S3:TNG - iRODS S3 Resource Plugin with Direct Streaming

Justin James
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
jjames@renci.org

Kory Draughn
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
korydraughn@renci.org

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

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

ABSTRACT

The iRODS S3 storage resource plugin has become very important to the iRODS ecosystem. Many production systems are now spanning local disk, local or remote object stores, and tape. Last year's release of the cacheless S3 plugin enjoyed immediate uptake.

This year's update shares the design and engineering underway for the iRODS S3 plugin to provide direct streaming into and out of S3-compatible storage. This rewrite uses the new iRODS IOStreams library[1] and in-memory buffering to make efficient multi-part transfers.

Keywords

iRODS, S3, AWS, streaming, multipart, data management

INTRODUCTION

iRODS has provided an interface to S3-compatible[2] storage since iRODS 2.2[3] through the compound resource (with child resources designated in the roles of cache and archive). Last year, we introduced the work to provide a 'cacheless' connection to S3-compatible storage that did not require a compound resource as parent[4]. This was better for configuration (fewer moving parts), performance (no additional replica needed), as well as for cost (no additional replica needed).

This year's progress revisits some remaining assumptions and addresses performance through direct streaming.

PRIOR LIMITATIONS AND MOTIVATION

While the existing cacheless plugin did not require a compound resource with an archive and cache resource, it still used cache files at the OS level. Because the S3 multipart protocol is different from the parallel transport mechanism in iRODS, the S3 plugin still collected the entire file 'locally' from one protocol before sending it on its way using the other protocol. This year's work addresses this performance bottleneck, but not completely. Some scenarios still require a local cache file.

Performance was also limited by having to read an entire object from S3, write to the local disk, and flush the object back to S3. This required multiple trips to the local disk, both for read and write.

iRODS UGM 2020 June 9-12, 2020, Virtual

In some cluster cases, communicating with the S3 endpoint is faster than the communicating with the local disk which means the performance is further limited by the performance of the local disk.

One last limitation of the cacheless S3 plugin is that it does not support the `dstream` interface directly.

Figure 1. Ongoing evolution of the iRODS S3 Plugin

This paper shares how the iRODS S3 Resource Plugin is migrating to a streaming plugin that streams connections directly from iRODS to the S3 backend and vice versa with as few interactions with the local disk as possible.

IMPLEMENTATION

In the new streaming S3 plugin, all reads and writes are handled by an `s3_transport` class which extends `irods::experimental::io::transport`.

As the primary goal is to remove the use of any local cache file, here we will discuss some different usage scenarios.

For normal gets and puts, no cache file is used. When the `RESOURCE_OP_READ` operation is called, a read is called to the `dstream` object. When a `RESOURCE_OP_WRITE` operation is called, this data is streamed directly to S3 via the `dstream` object. During this write, if a parallel transfer is performed in iRODS, a multipart upload is started to S3 and each transfer thread streams data directly to S3 for its part. If a single buffer write is being performed, then multipart is not used and data is streamed sequentially to S3.

In some circumstances covered later, a local cache file will still be used.

The iRODS S3 plugin is no longer using the `S3FS`[5] libraries. Both the cacheless and cache versions use `libs3`[6] directly which allows for more fine-grained control of the underlying transfer mechanisms.

The `s3_transport` code in this plugin is a proving ground for two new libraries that have been recently added to the iRODS core. The first is a space-limited circular buffer[7] with notifications for threads waiting to read and write. The other is the use of `dstream` within the data movement layer in iRODS.

PARALLEL PUT

When an iRODS object is opened in write only mode with the truncate flag set, a full file upload (PUT) is being performed. This is the 'usual' case for when files are put into iRODS.

Each thread creates the `dstream` and `s3_transport` objects when it receives the first call to the `RESOURCE_OP_WRITE` operation (Figure 2). The very first `s3_transport` object that is opened calls the S3 function `CreateMultipartUpload`. The `RESOURCE_OP_WRITE` operation simply forwards to `dstream.write()` which calls `s3_transport.send()`. On the `s3_transport.send()`, the data is written to a per-thread, in-memory circular buffer. The `s3_transport` object creates a thread to read the data from the buffer and stream it to S3.

