The XRootD Protocol
Version 3.1.0

Andrew Hanushevsky
28-May-2018

1 Contents

2 Request/Response Protocol

2.1 Format of Client-Server Initial Handshake

When a client first connects to the **XRootd** server, it must perform a special handshake. This handshake will determine whether the client is communicating using **XRootd** protocol or another protocol hosted by the server.

The handshake consists of the client sending 20 bytes, as follows:

```
kXR_int32     0
kXR_int32     0
kXR_int32     0
kXR_int32     4  (network byte order)
kXR_int32  2012  (network byte order)
```

XRootd protocol, servers should respond, as follows:

```
streamid:  kXR_char  smid[2]
status:    kXR_unt16    0
msglen:    kXR_int32 rlen
msgval1:   kXR_int32 pval
msgval2:   kXR_int32 flag
```

Where:

smid is the initial *streamid*. The *smid* for the initial response is always two null characters (i.e., '\0');

rlen is the binary response length (e.g., 8 for the indicated response).

pval is the binary protocol version number.

flag is additional bit-encoded information about the server; as follows:
kXR_DataServer - This is a data server.
KXR_LBalServer - This is a load-balancing server.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The particular response format was developed for protocol version 2.0 and does not convey all of the information to capture features provided by subsequent protocol versions. In order to provide backward compatibility, this response format has been kept. The recommended mechanism to obtain all of the information that may be needed is to "piggy-back" a kXR_protocol Request with the handshake (i.e. send the handshake and the request with a single write).
3) All twenty bytes *must* be received by the server at one time. All known TCP implementations will guarantee that the first message is sent intact if all twenty bytes are sent in a single system call. Using multiple system calls for the first message may cause unpredictable results.

2.2 Data Serialization

All data sent and received is serialized (i.e., marshaled) in three ways:

1. Bytes are sent unaligned without any padding,
2. Data type characteristics are predefined (see table below), and
3. All integer quantities are sent in network byte order (i.e, big endian).

XRootd Type	Sign	Bit Length	Bit Alignment	Typical Host Type
kXR_char8	unsigned	8	8	unsigned char
kXR_unt16	unsigned	16	16	unsigned short
kXR_int32	signed	32	32	long[1]
kXR_int64	signed	64	64	long long

Table 1: XRootd Protocol Data Types

Network byte order is defined by the Unix **htons()** and **htonl()** macros for host to network short and host to network long, respectively. The reverse is defined by the **ntohs()** and **ntohl()** macros. Many systems do not define the long long versions of these macros. **XRootd** protocol requires that the **POSIX** version of long long serialization be used, as defined in the following figures. The OS-dependent **isLittleEndian()** function returns true if the underlying hardware using little endian integer representation.

[1] As of this writing, the long type has taken on several meanings for 64-bit architectures. Some machines define a long to be 64-bits and int 32-bits while some others reverse the definition.

```
unsigned long long htonll(unsigned long long x)
       {unsigned long long ret_val;
        if (isLittleEndian())
          {*( (unsigned long *)(&ret_val) + 1) =
                     htonl(*(  (unsigned long *)(&x)));
           *(((unsigned long *)(&ret_val))) =
                     htonl(*( ((unsigned long *)(&x))+1) );
           } else {
           *( (unsigned long *)(&ret_val)) =
                     htonl(*(  (unsigned long *)(&x)));
           *(((unsigned long *)(&ret_val)) + 1) =
                     htonl(*( ((unsigned long *)(&x))+1) );
           }
       return ret_val;
      };
```

Figure 1: POSIX Host to Network Byte Order Serialization

```
unsigned long long ntohll(unsigned long long x)
       {unsigned long long ret_val;
        if (isLittleEndian())
           {*( (unsigned long *)(&ret_val) + 1) =
                          ntohl(*( (unsigned long *)(&x)));
            *(((unsigned long *)(&ret_val))) =
                          ntohl(*(((unsigned long *)(&x))+1));
           } else {
            *( (unsigned long *)(&ret_val)) =
                          ntohl(*( (unsigned long*)(&x)));
            *(((unsigned long*)(&ret_val)) + 1) =
                          ntohl(*(((unsigned long*)(&x))+1));
           }
        return ret_val;
       };
```

Figure 2: POSIX Network to Host Byte Order Serialization

More compact and efficient, though OS restricted (i.e., Solaris and Linux), versions of 64-bit network byte ordering routines are given in the following figure.

```
#if defined(__sparc) || __BYTE_ORDER==__BIG_ENDIAN
#ifndef htonll
#define htonll(x) x
#endif
#ifndef ntohll
#define ntohll(x) x
#endif
#else
#ifndef htonll
#define htonll(x) __bswap_64(x)
#endif
#ifndef ntohll
#define ntohll(x) __bswap_64(x)
#endif
```

Figure 3: Network and Host Byte Ordering Macros

2.3 Client Request Format

Requests sent to the server are a mixture of ASCII and binary. All requests, other than the initial handshake request, have the same format, as follows:

```
kXR_char  streamid[2]
kXR_unt16 requestid
kXR_char  parms[16]
kXR_int32 dlen
kXR_char  data[dlen]
```

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

requestid
: is the binary identifier of the operation to be performed by the server.

parms are parameters specific to the *requestid*.

dlen is the binary length of the *data* portion of the message. If no data is present, then the value is zero.

data are data specific to the *requestid*. Not all requests have associated data. If the request does have data, the length of this field is recorded in the *dlen* field.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) All **XRootd** client requests consist of a standard 24-byte fixed length message. The 24-byte header may then be optionally followed by request specific data.
3) Stream id's are arbitrary and are assigned by the client. Typically these id's correspond to logical connections multiplexed over a physical connection established to a particular server.

4) The client may send any number of requests to the same server. The order in which requests are performed is undefined. Therefore, each request should have a different *streamid* so that returned results may be paired up with associated requests.
5) Requests sent by a client over a single physical connection may be processed in an arbitrary order. Therefore the client is responsible for serializing requests, as needed.

2.3.1 Valid Client Requests

Requestid	Login?	Auth?	Redirect?	Arguments
kXR_admin	yes	yes	no	*args*
kXR_auth	y	n	n	*authtype, authinfo*
KXR_bind	n	n	n	*sessid*
kXR_chmod	y	y	yes	*mode, path*
kXR_close	y	-	n	*fhandle*
kXR_decrypt	y	y	n	
KXR_dirlist	y	y	y	*path*
KXR_endsess	y	-	n	*sessid*
kXR_getfile*	y	y	y	*path*
kXR_locate	y	y	y	*path*
kXR_login	n	n	n	*userid, token*
kXR_mkdir	y	y	y	*mode, path*
kXR_mv	y	y	y	*old_name, new_name*
kXR_open	y	y	y	*mode, flags, path*
kXR_ping	y	n	n	
kXR_prepare	y	y	n	*paths*
kXR_protocol	n	n	n	
kXR_putfile*	y	y	y	*mode, flags, path*
kXR_query	y	y	y	*args*
kXR_read	y	-	y	*fhandle, pathid, length, offset*
kXR_readv	y	-	y	*fhandle, pathid, length, offset*
kXR_rm	y	y	y	*path*
kXR_rmdir	y	y	y	*path*
kXR_set	y	y	y	*info*
kXR_sigver	y	y	n	*signature*
kXR_stat	y	-	n	*fhandle*
kXR_stat	y	y	y	*path*
kXR_statx	y	y	n	*pathlist*
kXR_sync	y	-	n	*fhandle*
kXR_truncate	y	-	n	*fhandle, length*
kXR_truncate	y	-	y	*path, length*
kXR_write	y	-	y	*fhandle, pathid, length, offset, data*
kXR_verifyw	y	-	y	*fhandle, length, offset, data*

Table 2: Valid Client Requests

*

*These requests aere currently not supported.

2.3.2 Valid Client Paths

The **XRootd** server accepts only *absolute paths* where a path may be specified. Relative paths must be resolved by the client interface prior to sending them to **XRootd**. This means that the interface must handle a virtual "current working directory" to resolve relative paths should they arise.

Path names are restricted to the following set of characters:

- Letters (upper or lower case),
- Digits (0-9), and
- Special characters: `!@#%^_-+=:./`

In general, paths may not contain shell meta-characters or imbedded spaces.

2.3.3 Client Recovery from Server Failures

A server failure should be recognized when the server unexpectedly closes its TCP/IP connection or does not respond for an extended period of time. Should this happen, the client may recover all operations by treating the termination of the connection or unresponsiveness as a redirection request (see page 33) to the initial **XRootd** server for all streams associated with the closed TCP/IP connections.

The initial **XRootD** server is defined as the *first* manager or the *last* meta-manager encountered. In the absence of any manager, the first data server encountered. See the **kXR_protocol** request on how to determine a node's type.

Because many clients are likely to be affected by a server failure, it is important that clients pace their reconnection to the initial **XRootd** server. One effective way to do this is to use the last three bits of the client's IP address as the number of seconds to wait before attempting a reconnection. It is up to the client to determine either the number of times or the time window in which reconnections should be attempted before failure is declared. Typical values are 16 attempts or 3 minutes, whichever is longer.

Note that it may not be possible to recover in this way for files that were opened in update mode. Clients who do not provide proper transactional support generally cannot recover via redirection for any read/write resources.

2.4 Server Response Format

All responses, including the initial handshake response, have the same format, as follows:

```
kXR_char  streamid[2]
kXR_unt16 status
kXR_int32 dlen
kXR_char  data[dlen]
```

Where:

streamid
: is the binary identifier that is associated with this request stream corresponding to a previous request.

status is the binary status code indicating how the request completed. The next section describes possible status codes.

dlen is the binary length of the *data* portion of the message. If no data is present, then the value is zero.

data are data specific to the *requestid*. Not all responses have associated data. If the response does have data, the length of this field is recorded in the *dlen* field.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.
3) Unsolicited responses are server requests for client configuration changes to make better use of the overall system. Since these responses do not correspond to any request, the *streamid* value has no meaning.
4) Unsolicited responses must be immediately acted upon. They should not be paired with any previous request.

2.4.1 Valid Server Response Status Codes

The following table lists all possible requests and their arguments. Grayed rows represent requests that are not currently supported.

Status	Response Data
`kXR_attn`	*Parameters to direct immediate client action*
`kXR_authmore`	*Authentication specific data*
`kXR_error`	*Error number and corresponding* ASCII *message text*
`kXR_ok`	*Depends on request* (**this is predefined to be the value 0**)
`KXR_oksofar`	*Depends on request*
`kXR_redirect`	*Target port number and* ASCII *host name*
`kXR_wait`	*Binary number of seconds and optional* ASCII *message*
`kXR_waitresp`	*Binary number of seconds*

Notes

1) Any request may receive any of the previous status codes.
2) The following sections detail the response format used for each status code.

2.4.2 Server kXR_attn Response Format

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 plen
kXR_int32 actnum
kXR_char  parms[plen-4]
```

Where:

plen is two bytes of padding required by the standard response format. These two bytes can be ignored for this particular response code.

plen is the binary length of the *parms* portion of the message (i.e., the subsequent bytes).

actnum
is the binary action code describing the action that the client is to take. These are:

kXR_asyncav - The file or file(s) the client previously requested to be prepared are now available.

kXR_asyncab - The client should immediately disconnect (i.e., close the socket connection) from the server and abort further execution.

kXR_asyncdi - The client should immediately disconnect (i.e., close the socket connection) from the server. Parameters indicate when a reconnect may be attempted.

kXR_asyncgo - The client may start sending requests. This code is sent to cancel the effects of a previous **kXR_asyncwt** code.

kXR_asyncms - The client should send the indicated message to the console. The parameters contain the message text.

kXR_asyncrd - The client should immediately disconnect (i.e., close the socket connection) and reconnect to the indicated server.

kXR_asynresp - The client should use the response data in the message to complete the request associated with the indicated streamid.

kXR_asynunav - The file or file(s) the client previously requested to be prepared cannot be made available.

kXR_asyncwt - The client should hold off sending any new requests until the indicated amount of time has passed or until receiving a **kXR_asyncgo** action code.

parms is the parameter data, if any, that is to steer client action.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **`kXR_attn`** response code to optimize overall system performance and to notify clients of any impending events. All responses *except* for **kXR_asynresp**, do not correspond to any client request and should not be paired up with any request.
3) When **`kXR_attn`** is received, the client must perform the requested action and indicated by the *actnum* value.

2.4.2.1 Server kXR_attn Response for kXR_asyncab Client Action

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 mlen
kXR_int32 kXR_asyncab
kXR_char  msg[mlen-4]
```

Where:

mlen is the binary length of the following action code and message.

msg is the message to be sent to the terminal. The *mlen* value, less four, indicates the length of the message. The ending null byte ('\0') is transmitted and included in the message length.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.
3) When **kXR_attn** is received with the **kXR_asyncab** action code, the client should close all physical connections, write the message (*msg*), if any, to standard error, and terminate execution.

2.4.2.2 Server kXR_attn Response for kXR_asyncdi Client Action

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 12
kXR_int32 kXR_asyncdi
kXR_int32 wsec
kXR_int32 msec
```

Where:

wsec is the number of seconds the client should wait before attempting to reconnect to the server.

msec is the maximum number of seconds the client should wait before declaring reconnect failure.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.
3) When **kXR_attn** is received with the **kXR_asyncdi** action code, the client should close the physical connection, wait *wsec* seconds, and attempt to reconnect to the server.
4) If a server reconnect fails, the client should wait either an additional *wsec* seconds or some other predetermined time and try again. If *msec* seconds have gone since the initial wait and the client has not reconnected to the server, a reconnect failure should be declared.
5) When a reconnect failure is declared, the client may either terminate the program or perform an internal redirection to a load-balancing server.
6) A reconnect is essentially a delayed redirect to the same server. The actions that must be carried out when reconnecting are identical to those that must be performed when reconnecting to a different server. Refer to the description of the **kXR_asyncrd** action for the set steps that the client must take to successfully reconnect.

2.4.2.3 Server kXR_attn Response for kXR_asyncgo Client Action

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 4
kXR_int32 kXR_asyncgo
```

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.
3) When **kXR_attn** is received with the **kXR_asyncgo** action code, the client may resume sending requests to the server.
4) The **kXR_asyncgo** code is sent to cancel the effects of a previously sent **kXR_asyncwt** code. Therefore, if the client is still waiting for the **kXR_asyncwt** interval to expire, the interval should be cancelled.

