E-iRODS Composable Resources

Terrell Russell[1], Jason Coposky[1], Harry Johnson[1], Ray Idaszak[1], Charles Schmitt[1]

(1) Renaissance Computing Institute, University of North Carolina, Chapel Hill

Abstract

RENCI has developed composable resources for Enterprise iRODS (E-iRODS) which allow for shareable, flexible definitions of storage resources. Using a well-defined tree metaphor to describe composite resources provides insight into existing resources as well as a powerful tool for envisioning new resource configurations. Defined resources can be shared and iterated within the community easily as they are plugins, external from the E-iRODS core.

Index Keyword Terms— *iRODS, E-iRODS, storage resources, data management, composability, plugins*

1. Introduction

Resources in iRODS have historically sat at the interface between the data management layer and logic of iRODS and the storage of physical bits on disk or tape. Resources are characterized by drivers that handle the translation to and from those physical media. With E-iRODS composable resources, drivers and management logic are encapsulated into a pluggable architecture that allows the community to develop new and interesting configurations outside the E-iRODS core.

Composable resources are included in the third beta of E-iRODS and are planned to be included in future releases of iRODS. They represent a retooling of how iRODS manages its relationship with physical media. Composable resources use a tree metaphor to describe their functionality and capabilities. They represent a more flexible architecture with which to construct as complex a configuration of storage devices as necessary.

Included are resource types representing:

- unix file system
- legacy compound (expected)
- random (expected)
- replication (expected)
- round-robin (expected)
- storage balancing (expected)
- tiered (expected)
- pass through (for testing)

2. Tree Metaphor

In computer science, a tree is a data structure with a hierarchical representation of linked nodes. These nodes can be named based on where they are in the hierarchy.

The node at the top of a tree is the root node. Parent nodes and child nodes are on opposite ends of a connecting link, or edge. Leaf nodes are at the bottom of the tree, and any node that is not a leaf node is a branch node. These positional descriptors are helpful when describing the structure of a tree. Composable resources are best represented using this tree metaphor.

3. Virtualization

In iRODS, files are stored as Data Objects on disk and have an associated physical path as well as a virtual path within the iRODS file system. iRODS collections only exist in the iCAT database and do not have an associated physical path (allowing them to exist across all resources, virtually).

Composable resources introduce the same dichotomy between the virtual and physical. E-iRODS resources are defined to be either *coordinating resources* or *storage resources*. These two different classes of resource map directly to the branch nodes and leaf nodes of a generic tree data structure. A coordinating resource has built-in logic that defines how it determines, or coordinates, the flow of data to and from its children. Coordinating resources exist solely in the iCAT and virtually exist across all E-iRODS servers in a particular Zone. A storage resource has a Vault (physical) path and knows how to speak to a specific type of storage medium (disk, tape, etc.). The encapsulation of resources into a plugin architecture allows E-iRODS to have a consistent interface to all resources, whether they represent coordination or storage.

This virtualization of the coordinating resources allows the logic of how to manage both the placement and the retrieval of Data Objects to exist independent of the types of resources that are connected as children resources. When E-iRODS tries to retrieve data, each child resource will “vote” by offering whether it can provide the requested data, and coordinating resources

will decide which particular storage resource (e.g. physical location) the read should come from. The specific manner of this vote is specific to the logic of the coordinating resource. For instance, a coordinating resource could optimize for reducing the number of requests made against each storage resource within some time frame or it could optimize for reducing latency in expected data retrieval times. We expect a wide variety of useful optimizations to be developed by the community.

An intended side effect of the tree metaphor and the virtualization of coordinating resources is the deprecation of the concept of a resource group. Resource groups in community iRODS could not be put into other resource groups. A specific limiting example was that of the compound resource where, by definition, it was a group and could not be placed into another group significantly limiting its functionality as a management tool.

Groups in E-iRODS now only refer to user groups.

4. Dynamic Functionality

4.1 Policy Enforcement Points (PEPs)

The code that defines a particular type of composite resource is developed as a plugin and is dynamically parsed when it is added to an E-iRODS system. When an exposed method on a resource type is first called, an operation within the resource plugin will dynamically construct a string representing a new policy enforcement point (PEP) relevant to its own functionality and invoke both the pre() and post() PEP for that operation. These well-defined PEPs can be used by rules to enforce more fine-grained policy. These strings are matched directly by name and are constructed as follows:

```
pep_$operation _pre()
pep_$operation_post()
```

For example, the derived PEP pre() and post() strings for open, close, read, and write would be:

```
pep_open_pre()
pep_open_post()
pep_close_pre()
pep_close_post()
pep_read_pre()
pep_read_post()
pep_write_pre()
pep_write_post()
```

Within each of these operations, the following four iRODS $-variables (dollar variables) are populated and made available:

```
$pluginInstanceName
$objPath
$phyPath
$replNum
```

4.2 Post Disconnect Maintenance Operations (PDMOs)

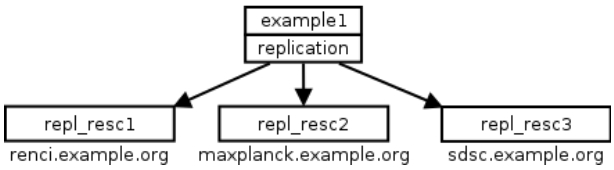
Since some operations may require multiple replicas to be made across potentially slow or latent connections, the plugin environment provides functionality for allowing “offline” or “post-disconnect” operations. Any work that needs to be done after the prompt is returned to the user can be done inside the function named post_disconnect_maintenance_operation() within the plugin.

