Distributed Shared Collection Communication Protocol

Michael Wan, Reagan Moore, Arcot Rajasekar
San Diego Supercomputer Center
mwan@sdsc.edu

The integrated Rule Oriented Data System (iRODS) is a data grid that automates application of management policies. The system is based on a distributed client/server model. The federation of servers is implemented as peer-to-peer servers with a central metadata catalog managing persistent state information and a rule engine that applies rules stored in a rule base. Clients issue requests to the iRODS system to invoke rules controlling the execution of remote operations. For each client request, iRODS sends the client message to the remote server where the data reside, the requested operations are then performed, and the result is sent back to the client.

The iRODS environment supports both server-driven and client-driven workflows. The workflows are controlled by rules that are expressed as event/condition/action-set. Actions include operations that may be performed at the remote servers.

To implement the rule based data grid and the workflow functionalities, the iRODS system supports a rich set of low level operations including the followings:

- I/O operations – these includes POSIX type I/O calls such as creat, open, read, write, lseek, close, etc and whole file type operations such as put, get, cp, replicate, checksum, etc.
- Micro-services that encapsulate multiple I/O operations, or that manipulate entire files, or that manipulate metadata, or that set administrative metadata. Currently 73 micro-services have been implemented. These are listed in Appendix A.
- Recovery procedures for micro-services that manage failure conditions when a desired set of operations do not complete. Currently 35 recovery micro-services have been implemented.

The operations/micro-services are invoked at a remote storage location through control messages that encapsulate all information required to perform the desired operations. The iRODS communication protocol must therefore support transmission of complex structures, which may be unique to a single micro-service.

This document defines the iRODS communication protocol, and the mechanisms used to serialize structures for transmission over a wide-area-network. We present the communication protocol including the types of messages that are sent over the network. We then present the approach used to serialize the structures needed to perform the remote operations. We close with a discussion of the overlap of this approach to the Open Grid Forum services.

iRODS Communication Protocol

The iRODS clients and servers communicate with each other by sending iRODS messages. A key feature of the protocol is that the messages used for communication from client to server are also used for communication between servers.

There are a total of six different types of messages that may be sent:
<table>
<thead>
<tr>
<th>Message type</th>
<th>Message function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RODS_CONNECT</td>
<td>Establish a socket connection</td>
</tr>
<tr>
<td>RODS_DISCONNECT</td>
<td>Disconnect the socket</td>
</tr>
<tr>
<td>RODS_API_REQ</td>
<td>Send the requested operation</td>
</tr>
<tr>
<td>RODS_API_REPLY</td>
<td>Reply with the operation results</td>
</tr>
<tr>
<td>RODS_REAUTH</td>
<td>Authenticate between federated data grids</td>
</tr>
<tr>
<td>RODS_VERSION</td>
<td>Verify version compatibility</td>
</tr>
</tbody>
</table>

Table 1. Types of messages sent from clients to servers and between servers

The ROCS_API_REQ is a generic message that is used to transmit structures required by any of the API/micro-services listed in Appendix A. The approach is to define a template (Packing Instruction) for each micro-service that defines how to parse the structures that have been serialized in a message body. Thus for each of the micro-services listed in Appendix A, there is a corresponding parsing template for parsing the structures that are being sent to the remote site.

The original implementation was based on serializing (packing) C structs into a tightly packed sequence of bytes and reconstructing (unpacking) the byte sequence into the C structs at the remote site. This is called the NATIVE_PROT encoding. This gives the best performance for sending information to a remote site. However, in order to support Java based interfaces, an XML-based syntax has also been defined, called the XML_PROT encoding. The content that is sent in the iRODS communication protocol can be packed using the NATIVE_PROT encoding or the XML_PROT encoding, depending on the communication protocol specified for the session. If the communication protocol is NATIVE_PROT, the contents of a C struct will be written into a buffer sequentially based on the Packing Instruction (PI) for that C struct. An assumption is that the receiving server will apply the same Packing Instruction to sequentially unpack the buffer into a C struct using the unpackStruct() routine. NATIVE_PROT is the default encoding because it is more efficient although the message header and the RODS_CONNECT_T type messages (see the section on Protocol for the initial connection) always use the XML_PROT encoding.