2.4.2.4 Server kXR_attn Response for kXR_asyncms Client Action

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 mlen
kXR_int32 kXR_asyncms
kXR_char  msg[mlen-4]
```

Where:

mlen is the binary length of the following action code and message.

msg is the message to be sent to the terminal. The *mlen* value, less four, indicates the length of the message. The ending null byte ('\0') is transmitted and included in the message length.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.
3) When **kXR_attn** is received with the **kXR_asyncms** action code, the client should simply write the indicated message to the terminal.

2.4.2.5 Server kXR_attn Response for kXR_asyncrd Client Action

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 plen
kXR_int32 kXR_asyncrd
kXR_int32 port
kXR_char  host[?token][plen-8]
```

Where:

plen is the binary length of the *parameter* portion of the message (i.e., the subsequent bytes).

port is the binary port number to which the client must connect. If the value is zero, the default **XRootd** port number must be used. If the value is negative, then the text after *port* contains a standard URL that must be used to effect a new connection. This should only occur if the client has indicated that URL redirection responses are acceptable during the most recent kXR_login request to the redirecting server.

host is the ASCII name of the to which the client must connect. The *host* does *not* end with a null (\0) byte. The host should be interpreted as a standard URL if *port* is negative (see above).

token is an optional ASCII token that, when present, must be delivered to the new host during the login phase, if one is needed. The *token*, if present, is separated from the *host* by a single question mark. The *token* does *not* end with a null (\0) byte.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.

3) When **kXR_attn** is received with the **kXR_asyncrd** action code, the client should perform the following steps:
    a. Decompose the response to extract the port number, host name, and possible token value.
    b. Physically close the connection to the current host, regardless of type.
    c. Establish a new physical connection with the indicated host at the specified or default port number.
    d. Perform the initial handshake, login with token (see **kXR_login** description), and authentication (see **kXR_auth** description).
    e. Re-establish all open files, as needed. Previously opened files may be re-opened all at once or when a request attempts to use the file.
    f. Re-issue any requests that were sent to the previous server but have not received a response.
4) Since **XRootd** allows multiple open files per physical connection, a **kXR_asyncrd** response can become somewhat complicated to handle. The client can re-open files immediately after a new connection is made or can re-open files as they are needed. In either case, the client must:
    g. Issue a **kXR_open** request using the same file name and options as was originally used.
    h. Use the returned file handle for all subsequent requests for that file (i.e., substitute the new *fhandle* for the old *fhandle*).
5) An **XRootd** server will never redirect a physical connection to a **rootd** server. This differs for logical connections where a logical connection may be so redirected.
6) After 256 redirect responses within 10 minutes on the same physical connection, the client should declare an internal system error since it is obvious that effective work is not being performed.

2.4.2.6 Server kXR_attn Response for kXR_asynresp Client Action

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 rlen
kXR_int32 kXR_asynresp
kXR_char  reserved[4]
kXR_char  streamid[2]
kXR_unt16 status
kXR_int32 dlen
kXR_char  data[dlen]
```

Where:

rlen is the binary length of the following action code and response.

streamid
is the stream identifier associated with a previously issued request that received a **kXR_waitresp** response.

status is the binary status code indicating how the request completed. The codes definitions are identical as to those described for synchronous responses.

dlen is the binary length of the *data* portion of the message. If no data is present, then the value is zero.

data are data specific to the request. Not all responses have associated data. If the response does have data, the length of this field is recorded in the *dlen* field.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events.
3) Unlike other asynchronous events, this response is associated with a previous request and the response data must be used to complete that request.
4) The *rlen-dlen* is *always* 16.

5) When **`kXR_attn`** is received with the **`kXR_asynresp`** action code, the client should remove the request paired with *streamid* from wait state and complete it using the response data.

2.4.2.7 Server kXR_attn Response for kXR_asyncwt Client Action

```
kXR_char  pad[2]
kXR_unt16 kXR_attn
kXR_int32 8
kXR_int32 kXR_asyncwt
kXR_int32 wsec
```

Where:

wsec is the number of seconds the client should wait before sending any more requests to the server.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.
3) When **kXR_attn** is received with the **kXR_asyncwt** action code, the client should queue any new requests (i.e., not send new requests) until *wsec* seconds have elapsed.
4) While waiting, the client should still be receiving messages from the server. It is possible for the server to send additional unsolicited responses even after a **kXR_asyncwt** has been sent. For example, the server may send a **kXR_asyncgo** request to cancel the effects of the **kXR_asyncwt** request before the *wsec* interval has gone by.

2.4.3 Server kXR_authmore Response Format

```
kXR_char  streamid[2]
kXR_unt16 kXR_authmore
kXR_int32 dlen
kXR_char  data[dlen]
```

Where:

streamid
: is the binary identifier that is associated with this request stream corresponding to a previous request.

dlen is the binary length of the *data* portion of the message (i.e., the subsequent bytes).

data is the data, if any, required to continue the authentication process.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.
3) The **kXR_authmore** response code is issued only for those authentication schemes that require several handshakes in order to complete (e.g., .x500).
4) When a **kXR_authmore** response is received, the client must call the appropriate authentication continuation method and pass it *data*, if present. The output of the continuation method should be sent to the server using another **kXR_auth** request. This handshake continues until either the continuation method fails or the server returns a status code of **kXR_error** or **kXR_ok**.
5) Refer to the description of the security framework for detailed information.

2.4.4 Server kXR_error Response Format

```
kXR_char  streamid[2]
kXR_unt16 kXR_error
kXR_int32 dlen
kXR_int32 errnum
kXR_char  errmsg[dlen-4]
```

Where:

streamid
: is the binary identifier that is associated with this request stream corresponding to a previous request.

dlen
: is the binary length of the *data* portion of the message (i.e., the subsequent bytes).

errnum
: is the binary error number indicating the nature of the problem encountered when processing the request.

errmsg
: is the human-readable null-terminated message that describes the error. This message may be displayed for informational purposes.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since the error message is null-terminated, *dlen* includes the null byte in its count of bytes that were sent.
3) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.

2.4.4.1 Server kXR_error Sub-Codes & Recovery Actions

The following table lists possible error sub-codes included in the *errnum* field as part of the **kXR_error** response:

Status	Meaning	Redirector Recovery	Server Recovery
kXR_ArgInvalid	A request argument was not valid	n/a	n/a
kXR_ArgMissing	Required request argument was not provided	n/a	n/a
kXR_ArgTooLong	A request argument was too long (e.g., path)	n/a	n/a
kXR_Cancelled	The operation was cancelled by the administrator	n/a	n/a
kXR_ChkLenErr	The close length does not equal the file size	n/a	n/a
kXR_ChkSumErr	The **kXR_verifyw** checksum does not match	n/a	n/a
kXR_DecryptErr	Data could not be decrypted	n/a	n/a
kXR_FileLocked	File is locked, open request was rejected	n/a	n/a
kXR_FileNotOpen	File if not open for the request (e.g., read)	n/a	n/a
kXR_FSError	The file system indicated an error	n/a	**A**
kXR_inProgress	Operation already in progress	**B**	**B**
kXR_InvalidRequest	The request code is invalid	n/a	n/a
kXR_IOError	An I/O error has occurred	n/a	**A**
kXR_isDirectory	Object being opened with **kXR_open** is a directory	n/a	n/a
kXR_NoMemory	Insufficient memory to complete the request	**C**	**B**
kXR_NoSpace	Insufficient disk space to write data	n/a	n/a
kXR_NotAuthorized	Client is not authorized for the request	n/a	n/a
kXR_NotFile	The object being opened with **kXR_open** is not a file.	n/a	n/a
kXR_NotFound	The requested file was not found	n/a	**D**
kXR_noserver	There are no servers available to process the request	n/a	n/a
kXR_overQuota	Space quota exceeded	n/a	n/a
kXR_ServerError	An internal server error has occurred	**C**	**A**
kXR_SigVerErr	Request signature could not be verified	**n/a**	**n/a**
kXR_Unsupported	The request is valid but not supported	n/a	n/a

A. Go back to the redirector and ask for a different server**. kXR_refresh** *should not* be turned on and "tried=" opaque value should indicate the hostname of the failing server.

B. Generally, this represents a programming error. However, should an operation subject to a callback response be retried prior to the callback, this status code may be returned. Clients should honor server's callback requests and wait for a callback response. Therefore, this error can be ignored as long as a callback is outstanding. Otherwise, it should be treated as a fatal error.

C. If the redirector is replicated, a different redirector should be tried. If all redirectors provide the same response, a fatal error should be reported. In the case of intermediate redirectors (i.e., a redirector transferring the request to another redirector), the recovery may be attempted by treating the intermediate as a server and performing the action outline in **A**.

D. Go back to the redirector and ask for a different server. **kXR_refresh** *should* be turned on and "tried=" opaque value should indicate the hostname of the failing server. This should normally be done only once.

2.4.5 Server kXR_ok Response Format

```
kXR_char  streamid[2]
kXR_unt16 kXR_ok
kXR_int32 dlen
kXR_char  data[dlen]
```

Where:

streamid
: is the binary identifier that is associated with this request stream corresponding to a previous request.

dlen
: is the binary length of the *data* portion of the message (i.e., the subsequent bytes).

data
: is the result, if any, of the corresponding request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.
3) The **kXR_ok** response indicates that the request fully completed and no additional responses will be forthcoming.

2.4.6 Server kXR_oksofar Response Format

```
kXR_char  streamid[2]
kXR_unt16 kXR_oksofar
kXR_int32 dlen
kXR_char  data[dlen]
```

Where:

streamid
: is the binary identifier that is associated with this request stream corresponding to a previous request.

dlen is the binary length of the *data* portion of the message (i.e., the subsequent bytes).

data is the result, if any, of the corresponding request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.
3) The **kXR_oksofar** response indicates that the server is providing partial results and the client should be prepared to receive additional responses on the same stream. This response is primarily used when a read request would transmit more data than the internal server segment size. Refer to the **kXR_getfile** and **kXR_read** requests.
4) Sending requests using the same *streamid* when a **kXR_oksofar** status code has been returned may produced unpredictable results. A client must serialize all requests using the *streamid* in the presence of partial results.
5) Any status code other than **kXR_oksofar** indicates the end of transmission

2.4.7 Server kXR_redirect Response Format

```
kXR_char  streamid[2]
kXR_unt16 kXR_redirect
kXR_int32 dlen
kXR_int32 port
kXR_char  host[?[opaque][?token]][dlen-4] | url
```

Where:

streamid
is the binary identifier that is associated with this request stream corresponding to a previous request.

dlen is the binary length of the *data* portion of the message (i.e., the subsequent bytes).

port is the binary port number to which the client must connect. If the value is zero, the default **XRootd** port number must be used. If the value is negative, then the text after *port* contains a standard URL that must be used to effect a new connection. This should only occur if the client has indicated that URL redirection responses are acceptable during the most recent kXR_login request to the redirecting server.

host is the ASCII name of the to which the client must connect. The *host* does *not* end with a null (\0) byte. The *host* should be interpreted as a standard URL if *port* is negative (see above).

opaque is an optional ASCII token that, when present, must be delivered to the new host as opaque information added to the file name[2] associated with the operation being redirected. The *opaque*, if present, is separated from the *host* by a single question mark. The *opaque* does *not* end with a null (\0) byte but may end with a question mark (see *token* below). Therefore, *opaque* may never contain a question mark.

token is an optional ASCII token that, when present, must be delivered to the new host during the login phase, if one is needed (i.e. established connections to the specified host may be re-used without a login). The *token*, if present, is separated from the *host* by a two question marks. The first question mark may be followed by *opaque* information. If none is

[2] In the case of **kXR_mv**, two file names are present. The opaque information must be added to the second of the two file names.

present, another question mark immediately follows the first one. The *token* does *not* end with a null (\0) byte.

url when a client indicates that it supports multi-protocol redirects, the server may respond with an actual url. In this case, the *port* value is set to -1.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.
3) After 256 redirect responses within 10 minutes on the same logical connection, the client should declare an internal system error since it is obvious that effective work is not being performed.
4) The client must be prepared to handle a redirect response at any time. A redirect response requires that the client
    i. Decompose the response to extract the port number, host name, and possible token value.
    j. Possibly close the connection of the current host, if the current host is a data server and this is the last logical connection to the server. Otherwise, if this is the first load-balancing server encountered in the operation sequence, the connection should remain open since a load-balancing server always responds with a redirect.
    k. Establish a new logical connection with the indicated host at the specified or default port number. If a physical connection already exists and is session compatible with the new logical connection; the existing physical connection should be reused and the next step (i.e. handshake and login) should be skipped.
    l. Perform the initial handshake, login with token (see **`kXR_login`** description), and authentication (see **`kXR_auth`** description).
    m. If the redirection occurred for a request using a file handle (i.e., *fhandle*) then a new file handle must be obtained.
        i. A **kXR_open** request must be issued using the same file name and options as was originally used.
        ii. The returned file handle must be used for the request that is to be re-issued as well as all subsequent requests relating o the file.
    n. Re-issue the request that was redirected.
5) Opaque data must be treated as truly opaque. The client should not inspect nor modify the data in any way.

2.4.8 Server kXR_wait Response Format

```
kXR_char  streamid[2]
kXR_unt16 kXR_wait
kXR_int32 dlen
kXR_int32 seconds
kXR_char  infomsg[dlen-4]
```

Where:

streamid
is the binary identifier that is associated with this request stream corresponding to a previous request.

dlen is the binary length of the *data* portion of the message (i.e., the subsequent bytes).

seconds
is the maximum binary number of seconds that the client needs to wait before re-issuing the request.

infomsg
is the human-readable message that describes the reason of why the wait is necessary. The message does *not* end with a null (\0) byte. This message may be displayed for informational purposes.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.
3) The client should wait the indicated number of seconds and retry the request.
4) Nothing prohibits the client from waiting for less time than the indicated number of seconds.

2.4.9 Server kXR_waitresp Response Format

