S3 Resource Plugin Cacheless and Detached

Justin James
Applications Engineer
iRODS Consortium

June 25-28, 2019
iRODS User Group Meeting 2019
Utrecht, Netherlands

Introduction (Legacy Operation)

The legacy S3 plugin must be used in conjunction with a compound resource and a unixfilesystem cache resource.

The following is a sample hierarchy of the S3 plugin.

```
s3compound:compound
├── s3archive:s3
└── s3cache:unixfilesystem
```

This required the iRODS administrator to create a cache cleanup rule.

The S3 plugin itself only implemented a few operations:

- irods::RESOURCE_OP_UNLINK
- irods::RESOURCE_OP_STAT
- irods::RESOURCE_OP_RENAME
- irods::RESOURCE_OP_STAGETOCACHE
- irods::RESOURCE_OP_SYNCTOARCH

All of the other operations were handled by the cache resource.

Introduction (New Modes of Operation)

The new plugin now supports three operating modes

	Archive	**Cacheless**
Attached	archive_attached (default)	cacheless_attached
Detached	N/A	cacheless_detached

This mode is set using the **HOST_MODE** parameter in the resource context string.

If the **HOST_MODE** is not set, the default is archive_attached, which operates as the legacy S3 plugin.

Note that "archive_detached" is not a valid entry.

Introduction (Archive vs Cacheless)

- Archive
  - The S3 resource acts in the archive role behind a compound resource.
    - Requires a cache resource which provides POSIX semantics.
    - Must be attached to a specific iRODS server.

- Cacheless
  - The S3 resource can be standalone.
    - May be detached from any specific iRODS server (see next slide).
    - The S3 plugin provides POSIX semantics with no cache resource and requires no explicit cache management policy.

Introduction (Attached vs Detached)

- Attached
  - Only the server that is defined as the host in the resource configuration will serve the request.

- Detached
  - All iRODS servers may serve a request for an object. This is appropriate if all servers have connectivity to the S3 backend.

Creating a Cacheless S3 Resource

Creating a cacheless S3 resource is very similar to creating a legacy/archive S3 resource.

As stated previously, the only differences is that the cacheless S3 may be a standalone resource and the HOST_MODE must be set to either "cacheless_attached" or "cacheless_detached".

The following is an example of creating a cacheless/attached S3 resource to Amazon S3.

```
iadmin mkresc s3resc s3 `hostname`:/irods-bucket/irods/Vault
"S3_DEFAULT_HOSTNAME=s3.amazonaws.com;S3_AUTH_FILE=/var/lib/irods/s3.keypair;S3_REGIONNAME=us-east-
1;S3_RETRY_COUNT=1;S3_WAIT_TIME_SEC=3;S3_PROTO=HTTP;ARCHIVE_NAMING_POLICY=consistent;HOST_MODE=cache
less_attached"
```

Implementation Details

To implement the cacheless S3 resource, we implemented many of the operations that the archive S3 resource had not implemented.

These include:

- irods::RESOURCE_OP_OPEN
- irods::RESOURCE_OP_READ
- irods::RESOURCE_OP_WRITE
- irods::RESOURCE_OP_CLOSE
- irods::RESOURCE_OP_LSEEK

Implementation Details

As a starting point we started with S3FS which is an open source FUSE mount point for S3.

https://github.com/s3fs-fuse/s3fs-fuse

- First we validated that S3FS was stable. We created an S3FS mount point and put demoResc's vault inside this mount point and ran a suite of tests. All the tests passed except for a few bundle test cases.

- This convinced us that S3FS was a good starting point for our plugin.

Implementation Details

The next step was to translate the FUSE operations to iRODS resource plugin operations.

The iRODS resource plugin operations follow POSIX semantics instead of FUSE semantics. To implement this we need to store additional state information about every open file:

- Create our own file descriptors when a client opens a file.
- Create an offset (off_t) for each open file to store the offset within the file.
  - Adjust this offset after reads, writes, seeks, etc.
  - SEEK_SET, SEEK_END, SEEK_CUR simply adjust the offset.

Implementation Details (Dealing with Global Variables)

S3FS uses a lot of global variables for S3 configuration, internal data structures, etc.

- These can conflict when using parallel uploads and downloads or when writing to more than one resource within the same agent (two S3 resources under a replication node).