5. Examples

The following three examples illustrate how some new composite resources can be constructed by a data grid administrator.

5.1 Replicates Data Objects to three locations

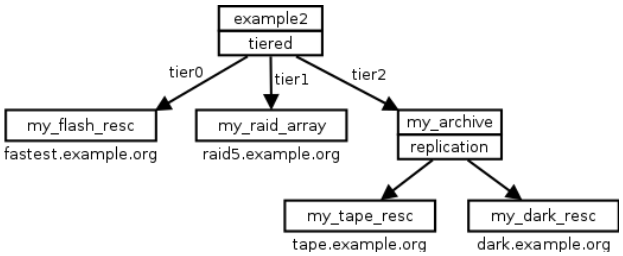
```
iadmin mkresc example1 replication
iadmin mkresc repl_resc1 "unix file system" renci.example.org:/Vault
iadmin mkresc repl_resc2 "unix file system" maxplanck.example.org:/Vault
iadmin mkresc repl_resc3 "unix file system" sdsc.example.org:/Vault
iadmin addchildtoresc example1 repl_resc1
iadmin addchildtoresc example1 repl_resc2
iadmin addchildtoresc example1 repl_resc3
```



This first example requires 7 iadmin commands to compose. The replication coordinating resource named “example1” is configured to have three children which would each receive a replica of every Data Object “put” into “example1”. The default replication coordinating resource code will populate one of the children resources, return to the calling function, and then use the PDMO to queue two replication events to populate the remaining children. It is the coordinating resource’s responsibility to provide a “voting” mechanism for which replica is returned when the Data Object is requested by a user (i.e. iget). This voting could be as simple as returning the first replica in the iCAT database or as complicated as keeping statistics on throughput or load or latency.

5.2 Tiered storage system with a dark archive

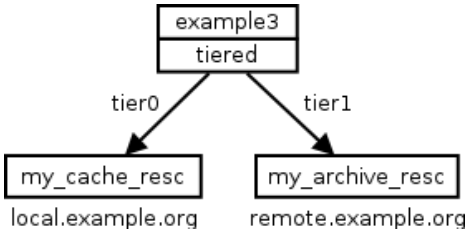
```
iadmin mkresc example2 tiered
iadmin mkresc my_flash_resc "unix file system" fastest.example.org:/Vault
iadmin mkresc my_raid_array "unix file system" raid5.example.org:/Vault
iadmin mkresc my_archive replication
iadmin mkresc my_tape_resc "unix file system" tape.example.org:/Vault
iadmin mkresc my_dark_resc "unix file system" dark.example.org:/Vault
iadmin addchildtoresc example2 my_flash_resc "tier0"
iadmin addchildtoresc example2 my_raid_array "tier1"
iadmin addchildtoresc example2 my_archive "tier2"
iadmin addchildtoresc my_archive my_tape_resc
iadmin addchildtoresc my_archive my_dark_resc
```



This example requires 11 iadmin commands to compose. The tiered coordinating resource named "example2" has three children, ordered with context strings specifying their tiered positions (0, 1, and 2). Here, they are ordered by speed with the fastest tier0 being a flash device, tier1 being a regular raid array of spinning disk, and a third tier2 being a replication coordinating resource with two children of its own. The tier2 "my_archive" is replicating onto a tape device of some kind as well as a write-only space known as a dark archive. This "my_dark_resc" could have policy around it specifying that only admins can write, and no user can read. A dark archive would then always 'vote' negatively requiring its parent to coordinate its own vote when responding to its own parent.

5.3 Simulates legacy compound resource type

```
iadmin mkresc example3 tiered
iadmin mkresc my_cache_resc "unix file system" local.example.org:/Vault
iadmin mkresc my_archive_resc "unix file system" remote.example.org:/Vault
iadmin addchildtoresc example3 my_cache_resc "tier0"
iadmin addchildtoresc example3 my_archive_resc "tier1"
```



This example requires 5 iadmin commands to compose. The tiered coordinating resource named "example3" has two children which are ordered by locality. The local "my_cache_resc" would be consulted first for any read request that comes into "example3". A write request would go into the cache resource first as well and then be replicated to its peer, the remote "my_archive_resc" via the internal logic of the tiered coordinating resource and employing the PDMO, just like example 5.1. This example can duplicate the existing functionality of the compound resource as defined in community iRODS.

6. Conclusion

The introduction of composable resources into the iRODS technology environment represents the capability to develop storage resources that have greater functionality than can be provided for today. By dynamically generating additional relevant PEPs when new resources are plugged into a running instance of E-iRODS, the benefits of the iRODS rule engine are preserved (if not magnified). This additional flexibility and functionality open the door for developers to generate new classes of storage resources that can easily be integrated into an existing E-iRODS deployment without requiring updates to the underlying E-iRODS release installation.

7. Acknowledgements

This work has been supported by RENCI. We would like to thank the iRODS team and community for input and guidance on the presented approach.