iRODS®

iRODS S3 API v0.2.0
with Multipart

Justin James
Applications Engineer
iRODS Consortium

May 28-31, 2024
iRODS User Group Meeting 2024
Amsterdam, Netherlands

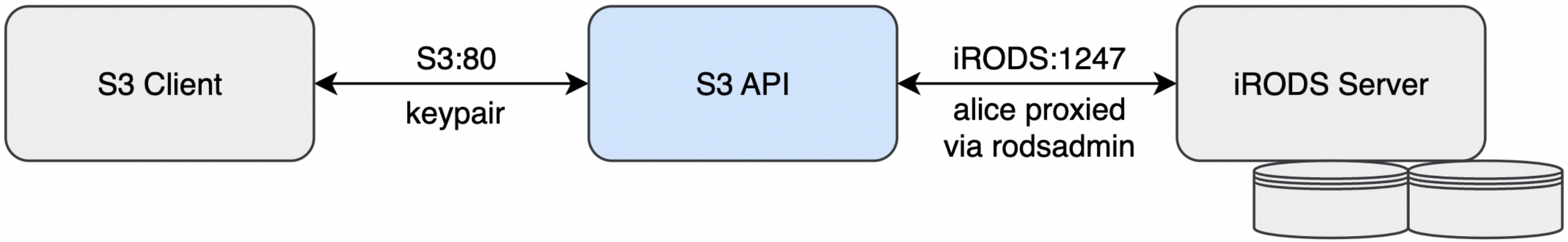
iRODS S3 API - Goals

- Present iRODS as the S3 protocol
  - Multi-user
  - Multi-bucket
- Load Balancer friendly
- Maintainable

iRODS S3 API - History

- iRODS S3 Working Group
  - https://github.com/irods-contrib/irods_working_group_s3
  - Initial email 2021073
- 2023 - Violet White implemented many of the endpoints
- v0.1.0 released Nov. 2023
- v0.2.0 released March 2024

iRODS S3 API - Architecture and Status



- Single binary
- Single configuration file
- Multi-user
- Multi-bucket
- Requires rodsadmin credentials

- Tests passing with:
  - AWS CLI Client
  - Boto3 Python Library
  - MinIO Python Client
  - MinIO CLI Client

iRODS S3 API - Changes Since UGM 2023

1. Added a docker-based testing framework
2. HMAC/Signatures now working for all clients
3. Implemented multipart uploads
4. Code migrated to framework used by HTTP API
    - coroutines removed / foreground and background task queues
    - boost::beast::http calls changed from sync to async
5. CMake files updated for consistency with other projects
6. Added support for range headers for *GetObject*
7. Refactor / reverse engineered *ListObjectsV2* to work exactly as Amazon S3
8. Updated code to handle HTTP chunked encoding
9. Implemented new endpoints
    - *HeadBucket*
    - *GetBucketLocation*
    - *GetObjectLockConfiguration*
    - *ListBuckets*
    - *GetObjectTagging*

iRODS S3 API - Status

- Implemented Endpoints
  - CompleteMultipartUpload
  - CopyObject
  - CreateMultipartUpload
  - DeleteObject
  - DeleteObjects
  - GetBucketLocation
  - GetObject
  - GetObjectLockConfiguration (stub)
  - GetObjectTagging (stub)
  - HeadBucket
  - HeadObject
  - ListBuckets
  - ListObjectsV2
  - PutObject
  - UploadPart

- Investigating
  - ListObjects
  - GetObjectAcl
  - PutObjectAcl
  - PutObjectTagging
  - UploadPartCopy
  - AbortMultipartUpload

