'). Note : the object name (filename), not the full iRODS path must be supplied.

Output parameters :
    locationsResult   - String - ICAT packaged query results

```
acGetDataObjLocations(*locCondition,*locationsResult)||
msiMakeQuery("DATA_REPL_NUM,DATA_NAME,DATA_PATH,
             COLL_NAME,RESC_NAME,RESC_LOC",
             *locCondition,*locationsQuery)
##msiExecStrCondQuery(*locationsQuery, *locationsResult)|nop##nop
```

acGetPhysicalDataObjMD5SUM : executes the OS's (g)md5sum utility on the physical content and returns the MD5 checksum
Input parameters :
    physicalPath   - String - filesystem path where the object is stored
Output parameters :
    physicalMD5   - String - the physical MD5 computed by the OS

```
acGetPhysicalDataObjMD5SUM(*physicalPath,*rodsHost,*physicalMD5)||
msiExecCmd(rodsMD5sum,*physicalPath,*rodsHost,null,null,*physicalMD5)|nop
```

acGetRandomString : executes rndString script and returns a pseudo random string of specified length using /dev/urandom.
Input parameters :
    strLength   - String - desired string length
Output parameters :
    string   - String - the random string

```
acGetRandomString(*strLength,*string)||
msiExecCmd(rndString,*strLength,*rodsHost,null,null,*string)|nop
```

acReplaceStaleReplica : trims (delete) a stale replica (wrong checksum encountered), and replicates over it from another fresh copy

Input parameters :

rodsPath	- String - iRODS content path
rescName	- String - resource in which content must be added

```
acReplaceStaleReplica(*rodsPath,*rescName,*replStatus)||
msiDataObjTrim(*rodsPath,*rescName,null,1,null,*trimStatus)
##msiDataObjRepl(*rodsPath,*rescName,*replStatus)|nop
```

acPolicyBasedReplicaReplacement : stale replica replacement can be either eager , or not (lazy)

- eager means the replacement will be done synchronously (immediately),
- lazy means the replacement will be done asynchronously (delayed , 1 minute by default)

Input parameters :

rodsPath	- String - iRODS content path
rescName	- String - resource in which content must be added
policy	- String - eager or lazy, see description above

```
acPolicyBasedReplicaReplacement(*rodsPath,*resc,*policy)||
ifExec(*policy == 'eager',
        acReplaceStaleReplica(*rodsPath,*resc,*replStatus),nop,
        delayExec(<PLUSET>1m</PLUSET>,
        acReplaceStaleReplica(*rodsPath,*resc,*replStatus),nop),nop)
```

9.6.5 External Scripts

Some of the utility Micro-services above rely on external scripts which are invoked using the msiExecCmd. For example, the acGetPhysicalDataObjMD5SUM Micro-service computes the replica's physical MD5SUM using the OS's GNU md5sum utility.

* referenced rodsMD5SUM script :

```
#!/bin/bash
# computes MD5 sum of given file.
# $1 : file path
#
E_BADARGS=65
E_NOFILE=66
UNAME="`uname -s`"
if [ $# -ne 1 ]
then
  exit $E_BADARGS
fi
if [ -f $1 ]; then
    if [ "${UNAME}" = "Linux" ]; then
        echo -n `md5sum $1 | sed -e 's/ .*//g'`
    elif [ "${UNAME}" = "SunOS" ]; then
        echo -n `gmd5sum $1 | gsed -e 's/ .*//g'`
    fi
    exit 0
else
    exit $E_NOFILE
fi
```

Appendix A. iRODS shell commands

The i-commands available in release 2.0 of iRODS are listed below, organized by type. The few parameters the i-commands need to operate (for connection to a server) can be set as user environment variables, or specified on the command line. There is a common set of command line options for the i-commands, so that each option (-a, -b, -c, etc) will mean the same thing (generally) in all of them. 'iinit' writes an automatic login file (scrambled password) for you in any window on your computer (actually, any computer with your same home directory), otherwise i-commands will prompt for your password. The options available for an i-command can be found by using the help option "-h". For example:

```
zuri% iinit –h
Creates a file containing your iRODS password in a scrambled form,
to be used automatically by the icommands.
Usage: iinit [-ehvVl]
 -e  echo the password as you enter it (normally there is no echo)
 -l  list the iRODS environment variables (only)
 -v  verbose
 -V  Very verbose
 -h  this help
```

Environment Variables (example values are shown for each variable)

irodsHost=localhost	The IP address of the metadata catalog server (iCAT).
irodsPort=1247	The port number used by the metadata catalog.
irodsDefResource=MzResc	The logical name of the default storage resource.
irodsHome=/Mzone/home/Mzrods	The user home collection within the Data Grid.
irodsCwd=/Mzone/home/Mzrods	The current working collection within the Data Grid.
irodsUserName=Mzrods	The user name known by the Data Grid.
irodsZone=Mzone	The name of the Data Grid.

User and File Manipulation i-commands

• iinit	Initialize - Store your password in a scrambled form for automatic use by other i-commands.
• iput	Store a file
• iget	Get a file
• imkdir	Like mkdir, make an iRODS collection (similar to a directory or Windows folder)
• ichmod	Like chmod, allow (or later restrict) access to your data objects by other users.
• icp	Like cp or rcp, copy an iRODS data object
• irm	Like rm, remove an iRODS data object
• ils	Like ls, list iRODS data objects (files) and collections (directories)
• ipwd	Like pwd, print the iRODS current working directory
• icd	Like cd, change the iRODS current working directory
• irepl	Replicate data objects.
• iexit	Logout (use 'iexit full' to remove your scrambled password from the disk)
• ipasswd	Change your irods password.
• ichksum	Checksum one or more data-object or collection from iRODS space.
• imv	Moves/renames an irods data-object or collection.
• iphymv	Physically move files in iRODS to another storage resource.
• ireg	Register a file or a directory of files and subdirectory into iRODS.
• irmtrash	Remove one or more data-object or collection from a RODS trash bin.

- irsync Synchronize the data between a local copy and the copy stored in iRODS or between two iRODS copies.
- itrim Trim down the number of replica of a file in iRODS by deleting some replicas.
- iexecmd Remotely Execute (fork and exec) a command on the server.
- imcoll Manage (mount, unmount, synchronize and purge of cache) a mounted iRODS collection and the associated cache.
- ibun Upload and download structured (e.g. tar) files.