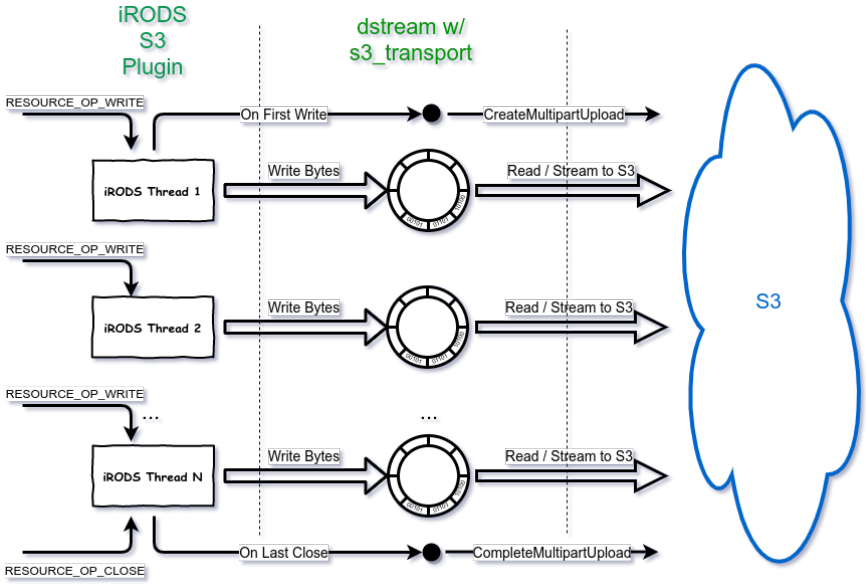

Figure 2. Streaming Parallel PUT

Circular Buffer

The blocking circular buffer is designed to improve performance and limit the amount of memory that is used when uploaded very large files. If the circular buffer is full, the `s3_transport.send()` waits until it can complete the write operation. The thread which reads from the circular buffer and streams to S3 will wait if the buffer is empty.

Each parallel transfer process/thread has its own circular buffer.

The size of the circular buffer is set in the resource context's `S3_CIRCULAR_BUFFER_SIZE` parameter. This size is in entries, not bytes. Each entry is equal to the size of the buffer sent to the plugin. The maximum number of bytes that may be used per iRODS Agent is `numberThreads * bufferSize * numberEntries`.

PARALLEL GET

When an object is opened in read-only mode (GET), the requested bytes are simply read from the S3 object. This is the 'usual' case for when files are retrieved from iRODS.

S3 allows random access reads for objects, so a call to `RESOURCE_OP_READ` translates directly to an S3 request to `GetObject` (Figure 3). Each thread creates the `dstream` and `s3_transport` objects when it receives the first call to the `RESOURCE_OP_READ` operation. The `RESOURCE_OP_READ` operation simply forwards to `dstream.read()` which calls `s3_transport.receive()`. The read operations are all synchronous.

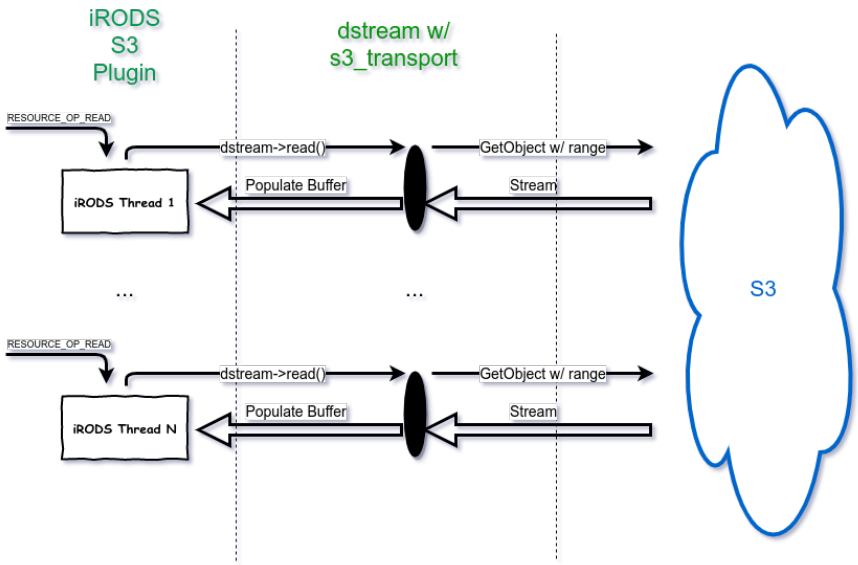

Figure 3. Streaming Parallel GET

CACHE FILE

In some cases, a local cache file will still be necessary. These include the following:

- The iRODS data object is opened in both read and write mode.

- The iRODS data object is opened in write-only mode but the object exists in S3 and is not being truncated.