iRODS S3 API - Configuration

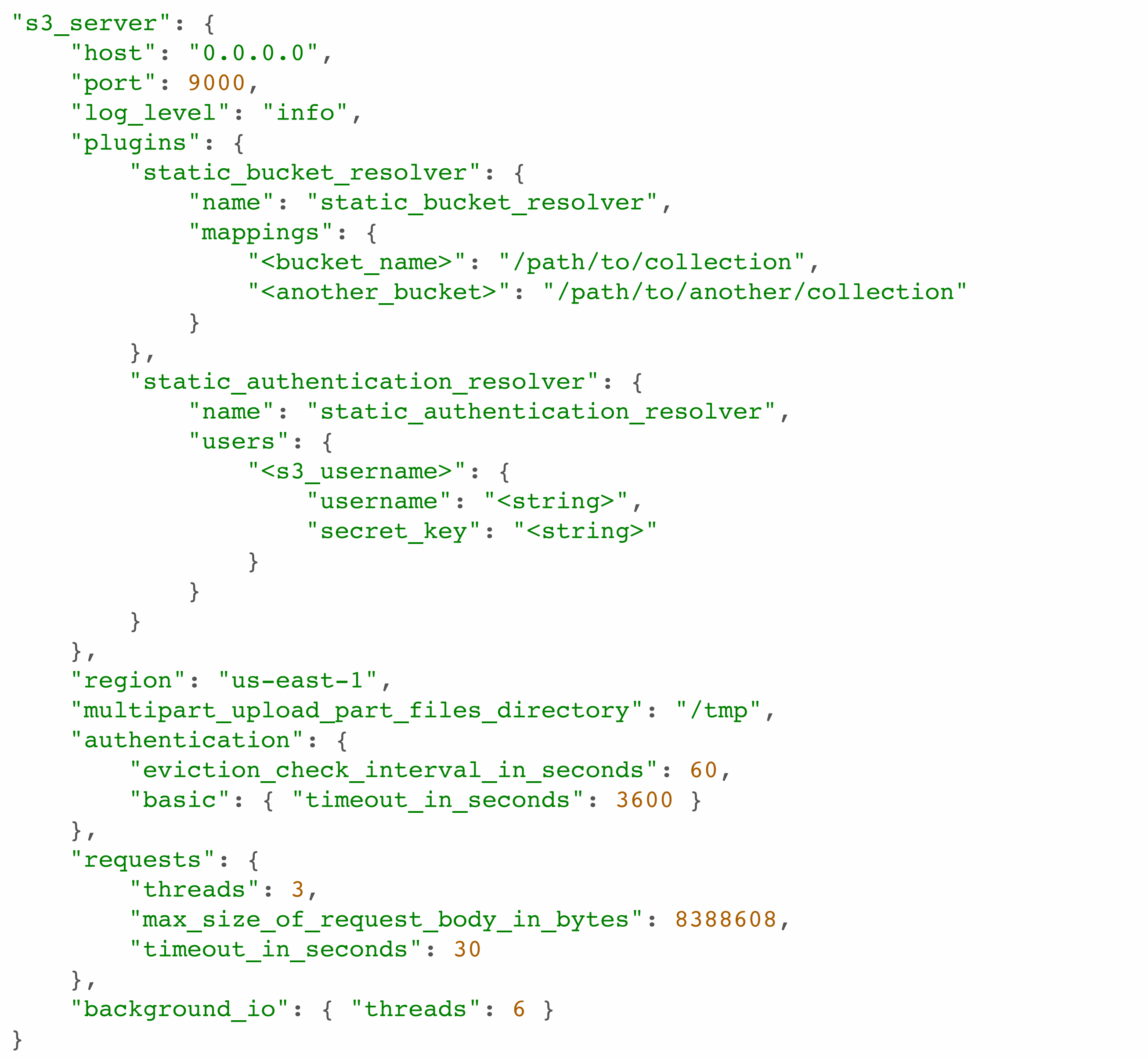
Single file which defines two sections to help administrators understand the options and how they relate to each other.

Modeled after NFSRODS.

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

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

iRODS S3 API - Configuration - s3_server



```
"s3_server": {
    "host": "0.0.0.0",
    "port": 9000,
    "log_level": "info",
    "plugins": {
        "static_bucket_resolver": {
            "name": "static_bucket_resolver",
            "mappings": {
                "<bucket_name>": "/path/to/collection",
                "<another_bucket>": "/path/to/another/collection"
            }
        },
        "static_authentication_resolver": {
            "name": "static_authentication_resolver",
            "users": {
                "<s3_username>": {
                    "username": "<string>",
                    "secret_key": "<string>"
                }
            }
        }
    },
    "region": "us-east-1",
    "multipart_upload_part_files_directory": "/tmp",
    "authentication": {
        "eviction_check_interval_in_seconds": 60,
        "basic": { "timeout_in_seconds": 3600 }
    },
    "requests": {
        "threads": 3,
        "max_size_of_request_body_in_bytes": 8388608,
        "timeout_in_seconds": 30
    },
    "background_io": { "threads": 6 }
}
```

iRODS S3 API - Configuration - irods_client

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

    "tls": { /* ... options ... */ },

    "enable_4_2_compatibility": false,

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

    "connection_pool": {
        "size": 6,
        "refresh_timeout_in_seconds": 600,
        "max_retrievals_before_refresh": 16,
        "refresh_when_resource_changes_detected": true
    },

