iRODS®

iRODS HTTP API

Kory Draughn
Chief Technologist
iRODS Consortium

June 13-16, 2023
iRODS User Group Meeting 2023
Chapel Hill, NC

Overview

- What is the iRODS HTTP API?
- Why is this necessary?
- Design
- Configuration
- Connection Pooling
- Parallel Writes
- General Performance
- Examples
- Remaining Work
- Future Plans

What is the iRODS HTTP API?

An experimental redesign of the iRODS C++ REST API.

Goals of the project ...

- Present a cohesive representation of the iRODS API over the HTTP protocol, effectively simplifying development of client-side iRODS applications for new developers
- Maintain performance close to the iCommands
- Remove behavioral differences between client-side iRODS libraries by building new libraries on top of the HTTP API
    - C, C++, Java, Python, etc - all languages produce identical behavior and results
- Absorbed by the iRODS server if adoption is significant

Why is this necessary?

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

Challenges ...

- Too many open ports raise security concerns
- Stability issues (e.g. crashing endpoints)
- Separation of endpoints increases complexity due to multiple layers
    - e.g. Interns found it difficult to understand how things are composed
- Pistache HTTP library lacks completeness/maturity/adoption
- Names of existing endpoints are fairly general which leads to difficulty in naming of new endpoints

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

To view the original document which kick-started this effort, click here.

Design - Early Decisions

- Single binary exposing one (or two) ports
- Boost.Beast
  - A C++ header-only library providing networking facilities for building high performance libraries and applications which need support for HTTP/1 and Websockets
  - First used by the iRODS S3 API
- Fixed set of URLs
  - Easy for users and developers to remember
- Renamed from REST to HTTP
  - The rules of REST are not clear
  - The rules of REST do not map well to the iRODS API
  - iRODS is stateful
  - Focus on designing the best API we can

Design - API URLs

Named based on concepts and entities defined in iRODS.

/authenticate
/collections
/data-objects
/info
/metadata
/query
/resources
/rules
/tickets
/users-groups
/zones

Operations are specified via parameters. This decision keeps URLs simple (i.e. **no nesting required**) and allows new/existing developers to guess which URL exposes the behavior they are interested in.

For example, if you want to modify a user, look at /users-groups. Or, perhaps you need to write data to a data object, then you'd use /data-objects.

Design - API Parameters

All URLs, except /authenticate, accept an **op** parameter.

- Mapped to a function responsible for executing the requested operation
- Shares common values where possible
  - e.g. stat, list, create, remove, etc

Common parameters used through out the API ...

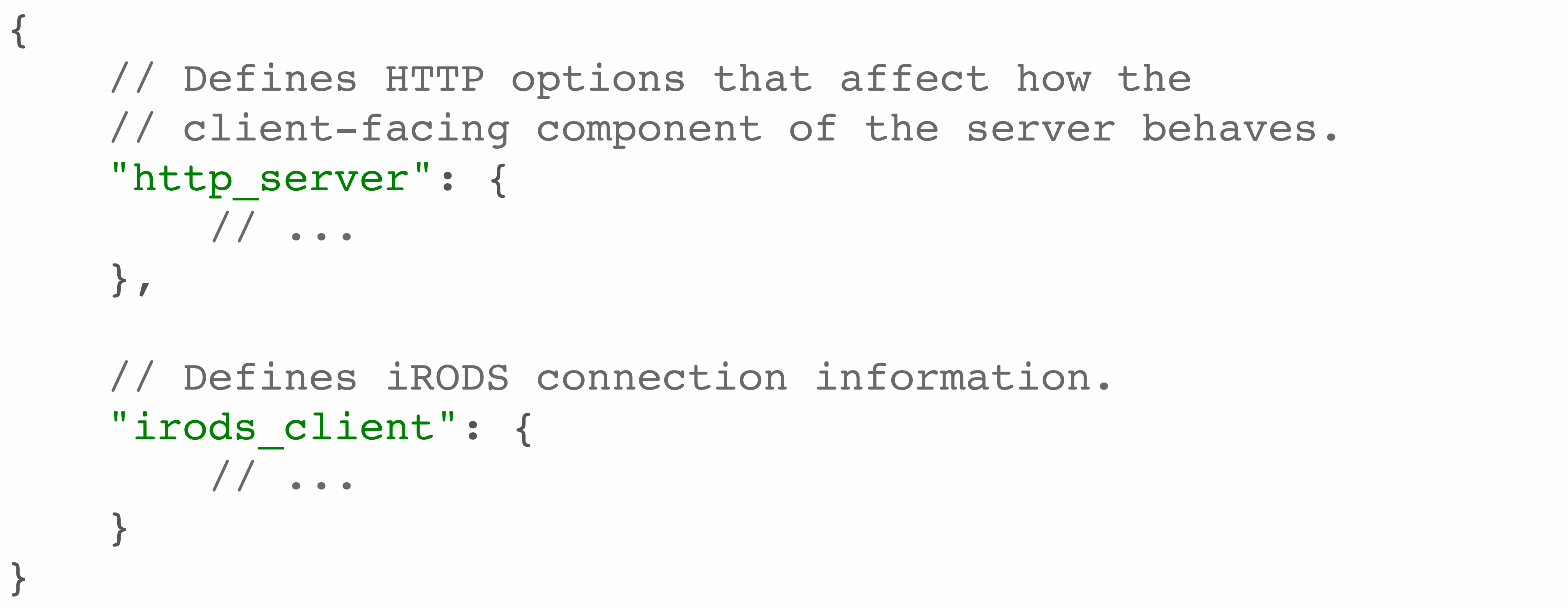
- lpath
- replica-number
- src-resource
- dst-resource
- offset
- count

