The XRootD Protocol Version 5.2.0

Andrew Hanushevsk

18-February-2026

©2004-2026 by the Board of Trustees of the Leland Stanford, Jr., University
All Rights Reserved
Produced under contract DE-AC02-76-SFO0515 with the Department of Energy
The protocol specification described in this document falls under BSD license terms.
The specification may be used for any purpose whatsoever.
Use of this specification must cite the original source -- xrootd.org.
Binary definitions in header file XProtocol.hh superceed any such definitions in this document.

1 Contents

1 Contents.......3
2 Request/Response Protocol.......7
2.1 Format of Client-Server Initial Handshake.......7
2.2 Data Serialization.......9
2.3 Client Request Format.......11
2.3.1 Valid Client Requests.......13
2.3.2 Valid Client Paths.......14
2.3.3 Client Recovery from Server Failures.......15
2.4 Server Response Format.......17
2.4.1 Valid Server Response Status Codes.......18
2.4.2 kXR_attn Response Format.......19
2.4.2.1 kXR_attn Response for kXR_asyncms Client Action.......20
2.4.2.2 kXR_attn Response for kXR_asynresp Client Action.......21
2.4.3 kXR_authmore Response Format.......23
2.4.4 kXR_error Response Format.......24
2.4.4.1 Error Codes and Recovery Actions.......25
2.4.5 kXR_ok Response Format.......27
2.4.6 kXR_oksofar Response Format.......28
2.4.7 kXR_redirect Response Format.......29
2.4.8 kXR_status Response Format.......33
2.4.8.1 Valid ResponseTypes.......35
2.4.9 kXR_wait Response Format.......36
2.4.10 kXR_waitresp Response Format.......37
2.5 Binary Definitions of Status, Error and Response Subcodes.......39
2.5.1 Response Status Codes.......39
2.5.2 kXR_attn Subcodes.......39
2.5.3 kXR_redirect Subcodes.......39
2.5.4 kXR_status Subcodes and Other Values.......39
2.5.5 Error Codes.......40
3 Transport Layer Security (TLS) Support.......41
3.1 Client-Server interactions to unilaterally use TLS.......42
3.2 Client-Server interactions to use TLS only when required.......42
4 Server Request Format.......43
4.1 kXR_auth Request.......43
4.2 kXR_bind Request.......46
4.2.1 TLS Considerations.......47
4.3 kXR_chkpoint Request.......49
4.3.1 kXR_ckpBegin, kXR_ckpCommit, and kXR_ckpRollback Subcodes.......51
4.3.2 kXR_ckpQuery Subcode.......53
4.3.3 kXR_ckpXeq Subcode.......55
4.4 kXR_chmod Request.......57
4.5 kXR_clone Request.......59
4.6 kXR_close Request.......61
4.7 kXR_dirlist Request.......63
4.8 kXR_endsess Request.......66
4.9 kXR_fattr Request.......67
4.9.1 Layout of *namevec*.......68
4.9.2 Layout of *valuvec*.......69
4.9.3 kXR_fattr Request – Delete Subcode.......71
4.9.4 kXR_fattr Request – Get Subcode.......73

4.9.5 kXR_fattr Request – List Subcode 75
4.9.6 kXR_fattr Request – Set Subcode 77
4.10 kXR_gpfile Request 79
4.11 kXR_locate Request 83
4.12 kXR_login Request 87
4.12.1 Additional Login CGI Tokens 90
4.13 kXR_mkdir Request 91
4.14 kXR_mv Request 93
4.15 kXR_open Request 95
4.15.1 Additional Open CGI Tokens 100
4.16 kXR_ping Request 101
4.17 kXR_pgread Request 103
4.17.1 Error recovery 106
4.17.1.1 Client 106
4.17.1.2 Server 106
4.17.2 Unaligned reads 107
4.17.3 Backward Compatability 108
4.18 kXR_pgwrite Request 109
4.18.1 Error recovery 112
4.18.1.1 Client 112
4.18.1.2 Server 112
4.18.2 Unaligned writes 114
4.18.3 Backward Compatability 114
4.19 kXR_prepare Request 115
4.20 kXR_protocol Request 119
4.20.1 Client's expect setting & Server's TLS Requirement Response 127
4.20.2 Protocol Security Requirements vs Response Implications 129
4.21 kXR_query Request 131
4.21.1 KXR_query Checksum Cancellation Request 135
4.21.2 KXR_query Checksum Request 137
4.21.2.1 Additional Query Checksum CGI Tokens 138
4.21.3 KXR_query Configuration Request 139
4.21.3.1 Format for Query Config cms 142
4.21.3.2 Format for Query Config proxy 142
4.21.3.3 Format for Query Config role 143
4.21.3.4 Format for Query Config xattrs 143
4.21.4 KXR_query Opaque Request 145
4.21.5 KXR_query Space Request 147
4.21.6 KXR_query Statistics Request 149
4.21.7 KXR_query Visa Request 153
4.21.8 KXR_query Xattr Request 155
4.22 kXR_read Request 157
4.23 kXR_readv Request 161
4.24 kXR_rm Request 165
4.25 kXR_rmdir Request 166
4.26 kXR_set Request 167
4.26.1 Valid kXR_set Values 169
4.27 kXR_sigver Request 171
4.27.1 Signing a request 173
4.27.2 Verifying a signed request 174
4.28 kXR_stat Request 175
4.28.1 Additional Stat CGI Tokens 179
4.29 kXR_statx Request 181
4.30 kXR_sync Request 183

