iRODS S3 Plugin with Direct Streaming

Justin James
Applications Engineer
iRODS Consortium

June 9-12, 2020
iRODS User Group Meeting 2020
Virtual Event

Existing Cacheless Limitations

- 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.

- Performance was limited by having to read an entire object from S3, write to the local disk, and flush the object back to S3.

- In some cases, the S3 is faster than the local disk which means the performance is further limited by the performance of the local disk.

- The existing cacheless plugin does not support dstream.

- Migrating to a streaming plugin that streams connections directly from iRODS to the S3 backend and vice versa.

$$\begin{tikzcd}\text{Compound} \arrow[r] & \text{Cacheless} \arrow[r] & \text{Streaming}\end{tikzcd}$$

The Streaming S3 Plugin (Summary)

- All reads and writes are handled by an s3_transport class which extends **irods::experimental::io::transport**.
- 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.
        - If parallel transfer is performed in iRODS, a multipart upload is started 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.

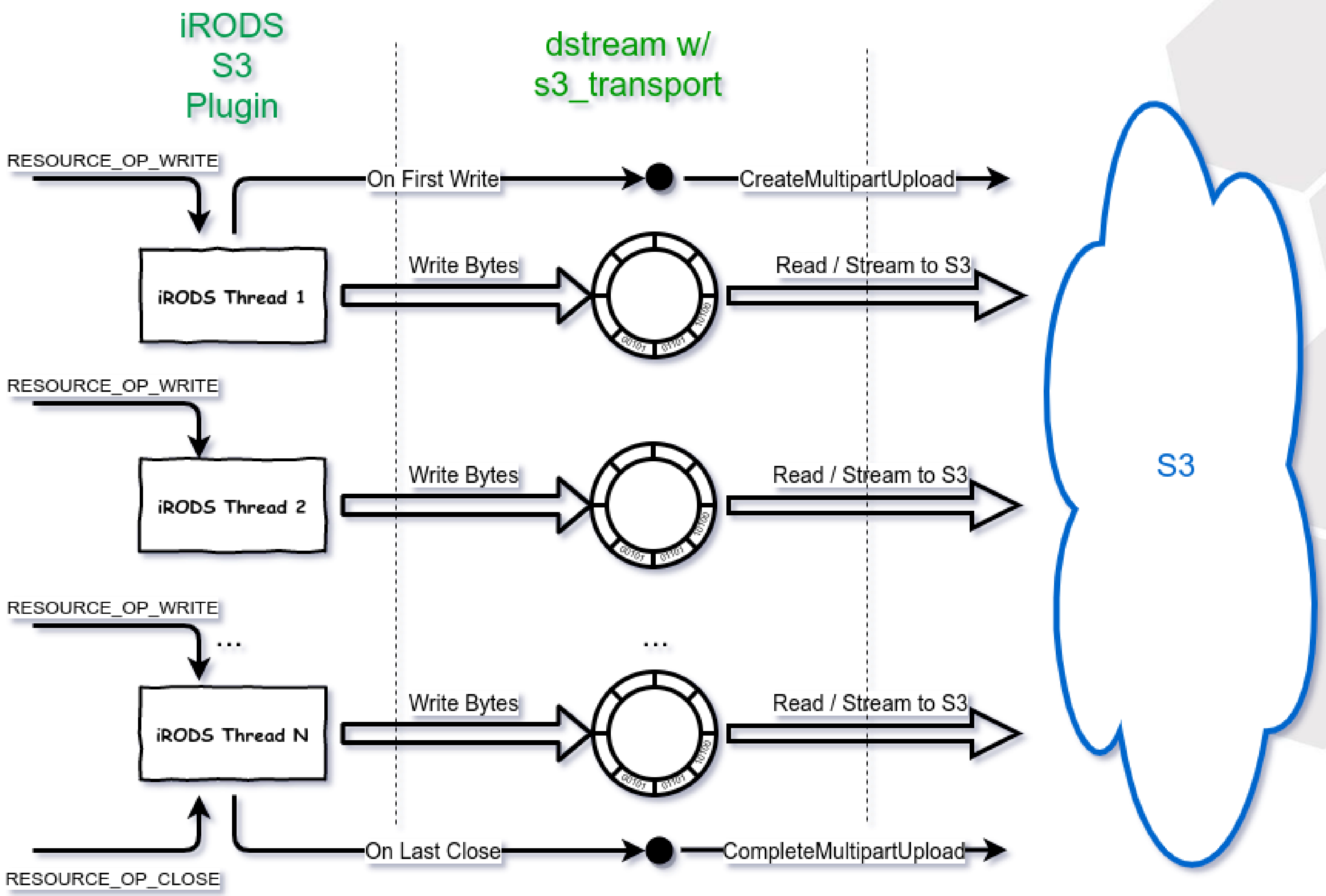
The Streaming S3 Plugin (Summary)

