Technology Update

Terrell Russell, Ph.D.
@terrellrussell
Chief Technologist, iRODS Consortium

June 9-12, 2020
iRODS User Group Meeting 2020
Virtual Event

In The Last Year

iRODS Release	Issues Closed
4.2.7	58
4.2.8	122

```
~/irods $ $ git shortlog --summary --numbered 4.2.6..4.2.8
    82  Kory Draughn
    36  Alan King
    13  Terrell Russell
    13  d-w-moore
    12  Jaspreet Gill
    10  Jason Coposky
     4  Justin James
     1  Ben Keller
     1  John Thiltges
     1  Matt Watson
```

In The Last Year

iRODS

Plugins

- Python Rule Engine Plugin
- **Storage Tiering Rule Engine Plugin**
- Auditing (AMQP) Rule Engine Plugin
- Update Collection Mtime Rule Engine Plugin
- **S3 Resource Plugin**
- Kerberos Authentication Plugin
- Curl Microservice Plugin
- **Hard Links Rule Engine Plugin**
- **Indexing Rule Engine Plugin**
- **Logical Quotas Rule Engine Plugin**
- Metadata Guard Rule Engine Plugin

Clients

- Python iRODS Client
- MetaInx
- **NFSRODS**
- Automated Ingest Framework
- **AWS Lambda for S3**

Active Development Work

- iRODS 4.2.9
- iRODS 4.3.0
- Metadata Templates Working Group
- Authentication Working Group
- Parallel Transfer Engine
- Logical Locking
- Policy Composition
- Publishing Capability
- NetCDF microservices
- C++-based REST API
- MetaInx and Indexing
- NFSRODS
- Testing Infrastructure

Working Groups

Technology Working Group

- Goal: To keep everyone up to date, provide a forum for roadmap discussion and collaboration opportunities

Metadata Templates Working Group

- Goal: To define a standardized process for the application and management of metadata templates by the iRODS Server
  - NIEHS / Data Commons
  - Utrecht / Yoda
  - Maastricht / DataHub+
  - Arizona / CyVerse

Authentication Working Group

- Goal: To provide a more flexible authentication mechanism to the iRODS Server.
  - SURF
  - NIEHS
  - Sanger
  - CyVerse
  - Utrecht

Philosophical Drivers

- Plugin Architecture
  - core is generic - protocol, api, bookkeeping
  - plugins are specific
  - policy composition
- Modern core libraries
  - standardized interfaces
  - refactor iRODS internals
    - ease of (re)use
    - fewer bugs
- Replicas as first class entities
  - logical locking
- Consolidation of data movement
  - dstreams all on 1247

Last Year and Next Year

- Core Libraries
  - Kory Draughn
- Logical Locking
  - Alan King
- Python Query Facilities
  - Daniel Moore
- Build and Test
  - Jaspreet Gill

Last Year's C++ Libraries

Goal: Provide standardized interfaces that simplify common iRODS tasks

- **filesystem**
  - server, plugins, icommands
- **iostreams**
  - server, indexing, S3 resource, icommands
- **thread_pool**
  - delay execution server, S3 resource
- **connection_pool**
  - delay execution server
- **query**
  - server, indexing, publishing, storage tiering
- **query_processor**
  - delay execution server, storage tiering

This Year's C++ Libraries: It's getting easier!

Nine new libraries:

- **key_value_proxy**
  - Provides a map-like interface over an existing keyValuePair_t.
- **lifetime_manager**
  - Guarantees that heap-allocated iRODS C structs are free'd at scope exit.
- **user group administration**
  - Simplifies management of iRODS users and groups.
- **shared_memory_object**
  - Simplifies access and management of shared memory.
- **with_durability**
  - A convenient retry mechanism for functions and function-like objects.
- **query_builder**
  - Enables query objects to be constructed lazily.
- **client_api_whitelist** *(server-side only)*
  - An interface for managing and querying the client API whitelist.
- **scoped_privileged_client** *(server-side only)*
  - Elevates the client's privileges for the duration of a scoped block.
- **scoped_client_identity** *(server-side only)*
  - Changes the client's identity for the duration of a scoped block.