Each iRODS message can contain up to 5 parts as described below:

1) **A 4 bytes integer** specifying the length of the Message Header that follows.

2) **The iRODS message header.** In the NATIVE_PROT encoding, the message header is represented by the msgHeader_t struct defined as follows:

```c
typedef struct msgHeader {
    char type[HEADER_TYPE_LEN];
    int msgLen;   /* Length of the main msg */
    int errorMsg; /* Length of the error struct */
    int bsLen;    /* Length of optional byte stream */
    int intInfo;  /* an additional integer info, for API, it is the apiReqNum */
} msgHeader_t;
```

In the XMO_PROT encoding, the message header is tagged with the name of each element. The iRODS subroutine “packStruct ()” packs C structs into the XML syntax. The following is a typical message header for an API request type message:

```xml
<MsgHeader_PI>
    <type>RODS_API_REQ</type>
    <msgLen>398</msgLen>
    <errorMsg>0</errorMsg>
</MsgHeader_PI>
```

iRODS
3) **The Main Message.** This is typically a C struct packed with the packStruct() subroutine. The length of this message is given in the "msgLen" element of the message header.

If the communication protocol is XML_PROT, each item in the C struct will be tagged with XML tags in the packing process. It uses the name given for each element in the Packing Instruction for the XML tag name. For example, the string MsgHeader_PI given below is the Packing Instruction for the msgHeader_t struct:

```c
#define MsgHeader_PI "str type[HEADER_TYPE_LEN];int msgLen; int bsLen;int errorLen;int intInfo;"
```

The packStruct() routine uses this Packing Instruction to pack a given msgHeader_t C struct and produces an XML message similar to the example given in 2). Note that the name of the tags used for each element are the names given in MsgHeader_PI.

The Packing Instruction contains:
- Name of the packing instruction
- A list of packing instructions separated by ";"
- Each packing instruction has two parts, the data type and the name of the struct or variable that is being packed.

4) **An Error message.** The error messages have been extended to support multiple levels of error analysis for tracking error returns through multiple levels of software. Thus instead of an error number, the error message is characterized by the rError_t struct which is defined as:

```c
typedef struct {
    int count;          /* number of error messages in the stack */
    rErrMsg_t **errMsg; /* an array of pointers to the rErrMsg_t struct */
} rError_t;
```

where the rErrMsg_t struct is defined by:

```c
typedef struct {
    int status;
    char msg[ERR_MSG_LEN];
} rErrMsg_t;
```

The rError_t struct is packed with the packStruct() routine and the length of the error message is given in the "errorLen" element of the message header. In most iRODS messages, errorLen is 0 meaning no the error message.

5) **A Byte Stream.** The "Main message" is followed by a "Byte Stream" which is a buffer of raw data (mostly for data transfer). The length of the "Bytes Stream" is given in the "bsLen" element of the message header. In most iRODS messages, bsLen is 0 meaning no raw data in being sent with the message.

**Communication Protocol Message Sequence**

The initial connection is managed through the following steps:
1) A rodsServer opens a socket and listens on a well known port
2) A rods client connects to the rodsServer.
3) The rods client sends a RODS_CONNECT_T iRODS message that includes a startupPack_t struct.
   Remember that a iRODS message can contain up to 5 parts. The information in the
   RODS_CONNECT message header is defined by:

   #define RODS_CONNECT_T "RODS_CONNECT"

   typedef struct startupPack {
       iRODSProt_t iRODSProt;
       int connectCnt;
       char proxyUser[NAME_LEN];
       char proxyRodsZone[NAME_LEN];
       char clientUser[NAME_LEN];
       char clientRodsZone[NAME_LEN];
       char relVersion[NAME_LEN];
       char apiVersion[NAME_LEN];
       char option[NAME_LEN];
   } startupPack_t;