- In some circumstances a local cache file will still be used.
- We are no longer using S3FS libraries. Both the cacheless and cache versions use libs3.
- The **s3_transport** code is a proving ground for a couple of libraries that have been added to the iRODS core:
  - Space limited circular buffer with notifications for threads waiting to read and write.
  - The use of dstream and transport in iRODS.

The Streaming S3 Plugin (Parallel Put)

- When an object is opened in write only mode with the truncate flag set, a full file upload (PUT) is being performed.
- Each thread creates the **dstream** and **s3_transport** objects when it receives the first call to the RESOURCE_OP_WRITE operation.
- The very first **s3_transport** object opened calls *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 circular buffer.
- The **s3_transport** object creates a thread to read the data from the buffer and stream it to S3.

The Streaming S3 Plugin (Parallel Put)



Parallel Put (Circular Buffer)

- The circular buffer is designed to limit the amount of memory that is used when uploaded very large files.
  - If the circular buffer is full, the 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 than may be used per agent is numberThreads * bufferSize * numberEntries.

The Streaming S3 Plugin (Parallel Get)

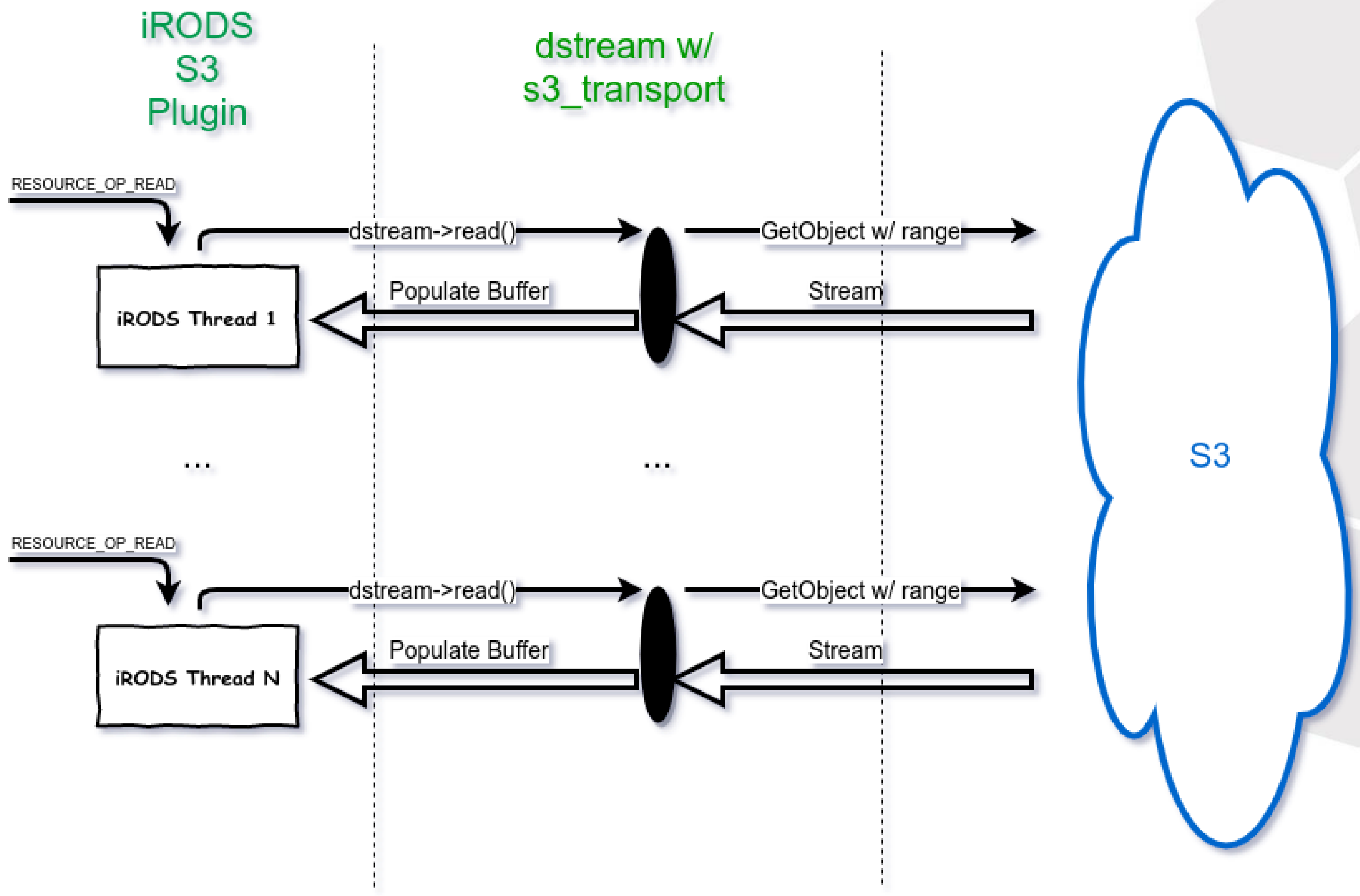
- When an object is opened in read only mode the requested bytes are simply read from the S3 object.
  - S3 allows random access reads for objects so a call to RESOURCE_OP_READ translates directly to a request to *GetObject*.

- 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.

The Streaming S3 Plugin (Parallel Get)



The Streaming S3 Plugin (Using a Cache File)

