iRODS S3 API: Presenting iRODS as S3

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

Violet White
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
vem@renci.org

ABSTRACT

S3 has taken over the storage world for a number of good reasons. Many software libraries, tools, and applications now read and write the S3 protocol directly. This paper describes a new iRODS client API that presents the iRODS namespace as S3. It will discuss the requirements, the design, the initial implementation, and future work.

Keywords

iRODS, s3, api, cpp

DESIRE / JUSTIFICATION

The iRODS Protocol has been around for more than 20 years and has proven reliable and robust. However, it is also relatively complex for users and developers who only want to move data. Amazon's introduction of the Simple Storage Service (S3) [1] in 2006 defined the S3 protocol on top of HTTP(S). Its relatively limited scope of moving and labeling data, while providing online access and near-perfect uptime, resiliency, and availability alongside other new Amazon Web Services (AWS) [2], came to dominate the online storage provider space. Many other tools grew the ability to interact with S3 and now new tooling is expected to be able to do the same upon initial release. In addition to being relatively comprehensible and easy to implement as a client, it also decouples authentication from authorization.

Implementing a new iRODS S3 API [3] would align with our Protocol Plumbing [4] efforts from the last couple years and will make iRODS more accessible and approachable to new developers as well as provide the iRODS Consortium a maintainable, familiar front door for new partners.

Because of this, the iRODS Consortium decided to implement such an API and present iRODS as S3. Our goals were to reuse as much code as possible (from other projects that we have written AND from other projects others have written), to provide a solution that is friendly to be deployed in a high availability (HA) environment, and to be maintainable for future developers.

Possible Futures

In July of 2021, an initial charter was sent to the iRODS Community, beginning the work of the newly designated iRODS S3 Working Group [5]. In it, four options were introduced as candidates for how to present iRODS as S3:

1. Update and maintain the MinIO-iRODS Gateway from BioTeam [6]

   This MinIO-based S3 front-end already existed and had been demonstrated, but had not been updated since its debut in 2018. It was never deployed in production and served only as a proof of concept. It was built with the GoRODS client library [7], which has been archived earlier this year. Presumably, BioTeam would have continued to own/maintain GoRODS and the MinIO-iRODS-gateway if this option was selected.

iRODS UGM 2023 June 13-16, 2023, Chapel Hill, NC

2. MinIO-iRODS-Gateway is converted to use the Go iRODS Client Library [8]

   Illyoung Choi, at the CyVerse project at the University of Arizona, has produced and is actively developing a pure Go iRODS client library. This new library could be ”swapped” for the GoRODS calls in the MinIO-iRODS-gateway. Presumably, then Arizona / Illyoung would own/maintain a fork of the MinIO-iRODS-gateway.

3. Add irods/gateway-irods.go to upstream MinIO project [9]

   Someone in the community (perhaps the Consortium itself) would port or implement the same work from Option 2 (convert to pure Go), but work with the MinIO community to get iRODS to be an officially supported gateway for the MinIO server directly.

4. New C++ implementation [3]

   The iRODS Consortium would work to implement the S3 specification directly with a new C++ client. This could be more performant (in the long run), but requires the most work and will require answers to many open questions.

At the end of this charter, Terrell Russell wrote ”I am leaning towards Option 3 as the best option both from a cost/benefit perspective, as well as exposure to a larger community and confidence that 'it just works'.”

Research

The Consortium investigated these four options and began implementation across four phases over the following two years. Phase 1 covered and discarded the first three options relatively quickly. Phase 2 selected and explored a C++ proof of concept while also investigating many alternative possibilities. Phases 3 and 4 explored multipart data transfer and authentication.

Phase 1

In July of 2021, Option 1's MinIO-iRODS-Gateway with GoRODS wrapping the C library was deemed limited and not as maintainable by either BioTeam or by the Consortium.

Later, the MinIO-iRODS-Gateway with pure go-irodsclient of Option 2 was determined to require an anonymous iRODS ticket functionality to handle the requests. The Consortium investigated and implemented a TicketBooth (or BoxOffice) application in October 2021 to handle creation and dispersal of these anonymous tickets. However, by November 2021 it was clear that such an application would require rodsadmin credentials making the TicketBooth no more functional than just using the C++ REST API directly. In addition, by February 2022, the use case of independently serving multiple users' files was proving impossible because the 'gateway' code fires 'too late' in the MinIO application to discriminate between different incoming users. 'Who' was logging in had already been determined by the MinIO server core.

In May 2022 [10], all of this design and implementation effort became moot when MinIO announced the deprecation of the gateway altogether. Their announcement explained that it was no longer worth it to support 'legacy POSIX-based' systems. Needless to say, this removed MinIO as a viable option.

