iRODS Delay Server Migration

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

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

ABSTRACT

The iRODS Delay Server can now be safely moved from one iRODS server to another without requiring a restart. This paper describes the requirements, the design goals, the algorithm, the implementation, and the effects of this new functionality.

Keywords

iRODS, delay rules, delay server, migration

DELAY QUEUE

The iRODS platform provides powerful data management capabilities through the combination of storage technology abstraction (via resource plugins), data discovery (via metadata), and policy enforcement (via the rule engine framework). The rules allow taking action on data, based on the metadata (and any other available inputs). iRODS policies (rules) can be fired in one of three ways:

1. now (upon request)
2. now (upon action)
3. later (via delay())

The iRODS Delay Server (`irodsDelayServer`), formerly known as the Rule Execution Server (`irodsReServer`), is the mechanism provided by iRODS to run rules at a later time than when they are enqueued (option 3 above). iRODS rules can be enqueued via `delay()` to execute at a later time. Any rules that are queued persist in the iCAT database and are processed by the `irodsDelayServer` in a priority order. The `irodsDelayServer` sleeps most of the time, but spawns an `irodsAgent` every 30 seconds (by default) to check the Delay Queue for any delayed rules that need to be run.

This paper will discuss the architecture of the iRODS Delay Server in iRODS 4.3.0 and the process by which the iRODS Consortium arrived at this design and implementation.

DELAY SERVER ARCHITECTURE

The iRODS 4.3.0 Delay Server Architecture is the culmination of design, preparation, and incorporation of smaller features for more than four years. 4.2.4 was released in 2018 while the first changes discussed here were included in 4.2.5 in 2019.

iRODS UGM 2022 July 5-8, 2022, Leuven, Belgium

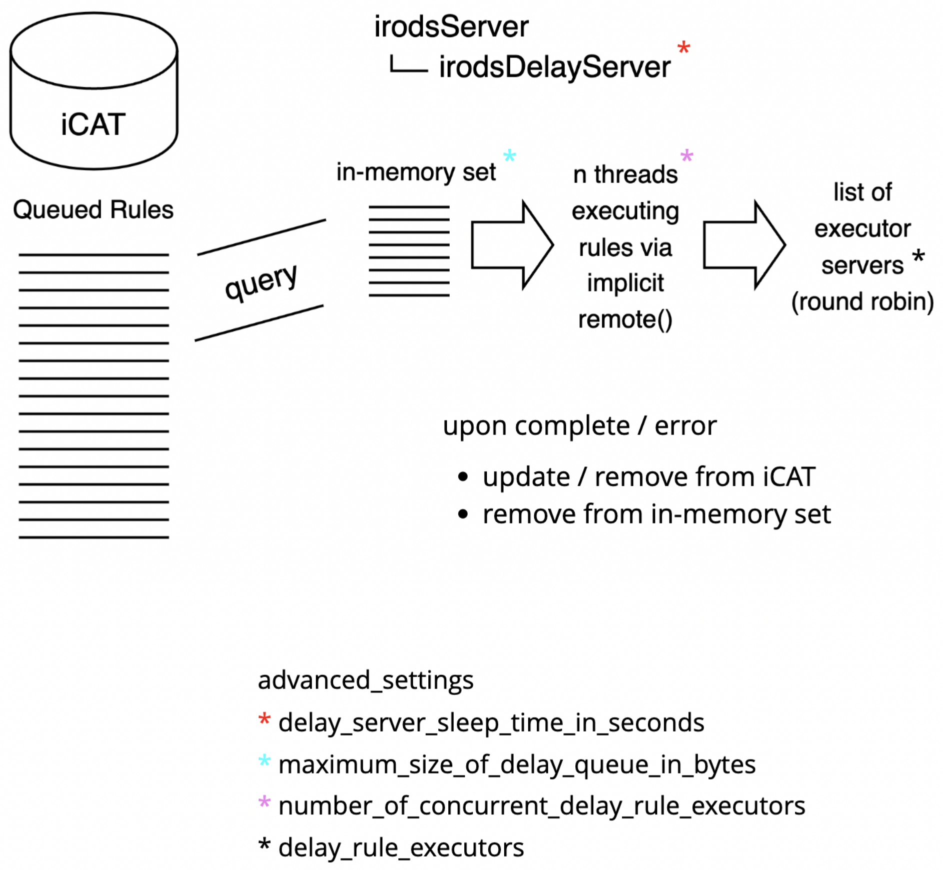
iCAT
Queued Rules
query
irodsServer
└─ irodsDelayServer *
in-memory set *
n threads executing rules via implicit remote() *
list of executor servers * (round robin)
upon complete / error
• update / remove from iCAT
• remove from in-memory set
advanced_settings
* delay_server_sleep_time_in_seconds
* maximum_size_of_delay_queue_in_bytes
* number_of_concurrent_delay_rule_executors
* delay_rule_executors