New API Plugin

Atomic Metadata Operations API Plugin

Executes a list of metadata operations on a single object atomically.

Features:

- Supports data objects, collections, users, and resources
- Provides a future proof interface by accepting JSON as input
- Supported by the iRODS Filesystem library
  - add_metadata(comm, path, container_holding_avus)
  - remove_metadata(comm, path, container_holding_avus)

Example JSON Input:

```
{
    "entity_name": "/tempZone/home/rods",
    "entity_type": "collection",
    "operations": [
        {
            "operation": "add",
            "attribute": "iRODS",
            "value": "is",
            "units": "awesome!"
        }, {
            "operation": "remove",
            "attribute": "ugm",
            "value": "2019"
        },
        // ... More Operations ...
    ]
}
```

Library / API Examples

Examples on using these libraries can be found at the following repository:

- https://github.com/irods/irods_api_examples

Help us make them better!

Replicas vs. Data Objects: Why It Matters

Data Object: a logical representation of data that maps to one or more physical instances (Replicas) of the data at rest in Storage Resources

Replica: an identical, physical copy of a Data Object

from training: https://github.com/irods/irods_training/blob/master/beginner/irods_beginner_training_2019.pdf

Operations which deal directly with replicas have completely separate implementations for moving data. Operations dealing with data objects still need access to replica information.

All of this has consistency and performance implications for moving data. In reality, all of these operations *should be* and *are* identical:

Open replica, *move* data to replica, *close* replica

Solution: Make replicas a proper entity within iRODS

Data Movement and Replica Status

But replicas have their own problems...

- A replica's status is wrong the moment it is created
- Replicas are either good or stale, even if it is not at rest

Solution: Intermediate replica status for data not at rest

Replica status should always reflect what's in the catalog, there's only one way to move data, and can be surfaced with a standardized interface - great! And it's even mostly implemented!

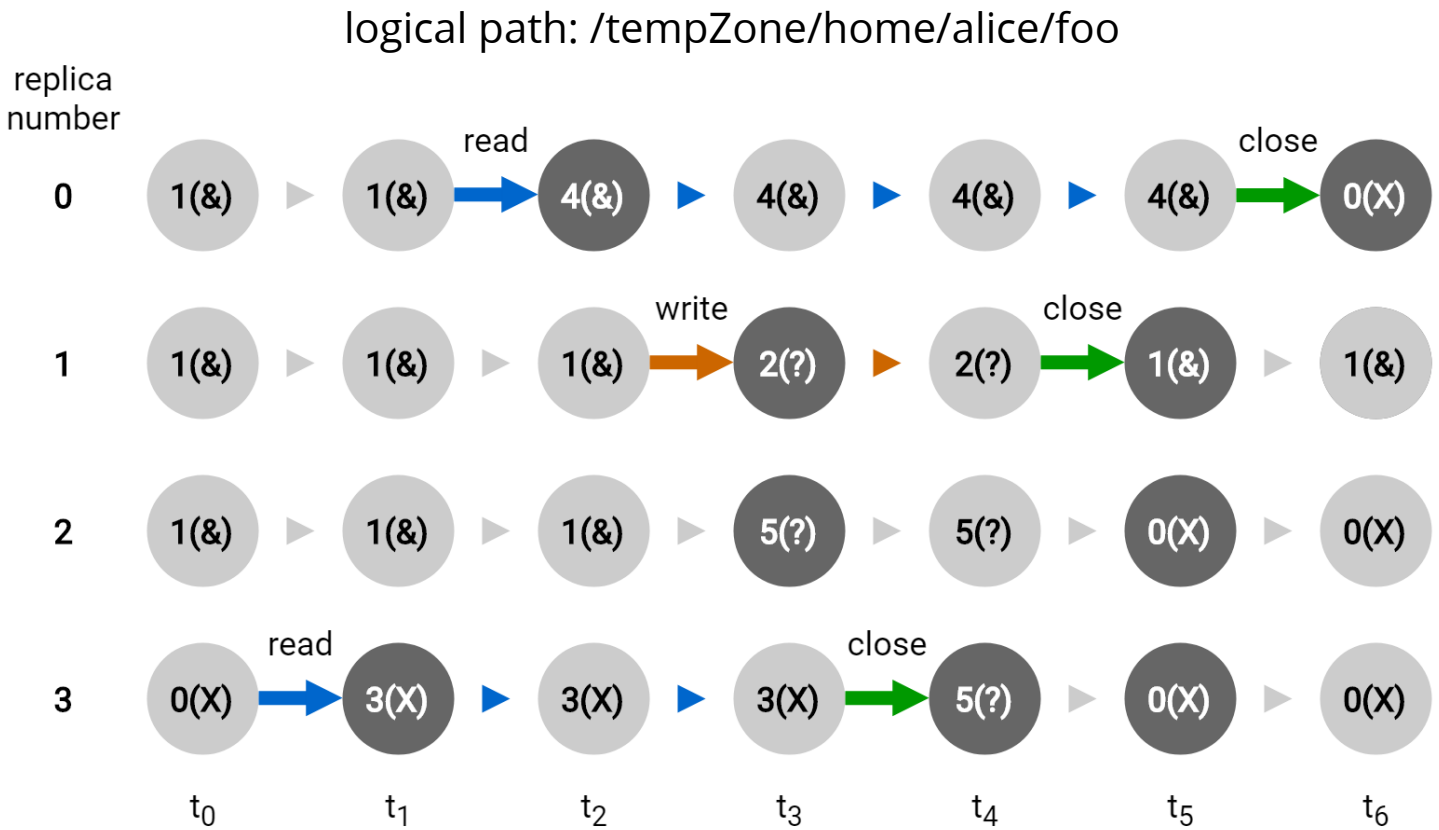
...but what about concurrent operations on different replicas represented by a single data object?