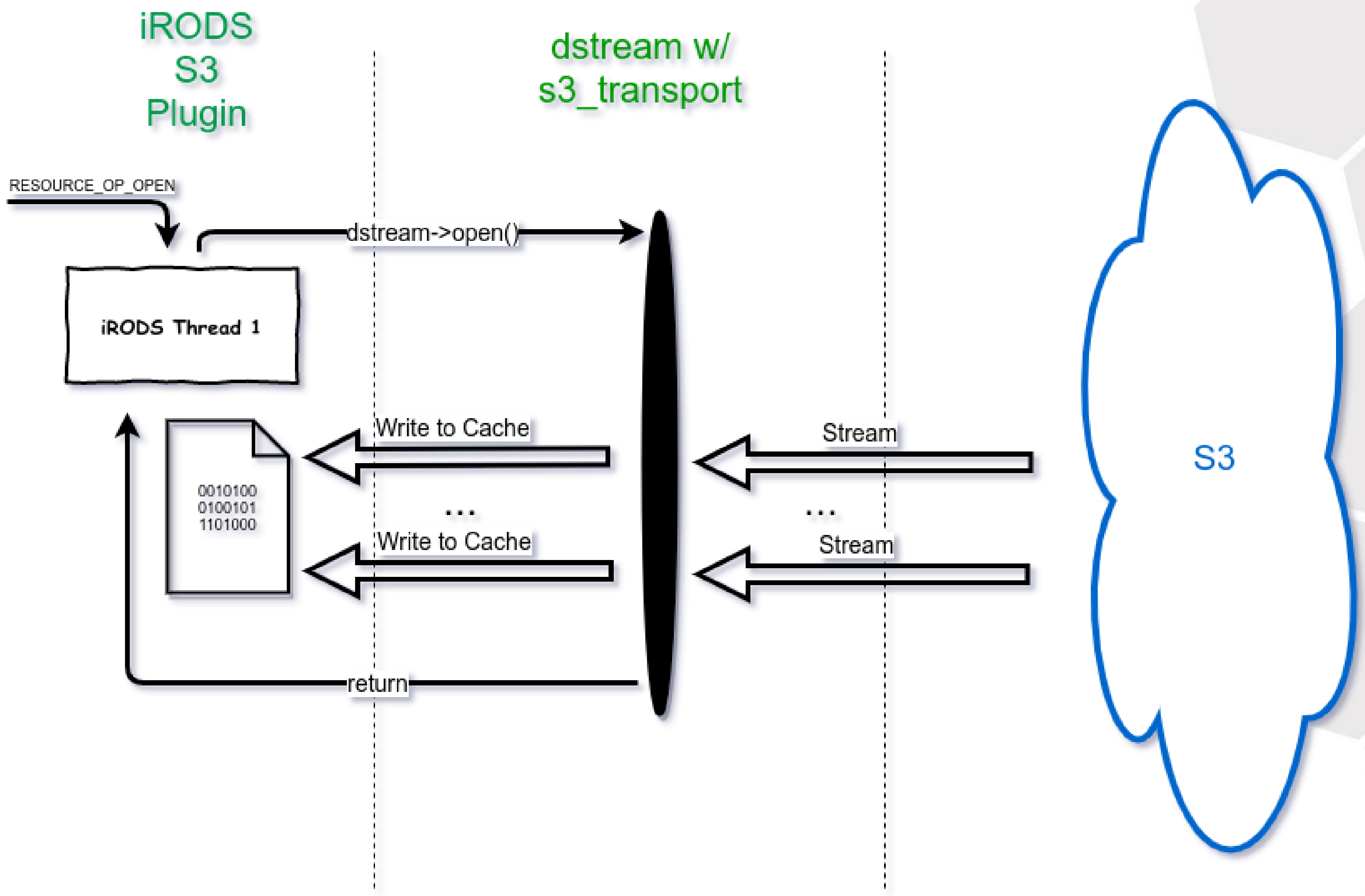
- In some cases a cache file will still be necessary. These include the following.
  - The object is opened in both read and write mode.
  - The object is opened in write-only mode but the object exists in S3 and is not being truncated.

- When the **s3_transport** object is created, it detects the need for a cache file and the object is downloaded to cache (if it exists and not truncated).