Figure 1. iRODS 4.3.0 Delay Server Architecture, incorporating updates from 4.2.5 to 4.3.0

4.2.5

iRODS 4.2.5 included updates that fixed the delay queue from being blocked by long-running rules (issue 4250 [1]). It also moved from processing rules directly from the catalog to holding an in-memory set of rules that had been fetched via query and then processed with fewer round trips to the database. This allowed a more efficient use of cores within a single machine to process the queue of delayed rules. In addition, the entire delay server itself was refactored to use threads instead of individual processes (issue 4251 [2]), reducing the memory footprint (issue 3782 [3] and issue 4266 [4]) and allowing reuse of some data structures.

The addition of the in-memory set required a new advanced setting named `maximum_size_of_delay_queue_in_bytes`. This gave the administrator control over how much memory is allocated to running the delay server. As subsequent queries are executed to pull rules from the catalog, if they do not fit into this in-memory set, then they are discarded until enough rules have been executed and there is room for additional rules to be added.

4.2.8

iRODS 4.2.8 included a refactoring of the delay server binary itself to use the newly available query processor (issue 4430 [5]). This reduced the length, redundancy, and complexity of the delay server by calling purpose-built library code. The delay server had been a full copy of the iRODS server source, which was wasteful of memory and required any bugfixes to be applied in two locations.

4.2.9

iRODS 4.2.9 included a change in location where the delay rule contexts were stored (issue 3049 [6]). Prior to 4.2.9, delay rule context was stored on the disk of the machine running the delay server at the time of enqueuing. A new text-like column was added to the `R_RULE_EXEC` table, namely `exe_context` of type `text` (PostgreSQL), `longtext` (MySQL), or `clob` (Oracle). By moving the context into the database, it separated the enqueuing machine from the execution machine. This allowed the delay server to be a different machine, if desired, at a later time.

4.3.0

iRODS 4.3.0 included two additional changes that allowed the delay server to be moved from one machine to another within a zone.

As part of the algorithm (discussed in a later section), the iRODS server must now regularly check to see if it is designated to run the delay server. There were other items that required regular work in the server, so it was decided to implement a cron-like facility within the iRODS server for these tasks.

One of the use-cases that was presented during discussion with the community included the ability to run a large number of concurrent delay rules at the same time, potentially on different hosts. To solve this need to scale up the processing of delay rules, we added an admin-defined list of eligible executors (servers) which could receive delay rules and context and run them in a distributed manner. To get this behavior by default, we also implemented an implicit `remote()` call (issue 4429 [7]) when processing the in-memory set which would send rules to a selected member of the list of executors (for now, in a round robin, or sequential, manner). The default and upgrade behavior is to have an empty list of executors which will use the local server as the only executor.

DESIGN GOALS

The sections above describe the work that was necessary to prepare for a future with a delay server migration algorithm running. The following design goals were defined through community discussion and brainstorming. There were four main goals.

1. No `irodsServer` restarts required. There are very large deployments under continuous load and a restart would be very disruptive, and probably require a maintenance window. This is unacceptable if a single delay server unexpectedly going away could cause the same disruption.
2. No double spends. Only one `irodsDelayServer` can be running in a particular Zone at any moment as it is the one pulling enqueued rules from the catalog, executing them, and then removing them from the catalog. This is paramount for ensuring organizational policy is followed, as a delay rule shall not be executed more than once.
3. Hands-free migration in case of disaster. If the `irodsDelayServer` is not coming back (the machine it is running on is dead or has been disconnected), then another machine in the Zone should be promoted to run the `irodsDelayServer` without any new action from the administrator.
4. Visibility. The delay server migration process should have as few moving parts and controls as possible, making it easy to interrogate and debug for the administrator.

