iRODS Capability: Automated Ingest

Hao Xu
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
xuh@cs.unc.edu

Alan King
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
alanking@renci.org

Terrell Russell
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
unc@terrellrussell.com

Jason Coposky
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
jasonc@renci.org

Antoine de Torcy
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
adetorcy@renci.org

ABSTRACT

The iRODS Automated Ingest Framework is a new iRODS client that has been designed to scale up to match the demands of data coming off instruments, satellites, or parallel filesystems and provide a front door to the policy-based data management platform of iRODS.

Initial testing shows promising flexibility and a roughly linear performance curve.

Keywords

iRODS, capability, ingest, Celery, Redis, data management

INTRODUCTION

The iRODS Automated Ingest Framework[1] has been designed to solve two major use cases: registering large amounts of existing data into an iRODS namespace without moving the source data (filesystem scanning) and ingesting new or updated data from a known location in a filesystem (a landing zone) into place within an iRODS Vault.

Based on the Python iRODS Client[3], Celery[4], and Redis[5], the goal of this framework is to scale up to match the demands of data coming off instruments, satellites, or parallel filesystems and provide a front door to the policy-based data management platform of iRODS[2].

For testing, this framework has been deployed manually. For enterprise customers, this framework is prototyped as Docker containers to be deployed and run on a Kubernetes cluster via Helm charts.

ARCHITECTURE

Overview

The motivation for this tool was to provide a parallel and distributed means of getting data into the iRODS catalog. Celery and Redis provide the coordination mechanism for concurrent 'stat' gathering from the source location as well as concurrent iRODS connections for an initial scan. They also provide a very fast insulating layer for a 'delta sync' when a data source is scanned again and many of the source files have not changed. In this case, the Redis cache reports that nothing has changed for a particular file, and the client determines that no connection to iRODS is necessary. Delta scans are much faster than initial scans for this reason.

iRODS UGM 2018 June 5-7, 2018, Durham, North Carolina, USA
[Authors retain copyright.]

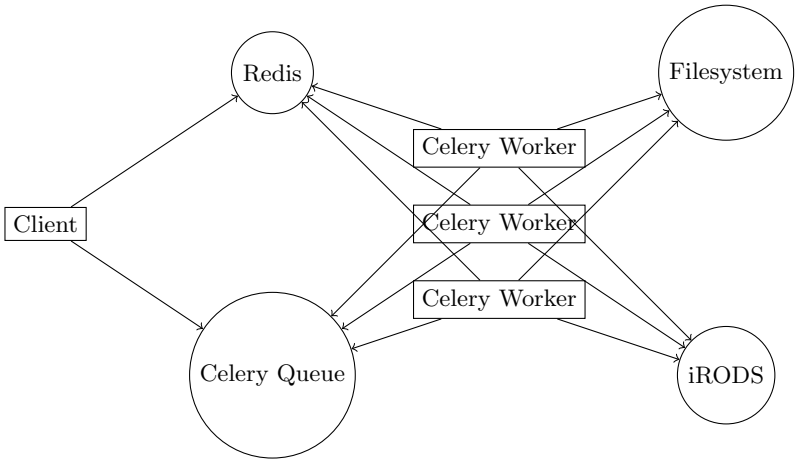

Figure 1. The number of Celery Workers can scale up to meet the required performance demand.

Client

The automated ingest client takes an initial path to scan, submits the job to the celery queue, and sets the following necessary Redis metadata. If the job is to be run a single time, the job name is added to the `singlepass` list. If the job is to be run continuously (scanning the same source again and again), then the job name is added to the `periodic` list. These two lists keep track of jobs that are running. If the job is in `singlepass`, a `restart` task is called one time synchronously to start the job. If the job is in `periodic`, a `restart` task is added to the `restart` queue which will start the work again once the current pass is complete.

The client can also stop a job and all tasks under that job.

Celery Queue

The `restart` task resets the `count` and `dequeue` lists, and the `tasks`, `retries`, and `failures` counters. The `restart` task calls a `sync_path` task that recursively and asynchronously walks the requested filesystem location (the source).

If the requested path is a directory, the `sync_path` task calls a `sync_dir` task that asynchronously creates and populates the metadata for that directory in the Redis cache, creates the collection in iRODS, and then lists and calls `sync_path` asynchronously on the immediate children of the directory. The `sync_dir` task compares the last sync time with the mtime and ctime of the directory, and if the directory has changed, the collection in iRODS is synchronized.

If the requested path is a file, the `sync_path` task calls a `sync_file` task that asynchronously creates and populates the metadata for that file in the Redis cache, and then puts or registers the file into iRODS. If the Redis entry already exists, the `sync_file` task compares the last sync time with the mtime and ctime of the file, and if the file has changed, the file contents and system metadata in iRODS are synchronized.

Before a task is added to the queue, the `count` counter is incremented and the task id is added to the task list. Each task has a retry handler, a failure handler, and an after return handler. The retry handler and failure handler increments the `retry` and `failure` counter respectively. The after return handler decrements the `count` counter and removes the task id from the `tasks` list. When the `tasks` counter is zero, it calls the cleanup function.

Redis

A Redis database can be used by Celery as a broker. Another Redis database is used to store metadata about jobs, including the `singlepass` and `periodic` lists, the `count` and `dequeue` lists, and the `tasks`, `retries`, and `failures`

counters. The `singlepass` list contains all the names of single pass jobs, the `periodic` list contains all the names of periodic jobs. These two lists are used to calculate running tasks for stopping a job. The `count` list contains all tasks created by the job and the `dequeue` list contains all tasks that are finished (with either success or failure). These two lists are used to calculate the list of running tasks. A running task can be stopped by name. The `tasks` counter is used to keep track of the remaining tasks. When the `tasks` counter reaches zero, the cleanup function is triggered. The `retries` and `failures` counters keep track of retried and failed tasks.