Solution: Logical locking

Logical Locking (irods/irods#3848)

Value	ils	Status	Description
0	X	stale	- data at rest may not match catalog
1	&	good	- data at rest matches catalog
2	?	intermediate	- data is not at rest
3	X	read lock	- allows open for read - locks out open for write - original status was stale
4	&	read lock	- allows open for read - locks out open for write - original status was good
5	?	write lock	- locks out all opens for this replica - when sibling replica marked intermediate

Pathological Concurrent Operation Scenario



0(X): stale
1(&): good
2(?): intermediate
3(X): read lock (stale)
4(&): read lock (good)
5(?): write lock

t0: 4 replicas; 3 good, 1 stale
t1: r3 opened for read; r3->3(X)
t2: r0 opened for read; r0->4(&)
t3: r1 opened for write; r1->2(?), r2->5(?)
t4: r3 closed/finalized; r3->5(?)
t5: r1 closed/finalized; r1->1(&), r2->0(X), r3->0(X)
t6: r0 closed/finalized; r0->0(X)

Python Rule Engine Plugin - Improved General Query

General Query facility provided by **/etc/irods/genquery.py**

The example below has:

- two return columns, "COLL_NAME" and "DATA_NAME"
- where clause matching DATA_NAME "like" a passed string variable

Original, more verbose syntax in 4.2.5:

```
from genquery import row_iterator, AS_LIST
def data_name_like (rule_args, callback, rei):
    q = row_iterator(["COLL_NAME","DATA_NAME"],
                     "DATA_NAME like '{}'".format(rule_args[1]),
                     AS_LIST,
                     callback )
    rule_args[:2] = [row for row in q] [0]
```

With improvements from Chris Smeele (Utrecht) in 4.2.8:

```
from genquery import Query
def data_name_like (rule_args, callback, rei):
    q = Query(callback,["COLL_NAME","DATA_NAME"],
              "DATA_NAME like '{}'".format(rule_args[1]))
    rule_args[:2] = q.first()
```

Python iRODS Client - General Improvements

These new features are available in v0.8.3

Queries can target federated zones

```
import irods.keywords as kw
from irods.models import DataObject
from datetime import timedelta, datetime
with iRODSSession(...) as session:
    q = session.query(DataObject.id) \
        .add_keyword(kw.ZONE_KW,'otherZone') \
        .filter(DataObject.modify_time > datetime.utcnow()-timedelta(seconds=3600))
    for row in q:  print( row[DataObject.id] )
```