When the `s3_transport` object is created, it detects any need for a cache file and the S3 object is downloaded to cache (if it exists and not truncated) (Figure 4). All iRODS reads and writes are performed directly on the cache file (Figure 5). When the last `close()` is performed on the iRODS data object, the cache file is flushed to S3 (Figure 6).

If the cache file is large enough, multiple upload threads are used and a multipart S3 upload is performed.

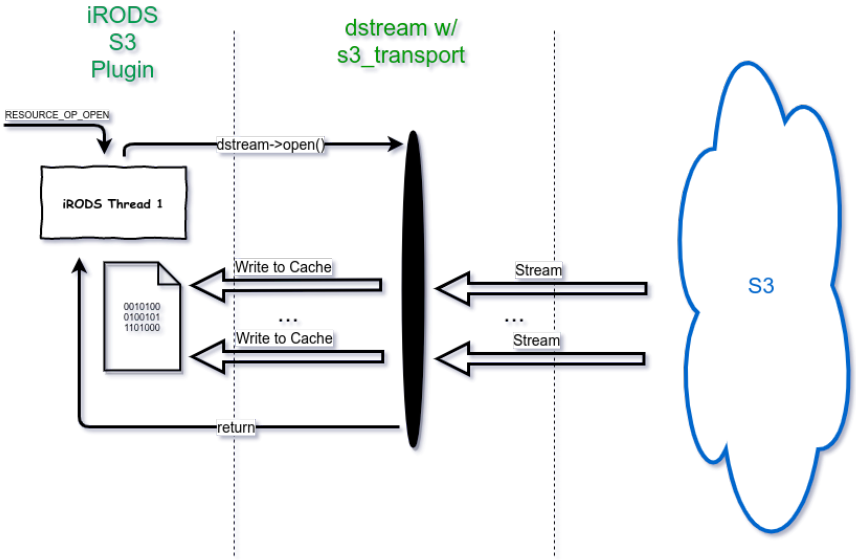

Figure 4. Streaming Cache Open

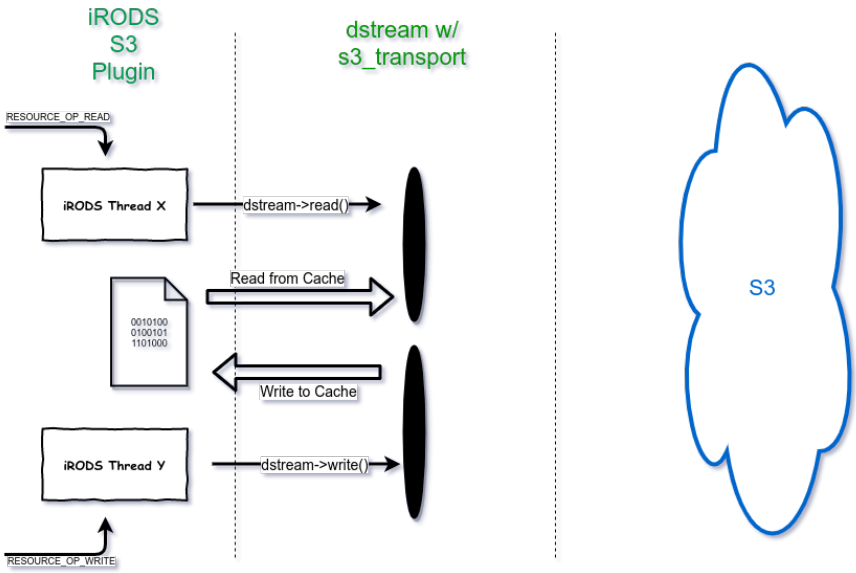

Figure 5. Streaming Cache Read / Write

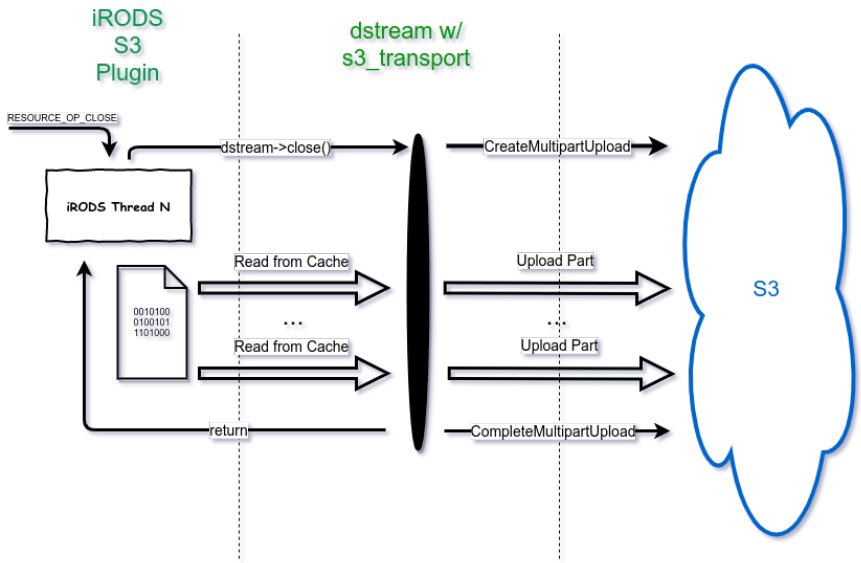

Figure 6. Streaming Cache Close

PERFORMANCE

We ran a battery of tests that compared the performance of uploads and downloads among the following three implementations:

- S3 Plugin w/ Streaming (2020)

- S3 Plugin w/ S3FS (Cacheless) (2019)

- Amazon AWS CLI Tool

Since iRODS generally uses 16 threads to transfer large files, the maximum number of threads for the S3 API was increased to 16 threads.

