Exploring the iRODS Native protocol, a hidden gem

Ton Smeele
Utrecht University
The Netherlands
a.p.m.smeele@uu.nl

ABSTRACT

The iRODS grid deploys a versatile protocol for client-server communications. While its XML serialization variant is used by many client applications, the more efficient Native variant is currently only available to C-language based clients. We discuss a design for an iRODS client library that also supports the Native protocol variant in languages other than C. We validate the usability of our design via an example implementation in Java. This example implementation is utilized at Utrecht University by a Java application that transfers data objects between unfederated iRODS zones.

Keywords

iRODS, data grid, distributed client server model, communication protocol

INTRODUCTION

iRODS deploys a distributed client/server communication model. Clients send requests to an arbitrary server in the grid. The fulfillment of such a request may require the connected server to consult other servers located in its own grid or in a federated grid. Additionally, when the client so desires, data file content is transmitted efficiently via peer-peer connections directly between the client and the resource server that manages the replica.

All grid communications are based on a single protocol, the *Distributed Shared Collection Communication Protocol*, more commonly known as the iRODS protocol [1]. Its power and flexibility is underpinned by decades of usage, without a need for major change. According to Arcot Rajasekar, the iRODS protocol shares its lineage with the SDSC Storage Resource Broker (SRB), an iRODS predecessor created in 1995 [2]. Parts of the SRB protocol design are based on a request/answer protocol that was deployed in Postgres95 [3]. To accommodate the exchange of more complex data structures as used in SRB and iRODS, significant changes have been applied.

Before a message is exchanged, it is serialized using either *Native* or *XML* encoding. Native is the original protocol implementation, where each element's value is packed as a sequence of bytes [1]. This method is supported by the C/C++ iRODS client library. For instance the iCommands communicate with an iRODS server via the native protocol. Unfortunately, this client library is not readily available to applications written in other programming languages. As a workaround, an XML-based serialization method has been added. Here, elements are serialized as tagged strings. Binary data is base64 encoded. Many iRODS clients depend on the XML protocol, for instance Jargon (Java), PRODS (PHP), PRC (Python), and the Go-iRODSClient (Go) [4][5][6][7].

Messages serialized using the Native protocol are smaller in size compared to messages encoded with the XML protocol. The larger message size produced by XML encoding is caused by a need to encode element names as tag in addition to the element values. In contrast, the Native protocol only encodes the element value. Furthermore, it encodes numerical and binary data more tightly. Despite being more efficient, support for the Native variant has remained limited to C/C++.

iRODS UGM 2025 June 17-20, 2025, Durham, NC

What does it take to implement a client library for the Native protocol? Our study aims to answer this question by creating a prototype of a iRODS client library in Java. We select Java, since it supports object oriented programming with strict type checking. These capabilities facilitate a clean implementation and fast debugging.

CLIENT LIBRARY ARCHITECTURE

We design our client library as a layered architecture, shown in Figure 1. The foundation is a layer that maintains a TCP/IP socket connection between the client application and an iRODS server. The middle layer is responsible for sending request messages and receiving response messages efficiently across this connection. Messages are (de)serialized using either the native or the XML protocol. The top layer implements a client application programming interface (API), that creates a message from a client request, and translates a received message back into a response handed over to the client. Each method present in this layer corresponds to an iRODS server supported API request type. Request arguments are encoded as iRODS data structures. Responses are decoded to Java classes suitable for consumption by client applications.

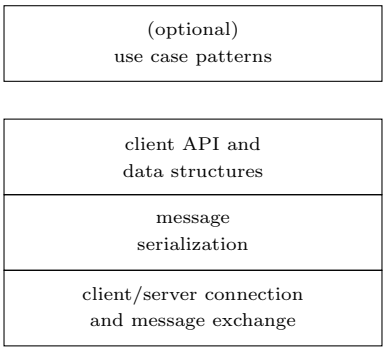

Figure 1. Architecture of MyRODS client library

Some interactions, for instance the transfer of replica data, require the execution of a sequence of multiple iRODS requests. We have developed a set of Java classes as a layer on top of the client library, to support common use case patterns, such as user authentication and parallel data transfer.