```
kXR_char  streamid[2]
kXR_unt16 kXR_waitresp
kXR_int32 4
kXR_int32 seconds
```

Where:

streamid
: is the binary identifier that is associated with this request stream corresponding to a previous request.

seconds
: is the *estimated* maximum binary number of seconds that the client needs to wait for the response.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the *streamid* value.
3) The client should wait the indicated number of seconds for the response. The response will be returned via an unsolicited response (**kXR_attn** with **kXR_asynresp**) at some later time which may be earlier than the time indicated in seconds. When the response arrives, the client must use the response data to complete the request that received the **kXR_waitresp**.
4) Nothing prohibits the client from waiting for different time than the indicated number of *seconds*. Generally, if no response is received after at least *seconds* have elapsed; the client should treat the condition as a fatal error.

3 Detailed Protocol Specifications

3.1 kXR_admin Request

Purpose: Perform an administrative function.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_admin**	**kXR_unt16**	**0**
kXR_char	*reserved*[16]	**kXR_int32**	*ilen*
kXR_int32	*rlen*	**kXR_char**	*resp*[*ilen*]
kXR_char	*reqs*[*rlen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

rlen is the binary length of the supplied request, *reqs*.

reqs is the request.

ilen is the binary length of the response, *resp*, that follows *ilen*.

resp is the response to the administrative request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The **kXR_admin** request is only valid for users who have successfully performed a **kXR_login** operation in an administrative role (i.e., logged in as administrator).
3) This request type is not currently supported. Use the local socket interface protocol to execute administrative requests.

3.2 kXR_auth Request

Purpose: Authenticate client's username to the server.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_auth**	**kXR_unt16**	**0**
kXR_char	*reserved*[12]	**kXR_int32**	**0**
kXR_char	*credtype*[4]		
kXR_int32	*credlen*		
kXR_char	*cred*[*credlen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed akXR_int32 with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

credtype
: the first four characters of the protocol name. If the protocol name is less than four characters, the name should be null terminated.

credlen
: is the binary length of the supplied credentials, *cred*.

cred are the credentials used to provide authentication information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Authentication credentials may be supplied by many means. The common mechanism used by **XRootd** is to use the classes in the **libXrdSec.so** library. See the "Authentication & Access Control Configuration Reference" for more information.
3) Refer to the description of the security framework on how a client authenticates to an **XRootd** server.

3.3 kXR_bind Request

Purpose: Bind a socket to a pre-existing session.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_bind**	**kXR_unt16**	**0**
kXR_char	*sessid*[16]	**kXR_int32**	**1**
kXR_int32	**0**	**kXR_char**	*pathid*

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

sessid is the session identifier returned by a previous **kXR_login** request.

pathid is the socket identifier associated with this connection. The *pathid* may be used in subqsequent **kXR_read**, **kXR_readv**, and **kXR_write** requests to indicate which socket should be used for a response or as a source of data.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *sessid* value should be treated as opaque data.
3) The socket issuing the **kXR_bind** request must neither have a session id (i.e., be logged in) nor be already bound.
4) Once a socket is bound to a session, if may only supply data for **kXR_write** requests or receive responses for **kXR_read** and **kXR_readv** requests.
5) Should the client close a bound socket, the client should issue a **kXR_unbind** request specifying the *pathid* of the socket that was just closed. Failure to do so may cause future **kXR_bind** requests to fail.
6) Each login session is limited to the number of bound sockets. Use the **kXR_Qconfig** sub-request code of **kXR_query** to determine the maximum number of sockets that can be bound to a login session.
7) Bound sockets are meant to support parallel data transfer requests across wide-area networks.

3.4 kXR_chmod Request

Purpose: Change the access mode on a directory or a file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_chmod**	**kXR_unt16**	**0**
kXR_char	*reserved*[14]	**kXR_int32**	**0**
kXR_unt16	*mode*		
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
is an area reserved for future use and must be initialized to null characters (i.e., '\0').

mode is the access mode to be set for *path*. The access mode is an "or'd" combination of the following values:

Access	Readable	Writeable	Executable
Owner	kXR_ur	kXR_uw	*not supported*
Group	kXR_gr	kXR_gw	*not supported*
Other	kXR_or	*not supported*	*not supported*

plen is the binary length of the supplied path, *path*.

path is the path whose mode is to be set.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) No **umask** is applied to the specified mode.

3.5 kXR_close Request

Purpose: Close a previously opened file, communications path, or path group.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_close**	**kXR_unt16**	**0**
kXR_char	*fhandle*[4]	**kXR_int32**	**0**
kXR_int64	*fsize*		
kXR_char	*reserved*[4]		
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request.

fsize
: the size, in bytes, that the file is to have. The close operation fails and the file is erased if it is not of the indicated size. An *fsize* of zero suppresses the check.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

3.6 kXR_decrypt Request

Purpose: Signal when the data stream is encrypted.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_decrypt**	**kXR_unt16**	**0**
kXR_char	*reserved*[16]	**kXR_int32**	**0**
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is a reserved field and should be set to zero.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The **kXR_decrypt** request should be considered as not fully specified. It is currently a place-holder for future enhancement.

3.7 kXR_dirlist Request

Purpose: Enumerate the contents of a directory.

Request

```
kXR_char   streamid[2]
kXR_unt16  kXR_dirlist
kXR_char   reserved[15]
kXR_char   options
kXR_int32  plen
kXR_char   path[plen]
```

Normal Response

```
kXR_char   streamid[2]
kXR_unt16  0
kXR_int32  dlen
kXR_char   dirname\n
           •
           •
           •
kXR_char   0
```

Normal Response w/ kXR_dstat

```
kXR_char   streamid[2]
kXR_unt16  0
kXR_int32  dlen
kXR_char   ".\n"
kXR_char   "0 0 0 0\n"
kXR_char   dirname\n
kXR_char   statinfo\n
           •
           •
           •
kXR_char   0
```

statinfo: id size flags modtime

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

options
: is, optionally, one or more of the following:
kXR_dstat - return stat information with each entry (protocol version 3+).

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*.

path is the path of a directory whose entries are to be listed.

dlen is the binary length of the data that follows *dlen*.

dirname
is an entry in the directory whose listing was requested.

statinfo
the **kXR_stat** information for the preceeding *dirname*. Refer to **kXR_stat** for details on the meaning of *id, size, flags,* and *modtime*. The *statinfo* is only returned when **kXR_dstat** is set and the server issuing protocol version 3 or higher.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) A directory may have multiple entries and the response contains all of the entries.
3) Each directory entry is suffixed with a new-line character; except for the last entry which is suffixed by a null character.
4) Since more entries may exist than is possible to send at one time, the **kXR_oksofar** protocol may be used to segment the response. Under no circumstances will a directory name be split across a response packet.
5) The server does not return the entries "." And "..".
6) An empty directory will return the eight-byte triplet {*streamid*, 0, 0}.
7) Cleints should always check if the server supports **kXR_dstat**. If the option is supported, the first entry will be a *dot* entry followed the zero stat information.

3.8 kXR_endsess Request

Purpose: Terminate a pre-existing session.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_endsess**	**kXR_unt16**	**0**
kXR_char	*sessid*[16]	**kXR_int32**	**0**
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

sessid
: is the session identifier returned by a previous **kXR_login** request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *sessid* value should be treated as opaque data.
3) The socket issuing the **kXR_endsess** request must be logged in and, optionally, authenticated.
4) If the *sessid* is all binary zeroes, the current session is terminated.
5) The server verifies that the process presenting the *sessid* actually received it on a previous **kXR_login**.

3.9 kXR_getfile Request

Purpose: Retrieve a complete file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_getfile**	**kXR_unt16**	*status*
kXR_int32	*options*	**kXR_int32**	*dlen*
kXR_char	*reserved*[8]	**kXR_int64**	*offset*
kXR_int32	*buffsz*	**kXR_char**	*data*[*dlen*-8]
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

status is the ending status of this request. Only the following two status codes indicate a normal ending:
kXR_ok - All of the data has been transmitted with error.
kXR_oksofar - Partial data has been transmitted without error; additional data should be expected on this stream.

options
is a bit vector representing the options that are to apply to the file transfer. The valid set of options are:
kXR_md5file - Compute and transmit an MD5 checksum for the file.
KXR_md5blok - Compute and transmit an MD5 checksum for each block.

reserved
is an area reserved for future use and must be initialized to null characters (i.e., '\0').

buffsz is the maximum binary length to be transmitted per file segment (i.e., buffer size). If *buffsz* is zero, 65,544 (i.e., 64K+8) is used.

plen is the binary length of the supplied path, *path*.

path is the path of the file to be retrieved.

dlen is the binary length of the data that follows with *dlen* never being greater than *buffsz*.

offset is the binary offset of where *data* was located within the file. Negative offsets indicate special non-file data is being transmitted. See the notes for more information.

data is the data associated with the file.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since a file may be much larger than the allowable buffer size, the file is sent in *buffsz* segments until the whole file is sent. This is accomplished using the **kXR_oksofar** status code. Each subsequent data segment is transmitted using a {*streamid*, *status*, *dlen*, *offset*, *data*} response. The last segment is indicated by a **kXR_ok**, if no error occurred.
3) Any status code other than **kXR_oksofar** indicates the end of transmission.
4) Sending requests using the same *streamid* when a **kXR_oksofar** status code has been returned may produced unpredictable results. A client must serialize all requests using the *streamid* in the presence of partial results.
5) When a 16-byte MD5 checksum is requested, it is transmitted either after the complete file is transferred or after each block, as specified by the options. An MD5 checksum will have a *dlen* of 24 and an *offset* of negative one (i.e., -1).
6) MD5 block checksums are always sent on the same TCP/IP connection that was used to send the block.
7) An empty file will return the eight-byte triplet {*streamid*, 0, 0}.
8) Empty files will not transmit MD5 checksums, even when so requested.
9) This request type is not currently supported.
10) The **kXR_getfile** request should be considered as not fully specified. It is currently a place-holder for future enhancement and may substantially change in functionality.

3.10 kXR_locate Request

Purpose: Locate a file.

Request	Normal Response
kXR_char *streamid*[2]	**kXR_char** *streamid*[2]
kXR_unt16 kXR_locate	**kXR_unt16 0**
kXR_unt16 *options*	**kXR_int32** *resplen*
kXR_char *reserved*[14]	**kXR_char** *info*[*resplen*]
kXR_int32 *plen*	
kXR_char *path*[*plen*]	

Where:

streamid
is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

options
are the options to apply when *path* is opened. The *options* are an "or'd" combination of the following values:

kXR_addpeers	- add eligible peers to the location output
kXR_nowait	- provide information as soon as possible
kXR_prefname	- hostname response is prefered
kXR_refresh	- update cached information on the file's location (see notes)

.
reserved
is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*.

path is the path of the file to be located. Opaque information appended to the path does not affect the request. Path may also start with an asterisk or be only an asterisk with the following meaning:

*****	- return all connected managers and servers
****path***	- return all managers and servers exporting *path*

resplen
is the byte length of the response that follows

info are zero or more node types, IPV6 hybrid addresses, and port numbers of nodes that have the file. The port number is to be used to contact the node.

Node Entry Response Format

*xy***[::***aaa.bbb.ccc.ddd.eee***]:***ppppp*
*xyhostname***:***ppppp*

Where:

x is a single character that identifies the type of node whose IP address follows. Valid characters are:
- **M** - Manager node where the file is online
- **m** - Manager node where the file is pending to be online.
- **S** - Server node where the file is online
- **s** - Server node where the file is pending to be online.

y is a single character that identifies the file access mode at the node whose IP address follows. Valid characters are:
- **r** - Read access allowed
- **w** - Read and write access allowed.

aaa.bbb.ccc.ddd.eee
is the IPv4 portion of the IPV6 node address, for IPV4 environments. Otherwise, a true IPV6 address is returned.

hostname
is the hostname for the node address. This format may only be returned when kXR_prefname is specified, but does not forbid an address reply.

ppppp is the port number to be used for contacting the node.

Notes
1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Option flags are the same as those defined for the **kXR_open** request.
3) The **kXR_refresh** voids the **kXR_nowait** option.

4) If the file resides in more than one location, each location is separated by a space.
5) The **kXR_nowait** option provides a location as soon as one becomes known. This means that not all locations are necessarily returned. If the file does not exist, a wait is still imposed.
6) If available, use the inet_ntop() and inet_pton() function to convert addresses to suitable format as these accepts traditional IPV4 address as well as IPV6 addresses.
7) Nodes identified as **M** or **m**, do not actually hold the file. These are manager nodes that know other locations for the file. To obtain the real file location, the client must contact each **M**(**m**) node and issue a **kXR_locate** request. The processes is iterative, as the response from an **M**(**m**) node may identified other **M**(**m**) nodes.
8) Clients should guard against circular references by setting an absolute depth limit in the number of **M**(**m**) to **M**(**m**) references they will accept before declaring an error. A limit of 4 covers a range of 16,777,216 possible locations.

3.11 kXR_login Request

Purpose: Initialize a server connection.

Request		**Normal Response** (server < 2.4.0 | client < 1.0)	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_login**	**kXR_unt16**	**0**
kXR_int32	*pid*	**kXR_int32**	*slen*
kXR_char	*username*[8]	**kXR_char**	*sec*[*slen*]
kXR_char	*reserved*		
kXR_char	*ability*	**Normal Response** (server >= 2.4.0 & client > 0.0)	
kXR_char	*capver*[1]	**kXR_char**	*streamid*[2]
kXR_char	*role*[1]	**kXR_unt16**	**0**
kXR_int32	*tlen*	**kXR_int32**	*slen*+16
kXR_char	*token*[*tlen*]	**kXR_char**	*sessid*[*16*]
		kXR_char	*sec*[*slen*]

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

pid
: is the process number associated with this connection.

username
: is the unauthenticated name of the user to be associated with the connection on which the login is sent.

ability are the client's extended capabilities represented as bit flags, as follows:

0b00000001	the client accepts full standard URL's in a redirection response. Unless the following ability is set, the protocol in the URL should remain xroot. This bit is also identified as **kXR_fullurl**.
0b00000011	the client accepts protocol changes in a full standard URL's in a redirection response. Unless the this ability is set, the protocol in the URL should remain xroot. This bit is also identified as **kXR_multipr**.
0b00000100	the client accepts protocol redirects during a **kXR_read** and **kXR_readv** requests.This bit is also identified as **kXR_readrdok**.

capver

is the client's capabilities combined with the binary protocol version number of the client. The capabilities reside in the top-most two bits while the protocol version number is encoded in the lower 6 bits. Currently, for capabilities two values are possible:

0b00*vvvvvv* - client only supports synchronous responses
0b10*vvvvvv* - (**kXR_asyncap**) client supports asynchronous responses

role is the role being assumed for this login. Valid roles are:
`kXR_useradmin` `0x01` - login as an administrator
`kXR_useruser` `0x00` - login as a regular user (the default)

tlen is the binary length of the supplied token, *token*. If no *token* is present, *tlen* is zero.

token is the token supplied by the previous redirection response that has initiated this login request plus other optional elements.

slen is the binary length of the information, *sec*, that follows *slen*.

sessid is the opaque session identifier associated with this login. The *sessid* is always present when the server protocol version if greater than or equal to 2.4.0 and the client protocol version if greater than 0.

sec is the null-terminated security information. The information should be treated as opaque and is meant to be used as input to the security protocol creation routine **XrdSecGetProtocol().**

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) If no security information is returned (i.e., slen is zero), the **XRootd** server does not require that the client authenticate.
3) If security information is returned, then the client must create the security context allowed by the security information, obtain credentials, and send them using an **kXR_auth** request.
4) Authentication must occur prior to any operation that requires authentication. See the table on page 10 for a list of requests that must be authenticated.
5) Logging in as an administrator suppresses any redirection attempts and limits the request set to **kXR_auth** and **kXR_admin**.
6) A subsequent **kXR_auth** request may revert the login into a normal user login should **xrootd** find that the authenticated user cannot assume the role of administrator.
7) Logging in as a normal user prohibits the use of the **kXR_admin** request.
8) Sending a **kXR_logi**n request on a previously authenticated connection destroys the authentication context; requiring that the connection be re-authenticated.
9) The sessid is used in **kXR_bind** and **kXR_endsess** requests,
10) Opaque information must be treated as truly opaque. The client must not inspect nor modify opaque information in any way.

3.11.1.1 Additional Login CGI Tokens

The following table lists additional cgi tokens that may be passed to further identify the client.

Token	Token Value
`xrd.cc`	the two character country code of the client's location
`xrd.if`	the client's interface speed in gigabits *gggg*[**.***mm*]
`xrd.ll`	the comma separated latitude and longtitude of the client in degree [**-**]*DDD*[**.***dddddd*] format
`xrd.tz`	signed timezone relative to UDT of client's location

3.12 kXR_mkdir Request

Purpose: Create a directory.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_mkdir**	**kXR_unt16**	**0**
kXR_char	*options*	**kXR_int32**	**0**
kXR_char	*reserved*[13]		
kXR_unt16	*mode*		
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
is an area reserved for future use and must be initialized to null characters (i.e., '\0').

options
are the options to apply when *path* is created. The *options* are an "or'd" combination of the following values:
kXR_mkpath - create directory path if it does not already exist

mode is the access mode to be set for *path*. The access mode is an "or'd" combination of the following values:

Access	Readable	Writeable	Searchable
Owner	kXR_ur	kXR_uw	kXR_ux
Group	kXR_gr	kXR_gw	kXR_gx
Other	kXR_or	*not supported*	kXR_ox

plen is the binary length of the supplied path, *path*.

path is the path of the of the directory to be created.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) When a directory path is created, as requested by the **kXR_mkpath** option, the directory permission specified in *mode* are propagated along the newly created path.
3) No **umask** applies to the specified mode.

3.13 kXR_mv Request

Purpose: Rename a directory or file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_mv**	**kXR_unt16**	**0**
kXR_char	*reserved*[14]	**kXR_int32**	**0**
kXR_int16	*arg1len*		
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

arg1len
: the length of the first component in paths. If *arg1len* is zero, then *paths* is scanned for spaces to delimit the components. See the notes for more information.

plen is the binary length of the supplied old and new paths, *paths*.

paths is the old name of the path (i.e., the path to be renamed) followed by a space and then the name that the path is to have.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Renames across file systems are not supported.

3) Protocol verson 3.1.0 introduced *arg1len* in order to specify the actual length of he first component to allow paths to have embedded spaces. When *arg1len* is non-zero then the *paths*+*arg1len* must point to a space character. All characters before *paths*+*arg1len* are used as the old name and all characters after *paths*+*arg1len*+1 is taken as the new name.
4) When *arg1len* is zero (pre-3.1.0 behaviour), then *paths* is scanned for the first space character and this becomes the breakpoint between the old name and the new name.

3.14 kXR_open Request

Purpose: Open a file or a communications path.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_open**	**kXR_unt16**	**0**
kXR_unt16	*mode*	**kXR_int32**	*resplen*
kXR_unt16	*options*	**kXR_char**	*fhandle*[4]
kXR_char	*reserved*[12]	[**kXR_int32**	*cpsize*]
kXR_int32	*plen*	[**kXR_char**	*cptype*[4]]
kXR_char	*path*[*plen*]	[**kXR_char**	*info*[*resplen*-12]]

Where:

streamid
is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

mode is the advisory mode in which *path* is to be opened. The *mode* is an "or'd" combination of the following values:

Access	Readable	Writeable	Executable
Owner	kXR_ur	kXR_uw	kXR_ux
Group	kXR_gr	kXR_gw	kXR_gx
Other	kXR_or	*not supported*	kXR_ox

options
are the options to apply when *path* is opened. The *options* are an "or'd" combination of the following values:

kXR_async - open the file for asynchronous i/o (see notes)
kXR_compress - open a file even when compressed (see notes)
kXR_delete - open a new file, deleting any existing file
kXR_force - ignore file usage rules
kXR_mkpath - create directory path if it does not already exist
kXR_new - open a new file only if it does not already exist
kXR_nowait - open the file only if it does not cause a wait
kXR_open_apnd - open only for appending
kXR_open_read - open only for reading
kXR_open_updt - open for reading and writing
kXR_posc - enable **P**ersist **O**n **S**uccessful **C**lose (**POSC**) processing

kXR_refresh	- update cached information on the file's location (see notes)
kXR_replica	- the file is being opened for replica creation
kXR_retstat	- return file status information in the response
kXR_seqio	- file will be read or written sequentially (see notes)

.

reserved
is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*.

path is the path of the file to be opened. The path can be suffixed with additional information necessary to properly process the request. See the following section on opaque information for more information.

resplen
is the byte length of the response that follows. At least four bytes will be returned.

fhandle
is the file handle for the associated file. The file handle should be treated as opaque data. It must be used for subsequent **kXR_close**, **kXK_read**, **kXR_sync**, and **kXR_write** requests.

cpsize is the compression page size. The *cpsize* field is returned when the **kXR_compress** or **kXR_retstat** have been specified. Subsequent reads must be equal to this value and read offsets must be an integral multiple of this value. If *cpsize* is zero, the file is not compressed and subsequent reads may use any offset and read length.

cptype is the compression algorithm used to compress the file. The *cptype* field is returned when the **kXR_compress** or **kXR_retstat** have been specified. If the file is not compressed, the first byte of the four byte field is a null byte (\0). For compressed files, subsequent reads must use this algorithm to decompress the data.

info is the same information that **kXR_stat** returns for the file. This information is returned only if **kXR_retstat** is set and the server is at protocol version 2.4.0 or greater. The *cpsize* and *cptype* fields are always returned and are only meaningful if **kXR_compress** has been specified. Otherwise, *cpsize* and *cptype* are set to values indicating that the file is not compressed.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Open fails if the path designates a directory.
3) No **umask** applies to the specified mode.
4) The **kXR_async** option tells the server to overlap file i/o with network requests as much as possible for this file. For instance, read requests may be done in parallel with other read requests sent on the same link. This option is only useful if the client is able to issue multiple requests (i.e., is not serializing the requests-response stream).
5) While the **kXR_async** option applies to write operations, as well. Server-side asynchronous opportunities are far more limited. The client needs to perform appropriate multiplexing of write requests with other requests to gain improved parallelism.
6) The **kXR_async** option imposes additional overhead on the server and should only be specified when the client can take advantage of request-response parallelism.
7) The **kXR_refresh** option imposes additional overhead on the server because it requires that the server obtain the most current information on the file's location before attempting to process the open request. This option should only be used as part of the error recovery process outlined in section "Client Recovery From File Location Failures".
8) The **kXR_refresh** option is ignored by any server not functioning as a primary redirecting server.
9) When a directory path is created, as requested by the **kXR_mkpath** option, the directory permission of 0775 (i.e., rwxrwxr-x) are propagated along the newly created path.
10) Only files may be opened using the **kXR_open** request code.
11) The **kXR_retstat** option is meant to eliminate an additional server request for file status information for applications that always need such information.
12) The **kXR_seqio** option is meant to be advisory. A server may choose to optimize data layout or access based on this hint. Misusing the hint may lead to degraded performance.
13) The **kXR_posc** option requests safe file persistence which persists the file only when it has been explicitly closed.

3.14.1 Passing Opaque Information

The **kXR_Open** request allows a client to pass opaque information to properly steer the open. The information may or may not be acted upon, depending on the server's capabilities. Opaque information is passed by suffixing the *path* with a question mark (**?**) and then coding the opaque information as a series of ampersand prefixed (**&**) variable names immediately followed by an equal sign (**=**) prefix value, as shown below:

```
path?&layer.directive=arg[,arg[,···]][&layer.directive=···]
```

Where:

layer

is the layer to which the directive is sent. Valid layer names are:

ofs	the logical file system layer
oss	the physical storage system layer.

directive

is the name of the specific directive

arg

are directive-specific arguments.

Notes

1) Unrecognized layer names or directive names are ignored.
2) Invalid values or arguments to a recognized directive normally result in termination of the request.
3) Refer to the documentation for a specific server extensions to determine the opaque information that can be specified.

Example

&ooss.cgroup=index&oofs.snotify=120,msg,0,imserv,xyzzy

3.15 kXR_ping Request

Purpose: Determine if the server is alive.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_ping**	**kXR_unt16**	**0**
kXR_char	*reserved*[16]	**kXR_int32**	**0**
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Use the **kXR_ping** request to see if the server is running.

3.16 kXR_prepare Request

Purpose: Prepare one or more files for access.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_prepare**	**kXR_unt16**	**0**
kXR_char	*options*	**kXR_int32**	*rlen*
kXR_char	*prty*	**kXR_char**	*resp*[*rlen*]
kXR_unt16	*port*		
kXR_char	*reserved*[12]		
kXR_int32	*plen*		
kXR_char	*plist*[*plen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

options
: are the options to apply to each *path*. The notes explain how these options can be used. The *options* are an "or'd" combination of the following:

kXR_cancel	- cancel a prepare request
kXR_coloc	- co-locate staged files, if at all possible
kXR_fresh	- refresh file access time even when location is known
kXR_noerrs	- do not send notification of preparation errors
kXR_notify	- send a message when the file has been processed
kXR_stage	- stage the file to disk if it is not online
kXR_wmode	- the file will be accessed for modification

prty is the binary priority the request is to have. Specify a value between 0 (the lowest) and 3 (the highest), inclusive.

port is the binary udp port number in network byte order to which a message is to be sent, as controlled by **kXR_notify** and **kXR_noerrs**. If port is zero and **kXR_notify** is set, notifications are sent via asynchronous messages via the connected server, if possible.

reserved
: is an area reserved for future use and must be initialized to null (i.e., '\0').

plen is the binary length of the supplied path list, *plist*.

plist is the list of new-line separated paths that are to be prepared for access. If only one path is supplied, it need not be terminated with a new line character (\n). If **kXR_cancel** is specified, then *plist* must be a prepare *locatorid*.

rlen is the binary length of the response, *resp*, that follows *rlen*.

resp is the response to request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The **kXR_prepare** request attempts to make the indicated files available for access. This may require that the files be brought in from a Mass Storage device.
3) The **kXR_prepare** request always executes asynchronously. Therefore, unless there are obvious errors in the request, a successful status code is immediately returned.
4) The system makes no guarantees that the files will be made available for access ahead of a future **kXR_open** request. Hence, the **kXR_prepare** request is treated as merely a hint.
5) The **kXR_prepare** request should normally be directed to a load-balancing server should one be present.
6) The when the prepare request has been accepted in the presence of the **kXR_stage** option, the server returns a request locator (i.e., *locatorid*) as the normal response. This *locatorid* should be treated as an opaque ASCII text string. The *locatorid* can be used to cancel the request at some future time and to pair up asynchronous messages with requests when **kXR_notify** has been set.
7) **kXR_coloc** is only meaningful in the presence of **kXR_stage** when more than one file has been specified.
8) Co-location of files is not guaranteed. When the **kXR_coloc** and **kXR_stage** options are set, an attempt will be made to co-locate all mentioned files in the request with the first file in the list of files.
9) Co-location may fail for many reasons, including but not limited to, files already present at different locations, files present in multiple locations, and insufficient space. The success if co-locations is implementation defined.

3.17 kXR_protocol Request

Purpose: Obtain the protocol version number, type of server, and possible security requirements.

Request		Normal Response	
kXR_char	*streamid*[2]	kXR_char	*streamid*[2]
kXR_unt16	kXR_protocol	kXR_unt16	0
kXR_int32	*clientpv*	kXR_int32	*dlen*
kXR_char	*options*	kXR_int32	*pval*
kXR_char	*reserved*[11]	kXR_int32	*flags*
kXR_int32	0	*Security Requirements*	
		kXR_char	'S'
		kXR_char	*rsvd*
		kXR_char	*secver*
		kXR_char	*secopt*
		kXR_char	*seclvl*
		kXR_char	*secvsz*
dlen: 8 *or* 14 + *secvsz**2		*Security Overrides*	
		{kXR_char	*reqidx*
		kXR_char	*reqlvl*}[*secvsz*]

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

clientpv
: the binary protocol version that the client is using. See the usage notes on how to obtain the correct value. The *clientpv* field is recognized only in protocol version 2.9.7 and above.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

options
: specifies what should be returned. Without any optios only the pval and glags should be returned. This is also he case if he server does not support support the return option or if no meaningful data exists for the specific request. The options are:

kXR_secreqs return protocol security requirements.

pval is the binary protocol version number the server is using.

flags is additional bit-encoded information about the server. The following flags are returned when *clientpv* is zero (i.e. not specified) or the server's protocol version is 2.9.6 or lower:

kXR_DataServer - This is a data server.
KXR_LBalServer - This is a load-balancing server.

The following flags are returned when *clientpv* is not zero (i.e. is specified) and the server's protocol version is 2.9.7 or above:

kXR_isManager - Has manager role.
kXR_isServer - Has server role.
kXR_attrMeta - Has the meta attribute (e.g. meta manager).
kXR_attrProxy - Has the proxy attribute (e.g. proxy server).
kXR_attrSuper - Has the supervisor attribute.

Security Requirements

If the server supports **kXR_secreqs** and the information is meaningful, at least 6 additional bytes are returned:

rsvd a reserved byte that should be set to zero.

secver the controlling security version. Currently, only version 0 is defined so the byte should be set to zero.

secopt security options:

kXR_secOFrce apply signing requirements even if the authentication protocol does not support generic encryption.

seclvl the default security level to be used. The next section defines each of 5 predefined security levels.

secvsz the number of security override doublets that follow. Security overrides allow a server to customize the predefined security level specified in *seclvl*. If there are no security overrides, this byte should be set to zero.

Security Overrides

A server may customize any predefined security level by returning alterations needed to the specified predefined security level. The informationis contained in a vector of doublets of size *secvsz*:

reqidx the request whose security requirements are to be changed. The request code is specified as a request index. Specifically, it is the **kXR** request code minux **kXR_auth** (the lowest numbered request code). Security requitements are explained in the following section.

reqlvl the security requirement that the associated request is to have:

kXR_signNone	the request need not be signed.
kXR_signLikely	a signing requirement is likely and depends on the request's context. If the request modifies data it should be interpreted as **kXR_signNeeded**. Otherwise, it should be interpreted as **kXR_signNone**.
kXR_signNeeded	the request must be signed.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The client should not rely on the response data length being 8. In the future, additional information may be returned.
3) The protocol version is defined by **kXR_PROTOCOLVERSION** in the header file that defines protocol values and data structures.
4) When the client specifies its protocol version in *clientpv*, the server may use that information to tailor responses to be compatible with the stated version. Since any number of **kXR_protocol** requests can be issued, the authoritative protocol version is considered to be the one in effect after the **kXR_login** request succeeds. After that time, the client's protocol version is immutable until the next login.
5) For **kXR_bind** requests, the client's protocol version is forced to be the same as that the base login stream to which the bind request refers.
6) When testing the bits in *flags* in the protocol response when *clientpv* is specified, the following order should be used:
   a. **`kXR_isManager`** -> role manager
      `kXR_attrMeta` -> role meta manager
      `kXR_attrProxy` -> role proxy manager
      `kXR_attrSuper` -> role supervisor
   b. **`kXR_isServer`** -> role server
      `kXR_attrProxy` -> role proxy server
   c. If none of the above, treat as role manager.
7) The protocol specifies that a client must affiliate with the *first* manager or the *last* meta-manager encountered. Client retry requests should be sent to the affiliated [meta] manager established during the connection phase.
8) Protocol version 2.9.7 provides for a mechanism to determine whether a connection target is a manager or a meta-manager. Clients using lower versions of the protocol do not have that capability and consequently treat managers and meta-managers identically. While this does not cause functional problems, it markedly reduces efficiency when retrying

requests in the presence of multiple meta-managers that control different sets of clusters.

9) Protocol version 3.1.0 introduced a mechanism to verify that requests came from an authenticated client. Pre 3.1.0 servers will never return security information when requested to do so. Servers that have no security requirements need not return any security information when requested to do so. When security information has not been returned the client should assume that no requirements exist.

3.17.1 Protocol Security Requiremenst Response Implications

The **xroot** protocol provides capabilities to verify that a request came from the previously authenticated client. The verification consists of prefixing a request with a **kXR_sigver** request that contains the cryptographic signature of the subsequent request to be verified. The specification of request signature and verification is explained in the **kXR_sigver** section. The **kXR_protocol** request allows a client to determine which requests need to be signed. The table below shows the signing requirements by request for each predefined security level.

Request	Compatible	Standard	Intense	Pedantic
kXR_admin	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_auth	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore
kXR_bind	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded
kXR_chmod	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_close	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded
kXR_decrypt	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore
kXR_dirlist	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_endsess	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded
kXR_getfile	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_locate	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_login	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore
kXR_mkdir	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_mv	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_open	kXR_signLikely	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_ping	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore
kXR_prepare	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_protocol	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore
kXR_putfile	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_query	kXR_signIgnore	kXR_signIgnore	kXR_signLikely	kXR_signNeeded
kXR_read	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_readv	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_rm	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_rmdir	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_set	kXR_signLikely	kXR_signLikely	kXR_signNeeded	kXR_signNeeded
kXR_sigver	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore
kXR_stat	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_statx	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_sync	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_truncate	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_verifyw	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded
kXR_write	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded

A server uses **kXR_protocol** request to specify the security level in effect and any specific overrides. Hence, the protocol provides a framework for, not an absolute definition of, security requirements.

Predefined security levels simplify handling of security requirements. The protocol pre-defines 5 security levels that can be specified in *seclvl*:

kXR_secNone	No security requirements exist.
kXR_secCompatible	A security requirement exists only for potentially destructive requests. (i.e. ones that modify data or metadata).
kXR_secStandard	A security requirement exists for potentially destructive requests. (i.e. ones that modify data or metadata) as well as certain non-destructive requests.
kXR_secIntense	A security requirement exists only for pq wide range of requests that may reveal metadata or modify data.
kXR_secPedantic	Security requirements apply to all requests.

For each request, one of three scenarios exist at each security level:

kXR_signIgnore	The request need not be signed.
kXR_signLikely	The request needs to be signed if it may modify data or metadata.).
kXR_signNeeded	The request must be signed.

The **kXR_signLikely** is the most problematic because it needs to be interpreted the context of what the request is actually doing. Only three requests need to be examined more deeply to determine whether or not they need to be signed.

kXR_open	must be signed if any of the options: **kXR_delete**, **kXR_new**, **kXR_open_updt**, **kXR_mkath**, and **kXR_open_apnd** has been specified.
kXR_query	must be signed if any of the options: **kXR_Qopque**, **kXR_qopaquf**, and **kXR_Qopaqug** have been specified.
kXR_set	must be signed if any request options (i.e. a non-default set operation) have been specified.

3.18 kXR_putfile Request

Purpose: Store a complete file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_putfile**	**kXR_unt16**	**0**
kXR_int32	*options*	**kXR_int32**	**0**
kXR_char	*reserved*[8]		
kXR_int32	*buffsz*		
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		
kXR_int32	*dlen*		
kXR_int64	*offset*		
kXR_char	*data*[*dlen*-8]		
•			
•			
•			
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

options
: is a bit vector representing the options that are to apply to the file transfer. The valid set of options are:
kXR_delete - deleting any existing file
kXR_force - ignore file usage rules
kXR_md5file - Compute and transmit an MD5 checksum for the file.
KXR_md5blok - Compute and transmit an MD5 checksum for each block.
kXR_new - create a new file only if it does not already exist

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

buffsz is the maximum binary length that will be transmitted per file segment (i.e., buffer size). If *buffsz* is zero, 65,544 (i.e., 64K+8) is used.

plen is the binary length of the supplied path, *path*.

path is the path of the file to be stored.

dlen is the binary length of the data that follows with *dlen* never being greater than *buffsz*.

offset is the binary offset of where *data* was located within the file. Negative offsets indicate special non-file data is being transmitted. See the notes for more information.

data is the data associated with the file.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since a file may be much larger than the allowable buffer size, the file is sent in *buffsz* segments until the whole file is sent. Therefore, a {*dlen*, *offset*, *data*} triplet is returned for each entry. When no more data exist, a *dlen* of zero is returned (i.e., there is no subsequent *data*).
3) When a 16-byte MD5 checksum is requested, it is transmitted either after the complete file is transferred or after each block, as specified by the options. An MD5 checksum will have a *dlen* of 24 and an *offset* of negative one (i.e., -1).
4) An empty file is created when the eight-byte triplet {*streamid*, 0, 0} is immediately sent.
5) An MD5 checksum must not be transmitted for an empty file.
6) This request type is not currently supported.
7) The **kXR_putfile** request should be considered as not fully specified. It is currently a place-holder for future enhancement and may substantially change in functionality.

3.19 kXR_query Request

Purpose: Obtain server information.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	*reqcode*	**kXR_int32**	*ilen*
kXR_unt16	*reserved1*[2]	**kXR_char**	*info*[*ilen*]
kXR_char	*fhandle*[4]		
kXR_char	*reserved2*[8]	**Delayed Response**[3]	
kXR_int32	*alen*	**kXR_char**	*streamid*[2]
kXR_char	*args*[*alen*]	**kXR_unt16**	**kXR_waitresp**
		kXR_int32	**4**
		kXR_int32	*seconds*

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request. Only kXR_Qvisa support *fhandle*.

reqcode
: is the binary code indicating the specific query being made. Valid codes are:

kXR_Qconfig	Query server configuration
kXR_Qckscan	Query file checksum cancellation
kXR_Qcksum	Query file checksum
kXR_Qopaque	Query implementation-dependent information
kXR_Qopaquf	Query implementation-dependent information
kXR_Qopaqug	Query implementation-dependent information

[3] A delayed response appears in protocol version 2.5.0 or higher. Earlier protocol versions did not use the delayed response mechanism.

kXR_QPrep	Query prepare status
kXR_Qspace	Query server logical space statistics
kXR_Qstats	Query server statistics
kXR_Qvisa	Query file visa attributes
kXR_Qxattr	Query file extended attributes

alen is the binary length of the supplied arguments, *args*.

args is the arguments to the query, specific to the *reqcode*.

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

seconds
is the binary identifier number of seconds by which a response should be delivered using the unsolicited response mechanism.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Responses to **kXR_Qspace** and **kXR_Qxattr** requests are documented in the Open File System (**ofs**) and Open Storage System (**oss**) configuration reference. Responses to **kXR_Qopaque** and **kXR_Qopaquf** are implementation dependent. This query type should not be used for portable programs.
3) Unstructured data may be passed using the **kXR_Qopaque**. The **kXR_Qopaquf** *reqcode* is meant for structured arguments (i.e., valid path and opaque information).
4) The **kXR_waitresp** response is not an error response but merely indicates that the response may take approximately seconds of time to deliver and will be reported using the unsolicited response mechanism (i.e., **kXR_attn** with **kXR_asynresp**). Refer to the description of each server response for detailed handling information.
5) A delayed response appears in protocol version 2.5.0 or higher. Earlier protocol versions did not use the delayed response mechanism.

3.19.1 KXR_query Checksum Cancellation Request

Purpose: Obtain server information.

Request		**Normal Response**	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	**kXR_Qckscan**	**kXR_int32**	**0**
kXR_char	*reserved*[14]		
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

U8

plen is the binary length of the supplied path, *path*.

path is the path of the file whose check sum is to be cancelled.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Only check sums requested by the current client may be cancelled.

3.19.2 KXR_query Checksum Request

Purpose: Obtain server information.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	**kXR_Qcksum**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*.

path is the path of the file to be stored.

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Each installation determines the type of checksum that will be returned. The algorithmic name of the checksum precedes the checksum value.

Returned Response

The general format for the **kXR_Qcksum** response is:

```
csname csvalue
```

Where:

csname
: is the algorithmic name of the checksum algorithm used. This name is selected by the administrator.

csvalue
: is the checksum name as a hexadecimal ASCII text string. The format is dependent on the algorithm used to compute the checksum

3.19.3 KXR_query Configuration Request

Purpose: Obtain server information.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	**kXR_Qconfig**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*qlen*		
kXR_char	*qry*[*qlen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

qlen is the binary length of the supplied query arguments, *qry*.

qry are the space-separated names of the variables to be returned. Current variables that may be queried are:

bind_max	maximum number of sockets that may be bound to login session.
chksum	checksum algorithm name supported by the server as "*n*:*name*" where *n* is the algorithm numeric id and *name* is it's name. If more than one algorithm is supported, they are listed, each separated by a comma.
cid	the globally unique cluster identification string.
cms	the current dynamic state of the cluster management service configuration. See the next section for the format.
pio_max	maximum number of requests that may be queued on a bound socket before the session stream must wait.
readv_ior_max	maximum amount of data that may be requested in a single **kXR_readv** request element.
readv_iov_max	maximum number of elements in a **kXR_readv** request vector.

role	the configured role. If no role has been configured "none" is returned
sitename	the site name associated with the server. If no sitename is associated, returned value is the token 'sitename'.
tpc	version number for third party copy protocol. If third party copy protocol is not supported, "tpc" is returned. Otherwise, an integer value is returned.
version	version identification string (implementation dependent).
wan_port	the preferred port number to connect to over for wide-area network access.
wan_window	socket buffer size (i.e., window) for the **wan_port** port.
window	socket buffer size (i.e., window) for the default port.

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Clients should avoid issuing configuration query requests to a redirector as this may not reflect the actual limits imposed by a server. Instead, configuration requests should be obtained for each server.

Returned Response

The general format for the **kXR_Qconfig** response consists of a list of new-line delimited value in 1-to-1 correspondence to the list of supplied variable:

```
Cvalue\n[Cvalue\n[. . .\n]]
```

Where:

Cvalue
is the corresponding value associated with the queried variable. If the variable has no value then the name of the variable is returned as *Cvalue*.

3.19.3.1 Format for Query Config cms

The general format for the **kXR_Qconfig cms** response consists of a space delimited list of host for which outbound connections have been configured and their current state. It should be interpreted in the context of the configured role (ie. **kXR_query role**):

```
                        host:port/state

state: c | d | s
```

Where:

host:*port*
: is the hostname or IP address of the endpoint to which a connection is to be made. For server roles the *host*:*port* should always be indicated as localhost:0 to indicate that the clustering service is local.

status is the status of the connection:
- **c** - the endpoint is connected.
- d - the endpoint is disconnected
- s - the endpoint is connected but is a suspended state.

3.19.3.2 Format for Query Config role

The general format for the **kXR_Qconfig role** response consists of a space delimited tokens describing the configured role. The role should be one of the following:

- **meta manager**
- **manager**
- **supervisor**
- **server**
- **proxy manager**
- **proxy supervisor**
- **proxy server**

3.19.4 KXR_query Opaque Request

Purpose: Obtain implementation-dependent server information.

Request		**Normal Response**	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	*querycode*	**kXR_int32**	*ilen*
kXR_char	*reserved1*[2]	**kXR_char**	*info*[*ilen*]
kXR_char	*fhandle*		
kXR_char	*reserved2*[8]		
kXR_int32	*qlen*		
kXR_char	*qry*[*qlen*]		

querycode: **kXR_Qopaque** | **kXR_Qopaquf** | **kXR_Qopaqug**

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

querycode
: is one of the specified query codes. Each code takes different arguments:
kXR_Qopaque - *qry* is passed to the filesystem plug-in without interpretation.
kXR_Qopaquf - *qry* is interpreted as a path and optional cgi string. If the path is valid, *qry* is passed to the filesystem plug-in.
kXR_Qopaqug - *qry* is passed along without inspection to the file plug-in associated with fhandle.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request. Only **kXR_Qopaqug** uses this field**.**

qlen is the binary length of the information, *qry*, that follows *qlen*.

qry is the information to be passed to the appropriate plug-in.

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) These subcodes provide a mechanism to use special implementation specific features. Use of these subcodes is not portable.

3.19.5 KXR_query Space Request

Purpose: Obtain server information.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	**kXR_Qspace**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*slen*		
kXR_char	*sname*[*slen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

slen is the binary length of the supplied path, *sname*.

sname is the logical name of the space whose statistics are to be returned.

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Each installation determines the type of logical spaces that exist and the values that can be returned for them.
6) The response to the **kXR_Qspace** request is documented in the Open File System (**ofs**) and Open Storage System (**oss**) configuration reference.
3) If *sname* is empty, the name "public" is used.

3.19.6 KXR_query Statistics Request

Purpose: Obtain server information.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	**kXR_QStats**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*alen*		
kXR_char	*args*[*alen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

alen is the binary length of the supplied arguments, *args*.

args is an optional list of letters, each indicating the statistical components to be returned. Valid letters are:

a - Return all statistics (default)
b - Buffer usage statistics
d - Device polling statistics
i - Server identification
l - Connection statistics
p - Protocol statistics
s - Scheduling statistics
u - Usage statistics
z – Synchronized statistics

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

2) Statistical information is returned as an XML text string. The XML schema is outlined below.
3) By default, the server returns statistical information that is readily available. The "z" option, informs the server that the information returned must be the accurate in real-time. This requires that the server synchronize activities before gathering information. While this is not this is not a resource intensive activity, it is one that may take considerable amount of elapsed time. The client using "z" option should be ready to wait a significant amount of time for a response.

Returned Response

The general XML schema for the **kXR_Qstats** response is:

```
<statistics tod="time" ver="version">details</statistics>

details: <stats id="sect">details</stats>[details]
```

Where:

time is the Unix **time()** value of when the statistics were generated.

vers is the xrootd version identification string.

setc is the section name assigned to the statistical information. Currently, the following section names should be expected to occur:

id	*arg*	**Information**
buff	**b**	- Buffer usage statistics.
cmsp		- Cluster Management Services
info	**i**	- Server identification.
link	**l**	- Connection (i.e., link) statistics.
ofsp		- Open File System layer
ossp		- Open Storage System layer
poll	**d**	- Device polling statistics.
proc	**u**	- Process usage statistics.
rootd	**p**	- Protocol statistics for rootd.
sched	**s**	- Scheduling statistics.
xrootd	**p**	- Protocol information for xrootd.

Notes

1) Each subsection is bracketed by **<stats>** and **</stats>** tags.

2) Sections appear in a server-defined order. The sections, corresponding to each requested letter code, are returned.
3) The detailed contents of each section beyond what is described here is implementation dependent.

3.19.7 KXR_query Visa Request

Purpose: Obtain server information.

Request		**Normal Response**	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	**kXR_Qvisa**	**kXR_int32**	*ilen*
kXR_char	*reserved1*[2]	**kXR_char**	*info*[*ilen*]
kXR_char	*fhandle*		
kXR_char	*reserved2*[8]		
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request.

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The response to the **kXR_Qvisa** request is documented in the Bandwidth Manager Configuration reference.

3.19.8 KXR_query Xattr Request

Purpose: Obtain server information.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_query**	**kXR_unt16**	**0**
kXR_unt16	**kXR_Qxattr**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*.

path is the path of the file whose extended attributes are to be returned.

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the requested information.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The response to the **kXR_Qxattr** request is documented in the Open File System (**ofs**) and Open Storage System (**oss**) configuration reference.

3.20 kXR_read Request

Purpose: Read data from an open file.