The tests were run against a local MinIO[8] server backed by an SSD drive. This was to simulate a case where the network throughput and storage latency is not a bottleneck so that we could compare the performance improvement by not using a cache file. The chunk size was set to 64MB.

Each upload and download was performed 6 times and the median time value was used to measure the performance.

Download

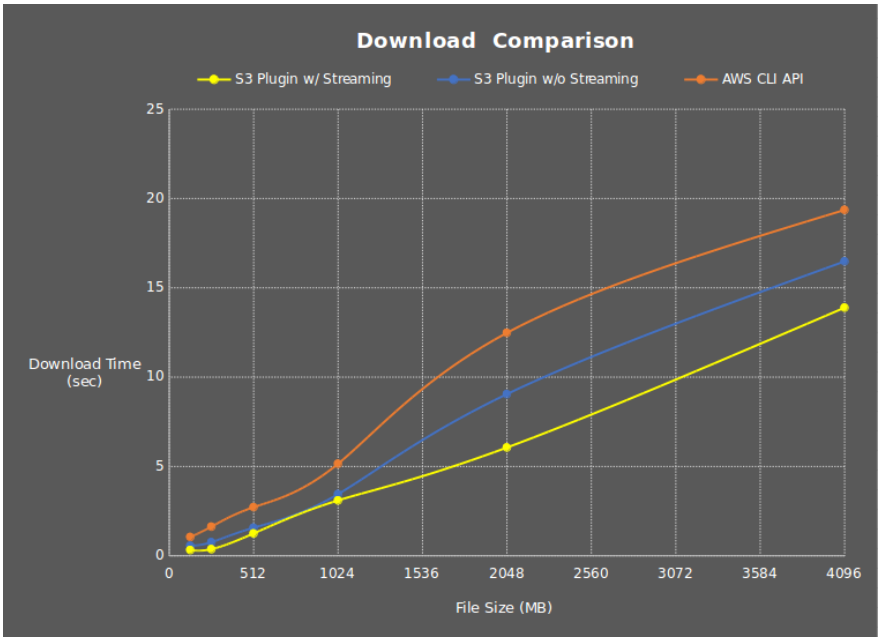

Figure 7. Download Performance Comparison

	128MB	256MB	512MB	1024MB	2048MB	4096MB
AWS S3 CLI	1.05	1.63	2.72	5.15	12.48	19.36
S3 Plugin w/ S3FS (2019)	0.56	0.76	1.57	3.45	9.05	16.48
S3 Plugin w/ Streaming (2020)	0.32	0.37	1.25	3.10	6.06	13.89

Table 1. Download times in seconds (median, n=6)

As seen in Figure 7 and Table 1, all three implementations were competitive with files under 1GB, but then the Streaming S3 Plugin (2020) began to outdistance the other two. With file downloads between 1GB and 4GB, the Streaming S3 Plugin (2020) consistently outperformed the S3FS implementation (2019) by about three seconds. The AWS CLI Tool was an additional couple seconds slower than S3FS.

It is also clear from this graph that download performance is relatively linear as the downloaded file grows in size.