Metadata i-commands
- imeta Add, remove, list, or query user-defined Attribute-Value-Unit triplets metadata
- isysmeta Show or modify system metadata
- iquest Query (pose a question to) the ICAT, via a SQL-like interface

Informational i-commands
- ienv Show current iRODS environment
- ilsresc List resources
- iuserinfo List users
- imiscsvrinfo Get basic server information; test communication.

Administration i-commands
- iadmin Administration commands: add/remove/modify users, resources, etc.

Rules and Delayed Rule Execution i-commands
- iRule Submit a user defined Rule to be executed by an irods server.
- iqstat Show pending iRODS Rule executions.
- iqdel Removes delayed Rules from the queue.
- iqmod Modifies delayed Rules in the queue.

As an example of the icommands, we list the help package for the iquest command. For each icommand, invoking the –h parameter will display the input parameters and provide usage examples.

Usage : iquest [[hint] format] selectConditionString

format is C format restricted to character strings. selectConditionString is of the form: SELECT <attribute> [, <attribute>]* [WHERE <condition> [AND <condition>]*] attribute can be found using iattrs command condition is of the form: <attribute> <rel-op> <value> rel-op is a relational operator: eg. =, <>, >,<, like, not like, between, etc., value is either a constant or a wild-carded expression. One can also use a few aggregation operatos such as sum,count,min,max and avg. Use % and _ as wild-cards, and use \ to escape them Options are:

-h this help

Examples:

iquest "SELECT DATA_NAME, DATA_CHECKSUM WHERE DATA_RESC_NAME like 'demo%'"

iquest "For %-12.12s size is %s" "SELECT DATA_NAME , DATA_SIZE WHERE COLL_NAME = '/tempZone/home/rods'"

```
iquest "SELECT COLL_NAME WHERE COLL_NAME like '/tempZone/home/%'"

iquest "User %-6.6s has %-5.5s access to file %s" "SELECT USER_NAME,
DATA_ACCESS_NAME, DATA_NAME WHERE COLL_NAME = '/tempZone/home/rods'"

iquest " %-5.5s access has been given to user %-6.6s for the file %s" "SELECT
DATA_ACCESS_NAME, USER_NAME, DATA_NAME WHERE COLL_NAME =
'/tempZone/home/rods'"

iquest "SELECT RESC_NAME, RESC_LOC, RESC_VAULT_PATH, DATA_PATH WHERE
DATA_NAME = 't2' AND COLL_NAME = '/tempZone/home/rods'"

iquest "User %-9.9s uses %14.14s bytes in %8.8s files in '%s'" "SELECT USER_NAME,
sum(DATA_SIZE),count(DATA_NAME),RESC_NAME"

iquest "select sum(DATA_SIZE) where COLL_NAME = '/tempZone/home/rods'"

iquest "select sum(DATA_SIZE) where COLL_NAME like '/tempZone/home/rods%'"

iquest "select sum(DATA_SIZE), RESC_NAME where COLL_NAME like
'/tempZone/home/rods%'"
```

Appendix B. iRODS Session Variable Mapping

The Data Variable Mapping defined in the core.dvm file located in the server/config/reConfig directory provides a mapping from an external variable name (logical) to an internal variable name in the Session Memory $. Each mapping consists of three parts separated by "|" symbol:

external variable name, action-list, and internal variable name

The action-list can be empty. If not, then the specified mapping is used when an action that invokes this mapping is in the list. The mappings are searched top-down in a file. We list the contents of the core.dvm file in Table 1. The variable names that can be used for conditions and as input parameters on Micro-services are listed in column 1. Note that one of the $ variables (rescName) is used by a specific action (resc_modify) and is stored in a separate structure within the rei memory structure. The meaning of most of the variables is transparent. More detailed explanations are given in the code using the variables.

Table 1. Mapping of External $ variables to Internal $ variable names in the REI structure

External $ variable	Action	Internal $ variable in REI structure
otherUser		rei->uoio->user
otherUserName		rei->uoio->userName
otherUserZone		rei->uoio->rodsZone
otherUserType		rei->uoio->userType
otherSysUidClient		rei->uoio->sysUid
rescName	resc_modify	rei->rgi->rescInfo->rescName
objPath		rei->doi->objPath
rescName		rei->doi->rescName
destRescName		rei->doi->destRescName
backupRescName		rei->doi->backupRescName
dataType		rei->doi->dataType
dataSize		rei->doi->dataSize
chksum		rei->doi->chksum
version		rei->doi->version
filePath		rei->doi->filePath
replNum		rei->doi->replNum
replStatus		rei->doi->replStatus
dataOwner		rei->doi->dataOwnerName
dataOwnerZone		rei->doi->dataOwnerZone
dataExpiry		rei->doi-dataExpiry
dataComments		rei->doi->dataComments
dataCreate		rei->doi-dataCreate
dataModify		rei->doi-dataModify
dataAccess		rei->doi->dataAccess
dataAccessInx		rei->doi->dataAccessInx
dataId		rei->doi->dataId
collId		rei->doi->collId
rescGroupName		rei->doi->rescGroupName
statusString		rei->doi->statusString
dataMapId		rei->doi->dataMapId

userClient		rei->uoic
userNameClient		rei->uoic->userName
rodsZoneClient		rei->uoic->rodsZone
userTypeClient		rei->uoic->userType
sysUidClient		rei->uoic->sysUid
hostClient		rei->uoic->authInfo->host
authStrClient		rei->uoic->authInfo->authStr
userAuthSchemeClient		rei->uoic->authInfo->authScheme
userInfoClient		rei->uoic->userOtherInfo->userInfo
userCommentClient		rei->uoic->userOtherInfo->userComments
userCreateClient		rei->uoic-userOtherInfo->userCreate
userModifyClient		rei->uoic-userOtherInfo->userModify
userProxy		rei->uoip
userNameProxy		rei->uoip->userName
rodsZoneProxy		rei->uoip->rodsZone
userTypeProxy		rei->uoip->userType
sysUidProxy		rei->uoip->sysUid
hostProxy		rei->uoip->authInfo->host
authStrProxy		rei->uoip->authInfo->authStr
userAuthSchemeProxy		rei->uoip->authInfo->authScheme
userInfoProxy		rei->uoip->userOtherInfo->userInfo
userCommentProxy		rei->uoip->userOtherInfo->userComments
userCreateProxy		rei->uoip->userOtherInfo->userCreate
userModifyProxy		rei->uoip->userOtherInfo->userModify
collName		rei->coi->collName
collParentName		rei->coi->collParentName
collOwnername		rei->coi->collOwnerName
collExpiry		rei->coi-collExpiry
collComments		rei->coi->collComments
collCreate		rei->coi-collCreate
collModify		rei->coi-collModify
collAccess		rei->coi->collAccess
collAccessInx		rei->coi->collAccessInx
collMapId		rei->coi->collMapId
collInheritance		rei->coi->collInheritance
zoneName		rei->rgi->rescInfo->zoneName
rescLoc		rei->rgi->rescInfo->rescLoc
rescType		rei->rgi->rescInfo->rescType
rescTypeInx		rei->rgi->rescInfo->rescTypeInx
rescClass		rei->rgi->rescInfo->rescClass
rescClassInx		rei->rgi->rescInfo->rescClassInx
rescVaultPath		rei->rgi->rescInfo->rescVaultPath
numOpenPorts		rei->rgi->rescInfo->numOpenPorts
paraOpr		rei->rgi->rescInfo->paraOpr
rescId		rei->rgi->rescInfo->rescId
gateWayAddr		rei->rgi->rescInfo->gateWayAddr
rescMaxObjSize		rei->rgi->rescInfo->rescMaxObjSize