"IN" operator

```
from irods.column import In
from irods.models import User, Collection
query_results = [ u[User.name] for u in session.query(User) \
                               .filter(User.zone == 'myZone')
]
for coll in session.query(Collection.name) \
                   .filter(In(Collection.owner_name,query_results)):
    print (coll)
```

Python iRODS Client - General Improvements

This query involves a single column multiple times:

```
with iRODSSession(...) as session:
  x = [ i for i in session.query(DataObject.id,Collection.name,DataObject.name)\
     .filter( Like(DataObjectMeta.name, 'criterionX\_%'), DataObjectMeta.value < '4')\
     .filter( Like(DataObjectMeta.name, 'criterionY\_%'), DataObjectMeta.value > '6')\
  ]
  print(x)
```

The equivalent iquest can be seen here:

```
$ iquest "select DATA_ID, COLL_NAME, DATA_NAME where \
  META_DATA_ATTR_NAME like 'criterionX\_%' and META_DATA_ATTR_VALUE < '4' and \
  META_DATA_ATTR_NAME like 'criterionY\_%' and META_DATA_ATTR_VALUE > '6' "
```

imeta provides a simpler usage if the attribute names are known:

```
$ imeta qu -d 'criterionX_a' '<' 4 and 'criterionY_b' '>' 6
```

iRODS Build and Test - History

July 2011

- Python → Node.js → RabbitMQ → Celery → Eucalyptus

October 2012

- Python → Node.js → ssh → OpenStack

January 2013

- Hudson → Python → OpenStack

October 2013

- Hudson → Python → vSphere long-running VMs

Spring 2015

- Jenkins → Python → Ansible → zone_bundles → vSphere dynamic VMs

Spring 2017

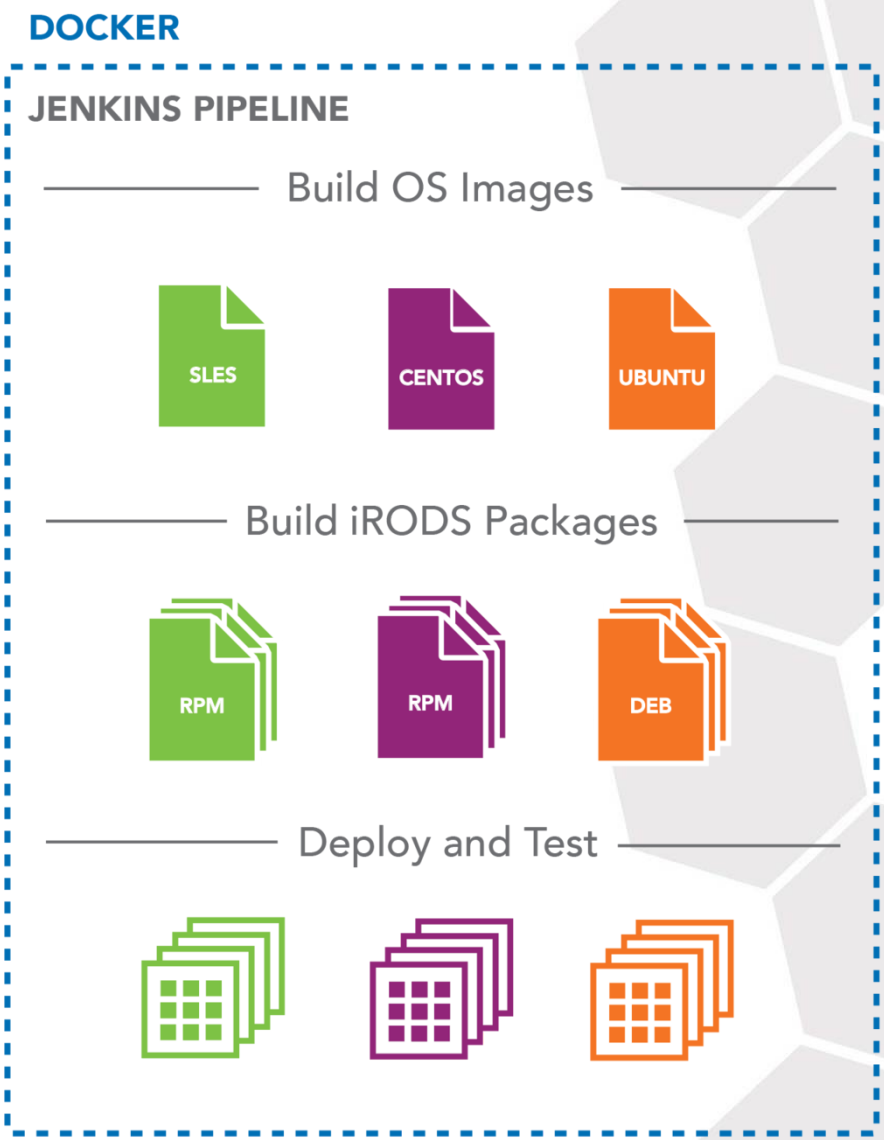
- Moved iRODS build/test logic from Ansible to python modules (per-repository)
- Consolidated to two parameterized Jenkins jobs

