iRODS interfaces: API and Clients based on Jargon-core for Developers, Grid Users, and Administrators

Mike Conway

Data Intensive Cyber Environments Center (DICE Center), University of North Carolina at Chapel Hill

Abstract

The development of a revised Java API for iRODS is well underway. Many interfaces and integration use cases based on Jargon-core have been demonstrated, and several applications based on this revised API are in deployment. This paper describes the new Jargon-core API, as well as a sample of interesting clients and capabilities that promise to make iRODS more accessible to developers, grid users, and grid administrators.

Index Keyword Terms— Jargon, Java, iRODS, Web Services, REST, iDROP

1. Introduction

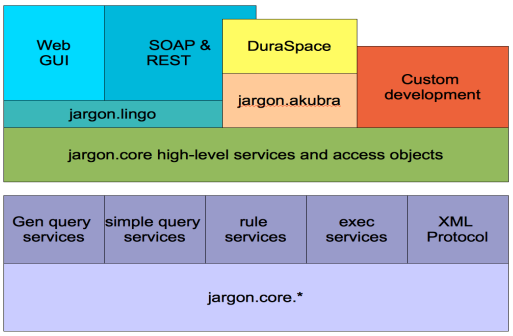
The Jargon API, originally developed by Lucas Gilbert, is a thin-client implementation of the iRODS [2] XML client protocol [3]. This API addressed many difficult problems in the implementation of the iRODS XML protocol. An assessment of the state of Jargon, as well as a vision of future directions, was presented at the 2010 iRODS User Meeting [4].

Since the writing of the 2010 assessment, work has proceeded on 'hardening' the existing API (which will be referred to as 'Jargon-trunk'), as well as development of a revised API (which will be referred to as 'Jargon-core'). The two API have evolved together, each effort informing the other. Jargon-trunk is gradually being re-factored with an eye towards convergence with the Jargon-core API. Enhancement requests and bug reports from users of Jargon are being incorporated into the design of Jargon-core.

The 2010 assessment of the Jargon API observed the following [4]:

- Jargon did a good job of navigating the XML protocol, and had encoded a good deal of real-world experience with iRODS protocol operations.
- Jargon was very difficult to use, both in design, and in the amount of low-level interaction required.
- Jargon had not kept up with evolving best practices and frameworks.

A conception of Jargon as part of a larger stack was discussed in the 2010 assessment. A figure was included that depicted a stack concept for Jargon-core. This stack concept has been an important driver for the development of the new API. At the writing of this paper, each element in the stack has been at least demonstrated with a running proof-of-concept. This paper will talk about developments in the Jargon-core stack, and highlight some of the possibilities going forward.



2. Jargon-core API Details

2.1. Jargon-core development infrastructure

Ease of development includes the idea that the Jargon-core code is easy to obtain and build. The RENCI/DICE collaboration [1] is intended to establish a robust, sustainable software environment for iRODS. As a result of this effort, Jargon-core has moved from the prior Subversion/Bugzilla environment to a comprehensive environment that includes:

- GForge project management and issue tracking software [5].
- A transition from ANT builds to Maven, including distribution of Jargon-core artifacts via a Nexus Maven repository [6].
- Continuous Integration support via Hudson [7].
- Distributed version control via git [8].
- An evolving test environment that will include a rich grid topology for functional testing.

The enhanced environment means that an interface like iDrop can be checked out of git, and with a simple 'mvn install' command, iDrop can be built, automatically obtaining necessary code in the Jargon-core software stack.

2.2. Jargon-core API highlights

In response to the findings of the initial 2010 assessment, several goals were articulated [4]:

- A higher-level API was needed that hid details of iRODS interaction, and provided easy methods to accomplish typical tasks.
- An API that enabled familiar development practices was needed. Examples given included “inversion of control” patterns, “POJO” domain objects, and test-driven development.
- Out-of-the-box administrative and archivist's interfaces were needed.
- A better toolset for integration was needed, including REST and SOAP capabilities.

The Jargon-core API has evolved, based on the findings of the 2010 analysis. At the writing of this paper, in preparation for the iRODS 2011 User Meeting, a beta release of Jargon-core is planned. At this time, there are around 650 unit and functional tests being run against the codebase, as well as several hundred additional tests in higher level libraries and interfaces. The work that remains before a full release has more to do with completeness of feature coverage and performance optimization than with the stability of the existing API.