These four elements led to the following approach.

APPROACH

The following three operational decisions were made to satisfy the design goals.

1. Use the transactional database to store zone-wide information. This provides a single source of truth and allows for confidence that all servers can act on the same information.

2. Split roles of leader and successor. To allow for the asynchronous nature of multiple servers agreeing on their duties as assigned, splitting the leader from the successor allows each server to run appropriate code and make correct decisions.

3. Run an identical algorithm on all iRODS servers. Each server is responsible for their own behavior but together, they provide a predictable, consistent response during a migration event.

The roles are designated as zone-wide settings in the catalog in the `R_GRID_CONFIGURATION` table.

namespace	option_name	option_value
delay_server	leader	<hostname>
delay_server	successor	<hostname>

Table 1. New configuration options

DEMONSTRATION

The following sequence of administrator shell commands shows the interrogation of the delay server status from the catalog, the setting of the new delay server, and then watching as the values change as the servers execute the algorithm and perform a clean delay server migration.

```
$ hostname
05f4be918c0f

$ iadmin get_delay_server_info
{
    "leader": "other.server.example.org",
    "successor": ""
}

$ iadmin set_delay_server $(hostname)

$ iadmin get_delay_server_info
{
    "leader": "other.server.example.org",
    "successor": "05f4be918c0f"
}

$ iadmin get_delay_server_info
{
    "leader": "05f4be918c0f",
    "successor": ""
}
```

Demonstrated are the two new `iadmin` subcommands, `get_delay_server_info` and `set_delay_server`.

The help text for these two subcommands are as follows:

```
$ iadmin h get_delay_server_info
```

```
get_delay_server_info

Prints information about the delay server as JSON.

This command allows administrators to identify which server is running the
delay server and if the delay server is being migrated.

This information is retrieved from the R_GRID_CONFIGURATION database table.

Example Output:

    {
        "leader": "consumer-1.irods.org",
        "successor": ""
    }

$ iadmin h set_delay_server
set_delay_server HOSTNAME

Set the delay server for the local zone in R_GRID_CONFIGURATION.

The hostname entered will be saved as the 'successor'.

Each iRODS server will periodically check the catalog to determine
if it should promote itself to be the delay server for the local zone.

This mechanism allows for graceful delay server migration without downtime.
```

ALGORITHM

This algorithm is designed to run regularly on all servers in a zone. It is in charge of starting and then stopping a local instance of the `irodsDelayServer`.

```
if self == leader
    if successor defined and not self
        gracefully finish and exit
    else
        if necessary, start irodsDelayServer
else if self == successor
    run health check on leader
    if leader is not running
        promote self to leader in iCAT
    else
        save health stats
else
    if necessary, gracefully finish and exit
```

There are three roles a particular server can find itself in, leader, successor, or neither.

1. Leader.

If the server running the algorithm checks the catalog and determines that the designated leader hostname matches its own hostname, then the first stanza is executed. If there is a successor designated in the catalog as well, and it is not also the same as this server's hostname, then a migration has been requested by the administrator and it is time for this server to request the local `irodsDelayServer` complete any delay tasks that are already being executed and then gracefully exit.

Otherwise, if the server is not currently running an `irodsDelayServer`, then it should start a local instance.

2. Successor.

   If the check of the catalog produces a hostname for successor that matches the local hostname, then it becomes the job of this server to monitor the leader for signs of health. If the leader is still running its own `irodsDelay-Server` (meaning that it has not yet gracefully exited), then the successor should save into the catalog that it has checked once and plans to try again on the next check. If the leader is not running (or if the health check is deemed to have failed because of repeated non-response), then the successor promotes itself to leader within the catalog and returns. The next time through the algorithm this server will find it matching the 'leader' condition (above).

3. Neither.

   If the check of the catalog produces no match for either leader or successor with the local hostname, then this server should tell any local `irodsDelayServer` to complete any work it has and then gracefully exit.