freeSpace		rei->rgi->rescInfo->freeSpace
freeSpaceTime		rei->rgi->rescInfo->freeSpaceTime
freeSpaceTimeStamp		rei->rgi->rescInfo->freeSpaceTimeStamp
rescInfo		rei->rgi->rescInfo->rescInfo
rescComments		rei->rgi->rescInfo->rescComments
rescCreate		rei->rgi->rescInfo-rescCreate
rescModify		rei->rgi->rescInfo-rescModify
connectCnt		rei->rsComm->connectCnt
connectSock		rei->rsComm->sock
connectOption		rei->rsComm->option
connectStatus		rei->rsComm->status
connectApiTnx		rei->rsComm->apiInx
connectWindowSize		rei->rsComm->windowSize
connectReconnFlag		rei->rsComm->reconnFlag
connectReconnSock		rei->rsComm->reconnSock
connectReconnPort		rei->rsComm->reconnPort
connectReconnAddr		rei->rsComm->reconnAddr
ConnectCookie		rei->rsComm->cookie

Append C: iRODS Micro-services

The Micro-services are organized into categories related to function:

- Administrative tasks
- Workflow controls
- Low-level Data Object manipulation (Posix style operations)
- Data Object manipulation tasks
- Collection manipulation tasks
- Proxy command tasks
- iCAT system services
- iCAT manipulation tasks
- Rule-oriented Database Access (RDA) tasks
- XMessaging system tasks
- E-mail tasks
- Metadata manipulation tasks
- User tasks
- System tasks
- ERA tasks (Electronic Records Archive)
- XML tasks
- HDF5 tasks
- Property manipulation tasks
- Web services
- BNL tasks (French National Library)

For each category, the corresponding Micro-services are listed, along with their input parameters, output parameters, and status information. The input and output parameters have specific required data types. Parameters passed between Micro-services, information sent over the network between client and server, information sent over the network between servers, and information stored in the msParam structure in memory are all typed. The instructions for packing the parameters (serializing into a bit stream) can be found in the file lib/core/include/rodsPackTable.h.

Administration Micro-services	Meaning / Input variable type	Input variables used	Output variables
msiAdmChangeCoreIRB	**change the core.irb file (can be invoked through iRule)**		
		Requires iRODS administration privilege	
	STR_MS_T	newFileNameParam - new core file name without the .irb extension	retval - 0 on success
msiAdmAppendToTopOfCoreIRB	**prepend another irb file to the core.irb file**		
		Requires iRODS administration privilege	
	STR_MS_T	newFileNameParam - prepended core file name without the .irb extension	retval - 0 on success
msiAdmAddAppRuleStruct	**add application level IRB Rules and DVM and FNM mappings to the Rule engine.**		
		Requires iRODS administration privilege	
	STR_MS_T	irbFilesParam - application Rules file name without the .irb extension	
	STR_MS_T	application $-variable file name mapping without the .dvm extension	
	STR_MS_T	fnmFilesParam - pplication microService mapping file name without the .fnm extension	retval - 0 on success
msiAdmClearAppRuleStruct	**clear application level IRB Rules and DVM and FNM mappings that were loaded into the Rule engine.**		
		Requires iRODS administration privilege	retval - 0 on success
msiAdmShowIRB	**display the currently loaded Rules**		
			retval - 0 on success
			rei->MsParamArray->MsParam->RuleExecOut->stdout
msiAdmShowDVM	**display the currently loaded variable name mappings**		
			retval - 0 on success
			rei->MsParamArray->MsParam->RuleExecOut->stdout
msiAdmShowFNM	**display the currently loaded microServices/Actions namemappings**		
			status - 0 on success
			rei->MsParamArray->MsParam->RuleExecOut->stdout

Workflow Micro-services			
nop, null	**no action**		
			retval - 0 on success
cut	**not to retry any other applicable Rules for this action**		
			retval - 0 on success
succeed	**exit with success immediately**		
			retval - 0 on success
fail	**fail immediately, recovery and retries are possible**		
			retval - 0 on success
msiGoodFailure	**useful when you want to fail but have no recovery initiated.**		
			retval - 0 on success
msiSleep	**sleep**		
		sec - seconds to sleep	
		microSec - micro-seconds to sleep	retval - 0 on success
whileExec	**while loop**		
	STR_MS_T	condition - logical expression (true or false)	
	STR_MS_T	whileBody - Micro-service/Rule list separated by ##	
	STR_MS_T	recoveryWhileBody - Micro-service/Rule list separated by ##	retval - 0 on success
forExec	**for loop with initial, step and end condition**		
	STR_MS_T	initial - initial assignment for loop variable condition - logical expression	
	STR_MS_T	step - increment of loop variable	
	STR_MS_T	forBody - Micro-service/Rule list separated by ##	
	STR_MS_T	recoveryForBody - Micro-service/Rule list	retval - 0 on success
forEachExec	**for loop iterating over a row of tables or a list**		
	STR_MS_T or StrArray_MS_T or IntArray_MS_T or GenQueryOut_MS_T	inlist - either comma separated string or array of strings or array of integers or iCAT query result	
	STR_MS_T	body - Micro-service/Rule list separated by ##	
	STR_MS_T	recoveryBody - Micro-service/Rule list	retval - 0 on success
break	**breaks out of while, for and forEach loops**		