```
Request                                  Normal Response
kXR_char   streamid[2]                   kXR_char   streamid[2]
kXR_unt16  kXR_read                      kXR_unt16  status
kXR_char   fhandle[4]                    kXR_int32  dlen
kXR_int64  offset                        kXR_char   data[dlen]
kXR_int32  rlen
kXR_int32  alen

struct readahead_list
      { kXR_char   fhandle2[4];
        kXR_int32  rlen2;
        kXR_int64  roffset2;
      };

struct read_args
      { kXR_char        pathid;
        kXR_char        reserved[7];
        readahead_list  rvec[(alen-8)/16];
      };
```

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

status is the ending status of this request. Only the following two status codes indicate a normal ending:

kXR_ok - All of the data has been transmitted without error.

kXR_oksofar - Partial data has been transmitted without error; additional data should be expected on this stream.

offset is the binary offset from which the data is to be read.

rlen is the binary *maximum* amount of data that is to be read.

alen is the binary length of the arguments that follow the request header. These arguments may include the *pathid* and read-ahead request list, **struct read_args**. If no data is to be pre-read, *alen* must be set to less than or equal to eight (typically zero).

pathid is the pathid returned by **kXR_bind**. The response data is sent to this path, if possible.

fhandle2
is the file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the pre-read request. Each *fhandle2* is treated separately allowing pre-reads to occur from multiple files.

rlen2 is the binary *maximum* amount of data that is to be pre-read. The *rlen2* should correspond to the intended amount of data that will be read at *offset2* in the near future.

offset2 is the binary offset from which the data is to be pre-read. The *offset2* should correspond to the intended offset of data that will be read in the near future.

dlen is the binary length of the of the data, *data*, that was *actually* read.

data is the data that was read.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) If more data is requested than the file contains, the total of all *dlen*'s will be less than *rlen*.
3) Reading past the end of file with a valid offset will return a *dlen* of 0.
4) The *fhandle* value should be treated as opaque data.
5) Since a read may request more data than the allowable internal buffer size, the data is sent in fixed-sized segments until the request is satisfied. This is accomplished using the **kXR_oksofar** status code. Each subsequent data segment is transmitted using a {*streamid*, *status*, *dlen*, *data*} response. The last segment is indicated by a **kXR_ok**, if no error occurred.
6) Any status code other than **kXR_oksofar** indicates the end of transmission.

7) Sending requests using the same *streamid* when a **kXR_oksofar** status code has been returned may produced unpredictable results. A client must serialize all requests using the *streamid* in the presence of partial results.
8) The **kXR_read** request allows you to also schedule the pre-reading of data that you will ask for in the very near future. Pre-reading data may substantially speed up the execution because data will be available in memory when it is actually asked for. On the other hand, requesting data that you will not need will simply cause a general slow-down of the complete system.
9) The pre-read request is considered only a hint. The system may or may not honor the pre-read request, depending on the current system load.
10) To schedule a pre-read without actually reading any data, issue a **kXR_read** request with *rlen* and *offset* set to zero and **readahead_list** filled out to reflect what data should be pre-read.

3.21 kXR_readv Request

Purpose: Read data from one or more open files.

Request

```
kXR_char   streamid[2]
kXR_unt16  kXR_readv
kXR_char   reserved1[15]
kXR_char   pathid;
kXR_int32  alen

struct read_list
      { kXR_char   fhandle[4];
        kXR_int32  rlen;
        kXR_int64  offset;
      };
```

Normal Response

```
kXR_char   streamid[2]
kXR_unt16  status
kXR_int32  dlen
kXR_char   data[dlen]
```

Where:

streamid
is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

status is the ending status of this request. Only the following two status codes indicate a normal ending:
kXR_ok - All of the data has been transmitted without error.
kXR_oksofar - Partial data has been transmitted without error; additional data should be expected on this stream.

alen is the binary length of the arguments that follow the request header. These arguments may include the *pathid* and read request list, **struct read_args**. The maximum allowed value for *alen* is 8200. This allows up to 512 read segments.

pathid is the pathid returned by **kXR_bind**. The response data is sent to this path, if possible.

fhandle
is the file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the read request. Each *fhandle* is treated separately allowing reads to occur from multiple files.

rlen is the binary *maximum* amount of data that is to be read. Less data will be read if an attempt is made to read past the end of the file.

offset is the binary offset from which the data is to be read..

dlen is the binary length of the of the response data, *data*.

data is the response data. The response data includes **read_list** headers preceding the actual data that was read.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Each **read_list** element represents a read request. All of the read requests are aggregated into one or more responses. Read data is always prefixed by its corresponding **read_list** element. However, the *rlen* value in the element indicates the actual amount of data that was read.
3) If an element requests more data than the file contains, the returned *rlen* will be smaller than the *rlen* in the request element.
4) Reading past the end of file with a valid offset will return a request element whose *rlen* is 0 with no data following.
5) The *fhandle* value should be treated as opaque data.
6) Since a read may request more data than the allowable internal buffer size, the data is sent in fixed-sized segments until the request is satisfied. This is accomplished using the **kXR_oksofar** status code. Each subsequent data segment is transmitted using a {*streamid*, *status*, *dlen*, *data*} response. The last segment is indicated by a **kXR_ok**, if no error occurred.
7) Any status code other than **kXR_oksofar** indicates the end of transmission.
8) Sending requests using the same *streamid* when a **kXR_oksofar** status code has been returned may produced unpredictable results. A client must serialize all requests using the *streamid* in the presence of partial results.
9) The server may return the read elements in any order.

3.22 kXR_rm Request

Purpose: Remove a file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_rm**	**kXR_unt16**	**0**
kXR_char	*reserved*[16]	**kXR_int32**	**0**
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*.

path is the path of the of the file to be removed.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

3.23 kXR_rmdir Request

Purpose: Remove a directory.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_rmdir**	**kXR_unt16**	**0**
kXR_char	*reserved*[16]	**kXR_int32**	**0**
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*.

path is the path of the of the directory to be removed.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The directory must be empty (i.e., no entries other than "." And "..").

3.24 kXR_set Request

Purpose: Set server information.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_set**	**kXR_unt16**	**0**
kXR_char	*reserved*[15]	**kXR_int32**	**n**
kXR_char	*modifier*	**kXR_char**	*resp*[n]
kXR_int32	*dlen*		
kXR_char	*data*[*dlen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

modifier
: set request modifier and should be initialized to zero.

dlen is the binary length of the supplied value, *data*.

data is the value to set.

resp is the response value to the specific set requested.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Set processing takes a command-like string in the *data* field. The following documents valid set arguments.

3.24.1 Valid kXR_Set Values

```
appid apmsg

monitor {off | on} [appid] | info [info]}
```

Where:

appid *apmsg*
includes *apmsg* in the server's log. This request is meant to be used to identify the start and stop if certain application processes for rudimentary monitoring purposes. Up to 80 characters will be recorded.

monitor
control monitor settings with respect to the application.
off - turns off monitoring for the application.
appid - includes up to 12 characters of application text in the monitor record.
on - turns on monitoring, if allowed by the configuration.
appid - includes up to 12 characters of application text in the monitor record.
info - insert information into the monitoring record, if information monitoring is active.
info - is up to 1024 characters of information

Response
is the unique four-character identification value that has been assigned to the *info* value.

Notes
1) Monitoring is enabled using the **xrootd.monitor** configuration directive. When monitoring is not enabled, the monitor set requests are ignored.
2) Use the returned identification value to tag future records in order to correlate related information.

3.25 kXR_sigver Request

Purpose: Provide a signature for the next request.

Request		Normal Response
kXR_char	*streamid*[2]	***None*** (see notes)
kXR_unt16	**kXR_sigver**	
kXR_unt16	*expectrid*	
kXR_char	*version*	
kXR_char	*flags*	
kXR_unt64	*seqno*	
kXR_char	*crypto*	
kXR_char	*reserved*[3]	
kXR_int32	*dlen*	

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request. The *streamid* should be identical to the *streamid* in the subsequent request.

expectrid
: is the *requestid* of the subsequent request.

version
: is version of the signature protocol being used. Currently, there is only one version so *version* should be set to zero.

flags request indicators:
: **kXR_nodata** the data payload is not included in the hash.

seqno is a monotonically increasing sequence number. Each **kXR_sigver** request should have a sequence number that is greater than a previous sequence number used on a particular TCP connection (i.e. login session).

crypto the cryptography used to construct te signature:
: **kXR_rsaKey** the rsa key encrypts the hash.
: **kXR_SHA256** The hash used is SHA-2.

reserved
is an area reserved for future use and must be initialized to null characters (i.e., '\0').

dlen is the length of the subsequent signature. This is normally an encrypted hash of the subsequent request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Only an error reponse is possible to a **kXR_sigver** request. This request simply supplies information used to verify that an authenticated client sent the subsequent request.
3) Only SHA-256 hashes and session key encryption are currently defined as supported.
4) Encryption should be done using the encryption facilities of the authentication protocol used.

3.25.1 Signing a request

When the response to **kXR_protocol** indicates that a particular request needs to be signed, the client should construct a **kXR_sigver** request and send it immediately ahead of the request that needs to be signed. The server should reject any request that should be signed but has been not signed or incorrectly signed.

A signature consists of a **SHA-256** hash of

- an unsigned 64-bit sequence number,
- the request header, and
- the request payload[4];

in that *exact* order. The *crypto* member of **kXR_sigver** should indicate **kXR_SHA256** and the **kXR_rsakey** should not be set.

The client should add one to the sequence number previously used in a **kXR_sigver** request on a particular TCP connection (i.e. login session) before using it in the current rquest. Sequence numbers should be monotonically increasing on each TCP connection.

Once the hash has been computed, it should be encrypted using the session key developed by the authentication protocol used during the login authentication process. If the authentication protocol does not support generic encryption, an unecrypted hash should be used if the server set the **kXR_secOFrce** option in the **kXR_protocol** response. If the option is not set in this situation the client should not sign the request.

The **kXR_sigver** request should be sent immediately before the request that it signed.

[4] When signing **a kXR_write** or **kXR_verifyw** request, the data payload should not be included in the hash and the **kXR_nodata** option should be set in **the kXR_sigver** option field.

3.25.2 Verifying a signed request

When the server receives a **kXR_sigver** request, it should assume that it is the signature for the following request. Note that **kXR_sigver** requests cannot be signed so a hold depth of one request is sufficient. Once the next request is received, the fllowing steps should be followed to verify its signature where sigver refers to the **kXR_sigver** request and its payload and thereq refers to the request whose signature is being verified. If a failure occurs at any step, the request should be rejected with an error sent to the client and the TCP connection can be closed.

- Verify that sigver.*seqno* is greater than the previous sigver.*seqno* the client used on the TCP connection. The saved value should only be udated if the signature is successfully verified.
- Verify that sigver.*streamid* equals thereq.s*treamid*.
- Verify that sigver.*expectrid* equals thereq.*requestid*.
- Verify that sigver.*version* matches the version being used.
- Verify that sigver.*crypto* specifies SHA-256.
- Verify that sigver.*crypto* does not specify the use of the rsa key.
- If the hash was encrypted (i.e. the authentication protocol supports genric encryption) successfully decrypt the hash using the session key via the authentication protocol used.
- Generate a new hash using the sigver.*seqno*, thereq, and the thereq payload[5] in that exact order.
- Verify that the new hash matches the decrypted hash.
- Upon success update the sequence number used so that the sequence number cannot be reused.

[5] When verifying **a kXR_write** or **kXR_verifyw** request, the data payload should not be included in the hash (i.e. the **kXR_nodata** option should be set in **the kXR_sigver** option field).

3.26 kXR_stat Request

Purpose: Obtain status information for a path.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_stat**	**kXR_unt16**	**0**
kXR_char	*opts*	**kXR_int32**	*ilen*
kXR_char	*reserved*[11]	**kXR_char**	*info*[*ilen*]
kXR_char	*fhandle*[4]		
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Default info: *id size flags modtime*\0

kXR_vfs *info*: *nrw frw urw nstg fstg ustg*\0

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

opts
: are stat processing options:
kXR_vfs - return virtual file system information.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path, *path*. If *plen* is zero then *fhandle* must hold the file handle value supplied by the successful response to the associated **kXR_open** request. The *fhandle* argument is ignored when a path is present.

path is the path whose status information is to be returned.

ilen is the binary length of the information, *info*, that follows *ilen*.

Default Response

info is the information about the requested path.

id is the OS-dependent identifier assigned to this entry. Uniqueness is not guaranteed. The *id* is numeric and convertible to a 64-bit value.

size is the decimal size of the data associated with the path whose information is being returned. The size may represent a number up to 2^{64}-1 (i.e., a long long).

flags identifies the entry's attributes as a decimal encoded 32-bit string. The entry should be assumed to be a regular file unless one or more of the following bits are set.

`kXR_xset`	- Either an executable file or a searchable directory.
`kXR_isDir`	- This is a directory.
`kXR_other`	- This neither a file nor a directory.
`kXR_offline`	- For files, the file is not online (i.e., on disk).
`kXR_poscpend`	- The file was created with **kXR_posc** and has not yet been successfully closed.
`kXR_readable`	- Read access allowed.
`kXR_writable`	- Write access allowed.

modtime
is the last modification time in Unix time units (i.e., seconds since 00:00:00 UTC, January 1, 1970).

Response for kXR_vfs

info the location information about the requested path.

nrw the number of nodes that can provide read/write space.

frw the size, in megabytes, of the largest contiguous area of r/w free space.

urw percent utilization of the partition represented by *frw*.

nstg the number of nodes that can provide staging space.

fstg the size, in megabytes, of the largest contiguous area of staging free space.

ustg percent utilization of the partition represented by *fstg*.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The returned string is compatible to the format returned by the root method Tsystem::GetPathInfo().
3) **kXR_stat** requests, without the **kXR_vfs** option, directed to a redirector (i.e., load balancer) referring to a non-file object may result in a non-deterministic response. That is, the response may indicate that the object does not exist when, in fact, it does exist but is not a file. Future versions may resolve the differences between redirectors and file servers.
4) **kXR_stat - kXR_vfs** requests need not specify an existing filesystem object. The specified path is used as a path prefix in order to filter out servers and partitions that could not be used to hold objects whose path starts with the specified path prefix.
5) **kXR_stat - kXR_vfs** requests directed to a redirector return the space values based on current conditions and should be treated as approximations. When the request is directed to an actual server, the server's actual space information is returned.

3.27 kXR_statx Request

Purpose: Obtain type information for one or more paths.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_statx**	**kXR_unt16**	**0**
kXR_char	*reserved*[16]	**kXR_int32**	*ilen*
kXR_int32	*plen*	**kXR_char**	*info*[*ilen*]
kXR_char	*paths*[*plen*]		

Where:

streamid
is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
is an area reserved for future use and must be initialized to null characters (i.e., '\0').

plen is the binary length of the supplied path list, *paths*.

paths is the new-line separated path list whose availability information is to be returned. If a single path is supplied, it need not end with a new line character (**\n**).

ilen is the binary length of the information, *info*, that follows *ilen*.

info is the information about the requested path consisting of a single binary character flag for each path in paths.

flags identifies the entry's attributes as a binary character. The entry should be assumed to be an immediately available regular file unless one or more of the following bits are set.

kXR_xset - Either an executable file or a searchable directory.
kXR_isDir - This is a directory.
kXR_other - This neither a file nor a directory, or does not exist.
kXR_offline - For files, the file is not online (i.e., on disk).

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) **kXR_statx** requests directed to a redirector (i.e., load balancer) referring to a non-file object may result in a non-deterministic response. That is, the response may indicate that the object does not exist when, in fact, it does exist but is not a file. Future versions may resolve the differences between redirectors and file servers.

3.28 kXR_sync Request

Purpose: Commit all pending writes to an open file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_sync**	**kXR_unt16**	**0**
kXR_char	*fhandle*[4]	**kXR_int32**	**0**
kXR_char	*reserved*[12]		
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

3.29 kXR_truncate Request

Purpose: Truncate a file to a particular size.

Request (Opened File)

