Refactoring Kanki - Towards a Modern Native iRODS Client Implementation

Ilari Korhonen
PDC Center for High Performance Computing, KTH
The 13th Annual iRODS User Group Meeting, June 10th, 2021
Background

• The Kanki project was started by me back in 2014 at my previous job while working with research IT at University of Jyväskylä, Finland.

• Problem was that there was (and as far as I know, still is) no other (even relatively current) native iRODS client with a graphical user interface, which could be deployed for all the major desktop platforms, Mac, Windows and Linux/X11.

• After presenting at the iRODS UGM 2015, Kanki was open sourced by the university (BSD 3-Clause) and has been available in GitHub at URL:

  [https://github.com/ilarik/kanki-irodsclient](https://github.com/ilarik/kanki-irodsclient)

• Sadly, since lack of time at my current work, at KTH, there has been modest progress since 2016, up until recently.

• The work was picked up again in late 2020 in collaboration with the iRODS Consortium and we have started the process of refactoring the client-side interfaces in the code from the old C-compatible APIs into a fully-fledged C++17 object-oriented interface.
About Kanki

• Written fully in C++ on top of the Qt framework, which makes the vast majority of the code already fully portable across Linux / MacOS / Windows / …

• Feature set includes the basic iRODS operations, a customisable metadata editor, for which one can write a (simple) schema definition (to be improved!), also an extensible iRODS search UI with the full capabilities of GenQuery.

• Uses the native iRODS client library for iRODS connectivity - in practise this means that one can build Kanki where Qt and irods-icommands are supported.

• Supports all the same authentication and transfer mechanisms as icommands, however UI support for configuring the authentication needs to be improved.

• Previous packaged builds were built against iRODS 4.1.x.

• From the development branches in GitHub, one can build it against 4-2-stable on Linux, currently. We are working on the MacOS build with the good people at University of Arizona / CyVerse.
The Future

• Since version 4.0.0 iRODS has been gradually refactored into a C++ code base, from the old iRODS 3.3.1 pure C code, but the client-side libraries so far have remained almost fully C-compatible - up until recently.

• This new development not only changes the API interface in the code from a procedural to an object-oriented one but also introduces new functionalities, including connection pooling and client-side thread pool management - in a coherent fashion within the client framework.

• Previously, introducing parallelism within a single iRODS client session has been a source of pain, due to the structure of an agent process and its synchronous nature.

• Nowadays parallelism in the server processes is increasing. Additionally, the introduction of facilities in the client library supporting asynchronous execution of tasks, makes it possible to painlessly run tasks in parallel, asynchronously.
Connection Pools and Thread Pools

• Newer versions of the iRODS client library include a connection pool facility, using which the client can instantiate and manage $n$ simultaneous connections to the same iRODS endpoint.

  irods::connection_pool + irods::connection_pool::connection_proxy

• From a successfully established connection pool one can take out a connection and reserve that for a task at hand, releasing the connection afterwards.

• A similar concept of a thread pool has been incorporated into the later client versions, based on Boost ASIO.

  irods::thread_pool

• One can instantiate thread pools of $m$ threads and spin up worker threads for tasks to be ran in parallel, possibly also “peeling off” connections from a connection pool reserved for the tasks.
In addition to the previously mentioned, in the later client versions, many object-oriented APIs have started appearing. In some cases these are “hidden” under the \texttt{irods::experimental} namespace.

- Filesystem API (\texttt{irods::experimental::filesystem})
  - previously all C-callable \texttt{rc*} functions
  - Kanki interfaced to these via its own intermediate OO interfaces.
- Streaming I/O API (\texttt{irods::experimental::io})
  - previously all C-callable functions, \texttt{rcDataObj[open/close/read/write/Lseek]}
  - Kanki had its own streaming API wrapping those.
- Query Builder API (\texttt{irods::experimental::query_builder})
  - previously the C-native iRODS GenQuery interface
  - Kanki had its own C++ query builder interface wrapping all that
- …
Programming Paradigm Changes

Scope-Based Resource Management

• Also known as RAII - Resource Acquisition Is Initialisation.
• Makes it easier to *not* leak memory.
• In old-school iRODS client code, data structures were simply regions of contiguous memory, possibly dynamically allocated.
• Pointers to these structures were managed in various ways and places, which leads to confusion and an easy way to end up leaking memory.
• C++ (especially from C++11 and later standards) provide mechanisms for managing the allocation and freeing resources (such as memory) simply by objects being either in or out of scope.
• Examples of this are the use of “smart” pointers and the management of locks, such as a mutex.
• All this makes the code inherently exception safe, since the unwinding of the stack is what enables all of this in the first place. *Exceptions are leveraged heavily in iRODS!*
Functional-Style Programming (Lambda Functions!)