			status - BREAK_ACTION_ENCOUNTERED_ERR
writeString	**write a string to stdout buffer**		
	STR_MS_T	where - buffer name (stdout or stderr)	rei->MsParamArray->MsParam-> RuleExecOut
	STR_MS_T	inString - string to write to buffer	retval - 0 on success
writeLine	**write a line (with end of line) to stdout buffer**		
	STR_MS_T	where - buffer name (stdout or stderr)	rei->MsParamArray->MsParam-> RuleExecOut
	STR_MS_T	inString - string to write to buffer	retval - 0 on success
assign	**assign a value to a parameter**		
	STR_MS_T	variable - msParam name or a $variable	
	STR_MS_T	value - expression to be computed	retval - 0 on success
ifExec	**if then else conditional branch**		
	STR_MS_T	condition: logical expression (true or false)	
	STR_MS_T	then - Micro-service/Rule list separated by ##	
	STR_MS_T	recoveryThen - Micro-service/Rule list	
	STR_MS_T	else - Micro-service/Rule list separated by ##	
	STR_MS_T	recoveryElse - Micro-service/Rule list separated by ##	retval - 0 on success
delayExec	**delay an execution of Micro-services or Rules**		
	STR_MS_T	delayCondition: condition for when to execute	
	STR_MS_T	body - Micro-service/Rule list separated by ##	
	STR_MS_T	recoveryBody - Micro-service/Rule list separted by ##	retval - 0 on success
remoteExec	**remote execution of Micro-services or Rules**		
	STR_MS_T	host - name of the server where the body is executed	
	STR_MS_T	delayCondition: condition for when to execute	
	STR_MS_T	body - Micro-service/Rule list separated by ##	
	STR_MS_T	recoveryBody - Micro-service/Rule list separted by ##	retval - 0 on success
applyAllRules	**apply all applicable Rules when executing a given Rule**		
	STR_MS_T	actionParam - the name of the action to execute	
	STR_MS_T	reiSaveFlagParam - 0 don't save rei, 1 save rei structure at every Rule invocation	

	STR_MS_T	allRuleExecFlagParam - 0 apply only to the actionParam invocationn, 1 apply recursively at all levels of invocation for every Rule inside the execution	retval - 0 on success

Data Object Low-level Micro-services	Can be called by client through iRule		
msiDataObjCreate	**create a data object**		
	DataObjInp_MS_T or STR_MS_T	dataObjName - path name of data object	retval - positive on success
	STR_MS_T	rsrcName - option resource name	irodsObjDesc [INT_MS_T] - descriptor index for the created object
msiDataObjOpen	**open a data object**		
			retval - positive on success
	DataObjInp_MS_T or STR_MS_T	dataObjName - path name of data object	irodsObjDesc [INT_MS_T] - descriptor index for the opened object
msiDataObjClose	**close an opened data object**		
			retval - 0 on success
	INT_MS_T or STR_MS_T	irodsObjDesc - descriptor index for the opened object	status [INT_MS_T] - positive on success
msiDataObjLseek	**lseek to a position within a data object**		
	DataObjLseekInp_MS_T or INT_MS_T or STR_MS_T	irodsObjDesc - descriptor index for the opened object	
	DOUBLE_MS_T or STR_MS_T	offset - byte offset for the seek	retval - positive on success
	INT_MS_T or STR_MS_T	whence - location of seek (SEEK_SET, SEEK_CUR, SEEK_END)	status [DOUBLE_MS_T or DataObjLseekOut_MS_T] - operation return status
msiDataObjRead	**read an opened data object**		

	DataObjReadInp_MS_T or INT_MS_T or STR_MS_T	irodsObjDesc - descriptor index for the opened object	retval - positive on success
	INT_MS_T or STR_MS_T	length - length of buffer to read, optional if use DataObjReadInp_MS_T	buffer [BUF_LEN_MS_T] - bytes read
msiDataObjWrite	**write a data object**		
	DataObjWriteInp_MS_T or INT_MS_T or STR_MS_T	irodsObjDesc - descriptor index for the opened object	retval - positive on success
	INT_MS_T or STR_MS_T	length - length of buffer to write, optional if use DataObjWriteInp_MS_T	status [INT_MS_T] - bytes written

Data Object Micro-services	Can be called by client through iRule		
msiDataObjUnlink	**delete**		
			retval - 0 on success
	DataObjInp_MS_T or STR_MS_T	dataObjName - path name of data object	status [INT_MS_T] - status of the operation
msiDataObjRepl	**replicate**		
	DataObjInp_MS_T or STR_MS_T	dataObjName - path name of data object	retval - 0 on success
	STR_MS_T	rsrcName - option resource name	status [INT_MS_T] - status of the operation
msiDataObjCopy	**copy**		
	DataObjInp_MS_T or STR_MS_T	dataObjName - path name ofsource data object	
	DataObjInp_MS_T or STR_MS_T	dataObjName - path name ofdestination data object	retval - 0 on success
	STR_MS_T	rsrcName - option resource name	status [INT_MS_T] - status of the operation
msiDataObjGet	**get**		
		Use only with iRule. Do not use with delayExec	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	retval - 0 on success

	STR_MS_T	inpParam2 - optional client's local file path	status [INT_MS_T] - status of the operation
msiDataObjPut	**put**		
		Use only with iRule. Do not use with delayExec	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	
	STR_MS_T	inpParam2 - optional resource	retval - 0 on success
	STR_MS_T	inpParam3 - optional client's local file path	status [INT_MS_T] - status of the operation
msiDataObjPutWithOptions	**put with options**		
		Use only from a client	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	
	STR_MS_T	inpParam2 - optional resource	
	STR_MS_T	inpParam3 - optional client's local file path	
	STR_MS_T	inpOverwriteParam - optional to overwrite content that already exists	retval - 0 on success
	STR_MS_T	inpAllCopiesParam - optional to force overwrite on all existing copies	status [INT_MS_T] - status of the operation
msiDataObjChksum	**checksum a data object**		
		Use only with iRule.	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	retval - 0 on success
	STR_MS_T	inpParam2 - optional flag (chksumAll, verifyChksum, forceChksum)	checksum [STR_MS_T]
msiDataObjPhymv	**move a data object from one resource to another**		
		Use onlywith iRule	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	
	STR_MS_T	inpParam2 - optional destination resourceName	
	STR_MS_T	inpParam3 - optional source resourceName	
	STR_MS_T	inpParam4 - optional replNum	retval - 0 on success
	STR_MS_T	inpParam5 - optional to specify IRODS_ADMIN_KW	status [INT_MS_T] - status of the operation
msiDataObjRename	**rename a data object**		
		Use only with iRule.	

