Enhancing iRODS Integration: Jargon and an Evolving iRODS Service Model

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

Abstract

Jargon is a pure-Java API that encapsulates an XML protocol defined by the iRODS Data Grid. Jargon allows integration with iRODS [1], and is evolving to provide new integration possibilities. This paper describes planned enhancements to the Jargon API developed by Lucas Gilbert.

***Index Keyword Terms*— Jargon, Java**

1. Introduction

iRODS is described by its creators as a type of "adaptive middleware that provides a flexible, extensible, and customizable data management architecture [2]." The iRODS system facilitates the creation of a distributed data grid across heterogeneous storage platforms. iRODS manages communication, metadata, security, auditing, federation, and other vital aspects of a distributed data grid with a unique policy-based approach. The iRODS system expresses data management policies as rules, which are high-level work-flows. These rules are composed of micro-services, which are small modules that perform data grid operations of various types [3].

Jargon, originally developed by Lucas Gilbert, is a pure Java API that allows thin-client connectivity to the iRODS Data Grid. Jargon handles low-level communication with iRODS using a native XML protocol. This protocol describes the sending of commands and data from a network client, as well as the receiving of status and data from the iRODS system [4]. Currently, Jargon is used to integrate a diverse set of custom applications and frameworks with iRODS.

As the number of grid-enabled applications grows, and as distributed systems evolve, so should the Jargon API. Web services using SOAP and REST are now common [5]. Messaging middleware, workflow tools, custom Java applications written on top of the Jargon API, and custom applications written using dynamic scripting languages are anticipated patterns of Jargon usage. By adhering to open standards and development practices, Jargon will become a useful tool, extending iRODS functionality to a wide array of audiences.

2. Recent Jargon Developments

Jargon is receiving new attention as community demand has grown. Jargon is actively used, therefore, efforts to update Jargon are proceeding carefully. Recent efforts include updating the code base to current standards, introducing unit testing, a large number of bug fixes, and refactoring activities.

3. Assessing Jargon

The most recent Jargon development has been done from the perspective of a developer who had an intermediate knowledge of iRODS, and no prior experience with the Jargon Java API. The experience provided valuable insights that have influenced Jargon development plans. These insights, and the resulting design choices, are the subject of this paper.

First, it must be said that the current Jargon does a very good job of navigating the iRODS XML Protocol. There are a myriad number of details that must be handled, and many of the difficult problems with low-level iRODS communication were solved by Lucas Gilbert in the initial versions of Jargon. The utility of the existing Jargon code is an asset that will enable the future evolution.

A primary issue is that Jargon is difficult to use without in-depth prior knowledge. Much of this is due to the complexity of the problems that iRODS addresses. Even so, Jargon exposes too many of the low-level details of iRODS in the public API.

Over time, Jargon has lost track of current best practices. Examples include the 'hand-rolled' nature of logging in Jargon, the lack of unit testing and measured code coverage, and the lack of a build and dependency management system such as Maven [7]. Many Jargon functions are now better supported in mature open-source libraries. One example is the Jargon support for HTTP file systems, which is significantly less capable than the Apache HTTP Client library [8].

Jargon has evolved to a point where refactoring is necessary. Small steps have already been taken, and will increase as releases proceed. This refactoring and enhancement will produce a set of libraries and capabilities to achieve Jargon's goals.

4. Jargon goals

4.1. Higher Level API

A primary goal in designing a follow-on version of Jargon will be to more effectively hide low-level details from API users. Only a few packages for domain objects and services should be presented to users as the public API, and there should only be one route to accomplish a task. This means that any reference to the iRODS XML protocol, or any semantics about connections or thread-safety should be hidden. The ideal would be a service level API, and interaction using familiar POJO's to represent domain data and actions. The strategy should be to leverage the existing Jargon code as much as possible, as there is a significant accumulation of real-world experience reflected in the code.

4.2. Enabling Familiar Development Practices

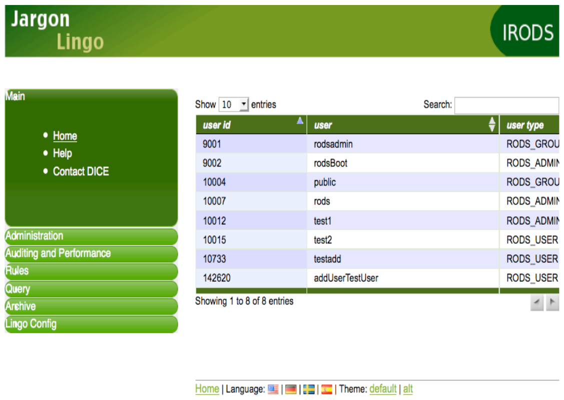
One important 'target audience' for Jargon will be a developer in another domain who is not intimately familiar with iRODS. This will likely be a developer who is used to developing web-facing or web service applications using existing best practices.