   The RODS_CONNECT_T type message is always packed using the XML protocol. The "iRODSProt"
   element in the startupPack_t struct specifies the protocol to be used for all subsequent iRODS messages.
   One of the following two protocols can be used:

       /* protocol */
       typedef enum {
           NATIVE_PROT,
           XML_PROT
       } iRODSProt_t;

   i.e., a value of 0 means native and a value of 1 means XML. Within the iCommands unix shell
   commands and the client C library, the default is NATIVE_PROT. The protocol can be changed to XML
   by setting the env variable "iRODSProt" to 1. In addition, if the env variable "iRODSLogLevel" is set to
   3 or lower, the generated XML will be output to stdout. Similarly, on the server side, the generated XML
   can be output to the log file by setting $spLogLevel=3 in the server/bin/start.pl file.

4) The rodsServer forks and execs a rodsAgent. The startupPack is passed to the rodsAgent through env
   variables.
5) The rodsAgent replies with a RODS_VERSION_T message which includes a version_t struct.

   #define RODS_VERSION_T "RODS_VERSION"

   typedef struct {
       int status;
       char relVersion[NAME_LEN];
       char apiVersion[NAME_LEN];
   } version_t;

   At this point, the client is connected to the server and a few general info type API calls can be made. But
   to be able to manipulate remote files, the client must authenticate himself/herself to the server.

6) An authentication call is issued. Note that authentication calls by clients are made just like any
other API calls by sending a RODS_API_REQ_T message to the server.

7) General API request. Clients make API calls by sending a RODS_API_REQ_T message to the server. Each API has an entry in the RcApiTable[] table in the apiTable.h file which specifies the API number, the Packing Instruction for the input and output structs, flags to indicate whether there are input/output "Bytes Streams", authorization levels and the server handler for this API.

The client packs the input struct based on the Packing Instruction given in the RcApiTable[] table. The message header, the packed main message and the "Bytes stream" (if any) are sent to the server as a single iRODS message defined by:

```
#define RODS_API_REQ_T "RODS_API_REQ"
```

The server receives the API request, calls the API's handle to process the request and returns the results in a RODS_API_REPLY_T type message to the client defined by:

```
#define RODS_API_REPLY_T "RODS_API_REPLY"
```

8) Other Server to Client message types:

a) disconnect - a client sends a RODS_DISCONNECT_T message to the server to disconnect

```
#define RODS_DISCONNECT_T "RODS_DISCONNECT"
```

b) re-authentication - needed for cross-zone connection. Note this is not yet implemented.

```
#define RODS_REAUTH_T "RODS_REAUTH"
```

**Packing/Unpacking Scheme Used in iRODS**

The heart of the communication protocol is the serialization of arbitrary C structs for transmission over the network. In the iRODS open source software distribution, the packStruct() and unpackStruct() routines in packStruct.c are used to pack and unpack arbitrary C structs.

1) **pack/unpack Data Types**

The struct can contain the following data types:

- **"char"** - 8 bit char
- **"bin"** - 8 bit binary data. Syntactically, it is treated the same as "char". But when it is used with the XML protocol, the binary data are encoded and decoded with base64 so that they are printable in the XML message.
- **"str"** - similar to char - null terminated string.
- **"piStr"** - packing instruction string. Syntactically, it is treated the same as "str". But this string is used as the packing instruction for other elements. Using this string, a dependent type element "?" can be used to store different data types.
- **"int"** - 32 bit integer or float
- **"double"** - 64 bit integer or float
- **"struct"** - a struct within the struct
- **"?"** - dependent type. The data type or packing instruction is dependent on the content of the other "piStr" type string.
For example, consider a struct to be packed in C that is defined by:

```c
struct foo1 {
    int myint1;
    int64_t myint64;
};
```

The packing instruction for this struct is:

"int myint1; double myint64;"