A few examples can illustrate the utility of the new API style. Consider the common use case of executing a GenQuery and returning the results. The following is the typical code using the prior Jargon-trunk API style:

```
MetaDataCondition[] condition = new
MetaDataCondition[1];
                condition[0] =
MetaDataSet.newCondition(metalAttrib,
        MetaDataCondition.EQUAL, metalValue);

String[] fileds = { StandardMetaData.FILE_NAME,
        StandardMetaData.DIRECTORY_NAME };
MetaDataSelect[] select =
MetaDataSet.newSelection(fileds);

MetaDataRecordList[] fileList =
irodsFileSystem.commands.query(condition, select, 100,
Namespace.FILE, false);
```

In the Jargon-trunk API, queries are built by assembling arrays of various types, then calling two-deep into the IRODSCommands object that also contains low-level connection methods. The returned object is an Array of MetaDataRecordList objects that mix results with an embedded reference to the iRODS connection.

In contrast, the Jargon-core API allows a more natural query by GenQuery string, returning a ResultSet object. The ResultSet has no connection to iRODS. The query itself is executed by an 'Access Object' that specializes in servicing iRODS Queries. The following example illustrates this approach:

```
IRODSGenQuery irodsQuery =
IRODSGenQuery.instance(queryString, 1000);

IRODSGenQueryExecutor irodsGenQueryExecutor =
accessObjectFactory.getIRODSGenQueryExecutor(irodsAcco
unt);

IRODSQueryResultSet resultSet =
irodsGenQueryExecutor.executeIRODSQuery(irodsQuery,
0);
```

The ResultSet object contains information about the query itself, including the original query and the internal translation. The ResultSet also has information on whether more rows are available, and what position in the results the current set of results holds. Notably, the query execution process can internally manage situations where the iRODS Agent needs to close the result set on the server side. This is especially common in situations where Jargon-core is

used in session-per-request situations like a Servlet container. This is an example where not only is the internal structure of the protocol hidden, but also details about the multiple steps required to accomplish an operation. Another simple example of this encapsulation of protocol details is illustrated by the following code that creates a new iRODS file:

```
public boolean createNewFile() throws IOException {
  try {
        fileDescriptor =irodsFileSystemAO.createFile(
                ...);

        ...
        // in irods the file must be closed, then
opened when doing a createnew
        this.close();
        this.open();
        ...
        return true;
}
```

Here a file that is newly created must be then closed and re-opened before it can be used from an input stream. This sort of protocol detail is representative of the types of frustrations that can occur when negotiating the iRODS protocol at a low level, and how sensible defaulting in a higher level API can help developers become more productive with Jargon.

As proposed in the 2010 Jargon review, the Jargon-core API has moved to a POJO domain model, and a variant of the DAO (Data Access Object) pattern. These patterns are now central to the organization of Jargon-core, and allow for enhanced testability, better organization, and easier expandability. Since the access objects that comprise the primary API to iRODS are implementations of defined Java interfaces, they may be easily mocked for testing when developing interfaces and higher-level libraries, as in this example:

```
CollectionAO collectionAO = mock(CollectionAO.class);

Mockito.when(collectionAO.findMetadataValuesByMetadata
Query(elements)).thenReturn(metaDataAndDomainData);
        Mockito.when(irodsAccessObjectFactory.getColl
ectionAO(irodsAccount)).thenReturn(collectionAO);
```

There is now a central factory for the various access objects, and these objects automatically share a connection per thread. This allows easy composition of services from multiple access objects. As new capabilities are implemented, these can be integrated into new access objects. This will allow better organization, ease of use, and more natural composition of services.

It is important to highlight the structural changes within Jargon-core in terms of connection handling. There have been a significant amount of reports from users about errors in Jargon and in the iRODS logs caused by abnormal termination of connections. The Jargon-trunk can spawn and clone connections at times without knowledge of the caller. In addition, Jargon-trunk will often rely on finalizers to close connections, versus direct action from the caller. This sometimes unpredictable generation of connections is revised in Jargon-core through a much more explicit model that manages connections in a session through a centralized cache, and that allows explicit closing of a particular connection, or all cached connections. The connection cache is based on a ThreadLocal, which also prevents the unpredictable results that can occur when a connection to iRODS is inadvertently shared between threads. The new IRODSSDSession holds a ThreadLocal connection cache, and a new IRODSProtocolManager interface defines an object that is asked for a connection, and a place to which connections are returned. This can be implemented in various ways, including as a connection pool or cache. The IRODSSession is also a central shared location where expensive data relating to the iRODS server or session can be kept. This includes properties controlling Jargon itself, and can also include server-side metadata, such as extensible metadata definitions. This contextual data about the server and session will be further defined in later releases, and can include default behavior, as well as overrides that can be injected at creation time.