	DataObjInp_MS_T or STR_MS_T	inpParam1 - source dataObj path	
	DataObjInp_MS_T or STR_MS_T	inpParam2 - optional destination object Path	retval - 0 on success
	INT_MS_T or STR_MS_T	inpParam3 - optional data type (=0 means data object, >0 means collection)	status [INT_MS_T] - status of the operation
msiDataObjTrim	**trim the number of replicas**		
		Use only with iRule.	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	
	STR_MS_T	inpParam2 - optional resourceName	
	STR_MS_T	inpParam3 - optional replNum	
	STR_MS_T	inpParam4 - optional minimum number of copies to keep	retval - 0 on success
	STR_MS_T	inpParam5 - optional to specify IRODS_ADMIN_KW	status [INT_MS_T] - status of the operation
msiPhyPathReg	**register a physical file into iRods**		
		Use only with iRule.	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	
	STR_MS_T	inpParam2 - optional destination resourceName	
	STR_MS_T	inpParam3 - optional physical path to be registered	retval - 0 on success
	STR_MS_T	inpParam4 - optional to specify COLLECTION_KW to indicate a directory is being registered	status [INT_MS_T] - status of the operation
msiObjStat	**stat an object to get its properties**		
		Use only with iRule.	retval - 0 on success
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	value [INT_MS_T] - COLL_OBJ_T or DATA_OBJ_T
msiDataObjRsync	**synchronize a data between iRods and local file**		
		Use only with iRule.	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	
	STR_MS_T	inpParam2 - optional rsync mode (IRODS_TO_LOCAL, LOCAL_TO_IRODS, IRODS_TO_IRODS)	

	STR_MS_T	inpParam3 - optional chksum value (RSYNC_CHKSUM_KW)	retval - 0 on success
	STR_MS_T	inpParam4 - optional to specify the destination path for the IRODS_TO_IRODS mode (RSYNC_DEST_PATH_KW)	status [INT_MS_T] - status of the operation
msiGetObjType	**find out if a given value is a data object, collection, resource, ...**		
			retval - 0 on success
		objParam - name of object	typeParam [STR_MS_T] - type of object
			USER_PARAM_TYP_ERROR if input parameter type doesn't match
msiCheckPermission	check authorization permission		
msiCheckOwner	check owner		

Collection Micro-services			
msiCollCreate	**create a collection**		
		Use only with iRule.	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	retval - 0 on success
	STR_MS_T	inpParam2 - flags integer (=1 means the parent collections will be created too)	status [INT_MS_T] - status of the operation
msiCollRepl	**replicate all files in a collection**		
	CollInp_MS_T or STR_MS_T	collection - used as the irods path of the collection to replicate	retval - 0 on success
	STR_MS_T	targetResc - resource where the replicated objects are stored	status [collOprStat_t] - operation status
msiRmColl	**delete a collection**		
		Use only with iRule.	
	DataObjInp_MS_T or STR_MS_T	inpParam1 - dataObj path	retval - 0 on success
	STR_MS_T	inpParam2 - optional forceFlag (irodsAdminRmTrash, irodsRmTrash)	status [INT_MS_T] - status of the operation

Proxy Command Micro-services			
msiExecCmd	**remotely execute a command**		
		Use only from a client	
	ExecCmd_MS_T or STR_MS_T	inpParam1 - specify the command to execute	
	STR_MS_T t	inpParam2 - optional o specify command arguements (cmArgv)	
	STR_MS_T	inpParam3 - optional STR_MS_T for the host address where the command is executed (execAddr)	
	STR_MS_T	inpParam4 - optional STR_MS_T to specify an iRODS file path (hintPath) where the file is stored	retval - 0 on success
	INT_MS_T or STR_MS_T	inpParam5 - optional to specify the resolved physical path from the hintPath	status [ExecCmdOut_MS_T] - status of command execution and stdout/sterr output

iCAT System Services:	
msiVacuum	**Postgres vacumm, done periodically to optimize indices and performance**
	retval - 0 on success
iCAT Micro-services	
msiCommit	commit the database transaction
msiRollback	roll back the database transaction
msiCreateUser	create a new user
msiDeleteUser	delete a user
msiAddUserToGroup	add a user to a group
msiCreateCollByAdmin	create a collection by administrator
msiDeleteCollByAdmin	delete a collection by administrator
msiRenameLocalZone	rename the local zone by updating tables
msiRenameCollection	rename a collection; used via a Rule with the above msiRenameLocalZone
msiExecStrCondQuery	execute a conditional query
msiExecGenQuery	execute a general query
msiMakeQuery	make a query

Rule-oriented Database Access Micro-services			
msiRdaToStdout	**call new RDA functions to interface to an arbitrary database returning results in standard-out**		
	STR_MS_T	inpRdaName - string with the name of the remote database being accessed	
	STR_MS_T	inpSQL - string, the SQL to use	
	STR_MS_T	inpParam1 - optional bind variable for the SQL	
	STR_MS_T	inpParam2 - optional bind variable for the SQL	
	STR_MS_T	inpParam3 - optional bind variable for the SQL	rei->MsParamArray->MsParam->RuleExecOut->stdout
	STR_MS_T	inpParam4 - optional bind variable for the SQL	retval - 0 on success
msiRdaToDataObj	**As above but store results in an iRods DataObject.**		
	STR_MS_T	inpRdaName - string with the name of the remote database being accessed	
	STR_MS_T	inpSQL - string, the SQL to use	
	STR_MS_T	inpParam2 - optional bind variable for the SQL	
	STR_MS_T	inpParam3 - optional bind variable for the SQL	
	STR_MS_T	inpParam4 - optional bind variable for the SQL	
	STR_MS_T	inpParam5 - optional bind variable for the SQL	
		inpOutObj - descriptor for writing results	retval - 0 on success
msiRdaNoResults	**As above, perform a SQL operation but without storing the resulting output.**		
	STR_MS_T	inpRdaName - string with the name of the remote database being accessed	
	STR_MS_T	inpSQL - string, the SQL to use	
	STR_MS_T	inpParam1 - optional bind variable for the SQL	
	STR_MS_T	inpParam2 - optional bind variable for the SQL	
	STR_MS_T	inpParam3 - optional bind variable for the SQL	
	STR_MS_T	inpParam4 - optional bind variable for the SQL	retval - 0 on success
msiRdaCommit	**Commit changes to the database.**		
	STR_MS_T	inpRdaName - string with the name of the remote database being accessed	retval - 0 on success
msiRdaRollback	**Rollback (don't commit) changes to the database.**		
	STR_MS_T	inpRdaName - string with the name of the remote database being accessed	retval - 0 on success
Xmessaging System Micro-			