Client/server connection and message exchange

For backward compatibility reasons, an iRODS client must always initiate an unencrypted connection. Once connected, the client may request the server to switch to encrypted communications, and vice versa. For instance, this mechanism is used by iRODS to securely communicate a PAM password over an otherwise unencrypted connection. We design encrypted communications as a secure socket layered over the existing socket. The socket switch takes effect after the response has been received and the channels are flushed.

iRODS supports communication between clients and servers with different hardware or software architectures. Regardless of architecture, data is exchanged in big endian format, in compliance with RFC 1700 [8]. Characters are UTF-8 encoded, in compliance with RFC 5198 [9]. To encode data types in big endian, we use the Java class `ByteBuffer`. The method `getBytes` of the class `String` facilitates encoding in UTF-8 character set. When decoding, ByteBuffer assists to meet word alignment requirements of the local architecture.

Virtually all exchanged messages use the layout shown in Figure 2. Each message consists of five components: the header size, header, message, error message, and byte stream. Once the header size, a 32 bit integer, is read, the receiving party knows how to read the header. The header contains data on the sizes of the remaining components, which allows those components to be read in one go. Empty components are allowed, they have a size value of zero.

While the last component is simply a byte array, the header, message, and error message are serialized named iRODS

Figure 2. Exchanged message layout

data structure types. The data types of the header and error message are fixed, they are respectively `MsgHeader_PI` and `RError_PI`. The data type of the message component varies by message type.

The iRODS protocol specifies that the header *must* be serialized using the XML protocol. In its first message, the client specifies the desired serialization protocol, Native or XML, for serialization of the message and error message components of subsequent messages. Hence a client library must always be able to support the XML protocol, while support of the Native protocol is optional.

The XML protocol has changed slightly as of iRODS 4.2.9. To fix a confusing specification in the protocol, iRODS now escapes an apostroph using the code `&apos;` whereas in prior releases this code would represent a backtick. The new encoding is used only when both client and server are version 4.2.9 or later. We use the server version, reported in an initial `RODS_VERSION` type response message, to choose the appropriate XML protocol variant.

Message serialization

In our message serialization layer, we deploy a strategy design pattern to implement all of the (de)serialization variants mentioned in the previous section. Serialization is done using the class `Packer` while deserialization is performed by `Unpacker`.

Our implementation must support the (de)serialization of six iRODS base data types: `bin` holds an arbitrary byte value, `char` depicts a byte used as a character, `str` is a null-terminated variable length character array, `int16` and `int` are respectively 16-bit and 32-bit size integers, and `double` represents a 64-bit size floating point value. In practice, the double type is used as a union to store either a floating point or, more often, a long integer value. We design an abstract class `Data`, with subclasses per iRODS data type. Each instance will store a variable name and its value.

We also need to support the serialization of composed data types. iRODS provides `struct` along with pointer reference and array as mechanisms to compose more complex data types from the iRODS base types. These concepts have semantics similar to their C language pendant. iRODS annotates its data type specifications with dimension information, so that the receiving party can allocate an appropriate amount of memory for each data element, for instance in case of indirect references. This information is important for client library implementations in programming languages such as C, where the programmer is responsible for memory management. As memory management is implicit in Java, our implementation may ignore dimensions for data received. However, it will need to ensure that any data sent does not exceed specified dimensions, for example the maximum size of a variable-length character string must be respected. We use a composite design pattern to represent the composition of data types.

The iRODS protocol requires messages to contain instances of predefined struct types, well-known by client and server. This approach allows messages to have a significantly smaller footprint, since details on the structure of data do not need to be included in each message. Note that this design assumes that client and server use a sufficiently compatible set of struct specifications.

```
GenQueryOut_PI = "int rowCnt; int attriCnt; int continueInx; int totalRowCount;
                  struct SqlResult_PI[MAX_SQL_ATTR];"
SqlResult_PI   = "int attriInx; int reslen; str *value(rowCnt)(reslen);"
```

Figure 3. Packing instructions for struct GenQueryOut_PI and a struct it depends on

