iRODS Logical Locking

Alan King
Renaissance Computing
Institute (RENCI)
UNC Chapel Hill
alanking@renci.org

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

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

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

ABSTRACT

iRODS 4.2.9 introduces Logical Locking by providing additional replica status values within the catalog. Previously, replicas in iRODS could only be marked 'Good' or 'Stale'. This did not capture the states of when data was in flight, or incomplete. This paper will explain the new Intermediate and Write-Locked states for iRODS replicas and how they are used to provide protection from uncoordinated writes into the system.

Keywords

iRODS, data management, logical locking, concurrency

INTRODUCTION

The iRODS Technical Overview[1] states:

> iRODS provides a logical representation of files stored in physical storage locations. We call this logical view a virtual file system and the capabilities it provides, Data Virtualization.

Interactions with data in an iRODS system are meant to mirror how one might interact with data in a traditional hierarchical file system. The purpose of the virtual file system is to present data across a number of geographically distributed servers under a unified namespace - in other words, a distributed system. A distributed system implies operational concurrency and operational concurrency implies race conditions.

The following document will describe how iRODS interacts with data, how the iRODS Consortium seeks to address concurrency in manipulation of data within an iRODS system, and planned efforts for protecting data access and manipulation as well as policy execution.

BACKGROUND

History of data representation and movement

The logical representation of data presented by the virtual file system is known as a Data Object, which the iRODS Beginner Training[2] defines as:

iRODS UGM 2021 June 8-11, 2021, Virtual

Data Object: a logical representation of data that maps to one or more physical instances (**Replicas**) of the data at rest in Storage Resources

A Replica could have one of two statuses:

- **Stale**: The replica is no longer known to be Good. Usually indicates that a more recently written-to replica exists. This does not necessarily mean that the data is incorrect but merely that it may not reflect the "truth" for this Data Object.
- **Good**: The replica is Good. Usually the latest replica to be written. When a replica is Good, all bytes and system metadata (size and checksum, if present) are understood to have been recorded correctly.

iRODS has historically presented an interface which interacts with data at the logical level - that is, with Data Objects. However, over time, it became clear that interacting directly with replicas is very important for iRODS users and administrators. Other operations were added to allow for replica manipulation which were very difficult to integrate with existing APIs.

As such, operations which deal directly with replicas have completely separate implementations for moving data from the logical level operations. Furthermore, operations dealing with Data Objects still need access to replica information (after all, the operations interact with the storage eventually, at which point it is interacting at the replica level).

On top of all of this, because replica status can only be Good or Stale, any replica which exists in the catalog but has not been finalized is being represented incorrectly because the data is still in flight. Good and Stale refer to data which is at rest. There is no status to represent in-flight data.

Unifying data movement with replicas as First Class Citizens

These differences in implementation have consistency and performance implications for moving data. In reality, all of these data moving operations should be (and are) identical:

open replica –> move data to replica –> close replica –> finalize system metadata in catalog

Every data moving operation in iRODS should implement the above steps. These should use the same set of low-level interfaces so that data movement is made consistent across operations. To make this possible, replicas must be treated as the operative entity in data movement, not Data Objects (which are one-to-many mappings to replicas). This requires that replicas be treated as First Class Citizens within the iRODS server where, historically, they have not - at least not in all instances.

Efforts are being made to change this in the server, but it has not yet been fully realized. We will take a slight detour to describe in detail one of the most important operations within an iRODS system: Replication. First it will be defined at an abstract level as it is intended to function, and then the implementation and interface in iRODS will be described. This highlights the importance of a unified mode of data movement within the system even while the requirements for the operation differ so greatly from others such as put, copy, and rename.