services			
msiXmsgServerConnect	**connect to the message server designated in iRODS environment file**		
			retval - 0 on success
			outConnParam [RcComm_MS_T] - connection descriptor
msiXmsgCreateStream	**create a stream for sending messages**		
		inConnParam - connection descriptor from connect	outXmsgTicketInfoParam [XmsgTicketInfo_MS_T] - information structure for the ticket generated for the stream
		inGgetXmsgTicketInpParam - integer expiration time	retval - 0 on success
msiCreateXmsgInp	**create a message**		
	uint or STR_MS_T	inMsgNumber - message serial number	
	uint or STR_MS_T	inMsgType - number of messages, =0 (SINGLE_MSG_TICKET) or = 1 (MULTI_MSG_TICKET)	
	uint or STR_MS_T	inNumberOfReceivers - number of receivers of the message	
	STR_MS_T	inMsg - message body	
	int or STR_MS_T	inNumberOfDeliverySites - number of receiving addresses	
	STR_MS_T	inDeliveryAddressList - list of host addresses (comma separated)	
	STR_MS_T	inDeliveryPortList - list of corresponding ports (comma separated)	
	STR_MS_T	inMiscInfo - other information	outSendXmsgInpParam [SendXmsgInp_MS_T] - Xmsg packet
	XmsgTicketInfo_MS_T	inXmsgTicketInfoParam - the outXmsgTicketInfoParam from stream creation	retval - 0 on success
msiSendXmsg	**send a message**		
	RcComm_MS_T	inConParam - connection descriptor from server connect	
	SendXmsgInp_MS_T	inSendXmsgInpParam - the outSendXmsgInpParam	retval - 0 on success
msiRcvXmsg	**receive a message**		

			outMsgType [STR_MS_T] - message type
	RcComm_MS_T	inConnParam - connection descriptor from server connect	outMSG [STR_MS_T] - message body
	XmsgTicketInfo_MS_T or STR_MS_T or uint	inTicketNumber - outXmsgTicketInfoParam from msiXmsgCreateStream or outXmsgTicketInfoParam->rcvTicket (a string which the sender passes to the receiver)	outSendUser [STR_MS_T] - sender information
	uint or STR_MS_T	inMsgNumber: message serial number to fetch	retval - 0 on success
msiXmsgServerDisConnect	**disconnect from the message server**		
	RcComm_MS_T	iConnParam - connection descriptor from server connect	retval - 0 on success

Email Micro-services			
msiSendMail	**send email!**		
	STR_MS_T	xtoAddr - address of the receiver	
	STR_MS_T	xsubjectLine - the subject of the message	
	STR_MS_T	sbody - the body of the message	retval - 0 on success
sendStdoutAsEmail	**send rei's stdout as email**		
	STR_MS_T	xtoAddr - addresss of the receiver	
	STR_MS_T	xsubjectLine - the subject of the message	retval - 0 on success

Key-Value (Attribute-value) Micro-services			
writeKeyValPairs			
msiPrintKeyValPair	**print key-value pairs to rei's stdout**		
	STR_MS_T	where - designate either stderr or stdout	
	KeyValPair-PI	inKVPair - KeyValPair list	retval - 0 on success
msiGetValByKey	**given a key and a keyValPair struct, extract the corresponding value**		
	KeyValPair_PI	inKVPair - KeyVallPair list	outVal [STR_MS_T] - value corresponding to key
	STR_MS_T	inKey - key	retval - 0 on success
msiString2KeyValPair	**Convert a %-separated key=value pair strings into keyValPair structure**		
			outKeyValPairP [KeyVal_Pair-MS_T] - keyValue Pair structure
	STR_MS_T	inBufferP - key=value paris separated by a %-sign	retval - 0 on success
msiStrArray2String	**Array of Strings converted to a string separated by %-signs**		
			outStr [STR_MS_T] - string separated by %-signs
	strArr_MS_T	inSAParam - array of strings	retval - 0 on success
msiAssociateKeyValuePairsToObj	**ingest object metadata into iCAT from a AVU structure**		
	KeyValPair-MS_T	metadataParam - the keyValPair structure	
	STR_MS_T	objParam - the name of the object	retval - 0 on success
	STR_MS_T	typeParam - the iCAT-type of the object	USR_PARAM_TYP_ERROR when the input parameter does not mathc the type
msiRemoveKeyValuePairsFromObj	**remove object metadata from iCAT using a AVU structure**		
	KeyValPair-MS_T	metadataParam - the keyValPair structure	
	STR_MS_T	objParam - the name of the object	retval - 0 on success
	STR_MS_T	typeParam - the iCAT-type of the object	USR_PARAM_TYP_ERROR when the input parameter does not mathc the type

Other User Micro-services			
msiExtractNaraMetadata	extract metadata from a NARA Archival Information Locator File		
msiLoadMetadataFromFile	bulk load of metadata from a file		
msiApplyDCMetadataTemplate	apply Dublin Core metadata template to extract metadata		
writeBytesBuf	write bytes into a buffer		
msiFreeBuffer	free space in a buffer		
writePosInt	write a positive integer into a buffer		
msiGetDiffTime	get the difference in time between two events		
msiGetSystemTime	get the current system time		
msiHumanToSystemTime	get the time since the last command		
msiGetIcatTime	get a time value from iCAT		
msiGetTaggedValueFromString	**get a tagged value from a string**		
	STR_MS_T	inTagParam - the tag to be matched	
	STR_MS_T	inStrParam - the source string	retval - 0 on success
msiExtractTemplateMDFromBuf	**extract AVU metadata from a buffer using template**		
			metadataParam [KeyValPair-MS_T] - extracted metadata in KeyVal Pair structure
			retval - 0 on success
	BUF_S_T	bufParam - input buffer from which meetadata is to be extracted	USER_PARAM_TP_ERROR when input parameters do not match the type,
	TagStruct_MS_T	tagParam - pre-tag and post-tag combinations that surround desired metadata	INVALID_REGEXP if the tags are not correct
msiReadMDTemplateIntoTagStruct	**load template file contents into Tag structure**		
			retval - 0 on success
			USER_PARAM_TYP_ERROR if the input parameters do not match the type,
	BUF_MS_T	tempObjBuf - template file	INVALID_REGEXP if the tags are not correct,
	TagStruct-S_T	tagStruct - tag-template	NO_VALUES_FOUND if there are no tags identified