If we give this packing instruction a name, such as structFoo1_PI, the packing instruction is written as:

```c
#define structFoo1_PI "int myint1; double myint64;"
```

Each packing instruction has a name that is used by iRODS internally. The packing instructions are listed in clientLib/include/packStruct.h. Each of these packing instructions is also included in the RodsPackTable[] table in the rodsPackTable.h header file. So to add a new packing instruction to iRODS, the packing instruction is defined in packStruct.h and then an entry is added in the RodsPackTable[] table in the rodsPackTable.h header file. Typically, these are packing instructions for the low-level exchange protocol between the client and server. For example, the StartupPack_PI is the packing instruction for the struct StartupPack_t, the startup packet sent from the client to the server during the initial connection handshake.

In addition, iRODS recognizes the packing instructions for the client-level API input/output structs. Each API has its own header file in the clientLib/include/api directory. For example, the rcGetMiscSvrInfo() API has a file named getMiscSvrInfo.h in this directory and MiscSvrInfo_PI in this file defines the packing instruction for the output struct of this call. Similarly, each one of these API packing instructions is included in the ApiPackTable[] table in the apiPackTable.h file.

In summary, the iRODS packing routines use the RodsPackTable[] table in rodsPackTable.h and the ApiPackTable[] table in apiPackTable.h to resolve and execute the packing instruction.

2) pack/unpack of structs within a struct

We can also pack structs within a struct. For example, consider a struct to be packed in C defined by:

```c
struct foo2 {
    char myChar;
    int64_t myint64;
    struct foo1 myStruct;
};
```

Note that "struct foo1" is the struct defined in the previous example and the packing instruction is structFoo1_PI. The packing instruction for "struct foo2" is:

```c
#define structFoo2_PI "char myChar; double myint64; struct structFoo1_PI;"
```

3) pack/unpack of arrays

The routines can also pack/unpack arrays of data types in a struct. For example, consider a struct to be packed in C defined by:

```c
struct foo3 {
```
char myChar[100];
int64_t myint64[200];
struct foo1 myStruct[5];
};

The packing instruction for this struct is:

#define structFoo3_PI "char myChar[100]; int64_t myint64[200]; struct structFoo1_PI[5];"

4) **pack/unpack of string**

The "str" type is a special case of "char". The "str" is assumed to be NULL terminated and the packing routine takes advantage of this to save space. For example, if the char array to be packed in C is,

char myChar[200];
and if the packing instruction is,

"char myChar[200];"
then the whole 200 characters will be packed.

But if the packing instruction is,

"str myChar[200];"
then only the string up to the NULL character will be packed which can potentially reduce the number of bytes sent over the network. The unpacking routine will still unpack the struct back to a char array that is 200 bytes long (i.e., malloc 200 bytes even if the strlen is shorter).

5) **pack/unpack of pointers**

Consider a struct to be packed in C that is defined by:

```c
struct foo5 {
    char *myChar;
    str *myStr;
    int *myint;
    struct foo1 *myStruct;
};
```

Suppose this struct contains a char pointer, a NULL terminated string pointer, an integer pointer and a pointer to struct foo1 and we define the packing instruction to be:

#define structFoo5_PI "char *myChar; str *myStr; int *myint; struct *structFoo1_PI;"

For the element myStr, "str *" means it is a pointer to a NULL terminated string. No further information is needed to pack/unpack the string. The characters pointed to by the pointer will be packed until a NULL character is encountered.

But in C, the element "char *myChar;" may be used to declare a pointer to a single character or to an array of characters. Similarly, the element "int *myint;" may mean a pointer to a single integer or to an array of integers. So we need to give the pack/unpack routine a hint as to the array size of the content represented by the pointer. We use an integer enclosed by () to represent the array size of the content. We name this integer the "hint dimension" of the pointer.

Therefore, if the packing instruction for "struct foo5" is:

#define structFoo5a_PI "char *myChar(10); str *myStr; int *myint(20); struct *structFoo1_PI(30);"
This means myChar is a pointer to an array of 10 characters, myInt is a pointer to an array of 20 integers and myStruct is a pointer to an array of 30 "struct foo1". If no hint is given, then it is a pointer to a single element.