Rules For Replication

Replication refers to copying a physical replica of an existing Data Object from one storage resource to another. Replication requires 3 inputs:

1. **Logical path**: Path to an existing Data Object with at least one replica which is at rest and can be read.

case	source	destination	result	reason
0	-	-	unchanged	No source replica exists
1	-	&	unchanged	No source replica exists
2	-	X	unchanged	No source replica exists
3	&	-	&	Replication allowed
4	&	&	unchanged	Destination replica must be Stale
5	&	X	&	Replication allowed
6	X	-	X	Replication allowed
7	X	&	unchanged	Destination replica must be Stale
8	X	X	unchanged	Source replica must be Good

Table 1. This table shows the nine cases of initial state of the replicas on the source and destination resources, the resulting replica state on the destination resource, and the reason for the result.

2. **Source resource**: The resource from which data will be copied/read.

3. **Destination resource**: The resource to which data will be copied/written.

The source resource must already have an existing, at-rest replica which can be read in order for replication to be *possible* in any case.

If the destination resource has no replica, replication to the destination resource is *allowed* in **all cases**. If this is true, a **new replica** will appear on the destination resource as a result of the replication. If this is not true, the destination resource has an existing replica and replication may or may not occur.

If the destination resource has an existing replica, replication would be performing a **replica update**. The following requirements must be true in order for updating the replica to be *allowed*:

1. The destination resource must not be the source resource

2. The destination replica must be Stale

3. The source replica must be Good

Table 1 shows all possible cases for replication between a single source and destination when the source and destination replicas do or do not exist. Some notes about the table:

- '&' represents a Good replica
- 'X' represents a Stale replica
- '-' indicates a replica which does not exist.
- All replicas are assumed to be at rest

REPLICATION MECHANISM

The iRODS server provides the rsDataObjRepl API in order to replicate Data Objects. Here is the signature:

```
int rsDataObjRepl(rsComm_t*, dataObjInp_t*, transferStat_t**);
```

As described in the Rules for Replication subsection above, replication requires a logical path, a source resource, and a destination resource. The following describes how these requirements can be satisfied.

Logical Path

As described in the general Rules for Replication subsection, the logical path must refer to an existing iRODS Data Object of which the to-be-determined-or-specified source replica must exist. The authenticated user must have at least write permission on the Data Object in order to read from the source replica and write to the destination replica. The other qualifications regarding source and destination replica statuses have already been described in Rules for Replication.

Source Resource

In order to specify a source resource, any of the following may be provided via `condInput`:

1. `RESC_NAME_KW` - Must be a root resource. If this is not true, the following error is returned: `DIRECT_CHILD_ACCESS`.
2. `RESC_HIER_STR_KW` - Must be a full resource hierarchy.
3. `REPL_NUM_KW` - Must be an integer representing the existing replica.

If none of the above is provided, any resource is eligible for use as the source resource.

Resource hierarchy resolution is then performed on a `OPEN` operation by the API to determine the full resource hierarchy. The following must be true of the resolved resource hierarchy (depending on which of the source resource inputs was provided):

1. The resolved resource hierarchy has a root which matches `RESC_NAME_KW`
2. No resource hierarchy resolution is performed.
3. The resolved resource hierarchy has an existing replica with a replica number which matches `REPL_NUM_KW`

If the resolved resource hierarchy does not meet the requirements of the provided inputs or the requirements of the source replica as defined by the Rules of Replication, the following error is returned: `SYS_REPLICA_INACCESSIBLE`.

Destination Resource

In order to specify a destination resource, any of the following may be provided via `condInput`:

1. `DEST_RESC_NAME_KW` - Must be a root resource. If this is not true, the error `DIRECT_CHILD_ACCESS` is returned.
2. `DEST_RESC_HIER_STR_KW` - Must be a full resource hierarchy.

If neither of the above is provided, the system will look for a `DEF_RESC_NAME_KW` (the default resource). If present, `DEST_RESC_NAME_KW` is set to this value. If not, any resource is eligible for use as the destination resource.

Resource hierarchy resolution is then performed on a `CREATE` operation by the API to determine the full resource hierarchy. The following must be true of the resolved resource hierarchy (depending on which of the destination resource inputs was provided):

1. The resolved resource hierarchy has a root which matches `DEST_RESC_NAME_KW`
2. No resource hierarchy resolution is performed.