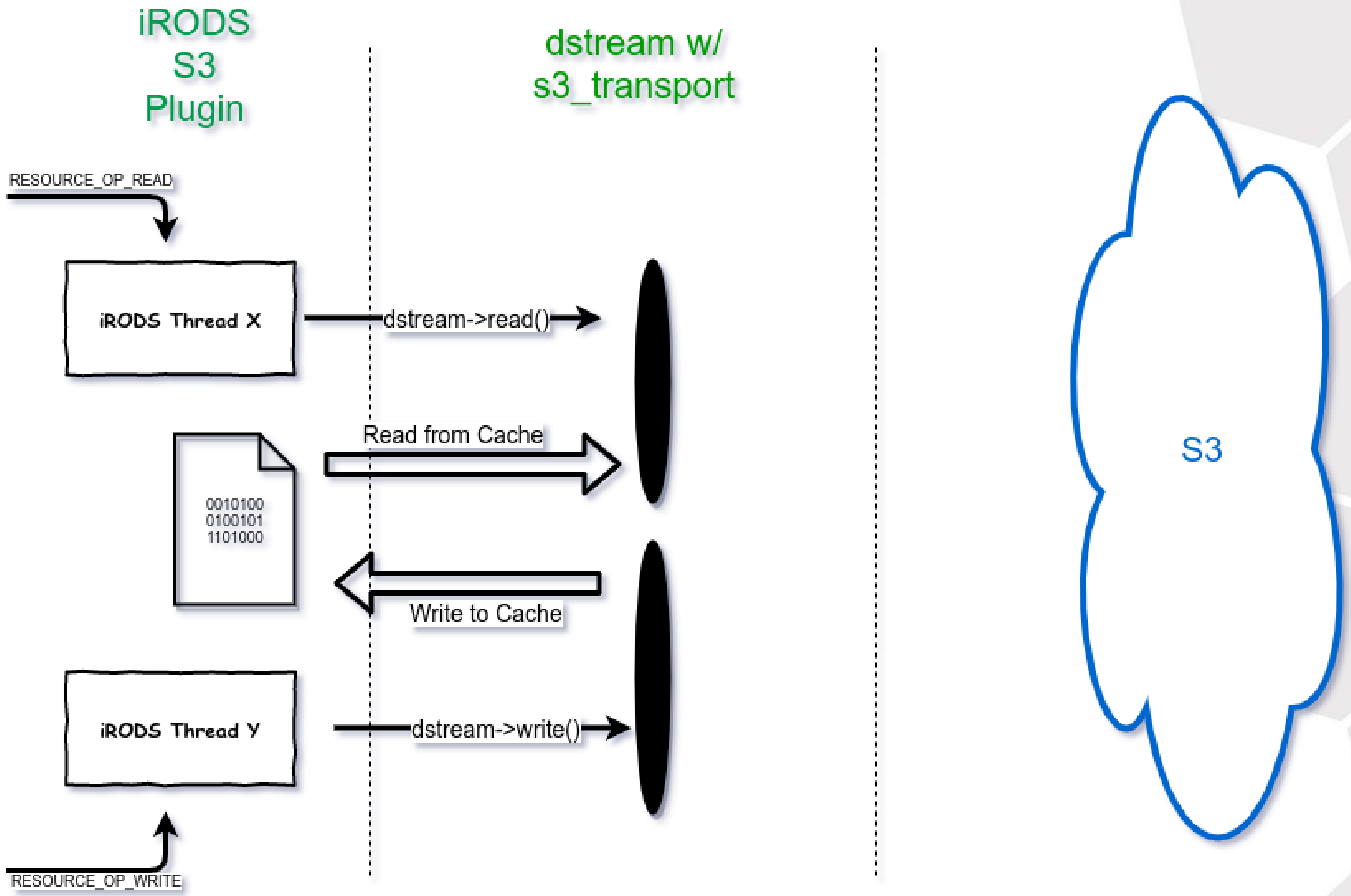
- All reads and writes are performed directly on the cache file.

- When the last close() is performed on the object, the cache file is flushed to S3.
  - If the cache file is large enough, multiple upload threads are used and a multipart upload is performed.