These practices should be reflected in the code, including:

- An “inversion of control” [8] pattern and development using the de-facto standard Spring container [9].
- The use of “POJO's” (Plain-Old-Java-Objects) [10].
- Facilities to enable test-driven development.
- Use of common build management practices, familiar libraries for logging, and other common practices.

4.3. Providing an Out-of-the-box Administrative and Archivist' Interface

iRODS has a large suite of tools, and a well-defined low-level interface. Like the Unix shell, icommands provide a knowledgeable user with a quick path to desired functionality [11], but can present some difficulty to occasional users. As the user base grows in size and diversity, it cannot be assumed that all users of iRODS will want to work with their data grid in this manner. It has become a common expectation that there will be web-based tools to interact with middleware platforms, including iRODS. A new, out-of-the-box administrative and archivist's interface is being developed on top of Jargon. The working name of this facility is “Jargon-Lingo”. At the time of this writing, a full-stack working prototype has been developed.



user id	user	user type
9001	rodsadmin	RODS_GROU
9002	rodsBoot	RODS_ADMI
10004	public	RODS_GROU
10007	rods	RODS_ADMI
10012	test1	RODS_ADMI
10015	test2	RODS_USER
10733	testadd	RODS_USER
142620	addUserTestUser	RODS_USER

Showing 1 to 8 of 8 entries

Figure 1 - Jargon web administrative interface

4.4 Enabling iRODS Integration

iRODS itself has many facilities for integration, including a driver architecture that allows many different storage types, and the ability to integrate databases and data streams into the grid. Jargon will provide an even richer integration environment at multiple levels:

- Java API level integration utilizing Jargon core libraries directly in custom applications. An example is the PoDRI project at UNC, which is integrating iRODS with DuraSpace using the Akubra API [12].
- Integration with dynamic scripting languages leveraging JVM dynamic language capabilities [13].
- Service integration with REST and SOAP interfaces on top of Jargon. An example is the integration of iRODS functionality with the Islandora project [14], where PHP scripts could act on the iRODS Data Grid using a service API.
- Integration of iRODS services in emerging cloud computing frameworks, such as jclouds [15].

In addition to the proposed Administrative and Archivist's interface, there will be a large number of custom interfaces for specific purposes. An example of this is an ongoing project to integrate the Islandora [11] Drupal module with iRODS, providing a simple, clean interface for many audiences.

5. Proposed Jargon Architecture

The following diagram illustrates the current Jargon design model, and reflects the above stated observations and goals. The remainder of this paper will discuss the properties of the proposed technology stack.

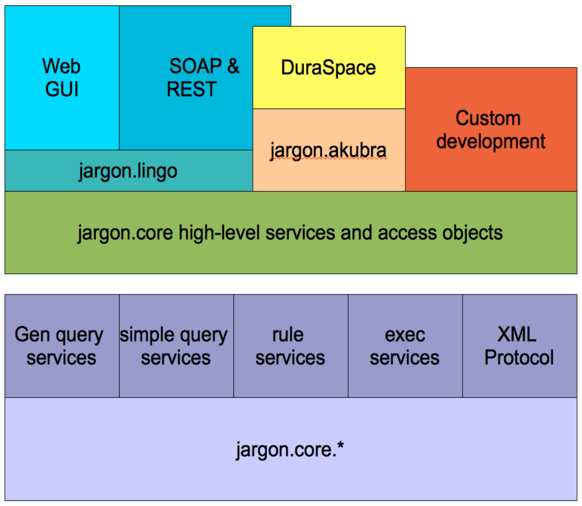


Figure 2 -Proposed Jargon Architecture

Jargon will evolve into a layered architecture, providing a clean separation of concerns, easier extension, and more effective testing through small, mockable units. Jargon will also move forward with the goal of effective test coverage at each level, providing a dependable toolkit as iRODS versions progress.

5.1. jargon.core.*

At the base of the API are the jargon.core libraries. Jargon, as it currently exists, will be transformed over time to become part of the low-level facilities in jargon.core, and this API will be made invisible to public users. Jargon refactoring activities have already begun, and will continue with the jargon.core model in mind.

Networking and low-level protocol handling will be encapsulated at this level, and this should enable easier optimization and tuning while shielding users from API changes. The development of a test suite dedicated to exercising the full iRODS XML at this level protocol will be of great assistance in validating Jargon-based applications as successive iRODS versions are developed.