For strings, the packing instruction of "str *myStr(10)" means it is a pointer to a NULL terminated string with a strlen of up to 9. The packing routine will pack only the string (including the NULL) in myStr and the unpacking routine will malloc exactly 10 bytes and then unpack the string.

Consider another string example. If the item to be packed in C is,

```c
char *myStr; /* pointer to an array of 5 strings stored in 200 bytes each */
```

the packing instruction will be,

"str *myStr(5)(200);"

6) **pack/unpack with dependent array size**

Often times, the array size represented by a pointer is not fixed, but depends on the value of another integer in the struct. For example, consider a struct to be packed defined by:

```c
struct foo6 {
    int intLen;
    int *myInt;  /* pointer to an array of integer with array size equal to "intLen" */
    int structLen;
    struct foo1 *myStruct; /* pointer to an array of struct with array size equal to "structLen" */
};
```

where the array sizes of myInt and myStruct depend on the value of the intLen integer. The packing instruction for this case will be:

```c
#define structFoo6_PI "int intLen; int *myInt(intLen); int structLen; struct *structFoo1_PI(structLen)"
```

The value of intLen will be evaluated at runtime and substituted in the packing instruction by the pack/unpacking routines. Note that the intLen element must be put before the myValue element for the dependency to be resolved.

7) **pack/unpack an array of pointers**

Within a struct, an array of pointers can be represented in one of the following two forms:

```c
struct foo7a {
    int *myInt[100]; /* an array of 100 (fixed size) pointers embedded in the struct */
};
```

or,

```c
struct foo7b {
    **myInt; /* pointer to an array of "intLen" pointers (a single pointer to an array of pointers */
};
```

We currently support only the second form ("struct foo7b") for now because it is probably more common. The first form can be done with a little more work on the packing instruction.
For the struct foo7b (2nd form), the packing instruction is:

#define structFoo7b_PI "int intLen; int *myInt[intLen];"

The string "int *myInt[intLen];" represents the packing instruction for a pointer to an array of "intLen" pointers. But what about the array size of the integer pointed to by each one these pointers? In this case, the "hint dimension" can be used. e.g., if each pointer represents a pointer to an array of 10 integer, the packing instruction will be:

#define structFoo7b_PI "int intLen; int *myInt[intLen](10);"

So, for pointers, the value enclosed by [] represents the array size of the pointer array and the value enclosed by () represents the array size of the content pointed to by each pointer.

8) packing dependent (free form) type.

This type allows a pointer in a struct to represent different data types depending on the content of another "piStr" type string. A "?" type and a "piStr" type always work in pairs with the "?" specifying that the pointer is a dependent type and the "piStr" element specifying to what type of pointer the dependent type belongs.

For example, if the struct to be packed is:

typedef struct MsParam {
    char *label;
    char *myType; /* this is the name of a packing instruction in rodsPackTable.h */
    void *inOutStruct; /* The pointer type depends on the content of myType */
    bytesBuf_t *inpOutBuf;
} msParam_t;

The packing instruction for this struct is:

#define MsParam_PI "str *label; piStr *myType; ?myType *inOutStruct; struct *BytesBuf_PI;"

Note that use of "str" to pack the "char *label" is more compact, as only the characters actually present are packed. Also note that use of the string "?myType" here means that the data type of the pointer depends on the content of myType. Also, the "piStr" element must be put before the "?" element for the dependency to be resolved.

The rcExecMyRule API, which is used by the "irule" command to execute client-initiated workflows, uses an array of msParam_t to pass input parameters to the various micro-services in the workflow being executed. The free form style input is needed because the rcExecMyRule API needs to support all types of micro-services with various input formats.

Packing/unpacking processes

The packing process involves writing the contents of the C struct into a buffer in the same order given in the Packing Instruction (PI) of the struct. If the element is a pointer, the content pointed to by the pointer is written. Multiple levels of struct-within-struct are handled by drilling down and packing the sub-struct whenever they are encountered.