4.31 kXR_truncate Request 185
4.32 kXR_write Request 187
4.33 kXR_writev Request 189
5 The Security Framework 191
5.1 Framework for Transport Layer Protocols 195
5.2 Request Verification 196
6 Document Change History 197

2 Request/Response Protocol

2.1 Format of Client-Server Initial Handshake

When a client first connects to the **XRootD** server, it should perform a special handshake. This handshake should 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 initial *streamid*. The *smid* for the initial response is always two null characters (i.e., '\0');

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

pval binary protocol version number.

flag additional bit-encoded information about the server; as follows:
kXR_DataServer - 0x00 00 00 01 This is a data server.
KXR_LBalServer - 0x00 00 00 00 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 should 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 *should* be received by the server at one time. All known TCP implementations should 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.

```
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

[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 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: Network and Host Byte Order Seialization

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

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

parms parameters specific to the *requestid*.

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

data 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 should 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	Value	Login?	Auth?	Redirect?	Arguments
kXR_auth	3000	y	n	n	*authtype, authinfo*
KXR_bind	3024	n	n	n	*sessid*
kXR_chkpoint	3012	y	-	n	*fhandle, length, offset*
kXR_chmod	3002	y	y	yes	*mode, path*
kXR_close	3003	y	-	n	*fhandle*
KXR_dirlist	3004	y	y	y	*path*
KXR_endsess	3023	y	-	n	*sessid*
kXR_fattr	3020	y	y	y	*Arguments vary by subcode*
kXR_gpfile	3005	y	optional	y	*Arguments vary by subcode*
kXR_locate	3027	y	y	y	*path*
kXR_login	3007	n	n	n	*userid, token*
kXR_mkdir	3008	y	y	y	*mode, path*
kXR_mv	3009	y	y	y	*old_name, new_name*
kXR_open	3010	y	y	y	*mode, flags, path*
kXR_pgread	3030	y	-	y	*fhandle, pathid, length, offset*
kXR_pgwrite	3026	y	-	y	*fhandle, pathid, length, offset*
kXR_ping	3011	y	n	n	
kXR_prepare	3021	y	y	n	*paths*
kXR_protocol	3006	n	n	n	
kXR_query	3001	y	y	y	*args*
kXR_read	3013	y	-	y	*fhandle, pathid, length, offset*
kXR_readv	3025	y	-	y	*fhandle, pathid, length, offset*
kXR_rm	3014	y	y	y	*path*
kXR_rmdir	3014	y	y	y	*path*
kXR_set	3018	y	y	y	*info*
kXR_sigver	3029	y	y	n	*signature*
kXR_stat	3017	y	-	n	*fhandle*
kXR_stat	3017	y	y	y	*path*
kXR_statx	3022	y	y	n	*pathlist*
kXR_sync	3016	y	-	n	*fhandle*
kXR_truncate	3028	y	-	n	*fhandle, length*
kXR_truncate	3028	y	-	y	*path, length*
kXR_write	3019	y	-	y	*fhandle, pathid, length, offset, data*
kXR_writev	3031	y	y	n	*fhandle, length, offset*

Table 2: Valid Client Requests

2.3.2 Valid Client Paths

The **XRootD** server accepts only *absolute paths* where a path may be specified. Relative paths should be resolved by the client interface prior to sending them to **XRootD**. This means that the interface should 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.

Any path may be suffixed by CGI information. The format corresponds to that defined in RFC 3875. However, the protocol does not allow URL encoded characters (i.e. %xx). The meaning of any CGI element that is not specified in this document is implementation specific.

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 29) 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 leading format, as follows:

```
kXR_char   streamid[2]
kXR_unt16  status
kXR_int32  xlen
kXR_char   xtend[xlen]
```

Where:

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

status binary status code indicating the request completion state. The next section describes possible status codes.

xlen binary length of the *xtend* portion of the message. If no *xtend* is present, then the value should be zero.

xtend data specific to the *requestid*. Not all responses have associated data. If the response does have data, the length of this field should be present in the *xlen* 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 should be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client should 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 should 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 responses and their arguments.

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 or URL
kXR_status	Depends on request
kXR_wait	Binary number of seconds & 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 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 two bytes of padding required by the standard response format. These two bytes can be ignored for this particular response code.

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

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

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

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

parms 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 should 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 should perform the requested action and indicated by the *actnum* value.

2.4.2.1 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 binary length of the following action code and message.

msg message to be sent to the terminal. The *mlen* value, less four, indicates the length of the message. The ending null byte ('\0') should be 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 should 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 following options should be implemented:
    a. simply write the indicated message to the terminal, or
    b. allow the application to register a callback to capture the message.

2.4.2.2 kXR_attn Response for kXR_asynresp Client Action

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

Where:

plen binary length of the following action code and response.

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

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

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

data 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 should 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 ould be used to either continue or complete that request, based on the *status* value.
4) The *rlen-dlen* should *always* equal a value of 16.

2.4.3 kXR_authmore Response Format

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

Where:

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

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

data 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 should be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client should 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 should 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 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
binary identifier that is associated with this request stream corresponding to a previous request.

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

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

errmsg
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 should 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 should appropriately pair responses with requests using the *streamid* value.

2.4.4.1 Error Codes and Recovery Actions

kXR_Error Status Code in *errnum*	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_AttrNotFound	The requested file attribute does not exist	n/a	n/a
kXR_AuthFailed	Authentication failed	**H**	**H**
kXR_BadPayload	The request arguments were malformed	n/a	n/a
kXR_Cancelled	The operation was cancelled by the system	n/a	n/a
kXR_ChkSumErr	The checksum does not match	n/a	n/a
kXR_Conflict	Request cannot be executed due to a conflict	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_fsReadOnly	The file system is marked read-only.	n/a	n/a
kXR_Impossible	The request cannot be executed due to exigent conditions	n/a	n/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_ItExists	Cannot create new object as it already exists	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	**E**
kXR_NotFile	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_noReplicas	No more replicas exist.	n/a	n/a
kXR_noserver	There are no servers available to process the request	**F**	n/a
kXR_overQuota	Space quota exceeded	n/a	n/a
kXR_overloaded	Server is overloaded	**C**	**D**
kXR_ReqTimedOut	Request could not be completed in time	n/a	**D**
kXR_ServerError	An internal server error has occurred	**C**	**A**
kXR_SigVerErr	Request signature could not be verified	**G**	**G**
kXR_TimerExpired	Special return code used for cache directives	n/a	m/a
kXR_TooManyErrs	Request has excessive errors to continue	n/a	**D**
kXR_TLSRequired	Request requires a TLS connection	n/a	n/a
kXR_Unsupported	The request is valid but not supported	n/a	E

A. Go back to the redirector and ask for a different server**. kXR_refresh** *should not* be turned on. The "tried=" **CGI** 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. The "tried=" **CGI** value should indicate the hostname of the failing server. This should normally be done only once.

E. If the redirector is a meta-manager or is virtual (i.e. actually a metalink) then go back to the redirector and ask for a different server. The "tried=" **CGI** value should indicate the hostname of the failing server. The **kXR_refresh** *should not* be turned on. For **kXR_NotAuthorized**, recovery should be attempted no more than three times.

F. If the redirector is virtual (i.e. actually a metalink), the follow the actions listed under **E**. Real redirectors have a real-time view of all available resources and the inability to allocate a resource indicates that none are useable for a request. Retrying the request is highly likely to be ineffective. Virtual redirectors only have a static view of resources and cannot determine if using another resource will succeed without actually trying to use that resource. Thus, all failures are retryable.

G. Signature verification errors due to transport corruption are retryable as such corruptions are transient. There is no way to determine if a failure is due to corruption or active compromise. The request should be retried once or twice.

H. Authentication failures may be due to server missconfiguration. If another server or redirector is available, the operation may be retried.

2.4.5 kXR_ok Response Format

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

Where:

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

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

data 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 should be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client should appropriately pair responses with requests using the *streamid* value.
3) The **kXR_ok** response indicates that the request fully completed and no additional responses should be forthcoming.

2.4.6 kXR_oksofar Response Format

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

Where:

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

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

data 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 should be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client should 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.
4) Sending requests using the same *streamid* when a **kXR_oksofar** status code has been returned may produced unpredictable results. A client should 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 kXR_redirect Response Format