- Care was taken to mitigate this by:
  - Using thread_local variables so that each thread has its own values.
  - Storing configuration in the irods::plugin_property_map.

Implementation Details (Download Peformance)

Problem:

- When iRODS does large file / parallel downloads, the plugin receives requests for bytes in a seemingly random order.
- If these individual download requests are performed separately, this does not optimize the S3 multipart download performance.
- S3FS core code does read-ahead and will retrieve more than is requested which helps download perfomance but it was still much slower than using the S3 CLI API.

Goals:

- Want full file downloads to be reasonably close to the performance when using the S3 CLI API.
- Want to be able to quickly service small requests in large files. (Do not download entire 100GB file when only requesting 1K of data.)

Implementation Details (Download Peformance)

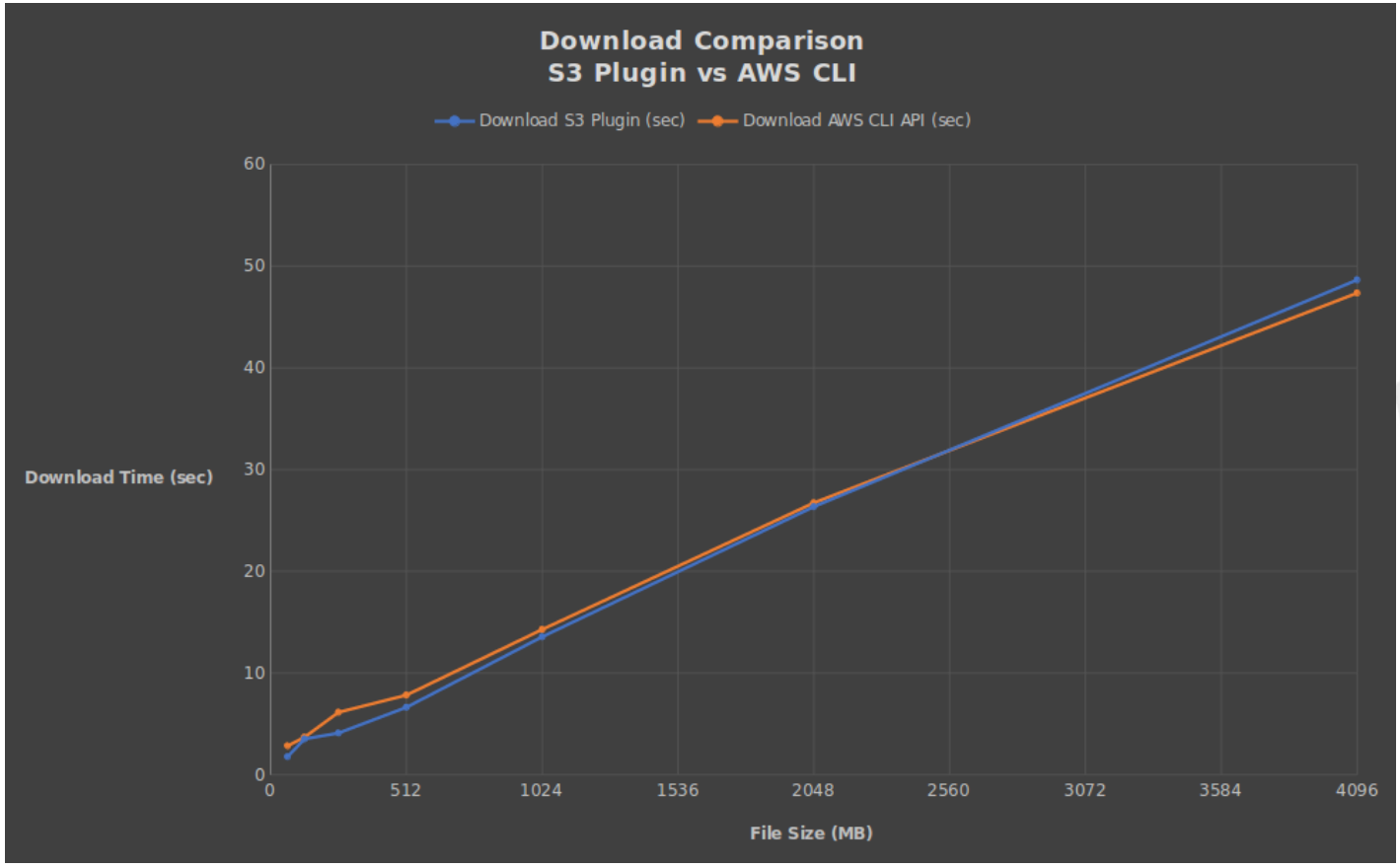
Solution:

- First read on an file only downloads the requested bytes.
- On the second and subsequent simultaneous read requests, this means we are likely doing a parallel download request of a full file.
  - At this point start a full multipart download.
  - All threads will wait for a notification that a multipart chunk has been downloaded.
  - On each multipart completion, the threads will check if their bytes have been downloaded. If so, return these bytes. If not, wait for the next notification.