    "resource": "<string>",
    "max_number_of_bytes_per_read_operation": 8192,
    "buffer_size_in_bytes_for_write_operations": 8192
}
```

iRODS S3 API - Multipart Options Considered

A. Multiobject - Parts written as separate objects. On *CompleteMultipartUpload*, parts are concatenated on the iRODS server.

- Efficient
- Unintentional execution of policy for each part
- Pollutes iRODS namespace
- Would require a concatenate API plugin

B. Store-and-Forward - Write each part to the mid-tier, then forward to iRODS on *CompleteMultipartUpload*.

- No extra policy triggered
- Requires a large amount of scratch space in the mid-tier
- Non-trivial *CompleteMultipartUpload*

C. Efficient Store-and-Forward - Write down / hold non-contiguous parts in the mid-tier, then send contiguous parts to iRODS when ready.

- Complicated - parts are not necessarily sent in order and can be resent
- Do not know part offsets so could only forward when all previous parts have been written
- Worst case almost the entire object would still need to be stored in the mid-tier

D. Store-and-Register - Write to a file accessible to iRODS and register when complete.

- Still requires writing individual part files since we do not know the part offsets
- Requires shared visibility between iRODS and S3 API

iRODS S3 API - Multipart Store-and-Forward

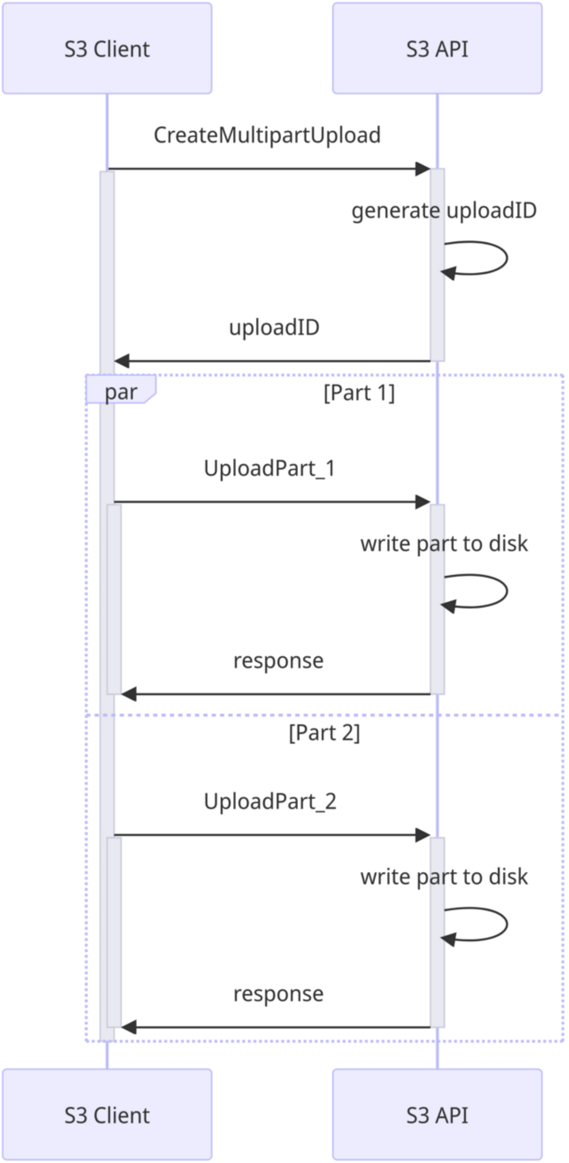
For now we have chosen the store-and-forward approach.

1. *CreateMultipartUpload*