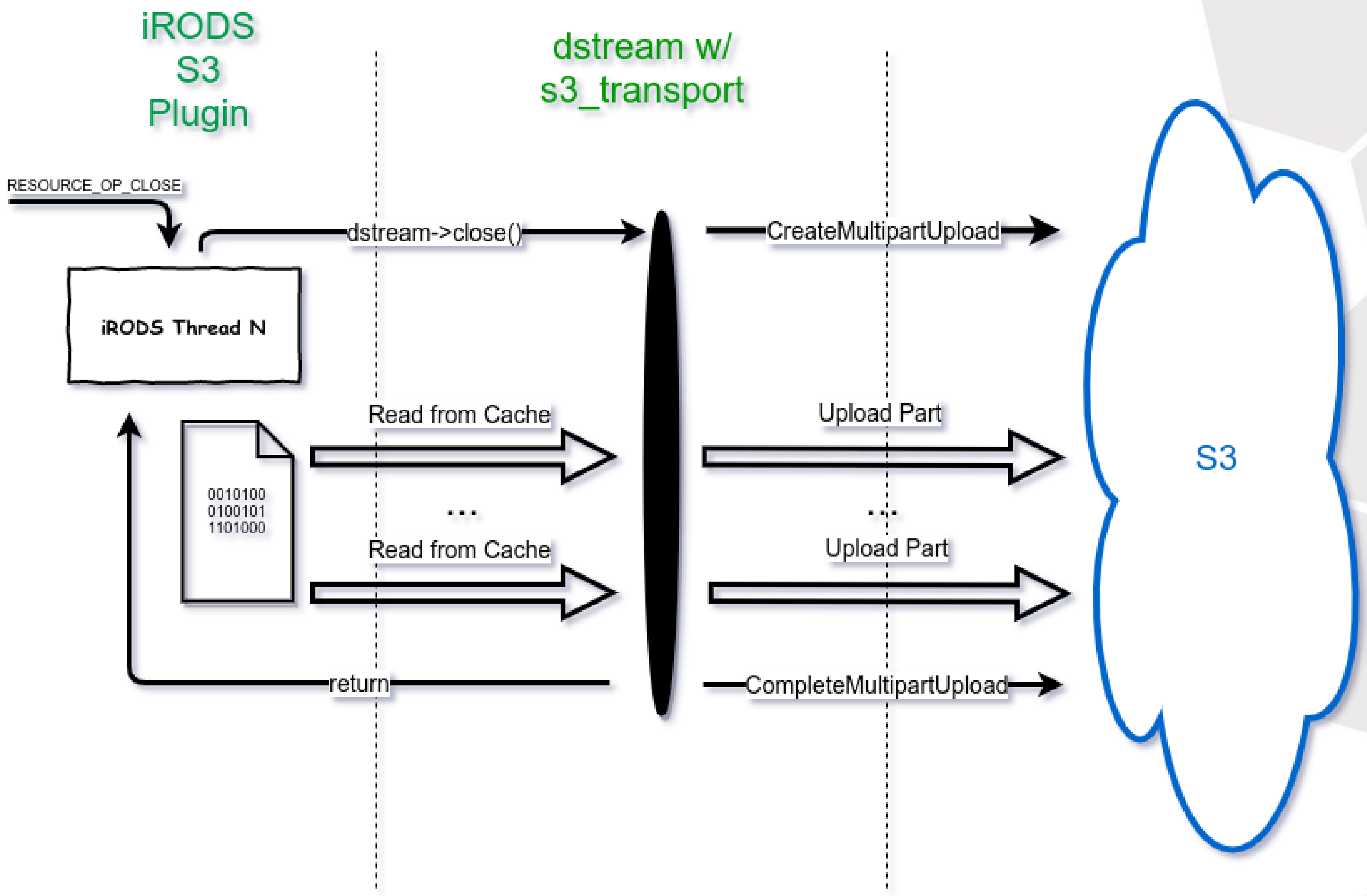
The Streaming S3 Plugin (Cache - First Open)



The Streaming S3 Plugin (Cache - Read/Write)