Phase 2

With the first three options off the table, the Consortium leaned into a new C++ implementation of an S3 API. This implementation would remove dependencies on other codebase(s) and companies and business models. This implementation would need to support multi-user *and* multi-bucket use cases. The Consortium began evaluation and selection of a C++ framework for handling the networking and programming interfaces in August of 2022. The list of candidates included Pistache [11], Oat++ [12], Drogon [13], and Boost.Beast [14]. Boost.Beast was selected in November 2022 and some initial endpoints were functional and responding by January of 2023, including configuration for both user mappings (from keypairs to iRODS users) and bucket mappings (from bucket names to iRODS collections).

Phase 2 - Alternate Illyoung Universes

While the C++ implementation was first coming together, Illyoung Choi at the University of Arizona and the CyVerse project was busy exploring additional alternate possibilities that would prevent the Consortium from having to build and support a novel codebase. The following were considered but eventually all discarded because they could not satisfy the dual requirement for multi-user and multi-bucket support:

- Add S3 protocol support to SFTPGo [16] (August 2022)
- Searching for existing S3 server in Go (September 2022)
- Add iRODS backend to Zenko [17] (October 2022)
- Add JuiceFS [18] frontend to iRODS (November 2022)
- Add JuiceFS frontend to SFTPGo (November 2022)
- GarageHQ [19] frontend to iRODS (January 2023)
- Using in-memory IBM s3mem-go [20] as inspiration (March 2023)

Phase 3

Once the decision was made for a novel C++ implementation of an S3 API, and the initial endpoints were coming together relatively quickly, attention was turned to moving data from a client into the API efficiently. This means addressing the Multipart Upload endpoint.

There were four options that the Consortium could see for implementing the S3 protocol's multipart upload. These were named 'Multiobject', 'Store-and-forward', 'Efficient Store-and-forward', and 'Store-and-register'.

The first of these, Multiobject, would write all uploaded parts individually to iRODS, and then a final step or 'complete' would trigger copy or concatenate or some other function in the server to stitch all the parts together as a single file in the iRODS namespace. The main benefit to this approach is that it is relatively simple and straightforward. However, the downsides are that a lot of additional policy would fire on the server (opens, reads, writes, closes for every part) which may trigger replication to distant, expensive disks (or continents). It also would require a new API plugin or function in the server to 'concatenate()' the individual parts.

The second option was Store-and-forward. This approach would write all the parts to storage local to the API, as a single file, and then send that entire file to iRODS where it would be written into managed storage. This is also very simple to implement, and does not fire additional policy in the server (the file is only written once in iRODS). However, since the file has to be sent 'twice', the client will 'see' a slow or delayed experience. Additionally, the API itself would have to provide potentially an enormous disk, realistically large enough to support the largest incoming file multiplied by the number of concurrent users.

A third option would be an opportunistic Efficient Store-and-forward algorithm that has a fallback of regular Store-and-forward. If the parts that are being uploaded are known to be contiguous, then they can be sent along to iRODS while any non-contiguous parts can be held in memory (or disk) in the API until enough parts have arrived that they too can be considered contiguous and eligible for sending to iRODS. This is elegant and would only require a single write to iRODS but has more complexity in the code since there are more states to be managed. This approach would also still require a relatively large disk for scratch space since if an early part was not delivered, no additional parts could be sent to iRODS until the earlier part had arrived. It is also noted here that the S3 protocol reserves the right to send a part number more than once which means that the S3 API would have to be able to recover in that scenario. This might only be possible if a resent part number is guaranteed to be exactly the same size as the initial part sent with that part number. If this is not the case, then no efficiency can be gained by this option over naive Store-and-forward.

The fourth option brainstormed was Store-and-register. In this scenario, the S3 API would write all the parts into their final order in storage that is also visible to iRODS. Then, the S3 API would issue a 'register' operation to have the file added as a replica to the iRODS catalog in-place. This is the simplest and the fastest but would only trigger the 'register' policy on the server. And, in addition, this adds a significant assumption and/or requirement of co-visibility of the S3 API's storage to the iRODS server.

These four options all have their upsides and their downsides and it was not clear which approach should be implemented first. It was decided that an initial release would not include multipart upload as more research needed to be done.

Phase 4

In June of 2023, with most of the endpoints implemented (see the next section), and multipart uploads set aside for a later date, there are now tests written and passing with the Python boto3 [15] client. However, more work needs to be done with the HMAC signature matching for other clients (AWS cli, MinIO's mc, etc.)