3.0. Client views

Jargon-core developments provide a foundation for a set of API and graphical interfaces that have been piloted, or are in development. These interfaces fall into two primary categories. First are 'data cloud' views. These are interfaces that deal with the storage and retrieval of data and metadata within iRODS. The goal is to create a set of interfaces and services that allow iRODS to be treated as a personal or organizational data cloud. Second are graphical interfaces and services for administration of an iRODS grid. Third are interfaces oriented towards archivists and curators. These are discussed in more detail below.

3.0.1. Personal and 'cloud views' of iRODS

In presenting a 'personal view' of the iRODS data grid, the capabilities provided by a client GUI application provide a baseline. Thus the iDrop GUI can act as a direct client of iRODS through the XML protocol, using the Jargon-core libraries. iDrop is targeted towards several use cases, and can be seen as:

1. A drag-and-drop desktop explorer model, where files can be easily moved, copied,

replicated, and inspected. This includes desktop drag-and-drop and copy-and-paste capabilities.

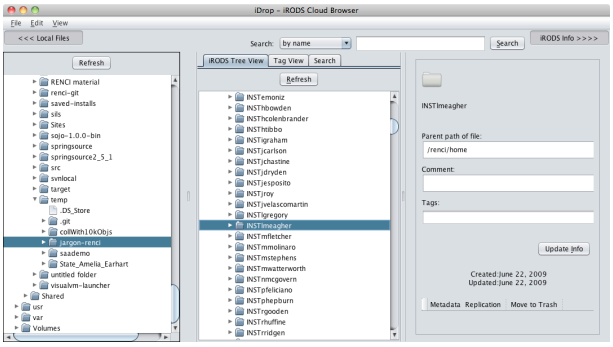
2. A transfer manager that can manage long running transfers in a reliable manner.
3. A multi-device synchronization service that can link local and iRODS folders across heterogeneous client devices.
4. An ingest tool, with the ability to gather metadata and to audit transfers.

Since a primary requirement is cross-platform deployment, iDrop is a Swing GUI that runs as a system tray application. The iDrop application utilizes a 'transfer engine' library built on Jargon-core. This transfer engine manages a queue of transfer operations across multiple grids, and can track the disposition of each file in a transfer. This includes file-by-file success/error information, with the ability to pause, cancel, and restart transfers. Since the transfers are asynchronous, the iDrop use can use drag-and-drop gestures to signal the desired operations, then iDrop can be closed. The client will run in the background and notify the user of transfer status as required.

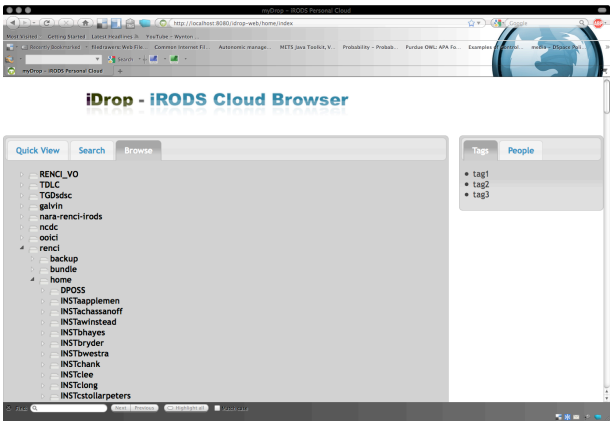
As work progresses, iDrop will include the ability to autonomously retry transfers when a network or agent error causes a disconnection, o~~r~~ ~~O~~r when a device is shut down mid-transfer. The goal is high-reliability through fault tolerance. Enhanced audit/control balancing features are planned for the near future, so that transfer integrity can be assured.

Since the iDrop queue is asynchronous, and since iDrop already has a model of periodic queue evaluation, it becomes possible to create a multi-platform synchronization service. In this mode, iDrop will initially display a wizard where users can select a per-device local synchronization directory. This directory will be periodically diff'd with iRODS, and synchronization jobs can be added to the transfer queue as needed.

An important goal of iDrop is to make transfers reliable, especially for users uncomfortable with Unix icommands. Fault tolerance is one strategy. A second part of the strategy is reporting of success and failure of transfers. iDrop already maintains a local database of transfer activity, and already receives file-by-file success and failure status reports. The iDrop database is being enhanced such that records are kept on restart attempts, and on matching of files transferred on each attempt versus the available source files. The result of a transfer will include a verification of each file transferred, and can possibly include checksum validation, or even a post transfer comparison of the source and target collections. In addition to the reporting in iDrop, it would be possible to add a manifest or other reporting to the target collection at the conclusion of a transfer.