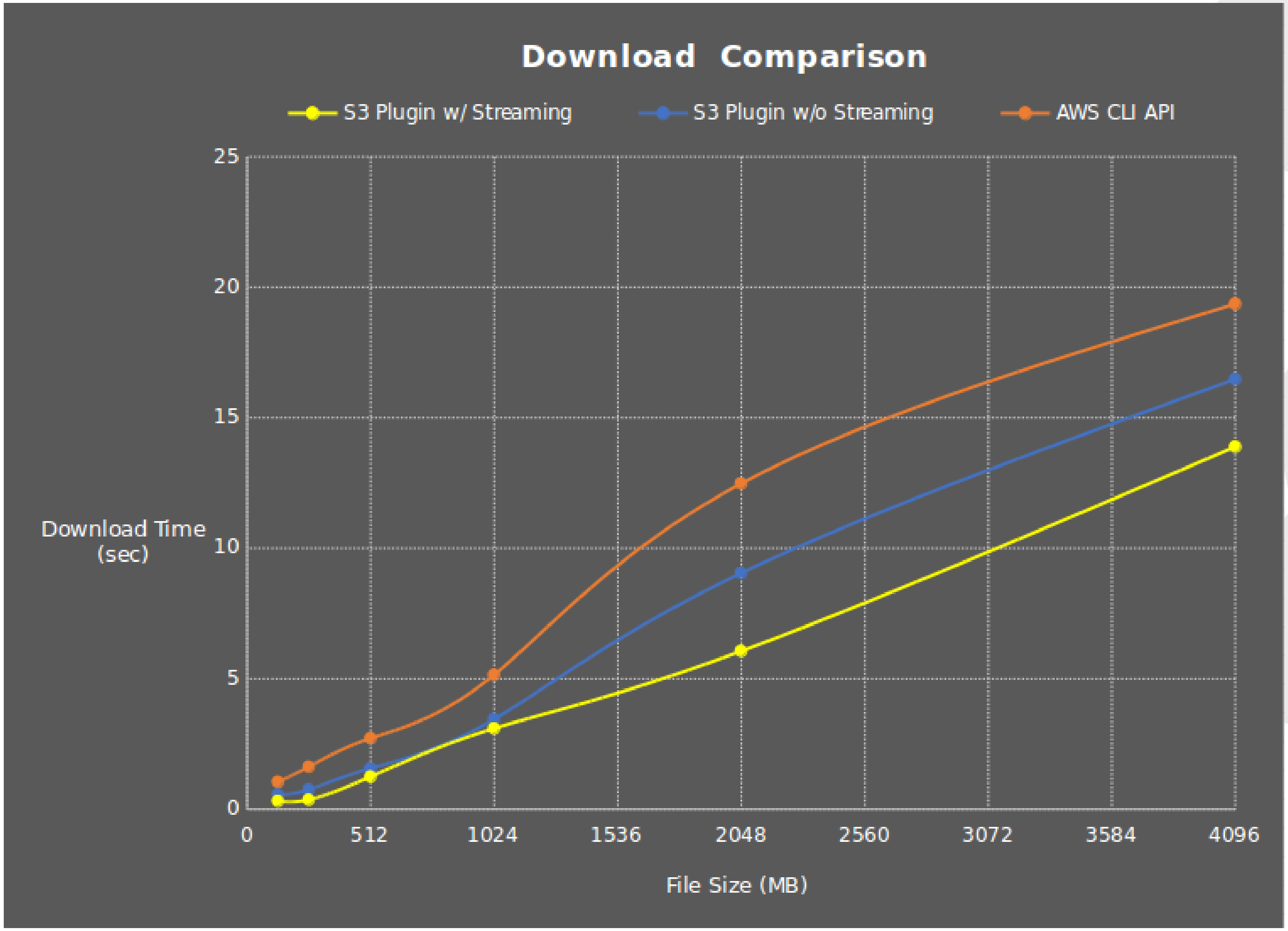
The Streaming S3 Plugin (Cache - Last Close)