System Micro- services	Can only be called by the server process		
msiSetDefaultResc	**set the default resource**		
		rei->doinp->condInput is used	
		rei->rsComm->proxyUser.authInfo.authFlag is used	
	STR_MS_T	xdefaulltRescList - a list of %-deliminted resourceNames	rei->rgi is set to a list of resources in the preferred order
	STR_MS_T	xoptionSTring - option (preferred, force, random) with random as default	retval - 0 on success
msiSetNoDirectRescInp	**set a list of resources that cannot be used by a normal user directly**		
		rei->doinp->condInput is used	
		rei->rsComm->proxyUser.authInfo.authFlag is used	retval - 0 on success
	STR_MS_T	xrescList - a list of %-delimintted resourceNames	USER_DIRECT_RESC_INPUT_ERR if resource is taboo
msiSetRescSortScheme	set the scheme for selecting the best resource to use		
msiSetMultiReplPerResc	set the number of copies per resource to unlimited		
msiSetDataObjPreferredResc	if the data has multiple copies, specify the preferred copy to use		
msiSetDataObjAvoidResc	specify the copy to avoid		
msiSetGraftPathScheme	Set the scheme for composing the physical path in the vault to GRAFT_PATH		
msiSetRandomScheme	set the the scheme for composing the physical path in the vault to RANDOM		
msiSetResource	set the resource from default		
msiSortDataObj	Sort the replica randomly when choosing which copy to use		
msiSetNumThreads	specify the parameters for determining the number of threads to use for data transfer		
msiSysChksumDataObj	checksum a data object.		
msiSysReplDataObj	replicate a data object.		
msiStageDataObj	stage the data object to the specified resource before operation.		
msiNoChkFilePathPerm	Do not check file path permission when registering		
msiNoTrashCan	Set the policy to no trash can.		
msiSetPublicUserOpr	Set a list of operations that can be performed by the user "public".		
msiCheckHostAccessControl	Set the access control policy.		
msiDeleteDisallowed	Set the policy for determining certain data cannot be deleted.		
msiSetDataTypeFromExt	get data type based on file name extension		

ERA - Electronic Records Archives Program			
msiRecursiveCollCopy	**Copy a collection and its contents recursively**		
	CollInp_MS_T or STR_MS_T	inpParam1 - the irods path of the destination collection	outParam [INT_MS_T] operation status
	CollInp_MS_T or STR_MS_T	inpParam2 - the irods path of the source collection	retval 0 on success
msiGetDataObjACL	**Get the access control list for a data object**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods path of the target object	outParam [BUF_LEN_MS_T] results
			retval 0 on success
msiGetCollectionACL	**Get the access control list for a collection**		
	CollInp_MS_T or STR_MS_T	inpParam1 - the irods path of the target collection	outParam [BUF_LEN_MS_T] results
	STR_MS_T	inpParam2 - Optional - Set it to "recursive" to perform the operation recursively	retval 0 on success
msiGetDataObjAVUs	**Retrieve metadata AVU triplets for a data object and return them as XML**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods path of the	outParam [BUF_LEN_MS_T] results

		target object	
			retval 0 on success
msiGetDataObjPSmeta	**Retrieve metadata AVU triplets for a data object and return them pipe separated**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods path of the target object	outParam [BUF_LEN_MS_T] results
			retval 0 on success
msiGetCollectionPSmeta	**Retrieve metadata AVU triplets for a collection and return them pipe separated**		
	CollInp_MS_T or STR_MS_T	inpParam - the irods path of the target collection	outParam [BUF_LEN_MS_T] results
			retval 0 on success
msiGetDataObjAIP	**Get an archival information package template for a data object**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods path of the target object	outParam [BUF_LEN_MS_T] results
			retval 0 on success
msiLoadMetadataFromDataObj	**Parse an iRods object (file) for new metadata AVUs and add them to the ICAT**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods path of the metadata file	outParam [INT_MS_T] operation status
			retval 0 on success
msiExportRecursiveCollMeta	**Export metadata AVU triplets for a collection and its contents**		
	CollInp_MS_T or STR_MS_T	inpParam - the irods path of the target collection	outParam [BUF_LEN_MS_T] results
			retval 0 on success
msiCopyAVUMetadata	**Copy metadata triplets from an iRODS object to another one**		
	STR_MS_T	inpParam1	outParam [INT_MS_T] operation status

		- the irods path of the source object	
	STR_MS_T	inpParam2 - the irods path of the destination object	retval 0 on success
msiGetUserInfo	**Return user information for one or more iRODS users**		
	STR_MS_T	inpParam1 - the target username. Can include wildcards	outParam [INT_MS_T] operation status
	BUF_LEN_MS_T	inpParam2 - Empty - a placeholder for the results	retval 0 on success
msiGetUserACL	**Return Access Control List for one or more iRODS users**		
	STR_MS_T	inpParam1 - the target username. Can include wildcards	outParam [INT_MS_T] operation status
	BUF_LEN_MS_T	inpParam2 - Empty - a placeholder for the results	retval 0 on success
msiCreateUserAccountsFromDataObj	**Parse an iRods object for new user accounts to create**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods file that contains the accounts to create	outParam [INT_MS_T] operation status

			retval 0 on success
msiLoadUserModsFromDataObj	**Parse an iRods object for user accounts to update**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods file that contains the account updates	outParam [INT_MS_T] operation status
			retval 0 on success
msiDeleteUsersFromDataObj	**Parse an iRods object for user accounts to delete**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods file that contains the accounts to delete	outParam [INT_MS_T] operation status
			retval 0 on success
msiLoadACLFromDataObj	**Parse an iRods object for access permissions to update/create**		
	DataObjInp_MS_T or STR_MS_T	inpParam - the irods file that contains the ACL updates	outParam [INT_MS_T] operation status
			retval 0 on success
msiGetAuditTrailInfoByUserID	**Get audit trail information by user identifier**		
	STR_MS_T	inpParam1 - the target user ID	outParam [INT_MS_T] operation status
	BUF_LEN_MS_T	inpParam2 - Empty - a placeholder for the results	retval 0 on success
msiGetAuditTrailInfoByObjectID	**Get audit trail information by object identifier**		
	STR_MS_T	inpParam1 - the target object ID	outParam [INT_MS_T] operation status
	BUF_LEN_MS_T	inpParam2	retval 0 on success