```
kXR_char   streamid[2]
kXR_unt16  kXR_truncate
kXR_char   fhandle[4]
kXR_int64  size;
kXR_char   reserved[4]
kXR_int32  0
```

Request (Closed File)

```
kXR_char   streamid[2]
kXR_unt16  kXR_truncate
kXR_char   reserved1[4]
kXR_int64  size;
kXR_char   reserved2[4]
kXR_int32  plen
kXR_char   path[plen]
```

Normal Response

```
kXR_char   streamid[2]
kXR_unt16  0
kXR_int32  0
```

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request. The *fhandle* argument is ignored when a path is present.

size is the binary size that the file is to have.

plen is the binary length of the supplied path, *path*.

path is the path of the of the file to be truncated.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

3.30 kXR_unbind Request

Purpose: Unbind a socket from a pre-existing session.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_unbind**	**kXR_unt16**	**0**
kXR_char	*pathid*	**kXR_int32**	**0**
kXR_char	*sessid*[15]		
kXR_int32	**0**		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

pathid is the socket identifier associated with the bound socket. This is the value returned by the **kXR_bind** request.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *sessid* value should be treated as opaque data.

3.31 kXR_write Request

Purpose: Write data to an open file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_write**	**kXR_unt16**	**0**
kXR_char	*fhandle*[4]	**kXR_int32**	**0**
kXR_int64	*offset*		
kXR_char	*pathid*		
kXR_char	*reserved*[3]		
kXR_int32	*dlen*		
kXR_char	*data*[*dlen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request.

offset is the binary offset to which the data is to be written.

pathid is the *pathid* returned by **kXR_bind**. The actual data is read from this path.

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

dlen is the binary length of the of the data, *data*, to be written.

data is the data to be written.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

3.32 kXR_verifyw Request

Purpose: Write data to an open file with checksum validation.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_verifyw**	**kXR_unt16**	**0**
kXR_char	*fhandle*[4]	**kXR_int32**	**0**
kXR_int64	*offset*		
kXR_char	*pathid*		
kXR_char	*vertype*		
kXR_char	*reserved*[2]		
kXR_int32	*dlen*		
kXR_char	*data*[*dlen*]		

Where:

streamid
: is the binary identifier that is associated with this request stream. This identifier will be echoed along with any response to the request.

fhandle
: is the file handle value supplied by the successful response to the associated **kXR_open** request.

offset is the binary offset to which the data is to be written.

pathid is the *pathid* returned by **kXR_bind**. The actual data is read from this path.

vertype
: identifies the checksum algorithm used; implying how many bytes of checksum data precede the data to be written; as follows:

Option	Bytes	Algorithm
kXR_nocrc	0	no crc computed
kXR_crc32	4	32-bit crc using the xroot supplied algorithm

reserved
: is an area reserved for future use and must be initialized to null characters (i.e., '\0').

dlen is the binary length of the of the data, *data*, to be written.

data is the data to be written.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.
3) The crc data must immediately precede the data to be written.
4) The length of the crc data plus the length of the data to be written must equal the *dlen* value.
5) Using an unsupported *vertype* causes the **verifyw** request to fail

4 The Security Framework

The **xrootd** protocol provides for a generic security framework in which virtually any security protocol can be used. The **xrootd** protocol neither specifies nor mandates that any particular security protocol be used. However, should a security protocol be implemented, the **xrootd** protocol specifies how the security protocol's data elements are to be framed and how client-server interactions during the security protocol's execution are to be handled.

The first two protocol steps that a client does after connecting to a server are:

1. sends the 20-byte handshake, followed by a
2. **kXR_login** request.

At this point, the server may require that a security protocol be used to authenticate the client. It is the server that initiates the authentication exchange. The client can never force an authentication to occur. The server mandates that authentication occur by responding to the **kXR_login** request with security information. This is an implicit action on the server's behalf.

Authentication is not required when the data length portion of the response to the **kXR_login** is exactly 16 bytes[6]. When the data length is greater than 16 bytes, the data portion of the response contains what is called a security token. This is essentially a list of the security protocols that the server can use to establish the client's identity along with possible configuration information that is specific to each protocol.

The format of the security token is:

```
ptoken: &P=protid[,protparms][ptoken]
```

Security Token Format

Where:

protid is a 1- to 7-character protocol name. This name is typically used to locate a shared library that implements the security protocol.

protparms
are optional protocol specific configuration parameters that must be supplied to the protocol's initialization routine when it is instantiated. The

[6] Protocol versions less than 2.4 used a different signifier. Refer to the kXR_login request code description for detailed information on deprecated protocol versions.

comma is required is *protparms* are present. The comma is optional otherwise. The *protparms* may not contain an ampersand (&).

In the reference implementation, the *protid* is specified by the **sec.protocol** configuration directive and *protparms* can come from the **sec.protocol** and the **sec.protparm** directives. The plug-in code that handles the protocol is then assumed to reside in **libXrdSec***protid***.so** which is dynamically loaded by **libXrdSec.so** which itself is loaded by the client when a security token has been returned in response to a login. See the "Scalla Security Configuration Reference" for more information.

Generally, the security token is handled by some class that sequences the authentication process. Only that class should be aware of the token's format. This class is responsible for loading one of the listed protocols and initiating the authentication sequence as defined by the security protocol. For consistency among implementations, it is recommended that protocols be considered from left to right and that protocols be successfully tried until one is found to succeed.

For instance, in the **krb5** security protocol, *protparms* defines the service principal whose ticket must be obtained and sent back to the server to prove the client's identity. The security token would appear as

&P=krb5,*srvname*

with *srvname* being the service principal name. If the service ticket must be forwardable, then the token would be sent as

&P=krb5,*srvname***,fwd**

Each protocol specifies its own *protparms* requirements. Refer to the "Scalla Security Configuration Reference" for more information for each available protocol.

The normal sequence in almost any security protocol is that one side generates data sends it to the other side that either accepts or rejects the data and may respond with other data which the receiver may or may not be required to respond to. This is a generalization of multiple exchanges. The xrootd protocol handles such exchanges without interpretation; as follows:

1 After the client-side security manager chooses a protocol and successfully initializes it with the *protparms*; that protocol must return some data that will be sent to the server. The data must be sent to the server as a **kXR_auth** request. This data is known as credentials.

2 The first eight characters of the initial credentials should contain the null terminated protocol identifier of the protocol that generated the credentials. Hence, the actual credential data starts eight bytes into the credential data packet in the first credential packet sent to the server. Subsequent packet layout is defined by the security protocol.

3 When the server receives the initial **kXR_auth** request; it should attempt to use a protocol handler that matches the protocol identifier contained in the credentials (i.e., in the first eight bytes). When the protocol handler is created, the credentials should be passed to its authentication method. If a match cannot be found or initialization fails, a **kXR_error** response must be sent. The connection should remain opened so that the client may try an alternate protocol without performing a new login. For subsequent **kXR_auth** requests, the same protocol used in the successful handling of the initial **kXR_auth** request must be used.

4 After the credentials are processed by the authentication method; three possibilities exist:
   a) The data is accepted and no more exchanges are needed. In this case, the response to the **kXR_auth** request must be a **kXR_ok**. The client is then considered to be fully authenticated.
   b) The data is not accepted and authentication failed. In this case, a **kXR_error** response must be sent with the connection remaining open.
   c) Finally, additional information is needed to complete the authentication. The authentication method supplies the data that data must be sent to the client who is expected to respond with a **kXR_auth** request. The data provided by the authentication method must be used as the body of a **kXR_authmore** response.

5 The client's action for each of the three possible responses is:
   a) Authentication exchanges are stopped when **kXR_ok** is received. The client is considered to be fully authenticated and may issue other requests.
   b) Authentication exchanges are stopped when a **kXR_error** is received. The client may declare a fatal error at this point.
   c) An authentication exchange should occur when **kXR_authmore** is received. The protocol's get credentials method should be called with the data in the **kXR_authmore** response. The method may then supply new data that must be sent to the server in a new **kXR_auth** request or indicate failure. In this processing continues with step 3.

4.1 Framework for Transport Layer Protocols

The **xrootd** security framework and the protocol elements that support it are geared to application layer security. Indeed, implementations are highly discouraged to expose the underlying transport to application code. This is necessary in order to allow multiple transports to be used in a transparent way.

Unfortunately, certain commonly available security implementations are either transport based (e.g., **ssl**) or rely on intimate knowledge of the transport out of expediency.

Currently, the **xrootd** protocol elements neither accommodate nor define direct transport layer interactions between the client and server parts of a security protocol; though nothing prevents implementations of doing so out of band or perhaps hijacking the **xrootd** connection for a limited amount of time. However, the reference implementation makes it impossible to do either.

That said, the reference implementation does provide a virtualized transport via a wrapper class called **XrdSecTLayer**. This class can be used to wrap transport layer security protocols implementations and carry out what appear to be transport layer interactions. The wrapper class virtualizes the interactions by appropriately framing all exchanges within the protocol defined in this document.

4.2 Request Verification

The protocol defines a mechanism to verify that a request came from a previously authenticated client. This is done using lightweight cryptographic signing. Signing requirements are completely controlled by the server that the client communicates with. The server uses the **kXR_protocol** request to inform the client of any signing requirements. The client uses the **kXR_sigver** request to sign a request when needed. When the server receives a **kXR_sigver** request it should use the information in the request to verify that the subsequent request actually came from an authenticated client. A server should reject any request where the signature cannot be verified or a request that needs to be signed was not preceeded by a **kXR_sigver** request. When a request is reject because it was improperly signed, the server should also close the connection to the client after reporting the error to the client.

5 Local Socket Administrative Protocol

Xrootd implementations may provide a local TCP socket for handling administrative functions. This section details the protocol used on this local socket. Refer to the xrootd configuration manual on information how to determine the location of the local socket.

5.1 Initiating an Administrative Session

To successfully initiate an administrative session, you must

1. Connect to an xrootd via its locally defined administrative socket.
2. Issue the login request.

Therefore, a successful login request must precede any other request.

5.2 General Request Format

All requests are transmitted on the local socket consist of new-line (\n) separated ASCII text records. Each request is structured as follows:

```
reqid command [target] [args] \n
```

Request Format

Where:

reqid is the text identifier token that is associated with this request. This token is echoed along with any response to the request. The *reqid* may not be longer than 15 characters.

command
is the command to be executed. The following sections document valid commands.

target is the pattern that identifies the connections to which command applies. Only commands that deal with connections have a *target* requirement. See the following section on the format of *target*.

args are *command* specific arguments.

5.2.1 Request Target Format

```
[user][*][@[pfx][*][sfx]]
```

Where:

user is the name of the user to which the request applies. If *user* ends with an asterisk, the request applies to all users that start with *user*. A single asterisk indicates all users.

pfx is the host name prefix to which the request applies. If nothing follows *pfx*, then the request applies only to host names matching *pfx*. If an asterisk follows *pfx* then the request applies to all host names that start with *pfx*.

sfx is the host name suffix to which the request applies. If nothing precedes *sfx*, then the request applies only to host names matching *sfx*. If an asterisk precedes *sfx* then the request applies to all host names that end with *sfx*.

Notes

1) The target specification allows you to route requests to particular connections. Each connection is identified in a uniform way, described in the following section.
2) In order to route a request to a single connection, the complete connection name must be specified.
3) The target is mandatory even if the request applies to all connections. In this case, specify a single asterisk for target.

5.2.1.1 Connection name format

```
user.pid:fd@host
```

Where:

user is the Unix username of the user.

pid is the user's process number that issued the request.

fd is the server's file descriptor number associated with the connection to ***user*:*pid*** at ***host***.

host is the host name, or IP address, where the user's request originated.

5.3 General Response Format

All responses transmitted on the local socket consist of new-line (\n) separated ASCII text XML records. Each response is structured as follows:

```
<resp id="reqid"><rc>code</rc>[xmltoks]</resp>\n
```
Response Format

Where:

reqid is the text identifier token of the request that is associated with this response.

code is the numeric code indicating success or failure of the request. Zero ("0") always indicates that the request succeeded. A non-zero value indicates that the request failed.

xmltoks
are additional XML response elements that are specific to the request.

5.3.1 Error Response Format

All error responses are structured as follows:

```
<resp id="reqid"><rc>code</rc><msg>msg</msg><resp>\n
```
Error Response Format

Where:

reqid is the text identifier token of the request that is associated with this response.

code is a non-zero numeric code indicating failure.

msg is a message explaining the failure.

5.4 Abort request for kXR_asyncab Client Action

Purpose: Send a **kXR_attn/kXR_asyncab** unsolicited response.

Request Format

reqid **abort** *target* [*msg*]**\n**

Normal Response Format

<resp id="*reqid*"**><rc>0</rc><num>***num***</num></resp>\n**

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be aborted.

num is the number of **kXR_attn/kXR_asyncab** unsolicited responses that were sent.

msg is optional message text to be sent to applicable clients.

Notes

1) When a client receives an unsolicited **kXR_attn/kXR_asyncab** response, it prints *msg* to the console or log, if present, and then immediately terminates execution. Therefore, all server connections are terminated.
2) The server does not close the associated socket until after the client closes the connection.
3) Use other requests to terminate connections to the immediate (i.e., single) server.

5.5 Close request

Purpose: Close client connections.

Request Format

```
reqid close target\n
```

Normal Response Format

```
<resp id="reqid"><rc>0</rc><num>num</num></resp>\n
```

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be closed.

num is the number of connections that were closed.

Notes

1) This request is local to the server and does not generate any unsolicited responses.
2) When the connection is closed, the client attempts to perform standard recovery actions.

5.6 cj request

Purpose: Cancel background job.

Request Format

```
reqid cj job key\n
```

Normal Response Format

```
<resp id="reqid"><rc>0</rc><num>num</num></resp>\n
```

Where:

reqid is the request-response association text identifier token.

job is the type of background job to be cancelled.

key is the key that identifies the particular background job to be cancelled.

num is the number of connections that were closed.

Notes

1) This request is local to the server and does not generate any unsolicited responses.
2) When a background job is cancelled, the client attempts to perform standard recovery actions.
3) Use the **lsj** request to list cancelable jobs.

5.7 Cont request for kXR_asyncgo Client Action

Purpose: Send a **kXR_attn/kXR_asyncgo** unsolicited response.

Request Format

reqid **cont** *target***\n**

Normal Response Format

<resp id="*reqid*"**><rc>0</rc><num>***num***</num></resp>\n**

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be resumed.

num is the number of **kXR_attn/kXR_asyncgo** unsolicited responses that were sent.

Notes

1) When a client receives an unsolicited **kXR_attn/kXR_asyncgo** response, and it is waiting due to a previous **kXR_attn/kXR_asyncwt** unsolicited response from the server; it must terminate the wait and resume normal communications with the server.
2) If the client is not waiting due to a previous **kXR_attn/kXR_asyncwt** unsolicited response, the client ignores the **kXR_attn/kXR_asyncgo** unsolicited response.
3) Use the pause request to send a **kXR_attn/kXR_asyncwt** unsolicited response.

5.8 Disc request for kXR_asyncdi Client Action

Purpose: Send a **kXR_attn/kXR_asyncdi** unsolicited response.

Request Format

reqid **disc** *target wsec msec***\n**

Normal Response Format

<resp id="*reqid*"**><rc>0</rc><num>***num***</num></resp>\n**

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be disconnected.

wsec is the number of seconds the client should wait before attempting to reconnect to the server.

msec is the maximum number of seconds the client should wait before declaring reconnect failure.

num is the number of **kXR_attn/kXR_asyncdi** unsolicited responses that were sent.

Notes

1) When a client receives an unsolicited **kXR_attn/kXR_asyncdi** response, it immediately disconnects from the server (i.e., closes the connection). It then waits *wsec* seconds and attempts to reconnect to the server. If the reconnection fails, it waits another *wsec* seconds and tries again. After *msec* seconds, it declares failure and executes the standard reconnection failure recovery steps.

5.9 Login request (*mandatory*)

Purpose: Create an administrative session.

Request Format

```
reqid login adminid\n
```

Normal Response Format

```
<resp id="reqid"><rc>0</rc><v>vers</v></resp>\n
```

Where:

reqid is the request-response association text identifier token.

adminid
is a string that uniquely identifies the administrative client instance. Typically, this would be of the form: "<*username*>.<*pid*>". Where,

<*username*> is the unix name of the process uid
<*pid*> if the process number as a ASCII string

vers is the protocol version number being used.

Notes

1) The login request must be the first request sent to the server.
2) This request is local to the server and does not generate any unsolicited responses.

5.10 Lsc request

Purpose: List client connections.

Request Format

```
reqid lsc target\n
```

Normal Response Format

```
<resp id="reqid"><rc>0</rc><conn>[conn]</conn></resp>\n
```

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be listed.

conn is a space separated list of connections. If no connections exist that match target, the list is empty. Otherwise, each connection has the form of:

```
user.pid:fd@host
```

Where:

user is the Unix username of the user.
pid is the user's process number.
fd is the server's file descriptor number associated with the connection to *user*:*pid* at *host*.
host is the host name, or IP address, of the user's machine.

5.11 Lsd request

Purpose: List detailed client connections.

Request Format

```
reqid lsd target\n
```

Normal Response Format

```
<resp id="reqid"><rc>0</rc>
     <c r="role" t="ctime" v="vers" m="[mon]">cname
       <io u="inuse"><nf>nfiles</nf>
         <p>pbytes<n>pcnt</n></p>
         <i>ibytes<n>wcnt</n></i>
         <o>obytes<n>rcnt</n></o>
         <s>stalls</s><t>tardies</t>
       </io>
      [ <auth p="prot">
         <n>[name]</n><h>[host]></h>
         <o>[org]</o><r>[role]</r>
       </auth> ]
     </c>
<resp>\n
```

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be listed.

role is the role assumed by the client connect. The '**a**' is used to designate an administrative role and '**u**' as a regular user role.

ctime is the server-local Unix time that existed when the connection was established (i.e., connect time).

vers is the client's version number that was specified during login. Refer to the description of the **login** request for information on how to interpret the version number.

mon the monitoring status of this connection. The letter '**f**' is used to indicate file-level monitoring and '**i**' as I/O-level monitoring. If no monitoring is in effect, the string is empty.

cname is the name of the connection as "*user*.*pid*:*fd*@*host*" (see **lsc**).

inuse the number of references to this connection. This is a close approximation of the number of concurrent requests that are active.

nfiles number of files the connection has open.

pbytes the number of bytes pre-read from files, by client request.

pcnt the number of pre-read requests.

ibytes the number of bytes read from the connection. This is an approximation of the number of bytes written to all files.

wcnt the number of write requests.

obytes the number of bytes written to the connection. This is an approximation of the number of bytes read from all files.

rcnt the number of read requests.

stalls number of times the connection stopped transmitting data in the middle of a request causing the request to stall.

tardies number of times the connection stopped transmitting data at a request boundary casing the request to be rescheduled.

prot the protocol name used for authentication.

name the client's distinguished name as reported by *prot*. If no name is present, the tag data is null.

host the client's host's name as reported by *prot*. If no host name is present, the tag data is null.

org the client's organization as reported by *prot*. If no organization is present, the tag data is null.

role the client's role name as reported by *prot*. If no role name is present, the tag data is null.

5.12 Lsj request

Purpose: List jobs.

Request Format

```
reqid lsj { * | chksum }\n
```

Normal Response Format

```
<resp id="reqid"><rc>0</rc>
          [<job id="jtype"> jkey<s>stat</s>
                <conn>[conn [. . .]]</conn>
          </job>] [...]]
</resp>\n
```

Where:

reqid is the request-response association text identifier token.

* lists all background jobs.

chksum
lists only check sum jobs.

jtype is the job type. Possible types are:
chksum - job requested via **kXR_query** - **kXR_Qcksum**

jkey the key unique identifying the job. Applicable keys are:
chksum - the logical filename of the file being check summed

stat is the job status. Possible values are:
a - actively executing
d - completed
w - waiting for resources
u - unknown status

conn is a space separated list of client connections that initiated the job. If no connections exist for the job, the list is empty. See the **lsc** request for the definition o f *conn*.

5.13 Msg request for kXR_asyncms Client Action

Purpose: Send a **kXR_attn/kXR_asyncdms** unsolicited response.

Request Format

reqid **msg** *target* [*msg*]**\n**

Normal Response Format

<resp id="*reqid*"**><rc>0</rc><num>***num***</num></resp>\n**

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be sent messages.

msg is optional message text to be sent to applicable clients.

num is the number of **kXR_attn/kXR_asyncms** unsolicited responses that were sent.

Notes

1) When a client receives an unsolicited **kXR_attn/kXR_asyncms** response, it prints the message to the console and continues normal execution. If the message is null, the client ignores the **kXR_attn/kXR_asyncms** unsolicited response.

5.14 Pause request for kXR_asyncwt Client Action

Purpose: Send a **kXR_attn/kXR_asyncwt** unsolicited response.

Request Format

reqid **pause** *target wsec***\n**

Normal Response Format

<resp id=*"reqid"***><rc>0</rc><num>***num***</num></resp>\n**

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be disconnected.

wsec is the number of seconds the client should wait before resuming communications with the server.

num is the number of **kXR_attn/kXR_asyncwt** unsolicited responses that were sent.

Notes

2) When a client receives an unsolicited **kXR_attn/kXR_asyncwt** response, it immediately pauses communications with the server for *wsec* seconds.
3) The client maintains the connection to the server during the pause interval. Use the **disc** request to tell the client to close the connection during the pause interval.
4) It is up to the client whether or not it pauses all communications or only communications with the immediate server.
5) Use the **cont** request to cancel the effects of a pause request.

5.15 Redirect request for kXR_asyncrd Client Action

Purpose: Send a **kXR_attn/kXR_asyncdrd** unsolicited response.

Request Format

reqid **redirect** *target host*[**?***token*]:*port*[**?***token*]**\n**

Normal Response Format

<resp id="*reqid*"**><rc>0</rc><num>***num***</num></resp>\n**

Where:

reqid is the request-response association text identifier token.

target is the pattern that identifies the connections to be sent redirections.

host is the DNS name or IP address of the target host.

token is the optional token that the client must transmit to the target host upon connection. The token may be specified after *host* or after *port*.

port is the port number that the client must use when connecting to host.

num is the number of **kXR_attn/kXR_asyncms** unsolicited responses that were sent.

Notes

1) When a client receives an unsolicited **kXR_attn/kXR_asyncrd** response, it switches all logical streams with the current host to the indicated host.
2) Switching logical streams is a complicated action. Refer to the description of the **kXR_attn/kXR_asyncrd** unsolicited response for details on the client's actions.

6 Document Change History

1 June 2005

- Add **kXR_bind** and **kXR_endsess** request codes.
- Explain how a *sessid* is returned in response to **kXR_login**.
- Add **kXR_open_apnd** and **kXR_retstat** options to **kXR_open**.

28 July 2005

- Document the administrative interface protocol.

16 Aug 2005

- Document the **lsd** administrative command.

25 Jan 2006

- Document the **cj** administrative command.
- Document the **lsj** administrative command.
- Add **kXR_Cancelled** subtype error code.
- Add **kXR_Qckscan** subtype request to **kXR_query**.

25 Jan 2006

- Document **kXR_readv**.
- Complete documentation of **kXR_bind**.
- Redefine the pre-read structure in **kXR_read** to include a *pathid* argument.
- Add a *pathid* to **kXR_write**.

5 Dec 2006

- Document **kXR_Qconfig** subcode of **kXR_query**.
- Document **kXR_unbind**.
- Explain ramification of not using **kXR_unbind** in the **kXR_bind** description.
- Clarify **kXR_open** request with respect **kXR_compress** and **kXR_retstat**.

25 Jan 2007

- Document **pio_max** variable for **kXR_Qconfig** sub-request of **kXR_query.**

26 Feb 2007

- Change **kXR_prepare** to reflect that the priority is really a char**.**

1 Aug 2007

- Document the **kXR_verifyw** request.
- Document the **kXR_replica**, **kXR_ulterior**, and **kXR_nowait** options.

Change History

26 Sep 2007

- Document **the kXR_locate** request.

15 Nov 2007

- Document the **kXR_nowait** option of the **kXR_locate** request.
- Document the **kXR_vfs** option of the **kXR_stat** request.

13 Mar 2008

- Document the **kXR_qspace** and **kXR_qxattr** options of the **kXR_query** request.

7 Apr 2008

- Document the **kXR_truncate** request.

12 May 2008

- Correct **kXR_Query** documentation w.r.t. the subcode location.
- Document the **kXR_QVisa** variant or **kXR_Query**.

20 Aug 2008

- Correct **kXR_coloc** and **kXR_fresh** options of the **kXR_prepare** request.
- Document the **kXR_Qopaque** and **kXR_Qopaquf** variants of **kXR_Query**.

26 Jan 2009

- Correct description of **lsj** admin command xml output.

8 Apr 2009

- Document **kXR_seqio** option of the **kXR_open** request.
- Add *fhandle* to the **kXR_stat** request to allow getting stat information based on a currently open file.

6 May 2009

- Describe the security framework as related to the protocol.

2 Jun 2009

- Describe the **kXR_posc** open flag and the **kXR_poscpend** stat response flag.

14 Jul 2009

- Alter description of **kXR_query** + **kXR_QStats** to indicate that other than the basic framing of the information, the actual XML package is implementation dependent.

9 Dec 2010

- Document missing field, *credtype*, in the **kXR_auth** request. This field was always there but somehow escaped documentation. Leaving it unset does not change the protocol but also does not allow the client to switch protocols mid-stream.

14 Jul 2011

- Expand description of information that **kXR_protocol** may return when the client optionally specifies its own protocol version number (new extension).

28 Mar 2012

- Correct diagrams and expand on descriptive text for **kXR_query**, **kXR_read**, **kXR_readv**, and **kXR_set**.
- Add missing *pathid* argument to **kXR_readv** request.
- Expand on the text describing responses to **kXR_redirect**.
- Add **tpc** to the list of configuration variables that may be queried.

21 Jun 2012

- Better explain possible error recovery actions.
- Add optional elements that should have been described:
  - zone field in **kXR_login**
  - port field in **kXR_prepare**
  - pathid field in **kXR_readv**

16 Jul 2013

- Minor corrections on the valid operations table.
- Document the **kXR_dstat** option of **kXR_dirlist**.
- Document the implementation dependent **kXR_query** requests and add **kXR_Qopaqug** to the list.
- Add **csname**, **sitename** and **version** to the list of configuration variables that may be queried.
- Describe full URL redirect responses.
- Document the **kXR_fullurl, kXR_multipr**, and **kXR_readrdok** settings in the **kXR_login** request.
- Describe how locate can return a hostname response (i.e. the **kXR_prefname** option in **kXR_locate**).
- Describe the **kXR_locate kXR_addperrs** option.
- Describe optional login tokens.

Change History

20 Nov 2013

- Document the kXR_Qconfig “cms” and “role” options.

3 Apr 2014

- Add better explaination on how to recover from server failures.

15 Oct 2014

- Correct type csname should be chksum in query config.

23 Mar 2015

- Document the **cid** option of **kXR_query** config.

10 Feb 2016

- Correct **kXR_dirlist** layout description.

10 Oct 2016

- Document **kXR_decrypt** request.
- Document **kXR_sigver** request.
- Document security reuirement response to **kXR_protocol**.
- Document extensions to **kXR_mv** to handle names with embedded spaces.

28 May 2018

- Correct layout of the **kXR_protocol** request.