Event Handlers

The automated ingest Celery workers can be deployed with a variety of options to describe their behavior around gathering, preparing, and sending information to iRODS. These options are described in event handler files and handed to the workers. This allows for custom behavior to be written for particular deployments of the Ingest Framework. The event handler methods made available to the workers include:

method	effect	default
pre_data_obj_create	user-defined Python	none
post_data_obj_create	user-defined Python	none
pre_data_obj_modify	user-defined Python	none
post_data_obj_modify	user-defined Python	none
pre_coll_create	user-defined Python	none
post_coll_create	user-defined Python	none
pre_coll_modify	user-defined Python	none
post_coll_modify	user-defined Python	none
as_user	takes action as this iRODS user	authenticated user
target_path	set mount path on the iRODS server which can be different from client mount path	client mount path
to_resource	defines target resource request of operation	as provided by client environment
operation	defines the mode of operation	Operation.REGISTER_SYNC
max_retries	defines max number of retries on failure	0
timeout	defines seconds until job times out	3600
delay	defines seconds between retries	0

Table 1. Available event handler methods

Where user-defined Python can be written, the event handler is just providing hooks for data preparation or bookkeeping. The other methods are either Celery-specific or iRODS-specific configuration options for the operation to be performed.

Operations

The event handler method `operation` defines the mode for any iRODS connection. The following six operations are available and determine whether data is transferred and how the iRODS catalog is updated:

operation	new files	updated files
Operation.REGISTER_SYNC (default)	registers in catalog	updates size in catalog
Operation.REGISTER_AS_REPLICA_SYNC	registers first or additional replica	updates size in catalog
Operation.PUT	copies file to target vault, and registers in catalog	no action
Operation.PUT_SYNC	copies file to target vault, and registers in catalog	copies entire file again, and updates catalog
Operation.PUT_APPEND	copies file to target vault, and registers in catalog	copies only appended part of file, and updates catalog
Operation.NO_OP	no action	no action

Table 2. Available event handler operation modes

SOURCE AS S3

The Automated Ingest Framework was originally designed for ingestion from mounted filesystems. However, during the framework's development, an additional use case of ingesting from an existing S3 bucket was presented. Once a suitable python library was identified (`minio`), ingesting data via the S3 protocol worked as expected.

FLEXIBILITY

With these different configuration options, the functionality required for a particular use case can be realized.

The two main use cases that are solved by this framework are the filesystem scanner (source files stay in place after scanning) and the landing zone (source files are moved aside in some manner after being scanned). The good way to think about the difference between these two ways of setting up the Automated Ingest Framework is to consider where the source of truth will be once the scanning has been performed.

With the filesystem scanner use case, the truth remains in the original source location since that data could continue to move as new science is performed and other systems are writing into that source location. This is most likely to be useful when putting jobs into the `periodic` list.

With the landing zone use case, the truth now lies in the iRODS Catalog. iRODS has a copy of the data under management and it is now owned and operated within a Vault that iRODS controls. This is most likely to be useful when putting jobs into the `singlepass` list.

Within these two different use cases, there is the opportunity for registering new physical replicas of already registered data, syncing updated data, or even only updating a delta if the source material has been appended.

These different settings, in addition to the pre- and post- methods for data object and collection creation and modification, provide a full programmatic surface for writing data preparation policy and harvesting of metadata from external sources on the way to having data ingested into iRODS.

EARLY PERFORMANCE

Initial Ingest

This data is preliminary, and we expect the rates to increase once we have more experience with enterprise configurations and topologies.

However, we see that over 4M files can be ingested by 32 workers in an hour and a half with this initial codebase (Table 3).

What is also notable is that the performance is relatively linear until the catalog provider itself is overwhelmed by the number of incoming concurrent connections (and their subsequent concurrent connections to the underlying iCAT database).

files	workers	minutes	files/worker/minute
1585094	32	36	1375.95
4000000	32	90	1388.89
13564110	32	299	1417.65

Table 3. Three early samples of rate of ingest.

Delta Sync

When scanning the same data source again, the Redis cache handles most of the load and since it is an in-memory data structure store, it never needs to touch the disk or the network. It can handle the lookups very quickly and prevent the system from needing to go to the iRODS catalog except where a source file has changed.

For the most common of loads (a few files per thousand have changed), the delta sync (Table 4) runs nearly 10x as fast as the initial ingest (Table 3).

files	workers	minutes	files/worker/minute
247641	32	0.633	12219.13
2337705	32	6.166	11847.76

Table 4. Syncing data already in Redis is much faster.

SUMMARY

The iRODS Automated Ingest Framework is a new client that is being tested as it is being built. It has been designed to solve a myriad of interesting data ingesting scenarios and scale out to keep up with incoming data rates from batteries of sensors, microscopes, sequencers, and satellites.

Early indications show that the performance is roughly linear as the number of workers scales up.

We are actively looking for additional use cases and look forward to increased community feedback.

REFERENCES

[1] iRODS Capability Automated Ingest. `https://github.com/irods/irods_capability_automated_ingest`

[2] Xu, H., Russell, T., Coposky, J., et al: iRODS Primer 2: Integrated Rule-Oriented Data System. In: Synthesis Lectures on Information Concepts, Retrieval, and Services. 131pp. Morgan Claypool. (2017)

[3] Python iRODS Client. `https://github.com/irods/python-irodsclient`

[4] Celery: Distributed Task Queue. `http://www.celeryproject.org/`

[5] Redis. `https://redis.io/`