- Generate a UUID for the upload_id
- Return the upload_id in the response.

2. *UploadPart*

- Write bytes to a local file *<multipart_upload_part_files_directory>/irods_s3_api_<upload_id>.<part_number>*



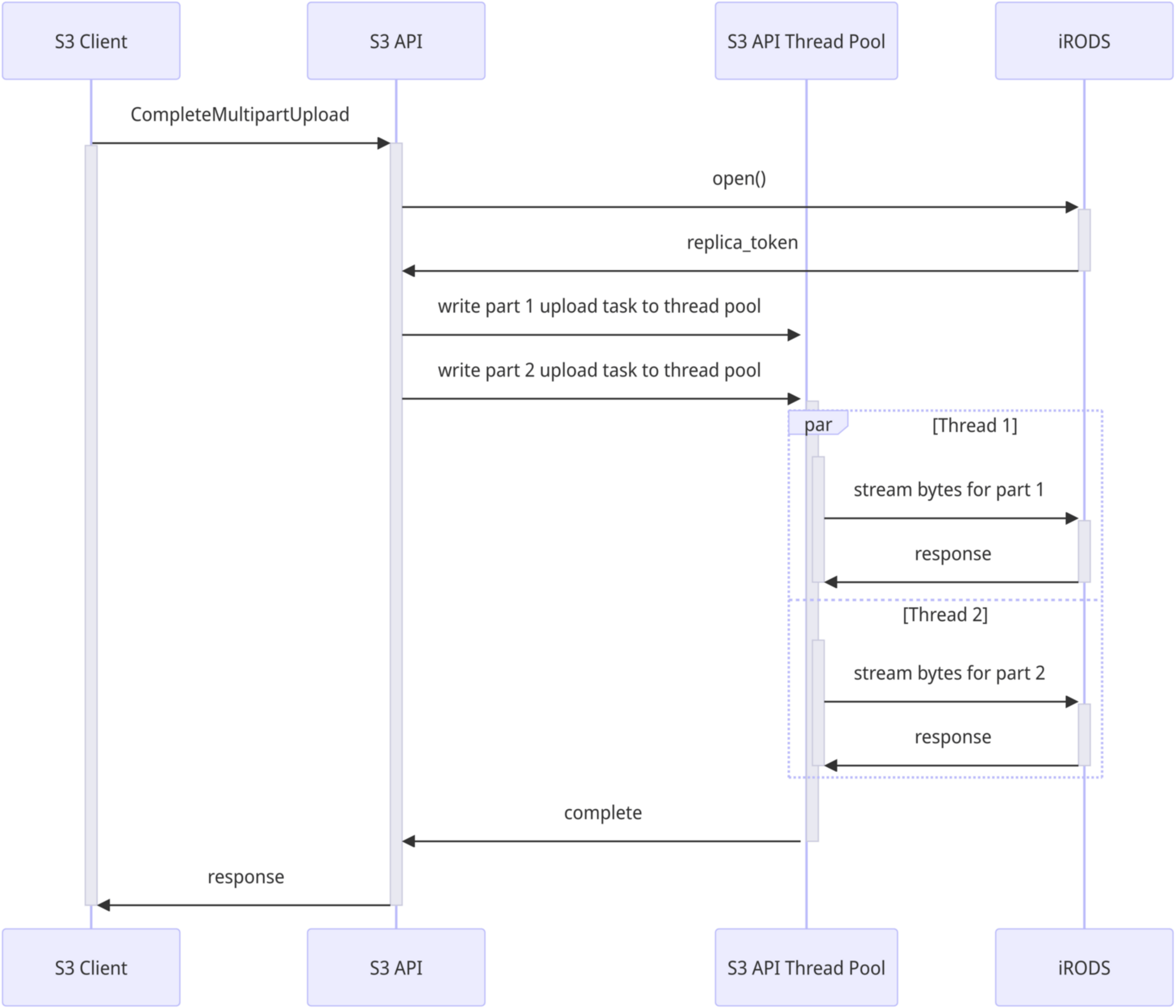
iRODS S3 API - Multipart Store-and-Forward

3. *CompleteMultipartUpload*

- Create the object in iRODS
- Determine the offset for each part
- Iterate through the parts and create tasks on the thread pool to upload parts to iRODS.
- When all parts are done, remove part files and send response to the client
- In a pure S3 environment, CompleteMultipartUpload does not move data and usually completes quickly. That is not the case here.

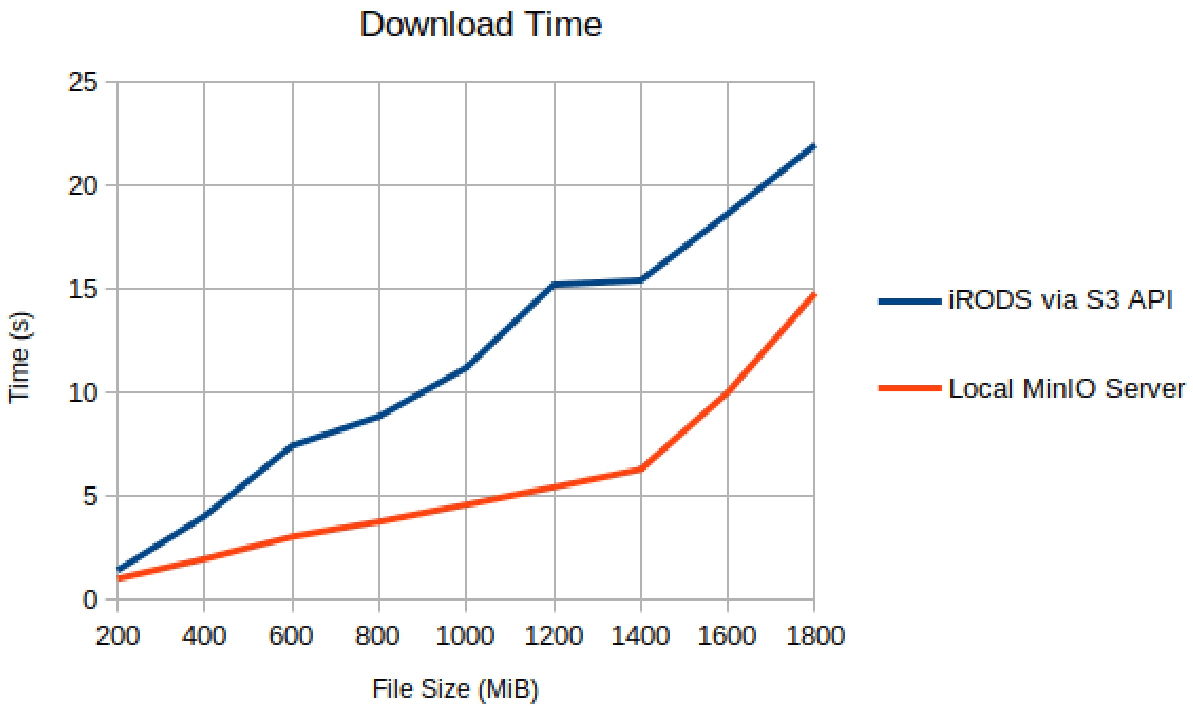
4. *AbortMultipartUpload* (not yet implemented)

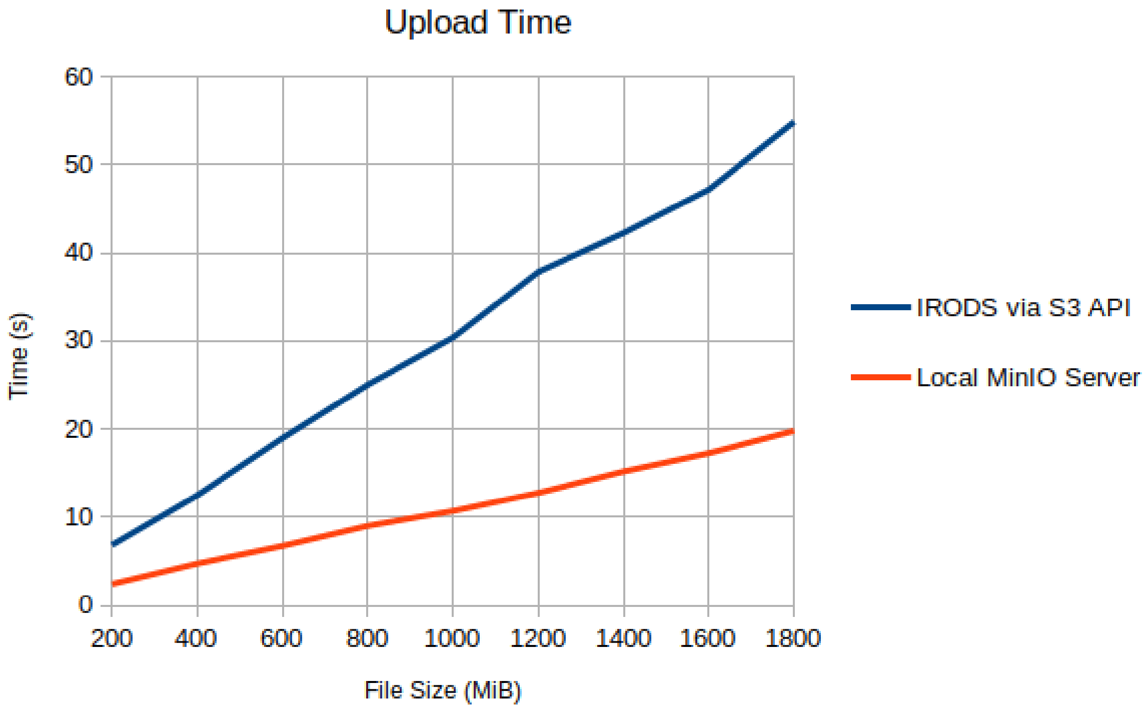
- Remove part files from mid-tier



iRODS S3 API - Performance Comparison

The following compares transfers to/from iRODS via the S3 API with transfers to/from a local MinIO server. The Boto S3 client was used for all cases.





Notes:

- The tests consisted of transfers of files from 200 MB to 1800 MB.
- The median of five runs is reported for each file size.
- Multipart uploads require two read/write cycles with store-and-forward.
- The S3 API was configured with 30 threads handling requests and 30 background threads.
- Performance degraded with large files when there was an insufficient number of background threads.