All predefined struct types are specified using *packing instruction* strings. The packing instruction is a ordered sequence of element names along with their base type and dimension. Figure 3 shows the packing instruction for

the predefined struct `GenQueryOut_PI`. Instances of this struct will consist of four integers, and an array of elements of the struct type `SqlResult_PI`. This example demonstrates several features of the packing instruction syntax. A specification can be nested by referencing another struct, such as `SqlResult_PI`. Instead of specifying a dimension using the literal value `50`, we may reference a constant such as `MAX_SQL_ATTR`. To specify the dimension of a dynamic array, which is only known at runtime, we can refer to the value of another instantiated variable (already in scope). Note for example, how the actual value of `rowCnt`, received at runtime as part of a `GenQueryOut_PI` data structure, will be used to drive the size of the subsequently received pointer array `value` in the embedded struct `SqlResult_PI`. Our implementation deploys a map `PackInstructionsConstants` to store all constants and builds a `Data` structure containing processed variables. Whenever a message is deserialized, the packing instruction of the expected data structure is parsed by class `ParsedInstruction`, searching the map and processed-data structure to resolve references as needed.

```
MsParam_PI     =  "str *label; piStr *type; ?type *inOutStruct; struct *BinBytesBuf_PI;"
```

Figure 4. Packing instruction for MsParam_PI includes a dependent type

In addition to the above mentioned features, packing instructions support a so-called *dependent type*. Here, the type of an embedded struct is determined dynamically at runtime based on the value of an instantiated variable, similar to the mechanism used for dynamic arrays. For this purpose, iRODS has added a meta data type `piStr` to the set of data types. This name is an acronym of *packing instruction string*. Values of this type are variable-length character arrays, restricted to the set of predefined struct names. Figure 4 demonstrates the use of dependent types. The symbol `?` is part of the packing instruction syntax, and indicates that the immediately following `piStr` variable must be dereferenced, to obtain the name of the target struct type. In the example case, at runtime, the value of variable `type` will specify the actual type of the subsequent embedded variable `inOutStruct`. We use the above-mentioned map and processed-data structure to implement this behavior.

Not all parts of a data structure need to be present in a message. Absent data structure elements are indicated by a nullpointer. The XML protocol variant serializes a nullpointer by simply omitting the tag for the corresponding element. Not packing any structure details, the Native variant of the protocol does not have this luxery. Instead it packs the UTF-8 representation of the string literal `"%@#ANULLSTR$%"` to depict a nullpointer. Note that, due to the iRODS protocol design, a serialized nullpointer cannot be distinguished from a serialized pointer to a string that bears a value equal to the nullpointer literal. In this odd case, we will interpret and unpack the structure as nullpointer.

Client API and data structures

iRODS communications follow a client/server model with typed messages. For instance, the first message sent by a client is a `RODS_CONNECT` type message, optionally followed by a `RODS_CS_NEG_T` message. Once connected, API calls use `RODS_API_REQ` type messages, and the server replies with `RODS_API_REPLY` type messages. When a client wishes to disconnect, it sends a `RODS_DISCONNECT` type message.

In general, the client sends a message, to which the server replies with a response message. We have encountered two exceptions to this rule. Firstly, the client must send two additional messages immediately after issuing a `RODS_CS_NEG_T` type request message, before it may expect a response. Secondly, when the client wants the session to end and sends a `RODS_DISCONNECT` type message, the server will not reply to this message. Instead, the client is expected to disconnect the socket. To support all use cases, including both exceptions, we make the client API layer responsible for message workflow decisions.

We design the client API as a central class `Irods`. Figure 5 demonstrates how Java applications instantiate this class and perform API calls. Each method of the `Irods` class represents an API call and returns the message component of a server reply as its result, transformed to Java classes more digestable for client applications. We customize the return type per method. For instance, the `rcMiscSvrInfo` call will return the received reply message as a `MiscSvrInfo` object. This approach facilitates compile-time type checking of client application calls. The remaining reply components, such as the errorMsg datastructure and the byteStream data, along with the `intInfo` returncode contained in the

```
Irods irods = new Irods(host, port);
RodsVersion version = irods.rcConnect(proxyUser, proxyZone, clientUser, clientZone);
if (irods.error) {
   throw new RuntimeException("Unable to connect, error is " + irods.intInfo);
}
MiscSvrInfo info = irods.rcMiscSvrInfo();
System.out.println("zone is " + info.rodsZone);
irods.rcDisconnect();
```

Figure 5. Example application using the client API

header, have a structure that is common across all API calls. They can be queried by client applications as object attributes.