- C++ is a multi-paradigm programming language, also supporting some constructs which enable some aspects of functional programming.
- So called lambda functions (anonymous functions) are perhaps the best example of these.
- Lambda expressions enable functions to be defined as literals and passed along as (anonymous) objects.
- How about a defining a lambda expression which would peel off a connection and performing some work on it, and then shipping this off to a thread pool for (asynchronous) execution?
Example 1

Kanki::RodsSession::makeColl()

Before

```cpp
int makeColl(const std::string &collPath,
             bool makeRecursive)
{
    collInp_t theColl;
    int status = 0;
    // preparations omitted

    commMutex.lock();
    status = rcCollCreate(rodsCommPtr, &theColl);
    commMutex.unlock();

    return status;
}
```

After

```cpp
int makeColl(const std::string &collPath,
             bool makeRecursive)
{
    int status = 0;
    // preparations omitted

    std::lock_guard mutexGuard(commMutex);
    namespace fs = irods::experimental::filesystem;
    try {
        fs::client::create_collection(*(commPtr()),
                                      collPath);
    }
    catch (fs::filesystem_error &e) {
        status = e.code().value();
    }
    return status;
}
```
Example 2

RodsDownloadThread::run()

Before

```cpp
void RodsDownloadThread::run() {
    int status = 0;
    // ...
    if (!(status = makeCollObjsList(this->objEntry, &collObjs))) {
        // ...
        for (int i = 0; i < collObjs.size(); i++) {
            Kanki::RodsObjEntryPtr curObj = collObjs.at(i);
            if (curObj->objType == DATA_OBJ_T) {
                // ...
                if ((status = downloadFile(curObj, dstPath, verify, overwrite)) < 0) {
                    reportError("iRODS get file error", curObj->getObjectFullPath().c_str(), status);
                }
                // ...
            }
            // ...
        }
        // ...
    }
    // ...
}
```
void RodsDownloadThread::run() {
    int status = 0;
    irods::thread_pool tank(RodsDownloadThread::defaultJobs);

    // …
    if ((status = makeCollObjList(this->objEntry, &collObjs, comm)) < 0)
        reportError("Download failed", this->objEntry->getObjectFullPath().c_str(), status);
    else {
        // …
        for (int i = 0; i < collObjs.size(); i++) {
            Kanki::RodsObjEntryPtr curObj = collObjs.at(i);
            if (curObj->objType == DATA_OBJ_T) {
                irods::thread_pool::post(tank, [=, &status] {
                    Kanki::RodsSession::connection_proxy conn = session->getConnection();
                    if (!conn)
                        reportError("iRODS connection failure", curObj->getObjectFullPath().c_str(), SYS_SOCK_CONNECT_ERR);
                    else if ((status = getObject(conn, curObj, dstPath, verify, overwrite)) < 0)
                        reportError("iRODS data stream error", curObj->getObjectFullPath().c_str(), status);
                });
            } // …
            } // …
        } // …
    } // …
} // …

After RodsDownloadThread::run()
Example 3

RodsQueueModel

Before

```cpp
RodsQueueModel::RodsQueueModel(/* omitted */) {
    // …
    connect(timer, &QTimer::timeout, this, &RodsQueueModel::refreshQueue);
    timer->start();
}

void RodsQueueModel::refreshQueue() {
    int status = 0;
    Kanki::RodsGenQuery query(this->conn);
    query.addQueryAttribute(COL_RULE_EXEC_ID);
    // …
    query.addQueryCondition(COL_RULE_EXEC_USER_NAME, Kanki::RodsGenQuery::isEqual, conn->rodsUser());
    if ((status = query.execute()) < 0) {
        // report errors to UI
    } else {
        // make data live
    }
```
Example 3

RodsQueueModel

After

RodsQueueModel::RodsQueueModel(/* omitted */) {
    // ...
    connect(timer, &QTimer::timeout, this, &RodsQueueModel::launchRefresh);
    timer->start();
}

void RodsQueueModel::launchRefresh() {
    irods::thread_pool::post(tank, [&] { this->refreshQueue(); });
}
Example 3

RodsQueueModel

After

```cpp
void RodsQueueModel::refreshQueue() {
  int status = 0;
  std::string queryStr = "SELECT RULE_EXEC_ID, RULE_EXEC_NAME, ..."; // omitted
  irods::connection_pool::connection_proxy conn = this->session->getConnection();
  // ...
  if (conn) {
    rcComm_t *comm = static_cast<rcComm_t*>(conn);
    try {
      for (const row_type &row : irods::experimental::query_builder().build(*comm, queryStr))
        tempData.push_back(row);
    } catch (const irods::exception &e) {
      status = e.code();
    } catch (const std::exception &e) {
      status = SYS_INTERNAL_ERR;
    }
    if (status < 0) {
      // report errors to UI
    } else {
      // make data live
    }
  }
}
```
• Let's see if it works!

• Built from the latest commit in GitHub…
Thanks go to

• All of you! For all of these years!

• Special acknowledgements lately:

  • Alan King, iRODS Consortium
  • Kory Draughn, iRODS Consortium
  • Terrell Russell, iRODS Consortium
  • Jason Coposky, iRODS Consortium

  • Tony Edgin, University of Arizona
  • Nirav Merchant, University of Arizona
  • Jeremy Frady, University of Arizona