If the resolved resource hierarchy does not meet the requirements of the provided inputs, the following error is returned: `SYS_REPLICA_INACCESSIBLE`.

If the destination resource has a replica which does not meet the requirements for a destination replica as defined in the Rules of Replication, the following error is returned: `SYS_NOT_ALLOWED`.

Updating all replicas

When the `ALL_KW` is provided, the API will gather a list of all existing replicas and update each one, serially at this time, according to all of the rules described above. Specifying a destination resource along with the `ALL_KW` is not allowed.

PROBLEMS WITH UNCOORDINATED, CONCURRENT OPERATIONS

Once data movement is unified across the iRODS server, the conversation about concurrency as it relates to system correctness can happen. We have identified three ways in which the state of the system can be made incorrect:

1. Uncoordinated, concurrent writing to a single replica can lead to **Data Corruption**.
2. Uncoordinated, concurrent writing to multiple replicas of the same Data Object can lead to **Truth Corruption**.
3. Uncoordinated, concurrent operation execution can lead to **Policy Violations**.

The iRODS Consortium is actively working to address these three situations in the server. The first two have been partially addressed in released server software and the third is to be addressed in a future release.

Data Corruption and Intermediate Replicas

The first problem listed above is that in-flight replicas can be opened and modified concurrently by multiple agents in an uncoordinated fashion. Also, as a result of iRODS historically creating replicas in the Good status, the catalog does not reflect the current, true state of the data from the moment it is registered in the catalog.

To address this, a third replica status has been introduced:

Intermediate: The replica is actively being written to. The state of the replica cannot be determined because the client writing to it still has the replica opened. Replicas which are marked Intermediate cannot be opened for read or write, nor can they be unlinked or renamed.

Replicas are created in the Intermediate status for data moving operations because the replica is considered in-flight until the time that it is finalized in the catalog. In this way, the replica status is accurately represented in the catalog and the system is able to respond appropriately to concurrent access attempts to this replica.

Truth Corruption and Logical Locking

However, it is unclear which replica for a given Data Object represents the Truth when multiple replicas are in flight at the same time. Protecting individual replicas is not enough to verify the correctness of the data at the logical level.

To address this, a new mechanism called iRODS Logical Locking (ILL) has been introduced which disallows opening any replica for a given Data Object when any one of the replicas are opened for write. This has been implemented using another new (fourth) replica status:

Write-Locked: One of this replica's sibling replicas is actively being written to but it is, itself, at rest. Replicas which are Write-Locked cannot be opened for read or write, nor can they be unlinked or renamed.

By only allowing coordinated modifications to a Data Object, assertions about the correctness of the logical Data Object can be made based upon the correctness of a replica because multiple replicas cannot simultaneously claim to be correct (while potentially holding different data).

Policy Violations and Operation Locking

We have not yet addressed the third problem of uncoordinated, concurrent policy invocations in iRODS. This is addressed in the Future Work section later in this paper.

SOLUTION - DESIGN

iRODS Logical Locking operates on the basic principles of a traditional read-write lock in computing[3]

1. The Data Object is locked when any of its replicas are opened.
2. The Data Object is unlocked when the opened replica is closed/finalized.

Here, we describe this process in more detail:

The Data Object is locked when any of its replicas are opened

1. Client requests to open a replica of a given Data Object.
2. iRODS checks the status of the replica to make sure it is not already locked (equivalent to trying to "acquire" the lock). "Locked" in this case means:
   (a) the replica is Read-Locked (this or a sibling replica is opened for read by an agent): the operation will fail if it is not an open-for-read.
   (b) the replica is Write-Locked (a sibling replica is Intermediate): the operation will fail.
   (c) the replica is Intermediate: the operation will fail for an uncoordinated open-for-write or any open-for-read. See the Implementation section below for details about concurrent write coordination.
   (d) If the open intends to create a new replica, the operation will fail if any replica is found not to be at rest (that is, Intermediate, Read-Locked, or Write-Locked).
3. Depending on the operation, the Data Object is then locked for this open. "Locked" in this case means:
   (a) **open-for-read**: the status of all replicas is set to read-lock
   (b) **open-for-write**: the status of the target replica is set to Intermediate and the statuses of the sibling replicas are set to Write-Locked
   (c) **open-for-create**: the statuses of any existing sibling replicas are set to Write-Locked
4. The target replica is then physically opened and the open is complete. If a new replica is supposed to have been created, the entry is created in the catalog in the Intermediate status.

As we will see later, there is still a race condition here: the TOCTOU[4] condition in the iRODS catalog itself. We have designed a solution for this which will be described later, but we mention it here to acknowledge its existence.

The Data Object is unlocked when the opened replica is closed/finalized

1. The client requests to close and finalize a replica of a given Data Object.
2. The Data Object is unlocked atomically along with updating the other system metadata. The final states of the replicas depends mainly on the operation requesting the close. The general cases for unlocking are shown here, but some operations (e.g. replication, phymv, etc.) may behave differently as they have different requirements:
   (a) **open-for-read**: if no other agents have any of the replicas opened, the replica statuses of each replica is restored to what it was before the open
   (b) **open-for-write**: there are two cases:
      i. success: the target replica is marked Good and all sibling replicas are marked Stale
      ii. failure: the target replica is marked Stale and all sibling replicas have their statuses restored to what they were before the open

SOLUTION - IMPLEMENTATION

Code Facilities

Two mechanisms were developed to implement Logical Locking:

1. `data_object_finalize` API plugin (finalize) - atomically applies updates to (multiple) rows in `R_DATA_MAIN`
2. `replica_state_table` (RST) - An in-memory, per-agent JSON structure which describes a snapshot of rows in `R_DATA_MAIN` for a particular data_id representing a Data Object

A library was developed under the `irods::logical_locking` namespace in the server on top of these utilities which makes surgical edits to the RST and updates the catalog via the finalize API to atomically update - and therefore unlock - Data Objects. As described in the design, Data Objects are locked on open and unlocked on close/finalize using this `logical_locking` library.

Refactoring iRODS Internals

Much refactoring was necessary in the iRODS server for a correct first attempt at an ILL implementation and to fix critical existing issues, all while maintaining the existing client-facing interfaces. In order to ensure that ILL is enforced across the API surface, the requirements described in "Unifying data movement with replicas as First Class Citizens" needed to be put in place. This accomplishes both the aforementioned consistency and correctness of data movement throughout the system, and a single entry point for using ILL.

A major shift was made internally in the iRODS server away from relying on `rsDataObjClose` to handle the logic of finalizing replicas as has been done historically. Each operation has its own requirements for how the replicas in a Data Object should appear for success or failure cases and in many cases they differ from those of other operations. The finalize API plugin is now used in conjunction with the `replica_close` API plugin to determine and control how replicas should be updated after the completion of an operation. `rsDataObjClose` can still be used to close and finalize a Data Object, but the resultant replica statuses are handled generically.

Coordinated, concurrent write support

To support concurrent writes to the same replica which enables parallel data transfer using multiple iRODS agents, another mechanism was developed called the Replica Access Registry (RAR). The RAR is held in shared memory so that agents spawned from a particular iRODS server can coordinate concurrent writes to particular replicas. When an iRODS agent opens a replica for write (or create) an entry is created in the RAR which associates an agent PID with a replica of a Data Object and is accessed via a universally unique identifier called a Replica Access Token (RAT). This token is part of the L1 descriptor held in the agent process which initiated the replica open. This token allows other iRODS agents connected to the same server to open the same Intermediate replica.

Using these tokens, clients can now implement parallel data transfer to a single replica over the main iRODS port. This has been demonstrated both in the C++ client API with the Parallel Transfer Engine (PTE) and in the Python iRODS Client (PRC)[6] version 1.0.0.