The primary entry point into the jargon.core functionality will be an iRODSProtocol object that encapsulates the raw network connection to iRODS, as well as the passing to and receiving of messages from the iRODS agent. Also, at this API level, the Jargon prototype includes new facilities for creating and keeping connections such that pooling and caching strategies can be plugged in. No code above the base jargon.core library will access the network connections to iRODS, and will only deal with XML messages.

5.2. jargon.core Mid-level Services

Above the infrastructure that handles connections to iRODS will be a set of mid level services. This intermediate layer will represent the major types of interactions that a client may have with iRODS. The jargon.core mid-level services are not a part of the public API, but do define common capabilities that can be combined by higher level services. Service will include:

- General Query Service – Provides a JDBC like interface to submit SQL-like queries and receive results resembling a JDBC ResultSet. The requests are for pre-defined columns using pre-defined relationships, and mirror the capabilities of the “iquest” icommand.
- Simple Query Service – Executes specific SQL statements permitted by iRODS and receives results resembling a JDBC ResultSet. This is somewhat like General Query, however, it can be used for more complex queries. Simple Query requires permitted SQL to be defined on the iRODS Server. Simple Query services can be used to optimize certain Jargon operations as the need arises.
- Rule Service – Executes rules on iRODS and return results. A philosophy in Jargon development is to use native iRODS functionality, as close to the data as possible, to deliver services to clients.
- Execution Service – Executes arbitrary scripts on an iRODS server from a known location.
- XML Protocol Actions – Executes actions, such as updates, and file operations using specific methods in the iRODS XML Protocol.

5.3. Connection Handling

The current Jargon code base attempts to share a connection between multiple threads, but since those threads access one common socket, the communications occur in a serialized fashion. One side effect of the current connection scheme in Jargon is that the “Command” class is forced to contain all the Jargon functionality in one place, with various levels of synchronization. Testing with the current arrangement, using VisualVM [16], reveals the following pattern for multiple threads sharing a connection in the current Jargon:

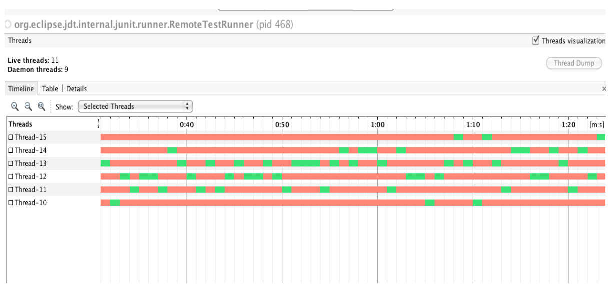


Figure 3 – Multiple threads sharing a connection

As you can see, even though multiple threads are accessing the connection, the actual communication with iRODS is single-threaded. The complications this multi-threaded connection access causes are clear, and the benefits of such sharing is doubtful. The relative efficiency of a connection per thread versus attempting to share a connection between multiple threads is an important area for study and testing, especially with connection pooling capability added to Jargon.

5.4. Access Objects

Jargon development should provide a familiar experience to Java mid-tier developers. One way to achieve that goal will be to utilize familiar design patterns. An added benefit will be that such design patterns have been battle-tested in many application deployments.

A primary design pattern for data enabled applications is the DAO Pattern [17]. As Sun describes this pattern in the J2EE Patterns Catalog:

"Use a Data Access Object (DAO) to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data."

The Jargon prototype uses an adaptation of the DAO pattern that is defined as a Jargon "Access Object". The Access Object framework will:

- Allow creation of Access Objects from a factory.
- Manage connection sharing such that multiple Access Objects in one thread may automatically utilize the same connection.
- Utilize jargon.core mid-tier services to accomplish tasks, and shield API users from details of each Access Object method.
- Use POJO domain objects for parameters and return values.

The concept of an "Access Object" in Jargon is inspired by a very common pattern of development using DAO objects and POJO domain objects with Hibernate [18]. The handling of session in Hibernate DAO's via a ThreadLocal Session object provides an attractive model for a cleaner codebase, treating an iRODS connection in a manner similar to a familiar JDBC connection to a database.

Jargon Access Objects are the lowest level of publicly usable API. Access Objects can be combined into higher level services, both within the Jargon API, and externally, by developers wishing to create new functionality. The following code snippet shows a User access object that utilizes a mid-level General Query service, and returns a User domain object.

```java
public User findById(final String userId) throws
JargonException,DataNotFoundException {

  iRODSGenQueryExecutorImpl iRODSGenQueryExecutorImpl
        = new iRODSGenQueryExecutorImpl(

   this.getiRODSProtocol());
  StringBuilder userQuery = new StringBuilder();

  userQuery.append(buildUserSelects());
  userQuery.append(" where ");

  userQuery.append(RodsGenQueryEnum.COL_USER_ID
       .getName);
  userQuery.append(" = '");
  userQuery.append(userId);
  userQuery.append("'");

                  ...

  iRODSQuery iRODSQuery
  iRODSQuery.instance(userQueryString, 500, 0);
  iRODSQueryResultSet resultSet;
  resultSet = iRODSGenQueryExecutorImpl
         .executeiRODSQuery(iRODSQuery,0);

                  ...

   List<String> row = resultSet.getResults().get(0);
   User user = buildUserFromResultSet(row);

   return user;

}
```