Use case patterns

We have named our client library implementation "MyRods"[1]. While developing and testing MyRods, we encountered a recurring need for certain sequences of client-server interactions. For convenience reasons, we have captured these patterns in a few macro-level classes.

The class `Hirods` extends the API class `Irods`. It adds methods to unburden the client application of roundtrips involved with native or PAM authentication. In addition, authenticated sessions can be cloned, for instance to support parallel data transfers. These secondary connections are typically managed and reused via an `IrodsPool` class. The `Hirods` method `getAvus` allows applications to retrieve all object, resource or user related metadata, and likewise sets of metadata can be added to an entity using method `addAvus`. Further, the `checkAccess` method queries the server to check if a user has the desired access to a collection or data object, similar to, although more generic than, the iRODS microservice `msiCheckAccess` [10].

The class `DataTransfer` is an abstract class for copying the content of a file to another file. Using a strategy pattern, the source and destination files may be specified as either an iRODS based `Replica` or a regular filesystem file (`LocalFile`). Data transfers are not limited to *put* and *get* type operations between a file and a data object in an iRODS zone. They can also be used to copy data between data objects in the same zone or across zones. A second strategy pattern is deployed to influence the method used for data transfer, with implementations for single threaded transfer and multi-threaded transfer over the zone-port.

Recently, our institute has commenced to migrate and consolidate data located on several local iRODS zones to a single third-party managed iRODS instance. We have developed the application `ipump` [2], based on our MyRods client library and patterns, to support this migration process.

Performance

We benchmark the efficiency of the Native iRODS protocol variant against the XML variant in terms of exchanged message size.

Our test comprises of a variety of common calls: connect to an iRODS 4.3.3 server, perform a native login, request status information on a data object, query data object names and their data size from a collection that contains one hundred data objects, download the content of a data object, and disconnect. We run this test for both protocol variants in turn. In our initial connect message, we specify the iRODS protocol variant to be used for the remainder of the calls. Figure 6 outlines the header and message component size of iRODS messages that have been exchanged

[1] See https://github.com/tsmeele/MyRODS
[2] See https://github.com/tsmeele/ipump

Direction (client)	iRODS Message type Type	Subtype	Header (bytes) Native	Header (bytes) XML	Message (bytes) Native	Message (bytes) XML	Native vs XML
sent	RODS_API_REQ	AUTH_REQUEST	132	132	0	0	100%
received	RODS_API_REPLY		140	141	64	153	69%
sent	RODS_API_REQ	AUTH_RESPONSE	133	134	29	119	64%
received	RODS_API_REPLY		139	139	0	0	100%
sent	RODS_API_REQ	OBJ_STAT	134	134	101	317	52%
received	RODS_API_REPLY		140	141	74	274	52%
sent	RODS_API_REQ	GEN_QUERY	133	134	81	423	38%
received	RODS_API_REPLY		142	142	2078	7787	**28%**
sent	RODS_API_REQ	GEN_QUERY	133	134	81	421	39%
received	RODS_API_REPLY		142	142	1116	3789	32%
send	RODS_API_REQ	GET_RESOURCE_INFO _FOR_OPERATION	136	136	112	328	53%
received	RODS_API_REPLY		140	141	76	150	74%
send	RODS_API_REQ	REPLICA_OPEN	136	136	104	321	53%
received	RODS_API_REPLY		142	142	1711	2345	75%
send	RODS_API_REQ	DATA_OBJ_READ	133	134	36	218	48%
received	RODS_API_REPLY		141	141	0	0	100%
send	RODS_API_REQ	DATA_OBJ_CLOSE	133	134	36	212	49%
received	RODS_API_REPLY		139	139	0	0	100%
sent	RODS_DISCONNECT		133	133	0	0	100%
		Total after connect	2601	2609	5699	16857	43%

Figure 6. Benchmark test results

after the initial connect.

The header component is always XML-serialized, regardless of the chosen protocol variant. This explains why there is almost no difference in size, except for one character in case the message component size, a value included in the header, requires an extra digit.

With regard to message component size, the Native protocol greatly outperforms the XML protocol, The difference is particularly striking in calls where many data values are being exchanged, such as a general query. Interestingly, while calls to transfer replica data do not exhibit any difference in size, the messages of the accompanying calls needed to open or close a replica display a factor two difference between the variants.