KNOWN LIMITATIONS

As with most features, ILL does not protect against rogue administrators.

Performance

Now that the server is enforcing coordination of writes across the iRODS Zone, this necessarily means more trips to the iRODS Catalog to query for information and modify rows to update replica statuses. These additional round trips to the database will introduce some marginal overhead, but for most data-write operations, they are still dwarfed by the time for actual data movement.

The Database Race

ILL is currently still prone to database race conditions in the iRODS Catalog due to a classic TOCTOU[4] problem which allows multiple winners for the race. The scenario arises when two agents arrive to lock a Data Object at the same time. The lock is first checked and then set if it passes. All at-rest, unopened Data Objects are eligible to be locked. The two agents will both find that the Data Object is unlocked and then go to the database to set the status. The database will not stop this and both agents will think that they have exclusive access to their respectively opened replicas of the Data Object.

We believe this can be solved by causing the check and set of the replica statuses to be a single database transaction. The system will lean upon the locking mechanism of the database and the lock cannot be "acquired" by more than one agent and so there can be only one winner.

FUTURE WORK

Policy Violation (Operation Locking)

The third problem listed before is violations in configured policy due to uncoordinated, concurrent policy invocations which incur data-modifying operations. We think that this will be solved by operation-aware logical locking, or **Operation Locking**. Where ILL is currently gated on opening and closing replicas of a Data Object, Operation Locking would be gated on the start and completion of a high-level operation.

As described before, open and close are considered "low-level" interfaces where things like put, replication, and physical move (phymv) are "high-level" and are built upon the low-level interfaces. Operation Locking would move the locking mechanism up from open into the higher level interfaces so that locking can be controlled across multiple data-modifying operations resulting from policy invocation. In this way, policy invocation will be made more correct and even enable coordinated, concurrent data movement within the policy-triggered operation(s). This will require extending ILL to make iRODS agents more aware of which lock(s) they have acquired.

An example of when this would be useful is when a client puts data into a resource hierarchy which includes a replication resource with multiple child resources. Without Operation Locking, the time between the multiple replication operations is subject to a race condition where other clients could be writing to the same Data Object.

Read Locks

The goal behind ILL is to implement a readers-writer lock[3] on Data Object resources via the replica status in the iRODS Catalog. A read lock will allow for multiple, **uncoordinated** agents to read from any replica in a Data Object at the same time. When a Data Object is Read-Locked, all data-modifying operations should be blocked (open for write, unlink, rename, etc.) Unlike write locks, which require coordination to lock and unlock properly, read locks will allow for uncoordinated reads. This means that ILL will need to be extended to track a list of agent PIDs and hostnames which are holding the lock. This may imply a refactor of the current lock implementation so that the solution can remain generic and possibly bring it more in line semantically with traditional locks/mutexes.

SUMMARY

After more than two years of planning, development, and testing, iRODS 4.2.9 introduces Logical Locking which protects the iRODS system from uncoordinated, concurrent writes to replicas of a particular Data Object. This is implemented by adding two additional possible replica statuses, Intermediate and Write-Locked.

REFERENCES

[1] iRODS Technical Overview (2016). `https://irods.org/uploads/2016/06/technical-overview-2016-web.pdf`

[2] iRODS Beginner Training (2019). `https://github.com/irods/irods_training/blob/5418c42d7d6c33d92b14acc02f9bbba5c414b779/beginner/irods_beginner_training_2019.pdf`

[3] Lamport, Leslie; *Concurrent Reading and Writing*; Communications of the ACM, November 1977, Vol. 20 No. 11, Pages 806-811 `http://lamport.azurewebsites.net/pubs/rd-wr.pdf`

[4] Time of Check Time of Use `https://cwe.mitre.org/data/definitions/367.html`

[5] Parallel Transfer Engine `need_url`

[6] Python iRODS Client `https://github.com/irods/python-irodsclient`