Upload

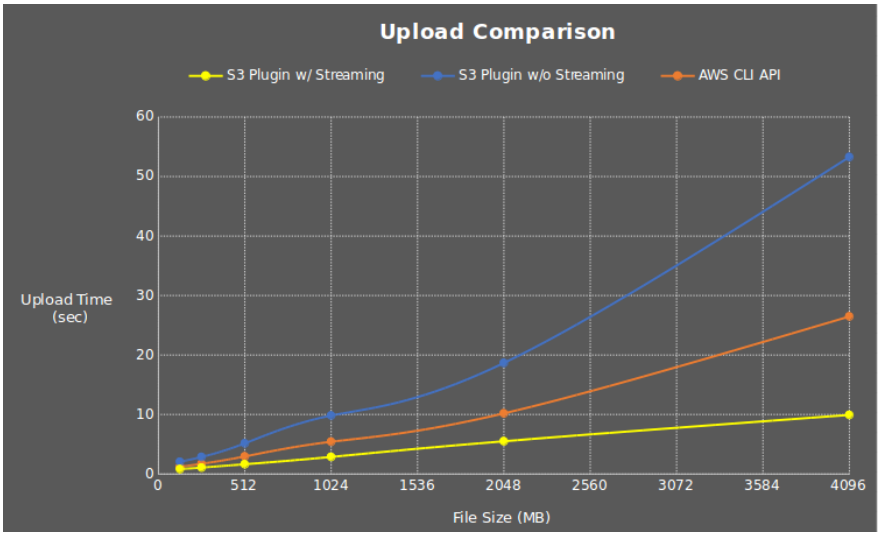

Figure 8. Upload Performance Comparison

	128MB	256MB	512MB	1024MB	2048MB	4096MB
S3 Plugin w/ S3FS (2019)	2.06	2.88	5.16	9.83	18.66	53.28
AWS S3 CLI	1.21	1.73	2.96	5.43	10.19	26.48
S3 Plugin w/ Streaming (2020)	0.83	1.09	1.65	2.88	5.51	9.93

Table 2. Upload times in seconds (median, n=6)

The Streaming S3 Plugin (2020) came out ahead for the upload comparison as well (Figure 8 and Table 2). However, the AWS CLI Tool came in second, with a trailing third for the S3FS implementation (2019) as the file sizes increased. For 4GB files, the Streaming S3 Plugin was 2.5x faster than the AWS CLI Tool.

The Streaming S3 Plugin, most notably, maintains a linear performance profile for upload due to the circular buffers providing consistent memory usage throughout and helping to mitigate the differences in storage medium performance by moving any bottleneck to the network.

These same tests were performed against an Amazon S3 backend, including live public network latencies. The results are not included here, but the relative performance among the three options remained.

FUTURE WORK

Development is almost complete but more testing and real-world usage will certainly bring new insights and optimizations.

With the current default iRODS settings, files between 32 MB and 80 MB are failing due to the S3 limitation on each multipart upload part (except the last) being at least 5 MB in size. Increasing the `transfer_buffer_size_for_parallel_transfer_in_megabytes` configuration setting is a manual workaround for this. This will be fixed and no configuration change will be necessary.

Additional work will also be required to implement the `RESOURCE_OP_READDIR` operation.

SUMMARY

The iRODS S3 Plugin has made a lot of progress in the last few years. It has moved from an archive class resource under a compound resource, to providing cacheless operations, to now providing direct streaming access to S3-compatible backends.

This progress has also increased its performance significantly. This year's streaming plugin is now almost 30% faster when downloading files than the AWS CLI Tool and 2.5x faster when uploading.

REFERENCES

[1] Draughn, Kory: iRODS IOStreams Library (2019). `https://github.com/irods/irods/issues/4268`

[2] Amazon S3 (2006) `https://en.wikipedia.org/wiki/Amazon_S3`

[3] Wan, Mike: Initial S3 File Driver commit (2009). `https://github.com/irods/irods-legacy/commit/2d204c14687340828483abecf8f73a8ea4dea944`

[4] James, Justin; Russell, Terrell; Coposky, Jason; iRODS S3 Resource Plugin: Cacheless and Detached Mode (2019) `https://irods.org/uploads/2019/James-iRODS-S3_Resource_Plugin_Cacheless_and_Detached-paper.pdf`

[5] s3fs-fuse: FUSE-based file system backed by Amazon S3 `https://github.com/s3fs-fuse/s3fs-fuse`

[6] libs3 `https://github.com/bji/libs3`

[7] Circular Buffer `https://en.wikipedia.org/wiki/Circular_buffer#References`

[8] MinIO: High Performance Object Storage `https://min.io/`