The two-phase promotion of a server to 'leader' in this algorithm should behave in the following scenarios:

1. Servers behaving as expected.

   Only the leader will ever start an `irodsDelayServer`. If requested, the leader will let go as soon as it has completed its current work. All other servers will do the same if not designated as the leader. The successor will not promote itself until the leader has let go.

2. The current leader is non-responsive.

   If the current leader has suffered an unrecoverable error or has been disconnected, then the successor performing a series of health checks can promote itself to leader. Only the successor can promote itself to leader.

3. The current leader has long-running jobs.

   The health check by the successor is implemented by asking for the PID of the `irodsDelayServer` on the leader. If the leader continues to answer that it is healthy, and just continuing to process its current rules, then the successor will simply wait and check again.

4. Fast switching of the successor by the administrator.

   If an administrator has changed the successor after the migration has begun, but before the entire algorithm has settled on a new leader and started the new `irodsDelayServer`, then as long as the checks on each machine are running in a well-spaced manner, then two servers should never decide they are both the leader at the same time. For additional confidence, a 'leader' confirmation may be performed prior to each time an `irodsDelayServer` retrieves new delay rules to execute.

LEARNED ALONG THE WAY

There were a few things that proved themselves tricky along the way to arriving at an implementation that satisfied the vision.

1. Database credentials.

   The decision to have the main `irodsServer`, via the cron-like facility, reach out and contact the database directly to gather the information about the leader and successor meant that every server has to have database credentials in their `server_config.json` file. This will be remedied in later releases.

2. Control plane as process / Blocking ourselves.

   The health check required by the successor to ask whether the current leader is still running an `irodsDelayServer` meant that a network request was sent but it could not be answered due to the main loop of the `irodsServer` waiting to fork an Agent to answer the request. This required moving the cron-like facility into its own thread, similar to the control plane, and unblocking the main server from answering incoming requests.

3. Who is the parent process?

   In learning about the control plane above, we learned to write down a PID-file to help other processes be able to identify the parent process and answer whether the `irodsDelayServer` was still running.

FUTURE WORK

This work is complete and functional and included as part of iRODS 4.3.0. We expect to learn a few things once deployments are in the real world, however, the following items are optimizations we have already identified and plan to include in future releases.

1. Remove database credentials requirement

   This requirement is a short-term limitation and will be removed relatively soon.

2. Detect and skip a redundant implicit `remote()`

   The current implementation is wrapping all calls in the `remote()` function. We expect that a simple check to compare the selected remote host against the current hostname will allow skipping a redundant call to `remote()`.

3. Advanced setting for sleep time between migration algorithm runs

   As shipped in 4.3.0, the delay server migration algorithm is set to execute every five seconds. We were trying to make sure that this new feature would be responsive enough to experimenting administrators before providing a configuration. We plan to introduce a new advanced setting for this value, including the possibility of configuring the server to never run the algorithm.

CONCLUSION

This paper discussed the design, implementation, and future work around the new delay server migration algorithm in iRODS 4.3.0. We expect this work to feature prominently in production workloads as many asynchronous tasks are being enqueued for later execution.

REFERENCES

[1] iRODS GitHub Issue 4250. ”irodsReServer blocks on in-progress jobs”
`https://github.com/irods/irods/issues/4250`

[2] iRODS GitHub Issue 4251. ”Use thread pool in rule execution server”
`https://github.com/irods/irods/issues/4251`

[3] iRODS GitHub Issue 3782. ”delayed rule queue processing slows down as the queue length grows”
`https://github.com/irods/irods/issues/3782`

[4] iRODS GitHub Issue 4266. ”Having at least 256 rules in the queue prevents new rules from being processed”
`https://github.com/irods/irods/issues/4266`

[5] iRODS GitHub Issue 4430. ”Refactor delay server as an irods::query_processor”
`https://github.com/irods/irods/issues/4430`

[6] iRODS GitHub Issue 3049. ”Move packedReis to db, add delay server boolean to server_config.json”
`https://github.com/irods/irods/issues/3049`

[7] iRODS GitHub Issue 4429. ”Add implicit remote() to delayed rule execution”
`https://github.com/irods/irods/issues/4429`