iRODS HTTP API

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

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

ABSTRACT

The iRODS Protocol has remained relatively static for more than 20 years. This is a testament to its original planning, but also means any redesign would carry a heavy upgrade and migration cost. Additionally, the protocol is novel to most developers and differs in implementation across client programming languages which hurts both approachability and adoption. This paper covers the design, implementation, and early performance results of a new HTTP API for interacting with iRODS.

Keywords

iRODS, http, api, cpp

WHAT AND WHY

The new iRODS HTTP API [1] is an experimental redesign of the relatively recent iRODS C++ REST API [2] originally developed and released in 2021. The HTTP API has renewed focus on consistency and familiarity, as well as maintainability into the future. The main goals of the project are to:

- Present a cohesive representation of the entire iRODS API over the HTTP protocol, effectively simplifying development of client-side iRODS applications for new developers. If all iRODS functionality is exposed via HTTP, then a significant number of existing developers can work with iRODS without having to climb a relatively steep learning curve.
- Maintain performance close to the iCommands. The iCommands have been around for nearly 20 years and have proven efficient for their focused responsibilities. Any new solution should have a similar (if not better) performance profile.
- Remove behavioral differences between client-side iRODS libraries by building new libraries on top of the HTTP API (including C, C++, Java, and Python). Since most languages now provide standard libraries around making HTTP calls, language-specific wrappers that call the HTTP API could be very thin, proving both easy to produce and easy to maintain.
- Potentially be absorbed by the iRODS server if adoption is significant. The world has chosen HTTP as the most-favored protocol for getting things done, and apart from large data movement, it should be considered as the primary candidate protocol for exposing the functionality of an iRODS namespace.

The iRODS C++ REST API proves that presenting iRODS as HTTP is possible, however, usage of the project over the last couple of years has uncovered some challenges.

iRODS UGM 2023 June 13-16, 2023, Chapel Hill, NC

- Too many open ports raise security concerns. The security personnel tasked with asking lots of questions and understanding the traffic on their networks are suspicious of a novel protocol and do not like opening the standard 200+ ports for iRODS parallel transfer.
- Stability issues (e.g. crashing endpoints). This would have gotten better over time, but the relatively high number of moving parts was already feeling brittle.
- Separation of endpoints increases complexity due to multiple layers. The design is good, but appears to be overkill for the work needing to be done today. We may yet end up with more abstraction in the future, but it felt to be too much without a corresponding requirement.
- Pistache HTTP library [3] lacks completeness/maturity/adoption. This iRODS HTTP API project needs to provide a solid foundation that other tools and layers can depend on. Pistache was not going to provide this stability and confidence in the near-term.
- The names of existing endpoints are fairly general which leads to difficulty in naming of new endpoints. Getting an API correct at the beginning is hard, and refactoring is even harder. The current endpoints had already boxed in our ability for providing consistency moving forward.

The iRODS HTTP API is aimed at resolving these issues by taking a different approach based on what we have learned from the community and the iRODS S3 API [4].

DESIGN

With the design goals firmly in place, an initial brainstorming document (in the form of a GitHub gist [5] in April 2023) laid out a fairly comprehensive listing of this new HTTP API effort. Some of the early decisions included:

- This new API would be run as a single binary exposing one (or two) ports. This will be easier to handle by administrators and be more expected by any interested security teams.
- The library of choice would be Boost.Beast [6], a proven C++ header-only library providing networking facilities for building high performance libraries and applications which need support for HTTP/1 and Websockets. It beat out other candidate libraries and was selected earlier in the year as the basis for the iRODS S3 API and proved the 'right answer'.
- The set of URL endpoints would be fixed and expected for iRODS users. They will represent the 'nouns' of iRODS and be easy to remember and 'guess'.
- The API itself would be renamed from REST to HTTP to reflect the reality that the rules of REST are not clear or consistent and that the statefulness of the iRODS API does not map as well to the singular nature of REST verbs.

API Endpoints and Operations

The endpoints of the new iRODS HTTP API will reflect the well-known nouns of iRODS and provide a stable baseline for exploration and experimentation (as well as documentation). These endpoints are named based on concepts and entities defined in iRODS (in alphabetical order):

- /authenticate
- /collections
- /data-objects
- /info

- /metadata
- /query
- /resources
- /rules
- /tickets
- /users-groups
- /zones

Operations associated with these endpoints are specified via parameters. This decision keeps URLs simple (i.e. top-level only, no nesting required) and allows both new and existing developers to make educated guesses as to which URL exposes the behavior they are interested in.

For example, if a developer wants to modify a user, they would explore `/users-groups`. Or, writing data to a data object would happen through the `/data-objects` endpoint.