As a complement to the iDrop Swing GUI, development has begun on a rich web interface. This interface is suitable for ad-hoc transfers of small amounts of data, or more casual use of iRODS. iDrop Web is currently a prototype under development. The web application is being developed using the Groovy/Grails framework [9].



iDrop web is meant to be a very clean, simple interface, with search and user tagging built in. This is especially appropriate for treating iRODS as a personal data cloud, where retrieval by search and tag are familiar methods. Sharing and 'social' aspects of data are also being considered in the prototype. A Java WebStart link located on the iDrop web interface will allow easy switching to the more capable GUI when needed.

The development of iDrop web will also create a REST-ful 'cloud API" for iRODS. Controllers and methods can be added to switch the rendering mode from HTML to JSON or XML for various functions. An example from the iDrop web prototype illustrates how Groovy/Grails controller code can be used to represent Jargon-core domain objects as JSON for such a REST-ful API:

```
def ajaxDirectoryListingUnderParent = {
  def parent = params['dir']

  def collectionAndDataObjectListAndSearchAO =
irodsAccessObjectFactory.getCollectionAndDataObjectListAndSearc
hAO(irodsAccount)

def collectionAndDataObjectList =
collectionAndDataObjectListAndSearchAO.listDataObjectsAndCollec
tionsUnderPath(parent)

def jsonBuff = []

collectionAndDataObjectList.each {


          ...

          def attrBuf = ["id":it.formattedAbsolutePath,
"rel":type]

          jsonBuff.add(["data":
it.nodeLabelDisplayValue,"attr":attrBuf,"state":state,"icon":ic
on, "type":type])
  }

render jsonBuff as JSON

}
```

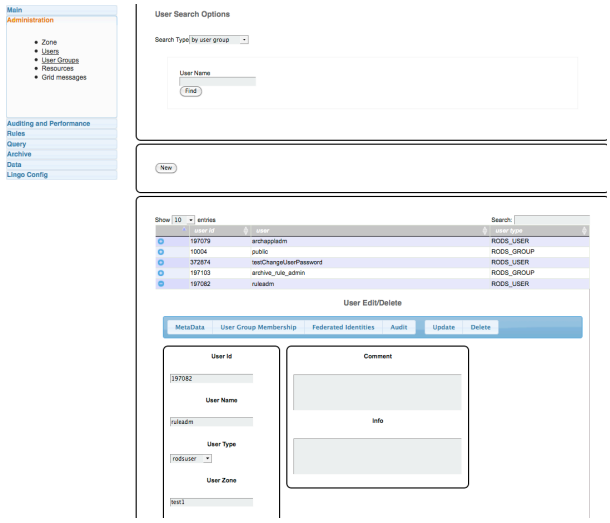
The availability of a simple REST-ful API for personal cloud data creates exciting new possibilities for light-weight uses of iRODS from all manner of platforms, languages, and devices. Certainly, HTTP data transfers are limited in size and speed, but uploads of photographs and other data from mobile devices provides an example of a use case where a REST-ful API can create new types of applications.

3.0.2. Lingo – administrative and web services for iRODS

While iDrop is targeted towards client views of the data and metadata stored on the grid, the "Lingo" projects are oriented towards grid administration and a more comprehensive service model that exposes a richer set of functionality.

At the heart of the project is a comprehensive, rich web interface that provides an integrated view of grid status, as well as access to the functions necessary to administer an iRODS grid. The Lingo web interface was developed into a running prototype, but was put aside for a time due to other priorities. As Jargon-core matures, it is now ready to take beyond the initial prototype stage to implementation.

It is important to note that iDrop, and the Lingo projects, serve two purposes. First, these projects will create out-of-the-box, user friendly interfaces that can serve a wide range of use cases. Importantly, these projects will be under the umbrella of ongoing support and development as Jargon-core and related services mature. The second purpose is as a driver of Jargon development itself. As new capabilities are required for both the data view and administrative/service view, they are pushed into the Jargon-core stack. This means that the API develops with comprehensive support for major use cases, and undergoes a large amount of functional testing in multiple situations on a regular basis. This will contribute to the stability, usability, and performance of the Jargon libraries.



The ability to expose administrative and other grid capabilities, including the transfer of data, as SOAP web services is an important item in the concept of a Jargon stack. The underlying components that would make up an iRODS service model are under development for the iDrop and Lingo interfaces. The remaining work will be to select a platform, identify the course-grained services that should be exposed, and to expose those services using the desired framework. That is a considerable undertaking, so the important activity in the shorter term is to prototype candidate services to ensure that the Jargon stack is evolving in a manner that will easily support exposing as web services. This has been done in initial testing using the Glassfish platform and the Metro framework.