Parameter names are not final and may change in the future.

Configuration - Top Level

Defines two sections to help administrators understand the options and how they relate to each other.

Modeled after NFSRODS.



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

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

Configuration - http_server

```
"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
    }
}
```

Configuration - irods_client

```
"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

iRODS clients connect and disconnect frequently.

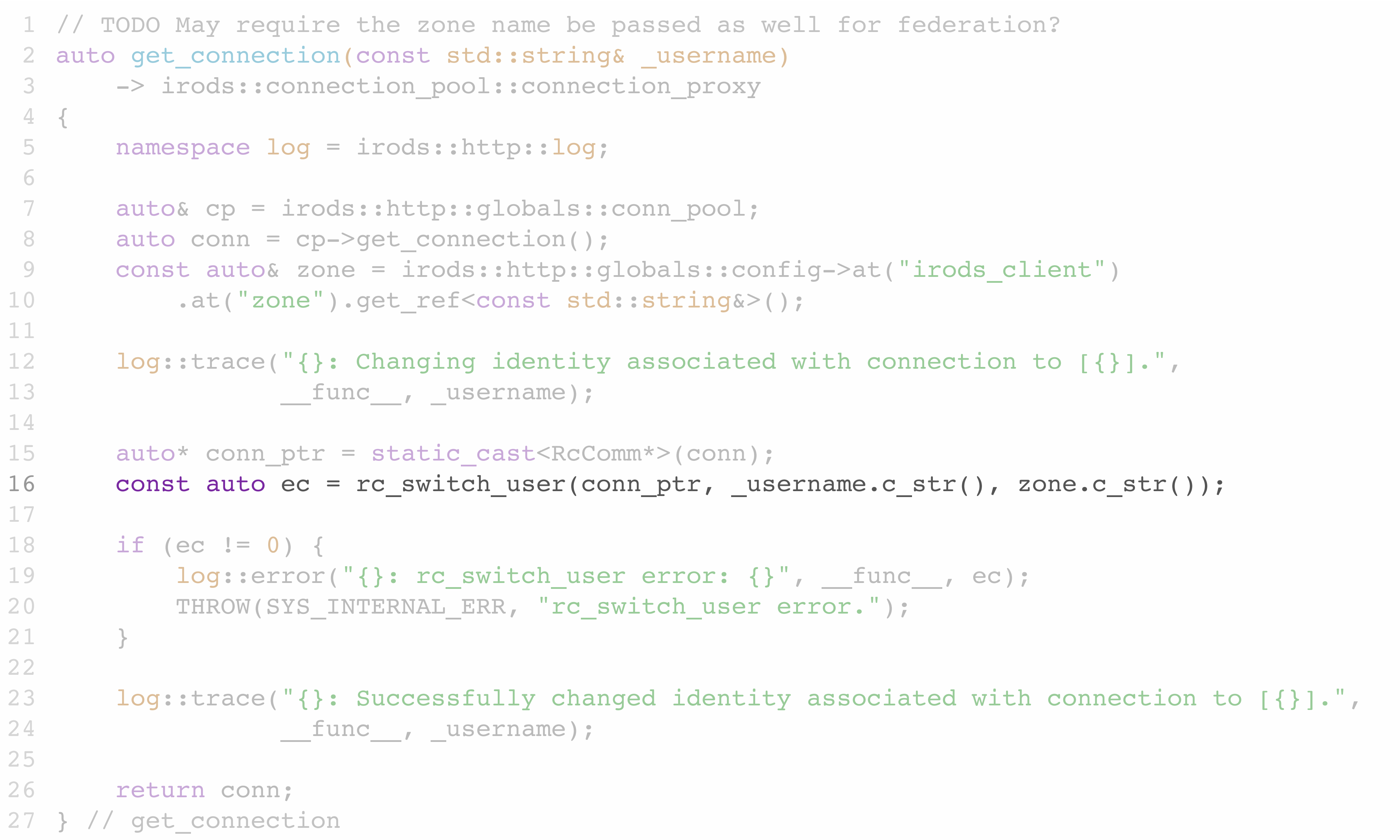
This kills performance!

This issue resulted in the following enhancements for iRODS 4.3.1 ...

- Proxy user support for irods::connection_pool and irods::client_connection
  - Not yet merged (see PR #7047 for details)
- rc_switch_user
  - Allows the identity associated with an *RcComm* to be changed in real-time
  - Original work can be found in PR #6691

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

Connection Pooling - Implementation