iRODS S3 API - Multipart - Enhancement - Efficient Store and Forward

In the future we may migrate to the efficient store-and-forward approach.
The design is not finalized but the following is a possible approach.

- As *UploadPart* requests are received, store the Content-Length in a table along with an upload status which is one of the following:
  - WRITING_TO_DISK
  - STREAMING_TO_IRODS
  - DONE_WRITING_TO_DISK
  - FINISHED

- When receiving the *UploadPart* request for part N, if the Content-Length of parts [1, N-1] are known, open a stream to iRODS, perform the appropriate seek(), and stream directly to iRODS. Set the status to STREAMING_TO_IRODS. When finished streaming set the status to FINISHED.

  Note that if the client uses chunked parsing, the Content-Length is not known until all chunks are parsed.

- If previous part sizes are not known, part N must be written to disk in the mid-tier. Set the status to WRITING_TO_DISK. Once the part has been completely written to disk, set the status to DONE_WRITING_TO_DISK.

- Have a background task examine all parts that are DONE_WRITING_TO_DISK. If all previous parts are of known size, set the status to STREAMING_TO_IRODS and begin streaming that part to iRODS. Once finished, set the status to FINISHED.

- *CompleteMultipartUpload* simply needs to wait until all parts are marked FINISHED.

Open question:

- What if a part is resent with a different size?

iRODS S3 API - Multipart - Enhancement - Store and Register

Another approach for an enhancement would be store-and-register where the initial part files are written to the iRODS server, combined into one file, then registered.

- An API would need to be created to write part files to the iRODS server. At this point these would simply be POSIX files, not data objects.
- During *CompleteMultipartUpload*, another API would be used to call copy_file_range() to concatenate all part files into one and register this new file in iRODS.

This approach has some challenges:

- This would not execute policy until registration. This is both good and bad. How do we enforce that the file is written to the correct server?
- What if the S3 API is configured to write to an object store?
- While it would be significantly faster than the current approach, *CompleteMultipartUpload* would still have to wait until the part files could be combined into one file.

One approach is to offer the option of efficient store-and-forward and store-and-register with caveats on how store-and-register can be used.

iRODS S3 API - Next Steps

- More testing / expand testing framework
- Implement multipart efficient store-and-forward
- Additional endpoints
  - Tagging
  - ACLs
- Additional plugins
  - Other bucket mappings
  - Other user mappings

iRODS S3 API

Release v0.2.0

- https://github.com/irods/irods_client_s3_api
- March 6, 2024
- Available via Docker Hub

Thank you!

Questions?