It is early days for this new API, but these endpoints should prove to be stable.

API Parameters

At this time, all endpoints, except `/authenticate`, accept an operation (`op`) parameter which is mapped to a function responsible for executing the requested operation. There can be multiple operations on each endpoint and represent the 'verbs' associated with that 'noun'.

The operations available across multiple endpoints will share common values as much as possible (e.g. stat, list, create, remove, etc.). This is designed to make the entire API more predictable, familiar, and approachable for both new and returning developers.

Some of the common parameters used so far include:

- lpath
- replica-number
- src-resource
- dst-resource
- offset
- count

CONFIGURATION

Tools to generate, read, parse, and manipulate JSON files have proven to be available nearly everywhere and has become the standard in the iRODS ecosystem. For these reasons, the iRODS HTTP API will be driven by a single JSON configuration file. It has two halves, one for each of the two 'roles' the API is playing: an HTTP server, and an iRODS client. This format is modeled after NFSRODS [7].

```
{
    // Defines HTTP options that affect how the
    // client-facing component of the server behaves.
    "http_server": {
        // ...
    },

    // Defines iRODS connection information.
    "irods_client": {
        // ...
    }
}
```

http_server

The HTTP server configuration currently allows for controlling the listening host and port, the log level, authentication options, and threading and timeout options. These are the preliminary values and we expect these to change as we explore deployment use cases, optimize for memory and bandwidth, and learn more about how this API plays with other software (including high-availability configurations and reverse proxies for TLS/SSL termination).

```
"http_server": {
    "host": "0.0.0.0",
    "port": 9000,

    "log_level": "warn",

    "authentication": {
        "basic": {
            "timeout_in_seconds": 3600
        }
    },

    "requests": {
        "threads": 3,
        "max_rbuffer_size_in_bytes": 8388608,
        "timeout_in_seconds": 30
    },

    "background_io": {
        "threads": 3
    }
}
```

irods_client

The iRODS client side of the configuration is very straightforward and will change less dramatically as we continue development. It covers the host, port, and zone, as well as the proxy `rodsadmin` information. This API must be deployed as an administrator since it holds a long-running connection pool with the iRODS server and uses the `rc_switch_user` functionality to proxy for regular user operation requests (read more in the next section). The remaining settings are to protect the iRODS HTTP API memory footprint.

```
"irods_client": {
```

```
    "host": "<string>",
    "port": 1247,
    "zone": "<zone>",

    "proxy_rodsadmin": {
        "username": "<string>",
        "password": "<string>"
    },

    "connection_pool": {
        "size": 6,
        "refresh_timeout_in_seconds": 600
    },

    "max_rbuffer_size_in_bytes": 8192,
    "max_wbuffer_size_in_bytes": 8192,

    "max_number_of_rows_per_catalog_query": 15
}
```

CONNECTION POOLING

Historically, iRODS clients have assumed a simple back and forth flow of messages with the iRODS server. Each operation traditionally required a connect and disconnect as well as authentication. For many use cases, this can kill performance on anything other than large transfers. This insight led to two enhancements being added to the iRODS 4.3.1 server. First, proxy user support for `irods::connection_pool` and `irods::client_connection`. This is not yet merged, but available as pull request #7047 [8]. Second, a new client call named `rc_switch_user` which allows the identity associated with an `RcComm` to be changed in real-time. This work can be found in pull request #6691 [9].

With these new facilities, the iRODS HTTP API can reuse existing iRODS connections to significantly boost performance.

The following code example shows how a client can use the connection pool, switch the associated user, and return a live connection to the caller.

```
// TODO May require the zone name be passed as well for federation?
auto get_connection(const std::string& _username)
    -> irods::connection_pool::connection_proxy
{
    namespace log = irods::http::log;

    auto& cp = irods::http::globals::conn_pool;
    auto conn = cp->get_connection();
    const auto& zone = irods::http::globals::config->at("irods_client")
        .at("zone").get_ref<const std::string&>();

    log::trace("{}: Changing identity associated with connection to [{}].",
               __func__, _username);

    auto* conn_ptr = static_cast<RcComm*>(conn);
    const auto ec = rc_switch_user(conn_ptr, _username.c_str(), zone.c_str());
```

```
    if (ec != 0) {
        log::error("{}: rc_switch_user error: {}", __func__, ec);
        THROW(SYS_INTERNAL_ERR, "rc_switch_user error.");
    }

    log::trace("{}: Successfully changed identity associated with connection to [{}].",
               __func__, _username);

    return conn;
} // get_connection
```

PARALLEL WRITES

Another aspect of providing a middle-tier API is the delicate business of handling parallel connections from the client (and through to the associated iRODS server). iRODS does not allow a data object to be written directly in parallel without coordination (in the form of a replica access token).

iRODS clients wanting to upload data in parallel are required to do the following:

1. Open a stream to the replica of interest.
2. Capture the Replica Access Token from the stream.
3. Open secondary streams.
    - Each stream must use its own connection
    - Each stream must target the same replica
    - Each stream must use the same open flags
    - Each stream must pass the Replica Access Token obtained from the stream in step (2)
4. Send bytes across streams.
5. Close secondary streams without updating the catalog.
6. Close the original stream normally.

