1. Goals of this Document

This document defines the protocol used by clients and servers running iRODS, which is open-source data management software. This protocol is application-specific and tightly coupled with the APIs provided by particular clients/servers. As such, some familiarity with iRODS, as well as basic technical experience, is assumed. Additionally, the description of the iRODS protocol provided in this document is not sufficient for a skilled engineer to implement an iRODS server or client from scratch. However, this document should allow a skilled engineer to send and interpret messages in the iRODS protocol comfortably enough to reverse engineer such an implementation, e.g., by observing network traffic in the course of a target iRODS workflow.

2. iRODS

iRODS stands for Integrated, Rule-Oriented Data System. It provides a platform for automated, large-scale, policy-based management of data with a focus on data integrity, secure collaboration, workflow automation, metadata-driven data discovery, and virtualization of key services. Although iRODS is a large and complex system, its protocol is minimalistic in the sense that common workloads can be achieved using a handful of basic patterns, which are described [TODO: Where is this described?].

3. The General Model of the iRODS Protocol [TO REVAMP]

iRODS zone topologies are often complex. However, as far as the iRODS protocol is concerned, communication is between two well-defined entities: the client and the server. Connections are initiated by the client. The transport layer protocol underlying an iRODS protocol workflow can either be plain TCP [TODO: How to describe this in more technical terms?] or TLS. After the initial request for connection, the client and server will decide whether a negotiation is required to decide the transport layer protocol. If it is, they will undergo that negotiation, and if it isn't they will default to TCP. If a negotiation is required, client and server inform each other of their POLICY regarding the choice between TCP and TLS. If client and server have incompatible policies (e.g., client requires TCP and server requires TLS), the negotiation will fail; otherwise, once the negotiation has completed, the client will send a request to authenticate as a user in a zone known by [TODO: Known by?] the server. This request is an API_REQ (see:). The specific control flow through which the protocol messages will move during authentication is determined by the specific API number being requested.

4. Encoding Schemes

The basic entities in the iRODS protocol are called packing instruction. A packing instruction is a single message that must be processed by either client or server. Packing instructions can be nested inside of other packing instructions, requiring recursive serialization and deserialization. To each packing instruction corresponds an in-memory C struct. There are two encoding schemes for packing instructions. The first works by removing padding from the byte-level representation of the corresponding C struct for transmission, and reintroducing padding to reconstruct that C struct upon receipt. This is called the native protocol. The second is a loosely XML-like encoding. Some differences between iRODS-flavored XML and standardized XML are that iRODS-flavored XML can never contain a tag specifying any schema, and that iRODS-flavored XML is always order-dependendent (specifically, tags must appear in the same order as their fields are declared in the corresponding C struct).

These two encoding schemes are completely equivalent in terms of the packing instructions they are able to express, however the XML protocol is in wider use among, e.g., independently developed iRODS clients. This is because XML is much easier for humans to read, although native protocol offers performance advantages. [TODO: This seems obvious but can we verify it?] ICommands are a client that ship with iRODS and represent the reference implementation of the iRODS protocol. They support both native- and XML-encoded protocol.

This document focuses on XML protocol to aid exposition. IRODS-dialect XML assumes UTF-8 encoding and does not decode special characters for punctuation and white space. For example, the string “ ” representing a tab would have to be replaced by "" before being transmitted to an iRODS server.

5. iRODS Message Structure

An iRODS message has as few as 2 and as many as 5 distinct parts, which are typically transmitted as separate TCP packets appearing in the order in which they are descrbed below.

- Prelude: Four bytes to be interpreted as a 32-bit integer indicating the length in bytes of the message header.
- Header: A packing instruction containing information about the message body. Here is an example of a header:

```
<MsgHeader_PI>
    <type>RODS_CONNECT</type>
    <msgLen>339</msgLen>
    <errorLen>0</errorLen>
    <bsLen>0</bsLen>
    <intInfo>0</intInfo>
</MsgHeader_PI>
```

Note that PI here stands for "packing instruction." Here, we note only the two most important fields of message headers: `type` and `msgLen`. `MsgLen` is a 32-bit integer which tells the recipient how long in the bytes the body corresponding to this header is. If `msgLen` is 0, then there is no message body. `MsgLen` MUST not be negative.

`Type` can only have one of six values, similar to an enum in programming languages like C or Java. The valid values for `type` are as follows:

```
- RODS_CONNECT: Sent by a client to start connection.
First message in all iRODS-protocol workflows.
- RODS_DISCONNECT: The message type sent by a client to end a connection.
- RODS_API_REQ: A request for an API provided by the server.
API number is indicated by `intInfo` field.
- RODS_API_REPLY: The response to a RODS_API_REQ.
Status codes will be indicated by `intInfo` field. 0 typically indicates success;
other values indicate an error code.
- RODS_VERSION: Following a connection, the first message
sent by the server during a connection handshake.
- RODS_REAUTH: As of version 4.3.0, this message type
is not used in either the server or iCommands.
```

- Body: A packing instruction containing more information about the message. Which packing instructions are valid is determined by information in the header, as well as by particular API implementations in the case of a `RODS_API_REQ` or `RODS_API_REPLY`. For example, a `RODS_CONNECT` must always be followed by a `StartupPack_PI`. Here is an example of a `StartupPack_PI` following a `RODS_CONNECT`:

```
<StartupPack_PI>
    <irodsProt>1</irodsProt>
    <reconnFlag>0</reconnFlag>
    <connectCnt>0</connectCnt>
    <proxyUser>rods</proxyUser>
    <proxyRcatZone>tempZone</proxyRcatZone>
    <clientUser>rods</clientUser>
    <clientRcatZone>tempZone</clientRcatZone>
    <relVersion>rods4.3.0</relVersion>
    <apiVersion>d</apiVersion>
    <option>iinit</option>
</StartupPack_PI>
```

As noted in section (1), this document will not provide an explanation of all packing instructions or their fields. Note, however, that not all fields are currently meaningful. For example, experimentation by the iRODS team suggests that the `option` field has no effect. Nonetheless, all fields must be included. The `option` field may be empty, i.e., it may appear as `<option></option>`. This is the case for `StartupPack_PI`s generated by the Python iRODS client. Additionally, under 4.3.0, `apiVersion` MUST be set to ‘d’.

- Error message stack. The `errorLen` field of the header indicates how long in bytes this component will be. The first four bytes of this component will be a 32-bit integer indicating the number of error messages included in the stack. [TODO: Decide on a good way to explain the structure of error messages, Wan doc exposes the actual C struct]. In most iRODS messages, there is no error stack and `errorLen` is 0 in the header.
- Byte stream. The `bsLen` field of the header indicates how long in bytes this component will be. This component contains arbitrary bytes and is mainly used for data transfer. Most messages have no byte stream and `bsLen` is 0 in the header.

6. Primitive Types

7. The Generic/Associate Array Types

The iRODS protocol features a family of packing instructions which implement random access by key values, like `HashMap<K,V>` types in Java or dictionaries in Python. These types differ in the types assigned to their keys and values. The following table summarizes this information:

Packing Instruction	Key Type	Value Type
InxIvalPair_PI	Integer	Integer
InxValPair_PI	Integer	String
KeyValPair_PI	String	String