Depending on the communication protocol used for the session, the struct can be packed using the
NATIVE_PROT encoding or the XML_PROT encoding. The NATIVE_PROT encoding which is the default protocol, packs everything without any tags or separators. Strings are copied directly from the C struct including the NULL character terminator. Integers and doubles are converted to the net format using ntoI() first before copying.

The packing algorithm for the XML_PROT protocol is the same except:

1) Each item in the C struct is tagged with an XML tag. The name given for each element in the Packing Instruction is used as the tag name. For example, if the Packing Instruction is:

   #define MyTest_PI "int myInt; str *myStr;", the output will look something like:

   <MyTest_PI>
   <myInt>3</myInt>
   <myStr>This is a test</myStr>
   </MyTest_PI>

2) Integers and doubles are converted into character strings as shown in the example above.

3) The way NULL pointers are handled is different. With XML encoding, we could represent a NULL pointer as a tag with empty content. For example, if the pointer myStr given above is NULL, we could represent it in XML as:

   <myStr></myStr>

   But this is indistinguishable from a zero length string (i.e., the content of a string containing a single null character). To distinguish between the two, a NULL pointer will NOT be packed at all and a tag with empty content (e.g., <myStr></myStr>) represents a string with zero length.
## Appendix A: iRODS Microservices

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>admChangeCoreIRB</td>
</tr>
<tr>
<td>admShowCoreIRB</td>
</tr>
<tr>
<td>delayExec</td>
</tr>
<tr>
<td>msiAddACLForDataToUser</td>
</tr>
<tr>
<td>msiAssociateKeyValuePairsToObj</td>
</tr>
<tr>
<td>msiAsynchExecution</td>
</tr>
<tr>
<td>msiCheckOwner</td>
</tr>
<tr>
<td>msiCheckPermission</td>
</tr>
<tr>
<td>msiCheckQuota</td>
</tr>
<tr>
<td>msiCommit</td>
</tr>
<tr>
<td>msiComputeChecksum</td>
</tr>
<tr>
<td>msiCreateAbstract</td>
</tr>
<tr>
<td>msiCreateCollByAdmin</td>
</tr>
<tr>
<td>msiCreateThumbNail</td>
</tr>
<tr>
<td>msiCreateUser</td>
</tr>
<tr>
<td>msiDataObjClose</td>
</tr>
<tr>
<td>msiDataObjCopy</td>
</tr>
<tr>
<td>msiDataObjCreate</td>
</tr>
<tr>
<td>msiDataObjLseek</td>
</tr>
<tr>
<td>msiDataObjOpen</td>
</tr>
<tr>
<td>msiDataObjRead</td>
</tr>
<tr>
<td>msiDataObjRepl</td>
</tr>
<tr>
<td>msiDataObjUnlink</td>
</tr>
<tr>
<td>msiDataObjWrite</td>
</tr>
<tr>
<td>msiDeleteCollByAdmin</td>
</tr>
<tr>
<td>msiDeleteData</td>
</tr>
<tr>
<td>msiDeleteDisallowed</td>
</tr>
<tr>
<td>msiDeleteUser</td>
</tr>
<tr>
<td>msiExtractMetadataForCongressionalRecords</td>
</tr>
<tr>
<td>msiExtractMetadataForDicomData</td>
</tr>
<tr>
<td>msiExtractMetadataForEmail</td>
</tr>
<tr>
<td>msiExtractMetadataForFitsImage</td>
</tr>
<tr>
<td>msiExtractTemplateMDFromBuf</td>
</tr>
<tr>
<td>msiFreeBuffer</td>
</tr>
<tr>
<td>msiGetClosestResourceToClient</td>
</tr>
<tr>
<td>msiGetDefaultResourceForData</td>
</tr>
<tr>
<td>msiGetNewObjDescriptor</td>
</tr>
</tbody>
</table>