Fall 2019

- Jenkins → Python → Docker

iRODS Build and Test - 2019 Architecture

- Dockerized Jenkins
- All configuration and setup in git
- Launches sibling Docker containers
  - Build OS Images
  - Build iRODS Packages
  - Deploy and Test
    - core, plugins, topology, federation
- Development is same as production



iRODS Build and Test - Progress

- [x] Increase coverage (more plugins in CI)
- [x] Move pipeline scripts to GitHub (no logic in Jenkins)
- [x] Address inconsistency (false reds / pyvmomi errors)
- [x] Containerize Jenkins (easier to test / update / redeploy)
- [x] Move from VMs to containers (speed / fewer moving parts)
- [ ] Parallelize the jobs (speed)
- [ ] Continuous Integration (speed / integrity / accountability)
- [ ] Make iRODS Jenkins public (accountability / confidence)

iRODS Build and Test - 4.2.8 release cycle

- iCAT database runs in its own container for every test
- Serialized Workflow
  - Docker by default creates max 31 networks
  - GitHub rate limit exceeded exception
  - We are still learning about Docker
- Operating Systems supported → Ubuntu 16, Ubuntu 18, and CentOS 7
- Databases supported → PostgreSQL, MySQL/MariaDB, and Oracle
- Number of Core Test Suites per OS per Database → 65
- Number of Plugins Tested per OS per Database → 12
- Topology → 4 combinations (Provider/Consumer, with/without SSL)
- Upgrade Topology Test → 2 combinations (Provider/Consumer)
- Federation → 1 combination (current vs. current)

iRODS Build and Test - Future

- Make iRODS Jenkins publicly accessible
- Investigate scaling out
- Increase coverage
  - more tests
  - more plugins
  - more operating systems (SLES 15)
- Conformance testing
- Approachable for community developers
  - Confidence
  - Acceptance Criteria

Philosophy to Policy

With the new libraries and first class replicas, we can rewrite 90% of the internals, and then fix the things that depend on them later, with little expectation of regression, because the interfaces remain the same.