This set of steps is now currently fully supported in the iRODS HTTP API through the use of a "Parallel Write Handle". However, this means the iRODS HTTP API server maintains state on behalf of the client.

Therefore, performing a parallel write requires the use of two operations. The first, `parallel_write_init`, instructs the server to allocate memory for managing the state of the upload. The second, `parallel_write_shutdown`, instructs the server to deallocate memory obtained via the earlier call to `parallel_write_init`.

Transfers of large files must use `multipart/form-data` as the HTTP content type. Failing to honor this series of steps will result in an error or corrupt data in the iRODS server.

The following example shows how to obtain, use, and then close the parallel write handle:

```
http_api_url="${base_url}/data-objects"

# Open 3 streams to the data object, file.bin.
```

```
transfer_handle=$(curl -H "Authorization: Bearer $bearer_token" "$http_api_url" \
  --data-urlencode 'op=parallel_write_init'                                      \
  --data-urlencode "lpath=/tempZone/home/rods/file.bin"                          \
  --data-urlencode 'stream-count=3'                                              \
  | jq -r .parallel_write_handle)

# Write "hello" (i.e. 5 bytes) to the data object.
# Notice we didn't specify which stream to use.
curl -H "Authorization: Bearer $bearer_token" "$http_api_url" \
  -F 'op=write'                                               \
  -F "parallel-write-handle=$transfer_handle"                 \
  -F 'count=5'                                                \
  -F 'bytes=hello;type=application/octet-stream'              \
  | jq

# Shutdown all streams and update the catalog.
curl -H "Authorization: Bearer $bearer_token" "$http_api_url" \
  --data-urlencode 'op=parallel_write_shutdown'               \
  --data-urlencode "parallel-write-handle=$transfer_handle"   \
  | jq
```

PERFORMANCE

The initial release of the iRODS HTTP API is tested here. These results are preliminary and we expect future releases to be more performant as the API is deployed into more diverse scenarios.

Throughput

This first test harness included two machines, one on a ”home” network, connected over the commodity internet, to another machine at ”work”. The test program is a custom-built, non-optimized Java application which speaks to the v0.1.0 iRODS HTTP API. The test program uses the multi-1247 parallel-write-handle approach mentioned in the section above. In Table 1, this Java application is compared to `iput` which was configured to use multiple high ports for traditional iRODS parallel transfer.

Client	Protocol	Average Elapsed Time (seconds)
iput (high ports)	iRODS	50.113
java application (multi-1247)	HTTP API + iRODS	51.975

Table 1. Upload speed comparison (n=5)

Each application used 4 threads to upload a 100 MiB test file into iRODS 4.3.0 five times and then the elapsed time was averaged. The Java application and HTTP API added a 3.7% overhead compared to `iput`. Additional testing showed that performance is sensitive to buffer sizes and the number of threads used. These results show that comparable performance to `iput` is within reach to many new clients with simple wrappers around the HTTP API.

Requests Per Second

This second test used ApacheBench [10] to measure Requests Per Second (RPS) supported by the HTTP API. Configured to send 2000 total requests, the HTTP API maintained 500 concurrent requests at all times. All of these tests were performed with a single development machine with 32 cores and 256 GiB of RAM. The iRODS server was a custom build of iRODS ˜4.3.1 (not yet released) which supported `rc_switch_user` and included updates to the connection pool library. The iRODS HTTP API had optimizations enabled in the build and was configured with 32 threads each for foreground and background processing.

Endpoint	Requests Per Second	50% of requests served (in seconds)
/authenticate	133.20	3.670
/data-objects	697.18	0.686
/resources	2599.53	0.167

Table 2. Requests Per Second (n=2000)

In Table 2, the response times for authenticating a new user using basic/native authentication, reading 8192 bytes of an iRODS data object, and stat'ing a resource are shown. Authentication is the heaviest amount of work and the HTTP API handled 133 requests per second with 50% of requests being served within 3.6 seconds. Reading 8k of data from a data object within iRODS (on a single server) was in the middle with nearly 700 requests per second with 50% of requests being served within two-thirds of a second. Executing a stat operation on a resource was much quicker with nearly 2600 requests per second with 50% of requests being served within one-sixth of a second.