```
// 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

iRODS does not allow a data object to be written to in parallel without coordination.

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 (1)
4. Send bytes across streams.
5. Close secondary streams without updating the catalog.
6. Close the original stream normally.

Parallel Writes

Fully supported through the use of a **Parallel Write Handle**.

This ultimately means, the iRODS HTTP API server maintains state on behalf of the client.

Performing a Parallel Write requires the use of two operations ...

- parallel_write_init
  - Instructs the server to allocate memory for managing the state of the upload
- parallel_write_shutdown
  - Instructs the server to deallocate memory obtained via a call to parallel_write_init

Large files must use multipart/form-data as the content type. Failing to honor this rule will result in an error or corrupt data.

Parallel Writes - Example

Demonstrates how to open 3 streams to a data object and write 5 bytes to it.

```
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
```

Parallel Writes - Java application vs iput

- Testing was carried out using two machines in different locations
  - Home network vs Office network
- Custom Java application built on top of the iRODS HTTP API
  - Not optimized
- Each application used **4 threads** to upload a **100 MiB** file into iRODS

Client	Time Elapsed
iput *(uses high ports)*	50.113s
Java application	51.975s

Performance is sensitive to buffer sizes and number of threads used.

General Performance - Test Environment and Setup

- Used ApacheBench to measure Requests Per Second (RPS)
  - Sent 2000 requests total
  - Maintained 500 concurrent requests at all times
- All testing was performed using a single machine
  - Development machine has 32 cores with 256 GiB of RAM
  - Custom build of iRODS ~4.3.1
    - Supports rc_switch_user and changes to connection pool library
  - iRODS HTTP API
    - Optimizations enabled
    - 32 threads for foreground processing
    - 32 threads for background processing

General Performance - Test Results

- /authenticate - Authenticating a new user using Basic/Native authentication
    - 133.2 RPS
    - 50% of requests took at least 3670 ms to serve
- /resources - Stat'ing a resource
    - 2599.53 RPS
    - 50% of requests took at least 167 ms to serve
- /data-objects - Reading 8192 bytes
    - 697.18 RPS
    - 50% of requests took at least 686 ms to serve

Examples - Stat'ing a collection

```
base_url="http://localhost:9000/irods-http/0.9.5"
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": {
    "error_code": 0
  },
  "modified_at": 1686499669,
  "permissions": [
    {
      "name": "rods",
      "perm": "own",
      "type": "rodsadmin",
      "zone": "tempZone"
    }
  ],
  "registered": true,
  "type": "collection"
}
```

Examples - Listing available Rule Engine Plugins

```
base_url="http://localhost:9000/irods-http/0.9.5"
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": {
    "error_code": 0
  },
  "rule_engine_plugin_instances": [
    "irods_rule_engine_plugin-irods_rule_language-instance",
    "irods_rule_engine_plugin-cpp_default_policy-instance"
  ]
}
```

Remaining Work

- Implement tests
- Improve performance of /authenticate endpoint
- Consider batch / bulk operations
- Add support for GenQuery2
- Clean up implementation for contributors
- Finish 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

- Consider adding options for enabling/disabling features, endpoints, etc
  - The iRODS C++ REST API supported this for all endpoints
    - Is this the responsiblity 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

Thank you!

Questions?

https://github.com/irods/irods_client_http_api