		- Empty - a placeholder for the results	
msiGetAuditTrailInfoByActionID	**Get audit trail information by action identifier**		
	STR_MS_T	inpParam1 - the target action ID	outParam [INT_MS_T] operation status
	BUF_LEN_MS_T	inpParam2 - Empty - a placeholder for the results	retval 0 on success
msiGetAuditTrailInfoByKeywords	**Get audit trail information by keyworkds in the comment field**		
	STR_MS_T	inpParam1 - the keyword string	outParam [INT_MS_T] operation status
	BUF_LEN_MS_T	inpParam2 - Empty - a placeholder for the results	retval 0 on success
msiGetAuditTrailInfoByTimeStamp	**Get audit trail information by time stamp**		
	STR_MS_T	inpParam1 - the beginning timestamp	outParam [INT_MS_T] operation status
	STR_MS_T	inpParam2 - the end timestamp	retval 0 on success
	BUF_LEN_MS_T	inpParam3 - Empty - a placeholder for the results	
msiSetDataType	**Set the data_type_name attribute of a data object by path or ID**		
	STR_MS_T	inpParam1 - the irods object ID	outParam [INT_MS_T] operation status

	STR_MS_T	inpParam2 - the irods object path	retval 0 on success
	STR_MS_T	inpParam3 - the attribute name	
msiGuessDataType	**Infer the data type of an object based on its extension and on the expandable ICAT list of known types**		
	STR_MS_T	inpParam1 - the irods object path	outParam [INT_MS_T] operation status
	STR_MS_T	inpParam2 - Empty - a placeholder for the result	retval 0 on success
msiGetCollectionContentsReport	**Return the number of objects in a collection by data type**		
	CollInp_MS_T or STR_MS_T	inpParam1 - the irods path of the target collection	outParam [INT_MS_T] operation status
	KeyValPair_MS_T	inpParam2 - Empty - a placeholder for the result	retval 0 on success
msiGetCollectionSize	**Return the object count and total disk usage of a collection**		
	CollInp_MS_T or a STR_MS_T	collPath - the irods path of the target collection	outKVPairs [KeyValPair_MS_T] results
			status [INT_MS_T] operation status
			retval 0 on success

XML (micro-services based on libxml2 and libxslt)			
msiLoadMetadataFromXml	**Parse an XML iRODS file to extract metadata tags**		
	DataObjInp_MS_T or STR_MS_T	targetObj - the irods path of the target object	retval 0 on success
	DataObjInp_MS_T or STR_MS_T	xmlObj - the irods path of the XML object	
msiXmlDocSchemaValidate	**Validate an XML file against an XSD schema, both iRODS objects**		
	DataObjInp_MS_T or STR_MS_T	xmlObj - the irods path of the XML object	outParam [INT_MS_T] validation result
	DataObjInp_MS_T or STR_MS_T	xsdObj - the irods path of the XSD object	retval 0 on success
msiXsltApply	**Apply an XSL stylesheet to an XML file, both iRODS objects**		
	DataObjInp_MS_T or STR_MS_T	xsltObj - the irods path of the XSL object	msParamOut [BUF_LEN_MS_T] results
	DataObjInp_MS_T or STR_MS_T	xmlObj - the irods path of the XML object	retval 0 on success

URL (micro-services based on libcurl)			
msiFtpGet	**FTP get a remote file and writes it to an iRODS object**		
	STR_MS_T	target - the remote URL	status [INT_MS_T] operation status
	DataObjInp_MS_T or STR_MS_T	destObj - the destination object's path	retval 0 on success

HDF	
msiH5File_open	open an HDF5 file
msiH5File_close	close an HDF5 file
msiH5Dataset_read	read a dataset from an HDF5 file
msiH5Dataset_read_attribute	read a dataset attribute from an HDF5 file
msiH5Group_read_attribute	read a group attribute from an HDF5 file

Properties	
msiPropertiesNew	create a new property
msiPropertiesClear	clear property

msiPropertiesClone	copy property
msiPropertiesAdd	add a property
msiPropertiesRemove	remove a property
msiPropertiesGet	get a property
msiPropertiesSet	set a property
msiPropertiesExists	check existence of a property
msiPropertiesToString	copy property to a string
msiPropertiesFromString	parse property from a string

Web Services			
msiGetQuote	**execute web service to get a stock quote**		
			outQuoteParam [STR_MS_T] - stock quotation as a float printed into a string
	STR_MS_T	inSymbolParam - stock symbol	retval - 0 on success
msiIp2location	**execute web service to convert IP address to a location**		
	STR_MS_T	inIpParam - the IP-address	outLocParam [STR_MS_T] - the location information
	STR_MS_T	inLicParam -the license string provided by http://ws.fraudlabs.com/	retval - 0 on success
msiConvertCurrency	**execute web service to convert currency**		
	STR_MS_T	inConvertFromParam - a 3-letter country code in structure char *countryCodeNames	outRateParam [STR_MS_T] - conversion rate as float printed into a string
	STR_MS_T	inConvertToParam - 3-letter country code in structure char *countryCodeNames	retval - 0 on success
msiObjByName	**execute web service to retrieve astronomy image by name**		
			outRaParam [STR_MS_T] - Right Ascension as float printed into a string
			outDecParam [STR_MS_T] - Declinationn as float printed into a string
			outTypParam [STR_MS_T] - type of object (star, galaxy, ...)
	STR_MS_T	inObjByNameParam - astronomical object name	retval - 0 on success
msiSdssImgCutout_GetJpeg	**execute web service to retrieve a Sloan Digital Sky Survey image cutout as a jpeg file**		

	STR_MS_T	inRaParam - right ascension as float printed into a string	
	STR_MS_T	inDecParam - declination as float printed into a string	
	STR_MS_T	inScaleParam - scaling factor as float printed into a string	
	STR_MS_T	inWidthParam - width of image as float printed into a string	
	STR_MS_T	inHeightParam - height of image as float printed into a string	outImgParam [BUF_LEN_MS_T] - image buffer
	STR_MS_T	inOptParam - optional parameters	retval - 0 on success

Guinot			
msiGetFormattedSystemTime	**get a formatted version of the system time**		
			outParam [STR_MS_T] - the formatted time
	STR_MS_T	inpParam - the desired output format	retval - 0 on success