This example Access Object illustrates a clean, higher-level object upon which services may be built. It is important to note that connection handling in this example is transparent, that no low-level protocol operations are visible at this layer, and that the operations of this method are easily tested with mock objects. This example also illustrates how Access Objects like this User Access Object make use of mid-level services, in this case a General Query Service. That General Query Service, in turn, relies on low-level jargon.core packages to turn the query into an XML protocol request, communicate the request to iRODS, and turn the XML protocol response from iRODS into a manageable object that resembles a familiar JDBC ResultSet for processing by the Access Object. Importantly, the caller of this Access Object does not see the underlying ResultSet, rather, the findUserById() method returns a POJO User object.

5.5 A Jargon Service Model

High-level Jargon services can be easily exposed as SOAP and REST using commodity open-source middleware such as Spring Web Services [19], Apache Axis [20], and Metro [21]. As lower level services are developed and tested, consideration will need to be given to the design of a REST/SOAP service model. This service model will allow iRODS to interact with a large number of external systems, and will be developed in the jargon.lingo libraries. The development of a service model is beyond the scope of this document, however, the Spring framework that is powering the web administrative GUI prototype would be a potential provider of REST-ful services, and would likely will not present a high technical hurdle. The Fedora Repository service model provides an excellent model for similar iRODS services [22].

Prototypes under development validate the basic approach outlined in this document, and it can be said with a level of confidence that a Jargon-based service layer providing both SOAP and REST-ful access to iRODS is quite feasible. Beyond the remaining technical hurdles, much consideration needs to be given to the use-cases, security model, and implications of such a facility.

6. Conclusion

This paper outlines some of the high-level design goals, and a proposed architecture for future Jargon development. At the writing of this paper, a working prototype does exist, and is being used for validation and experimentation. While still a work in progress, the prototype does provide valuable guidance for near-term Jargon refactoring. Jargon development will be guided by careful testing, community input, and current best practices.

Jargon, and the integration possibilities that it will enable, has the goal of making the iRODS Data Grid as familiar to developers as a database or messaging middleware platform, and a dependable tool to help manage the expanding need for secure sharing and preservation of data.

7. References

1. Jargon, A Java client API for the DataGrid, https://www.iRODS.org/index.php/Jargon
2. iRODS: integrated Rule Oriented Data System White Paper Data Intensive Cyber Environments Group University of North Carolina at Chapel Hill University of California at San Diego September 2008 Rajasekar, A., M. Wan, R. Moore, W. Schroeder
3. iRODS: integrated Rule-based Data System Rajasekar, A., M. Wan, R. Moore, W. Schroeder
4. *Packing/Unpacking Scheme Used in iRODS* Mike Wan, DICE
5. Restful web services vs." big"'web services: making the right architectural decision http://www2008.org/papers/pdf/p805-pautassoA.pdf [PDF] C Pautasso, O Zimmermann, F Leymann - 2008 – portal.acm.org
6. Apache Maven, http://maven.apache.org/
7. Apache HTTP Client , http://hc.apache.org/httpclient-3.x
8. Inversion of control containers and the dependency injection pattern, http://www.itu.dk/courses/VOP/E2006/8_injection.pdf [PDF], M Fowler - Actualizado el – itu.dk
9. Spring Framework, http://www.springsource.org/
10. Christopher Richardson, "What is POJO Programming?", Java Developer's Journal, http://java.sys-con.com/node/180374
11. iRODS icommands, https://www.iRODS.org/index.php/icommands
12. Akubra Project, http://www.fedora-commons.org/confluence/display/AKUBRA/Akubra+Project
13. New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine, http://java.sun.com/developer/technicalArticles/DynTypeLang/
14. Islandora Project, http://islandora.ca/
15. jclouds framework, http://code.google.com/p/jclouds/
16. VisualVM, http://java.sun.com/javase/6/docs/technotes/guides/visualvm/
17. DAO Pattern, http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
18. Generic Data Access Objects, https://www.hibernate.org/328.html
19. Spring Web Services, http://static.springsource.org/spring-ws/sites/1.5/
20. Apache Axis, http://ws.apache.org/axis/
21. Metro Web Services Framework, https://metro.dev.java.net/
22. Fedora Service Framework, http://fedora-commons.org/confluence/display/FCR30/Service+Framework