Requests that would traverse to a second or third iRODS server would take more time in a topology test environment. That was not tested here.

EXAMPLES

The following examples show some requests and responses from the iRODS HTTP API. Responses are in JSON and are shown here parsed through the command line tool `jq` for clarity and readability.

stat'ing a collection

```
base_url="http://localhost:9000/irods-http-api/0.1.0"
bearer_token=$(curl -sX POST --user 'rods:rods' "${base_url}/authenticate")

curl -sG -H "Authorization: Bearer $bearer_token" \
  "${base_url}/collections"                       \
  --data-urlencode 'op=stat'                      \
  --data-urlencode 'lpath=/tempZone/home/rods'    \
  | jq
```

```
{
  "inheritance_enabled": false,
  "irods_response": {
    "status_code": 0
  },
  "modified_at": 1686499669,
  "permissions": [
    {
      "name": "rods",
      "perm": "own",
      "type": "rodsadmin",
      "zone": "tempZone"
    }
  ],
  "registered": true,
  "type": "collection"
}
```

listing available rule engine plugins

```
base_url="http://localhost:9000/irods-http-api/0.1.0"
bearer_token=$(curl -sX POST --user 'rods:rods' "${base_url}/authenticate")

curl -sG -H "Authorization: Bearer $bearer_token" \
  "${base_url}/rules"                             \
  --data-urlencode 'op=list_rule_engines'         \
  | jq

{
  "irods_response": {
    "status_code": 0
  },
  "rule_engine_plugin_instances": [
    "irods_rule_engine_plugin-irods_rule_language-instance",
    "irods_rule_engine_plugin-cpp_default_policy-instance"
  ]
}
```

REMAINING WORK

This initial release covers basic endpoints and validates the approach for this new API. There are a number of items in the near term that need to be added to make this useful for others. This list includes:

- Implement tests
- Improve performance of /authenticate endpoint
- Consider batch / bulk operations
- Add support for GenQuery2
- Clean up implementation for contributors
- Fill out documentation
- Define good defaults for I/O-specific configuration properties
- Add support for Docker and Docker-Compose
- Expose SSL configuration properties for iRODS communication

FUTURE PLANS

Other considerations we are thinking about, but do not yet have clarity on, include the following possible future directions:

- Consider adding options for enabling/disabling features, endpoints, etc
  - The iRODS C++ REST API supported this for all endpoints
    - Is this the responsibility of the proxy (e.g. nginx, apache httpd)?
- Consider how to best support load balancers
  - Parallel Writes are stateful

- Consider how to deal with long running agents containing stale information
  - Should we refresh the connection after N number of API operations?
  - Should we refresh the connection after certain API operations?
    - e.g. Resource management operations

We are hoping that our approach of providing well-known and well-defined HTTP accessibility to the iRODS Protocol will bring in new developers and new use cases and better expose the flexibility and power of the iRODS server.

REFERENCES

[1] iRODS HTTP API `https://github.com/irods/irods_client_http_api`

[2] Coposky, J., Russell, T. (2021) iRODS Client: C++ REST API. iRODS User Group Meeting 2021. `https://irods.org/uploads/2021/Coposky-iRODS-C_Plus_Plus_REST_API-paper.pdf`

[3] Pistache: An elegant C++ REST Framework. `https://pistacheio.github.io/pistache/`

[4] Russell, T., White, V. (2023) iRODS S3 API: Presenting iRODS as S3. iRODS User Group Meeting 2023. `https://irods.org/uploads/2023/Russell-iRODS-S3-API_Presenting_iRODS_as_S3-paper.pdf`

[5] Draughn, K., Russell, T., King, A. (2023) Can the REST API be improved? `https://gist.github.com/korydraughn/78ec96120234659db1c2ba3235efa46c`

[6] Boost.Beast `https://github.com/boostorg/beast`

[7] Draughn, K., Russell, T., Mieczkowski, A., Coposky, J., Conway, M. (2020) iRODS Client: NFSRODS 1.0. iRODS User Group Meeting 2020. `https://irods.org/uploads/2020/Draughn-iRODS-NFSRODS_v1.0.0-paper.pdf`

[8] PR #7047. Client connection libraries allow deferring/changing the auth logic `https://github.com/irods/irods/pull/7047`

[9] PR #6691. Implemented new API plugin: rc_switch_user `https://github.com/irods/irods/pull/6691`

[10] ApacheBench `https://httpd.apache.org/docs/2.4/programs/ab.html`