Implementation Details (Download Peformance)

Results:

- Downloads times are very close to downloads using the S3 API using the same S3_MPU_CHUNK size and S3_MPU_THREAD count.
- If user only requests a small part of a large file, this is returned quickly. No full file download is performed.



Implementation Details (Upload Peformance)

Problem:

- When uploading files to S3, the full file must be uploaded.
- For large files, iRODS sends multiple data buffers in parallel. A naive approach would be to write these buffers directly to S3 but this requires rewriting parts of the file over and over ($O(n^2)$).

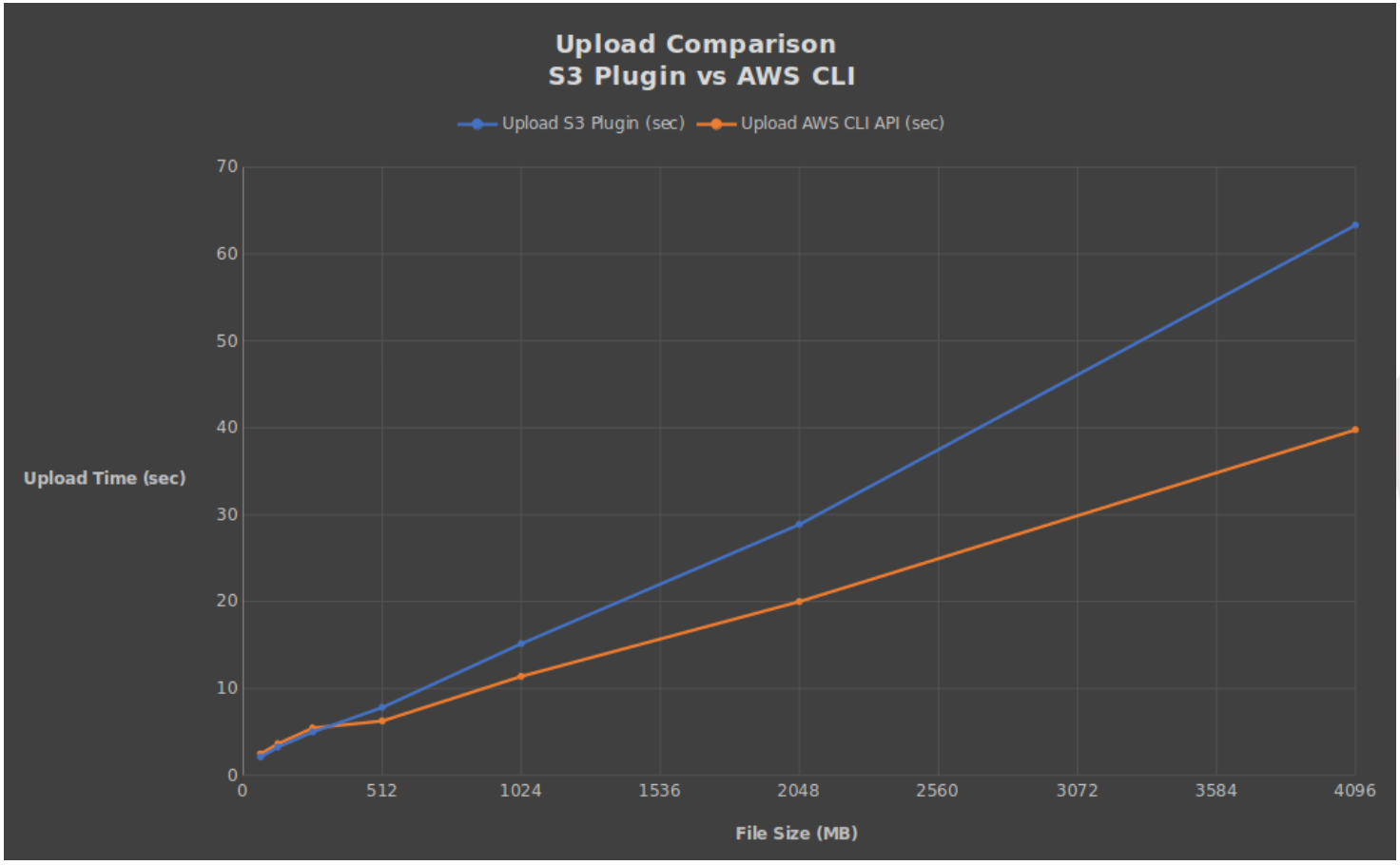
Solution:

- While the file is opened for writes, the writes are written to the local cache file.
- When the last close() is performed, flush the file to S3.

Implementation Details (Upload Peformance)

Results:

- File uploads are slightly slower using the S3 plugin than using the AWS CLI API but significantly better than the naive approach.
- We will investigate why this is the case and try to improve this performance in the next release of the S3 plugin.



Demonstration of the S3 Plugin

We will create a cacheless S3 resource. First, some assumptions:

- The access key and secret access keys have been saved to /var/lib/irods/news3resc.keypair with a newline separating the access key and secret access key.
- The S3 plugin has been installed on the server.
- A bucket named justinkylejames-irods1 has been created.

First let's create a cacheless S3 plugin

```
iadmin mkresc news3resc s3 `hostname`:/justinkylejames-irods1/irods/Vault
"S3_DEFAULT_HOSTNAME=s3.amazonaws.com;S3_AUTH_FILE=/var/lib/irods/news3resc.keypair;S3_REGIONNAME=us-east-
1;S3_RETRY_COUNT=1;S3_WAIT_TIME_SEC=3;S3_PROTO=HTTP;ARCHIVE_NAMING_POLICY=consistent;HOST_MODE=cacheless_at
tached"
```

Demonstration of the S3 Plugin

Create a simple test file and put it to the S3 resource:

```
$ echo 'this is a test file' > test.txt
$ iput -R news3resc test.txt
```

Using the S3 commands, check that the file exists:

```
$ aws s3 ls s3://justinkylejames-irods1/irods/Vault/home/rods/
2019-02-18 14:55:44         20 test.txt
```

Get the file using iRODS:

```
$ iget test.txt -
this is a test file
```

Demonstration of the S3 Plugin

Rename the data object and check that it has been renamed in S3:

```
$ imv test.txt newname.txt
$ ils -L
/tempZone/home/rods:
  rods              0 news3resc           20 2019-02-18.14:55 & newname.txt
        generic    /justinkylejames-irods1/irods/Vault/home/rods/newname.txt
$ aws s3 ls s3://justinkylejames-irods1/irods/Vault/home/rods/
2019-02-18 15:23:24         20 newname.txt
```

Remove the file:

```
$ irm -f newname.txt
$ ils
/tempZone/home/rods:
$ aws s3 ls s3://justinkylejames-irods1/irods/Vault/home/rods/
```

Demonstration of the S3 Plugin

Now for a more comprehensive test, I've created a file that is large enough to use the parallel file transfers. I have already created a 64M file that contains data that I will put into iRODS.

```
$ iput -R news3resc 64Mfile
```

Get the file:

```
$ iget 64Mfile 64Mfile2 -f
```

Compare the file we retrieved (64Mfile2) to the original file (64Mfile):

```
$ diff 64Mfile 64Mfile2
$ cksum 64Mfile 64Mfile2
1941261876 67108864 64Mfile
1941261876 67108864 64Mfile2
```

Next Steps

The cacheless S3 plugin has passed all CI tests. There are still some improvements to be made.

- If possible, improve the upload performance to mirror the AWS CLI performance.
- Some legacy S3 features have either not been implemented or or not tested (but still work in legacy/cache mode).
  - Comma separated list for S3_DEFAULT_HOST
  - ARCHIVE_NAMING_POLICY flag
- Enhance the S3 authentication options so that the credentials may be stored in the catalog or some other service like vault.
- Implement the RESOURCE_OP_READDIR operation which is used in things like recursive registrations.
- We plan to implement cacheless plugins for other iRODS archive resources (WOS, etc.)

Questions?