Internally

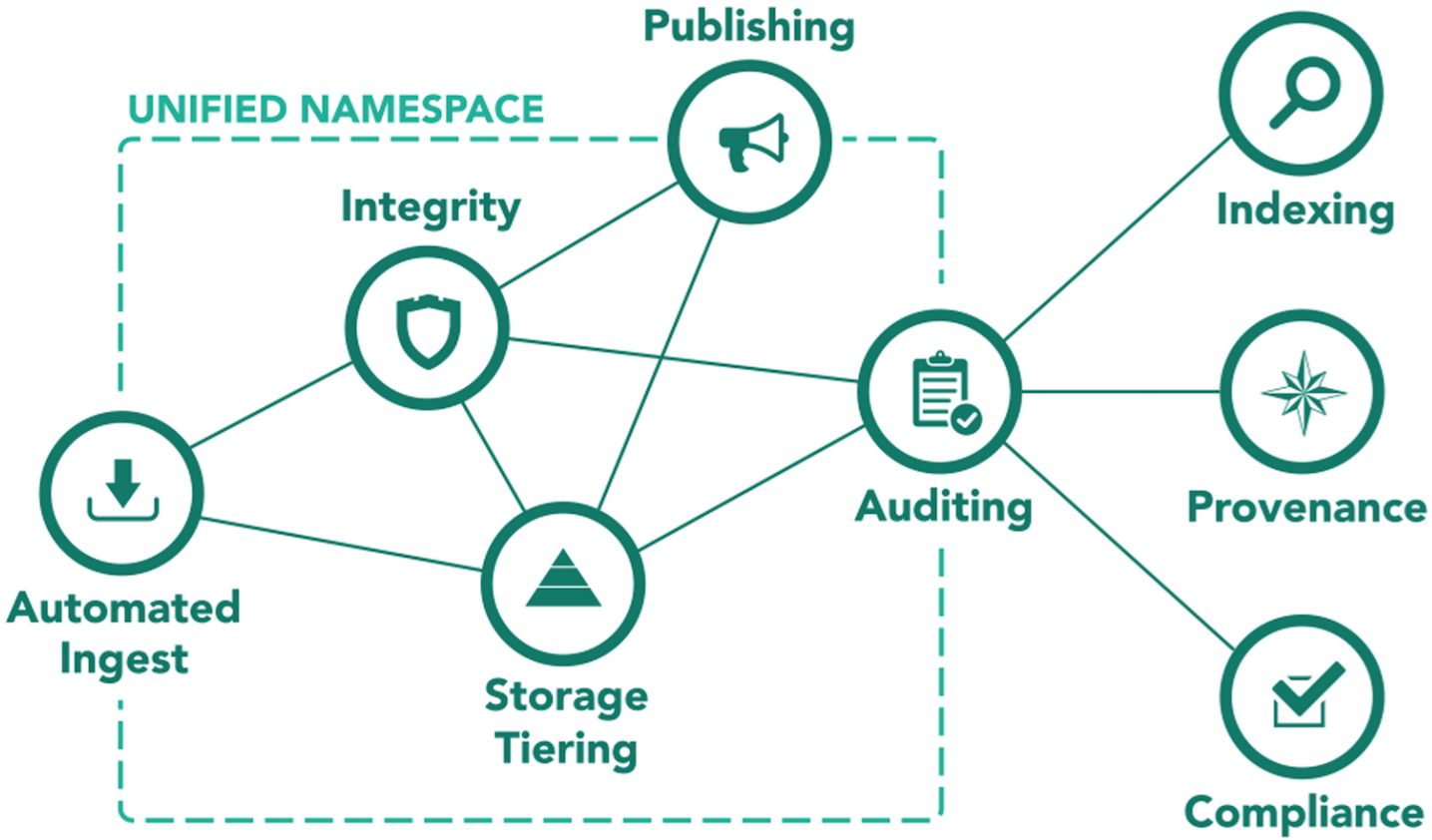
- We will have a new API... but not really
- Instead, we stepped back and built good tools
  - Allows us to refactor and go faster without breaking the 4.x API
  - This has turned out to be more powerful than originally expected

Externally

- It's a good story, the ability to compose policy into capabilities
- Can build smaller pieces of functionality which can be composed to help solve larger problems
- We don't have to worry about side effects

Continuation within the Rule Engine Plugin Framework allows administrators to break apart monolithic policy implementations into reusable components.

iRODS Data Management Model



Big Picture

Core

- 4.3.0 - Harden and Polish

Clients

- GUIs (Metalnx, et al.)
- Onboarding and Syncing (Automated Ingest)
- File System Integration (NFSRODS / SMBRODS)
- iRODS Console (alongside existing iCommands)

Continue building out policy components (Capabilities)

We want installation and management of iRODS to become about policy design, composition, and configuration.

Please share your:

- use cases
- pain points
- hopes and dreams

Open Source Community Engagement

Get Involved

- Working Groups
- GitHub Issues
- Pull Requests
- Chat List
- Consortium Membership

Tell Others

- Publish, Cite, Advocate, Refer