STATUS / IMPLEMENTATION

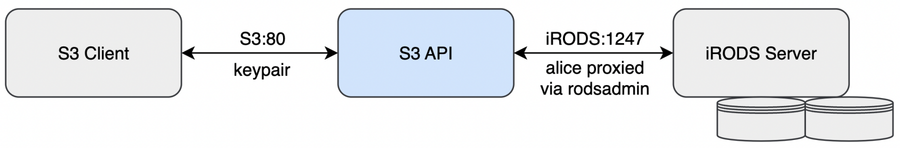

Figure 1. Architecture of the iRODS S3 API

The first implementation of the iRODS S3 API is a single binary currently requiring rodsadmin credentials (so it can proxy as the authenticated user). It has two configuration files, currently `irods_environment.json`, to define the iRODS environment and connection information, and `config.json`, to control the S3 features. This will probably be consolidated into a single configuration file as the settings mature and settle.

Architecture

The implemented endpoints at the time of this writing include:

- CopyObject
- DeleteObject
- GetObject
- HeadObject
- ListObjectsV2
- PutObject

The multipart endpoints are still under discussion, but include:

- CompleteMultipartUpload

- CreateMultipartUpload
- UploadPart

The Consortium has not yet looked into the following endpoints. They may be implemented later depending on community feedback and/or existing client implementations that require these endpoints:

- UploadPartCopy
- ListObjects
- DeleteObjects
- GetObjectAcl
- PutObjectAcl
- GetObjectTagging
- PutObjectTagging

Configuration

As stated, the configuration file for the S3 API will be consolidated and have 'two sides', one for the `s3_server`, and one for the `irods_client`.

```
{
    // Defines S3 options that affect how the
    // client-facing component of the server behaves.
    "s3_server": {
        // ...
    },

    // Defines iRODS connection information.
    "irods_client": {
        // ...
    }
}
```

The `s3_server` side will contain host and port information, control for logging level and the mapping system for both users and buckets. Initial designs for the mapping system will include a static resolver for both users and buckets, directly in this file, however, later this information could come from an external source and provide dynamic resolution for production deployments.

```
"s3_server": {
    "host": "0.0.0.0",
    "port": 80,

    "log_level": "warn",

    "bucket_mapping": {
        # local / static (JSON)
        # external / dynamic - iRODS or third party API
    },

    "user_mapping": {
        # local / static (JSON)
        # external / dynamic - iRODS or third party API
    },

    "threads": 3
}
```

The `irods_client` side contains the standard iRODS connection information including the host, port, and zone name, as well as the rodsadmin account information and some connection pooling information for efficiency when communicating with the iRODS server.

```
"irods_client": {
    "host": "<string>",
    "port": 1247,
    "zone": "<zone>",

    "proxy_rodsadmin": {
        "username": "<string>",
        "password": "<string>"
    },

    "connection_pool": {
        "size": 6,
        "refresh_timeout_in_seconds": 600
    }
}
```

NEXT STEPS

The next steps for this project include configuration consolidation and HMAC cleanup and confirmation. An initial release will require packaging as well as potential deployment instructions. Before that happens, testing will need to be done with multiple clients, perhaps in parallel, and even as an iRODS S3 resource. And of course, additional mappings for both user and bucket will be explored to see what satisfies the community's needs and expectations.

CONCLUSION

This new iRODS S3 API is exciting and will open the iRODS policy ecosystem to numerous additional existing S3 clients. Multiple systems will be able to take advantage of an iRODS Zone without needing to write code or learn the iRODS protocol. This new API is a key part of the Consortium's efforts around Protocol Plumbing.

REFERENCES

[1] AWS S3. `https://aws.amazon.com/s3`

[2] Amazon Web Services. `https://aws.amazon.com`

[3] iRODS S3 API. `https://github.com/irods/irods_client_s3_api`

[4] Russell, T. (2022) Protocol Plumbing: Presenting iRODS as WebDAV, FUSE, REST, NFS, SFTP, K8s CSI, and S3. Supercomputing 2022. `https://slides.com/irods/sc22-irods-protocol-plumbing`

[5] iRODS S3 Working Group. `https://github.com/irods-contrib/irods_working_group_s3`

[6] `https://github.com/bioteam/minio-irods-gateway`

[7] `https://github.com/jacquayj/GoRODS`

[8] `https://github.com/cyverse/go-irodsclient`

[9] `https://github.com/minio/minio/tree/master/cmd/gateway`

[10] `https://blog.min.io/deprecation-of-the-minio-gateway/`

[11] `https://github.com/pistacheio/pistache`

[12] `https://oatpp.io/`

[13] `https://drogon.org/`

[14] `https://github.com/boostorg/beast`

[15] `https://boto3.amazonaws.com/v1/documentation/api/latest/index.html`

[16] `https://github.com/drakkan/sftpgo`

[17] `https://github.com/scality/Zenko`

[18] `https://juicefs.com/en/`

[19] `https://garagehq.deuxfleurs.fr/`

[20] `https://github.com/IBM/s3mem-go`