Overall, the Native protocol variant is able to pack messages by a factor two to nearly four more efficient compared to the XML variant, depending on the mix of API call types used.

DISCUSSION AND CONCLUSION

Our MyRods implementation of the client API library bears some limitations. The current set of implemented API calls only supports connections from a client application to an iRODS server, it does not support connections between servers. Parallel data transfers are only supported via the zone-port. We have not yet implemented so called *portal-based* transfer, which involves the use of high-ports, as we expect this method to be deprecated in a future iRODS release[3].

Our benchmark test demonstrates that the Native variant of the iRODS protocol packs messages two to four times

[3]see https://github.com/irods/irods/issues/7949

more efficient compared to the XML variant. As the Native variant is only available to C/C++ based applications, unfortunately it remains a hidden gem.

Through our design and implementation of a Java-based iRODS client API library, we have demonstrated that the Native variant can be supported in other languages besides C/C++. The implementation in Java consists entirely of Plain Old Java Objects (POJO) and should therefore be portable across many operating system architectures.

Initial design and prototype development of the library has taken approximately sixty hours of development time. We have built an initial prototype to confirm, and adjust, our interpretation of the iRODS protocol specification. Once the lower two layers were established, we continued for another several weeks to complete the client API layer and support for the use case patterns.

Our approach, to internally use data structures that mimic the iRODS types, allowed for a rapid and consistent implementation of message serialization. Modeling of pointers as a separate data type turned out to be an important insight that simplified the serialization of elements.

Adding support for API calls required extensive inspection of iRODS server source code. The iRODS API is mostly documented inline in the source code. Sometimes information is fragmented as a result of a change in development practices since the introduction of the iRODS plug-in architecture. We recommend that iRODS returns to a more centralized definition of pack instructions and keywords, so that they are easier to locate and reuse. In addition, more complete, findable documentation on semantics of iRODS API arguments would be helpful for client application developers.

We learned that replicas share sufficient properties with regular files, to be modeled as specializations of a common supertype. More research will be needed to investigate the implications for high level client interfaces to iRODS. Can we simplify existing interfaces by using a different paradigm? For instance, should we continue to think of the entire iRODS zone as a filesystem? Or would it instead be appropriate to consider (and expose) each of its resources as separate filesystems?

REFERENCES

[1] M. Wan, R. Moore, and A. Rajesekar, "Distributed Shared Collection Communication Protocol." https://www.irods.org/index.php/iRODS_Protocol_Overview, May 2008.

[2] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC storage resource broker," in *CASCON First Decade High Impact Papers*, pp. 189–200, IBM Press, 2010.

[3] M. Stonebraker and L. A. Rowe, "The design of Postgres," in *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, vol. 15, pp. 340–355, ACM New York, NY, USA, 1986.

[4] M. Conway, "Enhancing iRODS Integration: Jargon and an Evolving iRODS Service Model," in *Proceedings iRODS User Group Meeting 2010*, (Chapel Hill, NC, USA), pp. 62–66, DICE, 2010.

[5] DICE-UNC, "PHP API for iRODS." https://github.com/DICE-UNC/irods-php.

[6] iRODS Consortium, "Python iRODS Client." https://github.com/irods/python-irodsclient.

[7] I. Choi, J. H. Hartman, and E. Skidmore, "Go-iRODSClient, iRODS FUSE Lite, and iRODS CSI Driver: Accessing iRODS in Kubernetes," in *iRODS User Group Meeting 2021 Proceedings*, (Virtual), pp. 63–72, iRODS Consortium, 2021.

[8] J. Reynolds and J. Postel, "Assigned numbers," RFC 1700, RFC Editor, Oct. 1994.

[9] J. Klensin and M. Padlipsky, "Unicode Format for Network Interchange," RFC 5198, RFC editor, Mar. 2008.

[10] A. Rajasekar, T. Russell, J. Coposky, A. de Torcy, H. Xu, M. Wan, R. W. Moore, W. Schroeder, S.-Y. Chen, M. Conway, and J. H. Ward, "iRODS 4.1 Microservices Workbook." https://irods.org/uploads/2015/01/irods4-microservices-book-web.pdf, May 2015.