For example, user administration functions were deployed as JAX-WS services on Glassfish, using very light-weight wrapping classes around jargon-core services. This is an example of the marshaling between

the Jargon-core domain objects and XML via SOAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:listUsersResponse
xmlns:ns2="http://websvc.lingo.jargon.irods.org/">


...

            <return>
                <count>0</count>
                <lastResult>false</lastResult>
                <comment/>
                <createTime/>
                <id>9001</id>
                <info/>
                <modifyTime/>
                <name>rodsadmin</name>
                <userDN/>
                <userType>RODS_GROUP</userType>
                <zone>test1</zone>
            </return>
            <return>
                <count>0</count>
                <lastResult>false</lastResult>
                <comment/>
                <createTime/>
                <id>10007</id>
                <info/>
                <modifyTime/>
                <name>rods</name>
                <userDN/>
                <userType>RODS_ADMIN</userType>
                <zone>test1</zone>
            </return>
            <return>
                <count>0</count>
                <lastResult>false</lastResult>
                <comment/>
                <createTime/>
                <id>1154810</id>
                <info/>
                <modifyTime/>
                <name>addUserUpdatedZone</name>
                <userDN/>
                <userType>RODS_USER</userType>
                <zone>test1</zone>
            </return>
           sResponse>
    </S:Body>
</S:Envelope>
```

3.0.3. Arch – archivist's interfaces to iRODS

It is beyond the scope of this paper to go into details on Arch and recent projects regarding policy-based preservation environments. However, it is worth mentioning that a class of interfaces has been identified that is oriented towards archivists who are using iRODS for policy-based preservation. Some aspects of this class of interfaces were demonstrated at the end of last year at NARA, and at the SAA annual meeting in Washington, DC. These demonstrations are discussed in a paper that has been submitted to the proceedings for the 2010 SAA Annual Conference [10].

4. Summary and looking forward

Much of the work on Java and interfaces has been on fundamentals. A solid, sustainable API design that is simple to use, easy to test, and easy to integrate– is the enabler for a large number of useful interfaces and services. There will be a shift in emphasis in the coming year from fundamental API development to such interface and integration activities. With the establishment of a richer test bed as a part of the RENCI/DICE collaboration, there will also be a greater emphasis on performance measurement and optimization as well as federated grid operations.

As illustrated by the use of Groovy/Grails, it is quite possible for PHP, Python, Ruby, Scala, and Groovy developers to leverage the Jargon-core API in their favorite language. This requires running the script on the JVM, but this can be a very high-performance runtime for the script language of choice. This will not necessarily appeal to die-hard Python developers, for example, but is quite viable for the common situation where a developer tasked with a web interface knows PHP, and is more productive using a familiar language with Jargon libraries.

The ability to use Jargon-core with dynamic scripting languages on the JVM also provides an attractive approach for ad-hoc reporting, utilities, and custom scripts for conversion.

Briefly, here are some other important future topics beyond the described interface and service development:

- An assessment of the low-level networking in Jargon, including the use of NIO. Measurement based optimization will be an important activity using the new test bed.
- Fedora repository integration via low level storage as well as Akubra.
- An assessment of the new JDK 1.7 NIO file system, with a potential implementation.

The primary goal for Jargon remains the same. Jargon and the Jargon stack are meant to make iRODS accessible to users, administrators, and software developers through friendly and reliable API, and through easy-to-use interfaces oriented towards grid users and grid administrators.

7. References

[1] *RENCI partners with DICE at UNC and UCSD to sustain iRODS software* available at: http://www.renci.org/news/releases/renci-partners-with-dice

[2]iRODS Introduction available at: https://www.irods.org/index.php/iRODS_Introduction

[3] *Distributed Shared Collection Communication Protocol*, Michael Wan, Reagan Moore, Arcot Rajasekar available at: https://www.irods.org/index.php/iRODS_Protocol_Overview

[4] *Enhancing IRODS Integration: Jargon and an Evolving IRODS Service Model,* Mike Conway, available at: http://irods.org/pubs/Meeting_1003/irods_meeting_1003_evolution_mconway2.pdf

[5] available at: http://gforge.org/gf/

[6] available at: http://maven.apache.org/

[7] avalable at: http://hudson-ci.org/

[8] available at: http://git-scm.com/

[9] available at: http://www.grails.org/

[10] *Policy-based Preservation Environments: Policy Composition and Enforcement in iRODS*, Mike C. Conway, Jewel H. Ward, Antoine De Torcy, Hao Xu, Arcot Rajasekar, and Reagan W. Moore