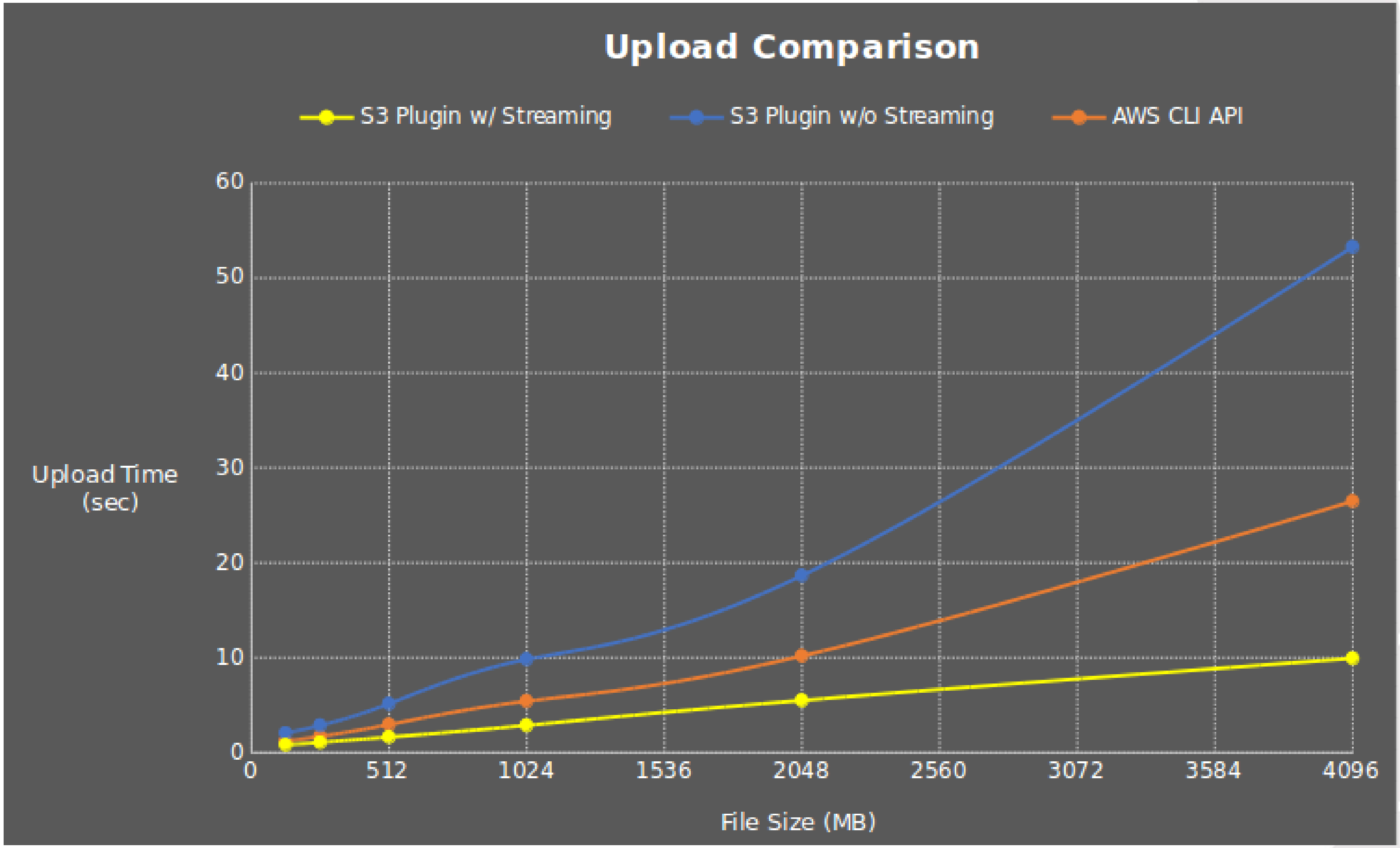
Implementation Details (Performance)

- We ran tests that compared the performance of uploads and downloads among the following:
  - Cacheless S3 resource which used S3FS
  - Streaming S3 resource
  - 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 server which is backed by a SSD drive. This was to simulate a case with the network throughput and storage latency is not a bottleneck so that we could compare the performance improvement by not using a cache file.
- Each upload and download was performed 6 times and the median time value was used to measure the performance.
- These same tests were performed with against an Amazon S3 backend. These results are not shown but the relative performance among the three options were similar.

Implementation Details (Download Peformance)



Implementation Details (Upload Peformance)



Next Steps

Development is almost complete but the final implementation details are being added.

- Currently with the default iRODS settings, files between 32 MB and 80 MB are failing due to the S3 limitation on each multipart upload part (except last) being at least 5 MB in size.
  - Increasing the *transfer_buffer_size_for_parallel_transfer_in_megabytes* configuration setting is a work-around for this.
  - This will be fixed and no configuration change will be necessary.
- Implementing the RESOURCE_OP_READDIR operation.

Questions?