```
kXR_char   streamid[2]
kXR_unt16  kXR_redirect
kXR_int32  dlen
kXR_int32  port | 0xffffffff | < 0
kXR_char   host[?[fcgi][?lcgi]][dlen-4] | url
```

Where:

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

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

port binary port number to which the client should connect. If the value is zero, the default **XRootD** port number should be used. If the value is negative, then the text after *port* contains a standard **URL** that should 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. See the usage notes when **0xffffffff** should be used as a negative port number.

host ASCII name of the to which the client should 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). See the usage notes describing the format of *host*.

fcgi optional ASCII CGI string that, when present, should be added to the end of any existing CGI information appened to the file name[2] associated with the operation being redirected. The *fcgi*, if present, is separated from the *host* by a single question mark. The *fcgi* does *not* end with a null (\0) byte but may end with a question mark (see *token* below). Therefore, *fcgi* may never contain a question mark. See the usage notes for more information.

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

lcgi optional ASCII CGI string that, when present, should be delivered to the new host during the login phase as additional login parameters. However, if an established connection to the specified host already exists it may be re-used without a login. The *lcgi*, if present, is separated from the *host* by a two question marks. The first question mark may be followed by *fcgi* information. If none is present, another question mark immediately follows the first one. The *lcgi* does *not* end with a null (\0) byte. See the usage notes for more information.

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 should be initialized to binary zero.
2) The *host*, either in its shortened form or in a URL redirect, may either be an FQDN or an IP address. IP addresses should be in IPv6 bracket format or in IPv4 octet format. However, using IP addresses should be avoided as they conflict with NAT devicess common in frwewalled sites, Kubernetes clusters using virtual networks, and addresses are not usually present in a host certificates SAN extension which makes TLS connection impossible.
3) CGI information (i.e. *fcgi* and *lcgi*) is a stream of 7-bit clean (ASCII) characters that encode zero or more assertions. These assertions are separated by the '&' character. Assertions are key-value pairs in the form key = value. Both key and value text must not contain '?', '&' or '=' characters. Hex encoding of characters (i.e. sequences such as % <hex> <hex>, where <hex> is a character in 0123456789abcdef) should not be supported. When parsing a CGI string, any key that is not understood should be ignored. Keys that start **xrd.** are reserved. Implementations are free to include other keys. It is recommended for keys to start with a reverse-DNS name under the developer's control; e.g., (org.example.my-new-key). For historic reasons a server should process a login string (i.e. *lcgi*) that starts with the '?' character by ignoring that character and processing the remaining login string.
4) Since requests may be completed in any order, the ordering of responses is undefined. The client should appropriately pair responses with requests using the *streamid* value.

5) 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.
6) The client should be prepared to handle a redirect response at any time. A redirect response requires that the client
    a. Decompose the response to extract the port number, host name, and possible token value.
    b. 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.
    c. 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.
    d. Perform the initial handshake, login with token (see **kXR_login** description), and authentication (see **kXR_auth** description).
    e. If the redirection occurred for a request using a file handle (i.e., *fhandle*) then a new file handle should be obtained.
        i. **A kXR_open** request should be issued using the same file name and options as was originally used.
        ii. The returned file handle should be used for the request that is to be re-issued as well as all subsequent requests relating o the file.
    f. Re-issue the request that was redirected.
7) Historically, clients tested the *port* for the exact value of **-1** (i.e. **0xffffffff**) to determine whether a redirect **URL** or a shortened host specification was present. This prevented additional information frm being passed in the *port* field. To provide backward compatability, a special **kXR_login** capability was introduced, **kXR_redirflags**, that indicates the client simply checks for a negative value and the low order 31 bits may be used as redirect flags. Servers should always use **-1** unless the client indicates that it is capable of handling any negative value by setting the **kXR_redirflags** capability in the login request.

8) The following redirect flags are defined:

Flag	Meaning
kXR_recoverWrts	Write recovery for copy targets is possible at the server that set this redirect flag.
kXR_collapseRedir	If the redirect target is in the same address group as the redirecting server, make the target the primary address for all future contacts for this address group.

9) Normally, the protocol limits write recovery to the server to which the write was directed. The **kXR_recoverWrts** flag allows write recovery to occur at a different server as long as that server is in the redirect path leading to a server executing a write operation.
10) A **DNS** host name may be assigned multiple addresses, each of which is a different physical endpoint. All servers under that **DNS** name should be considered to belong to the same address group. Any server in that address group may request that one of those servers be designated as the primary target using the **kXR_collapseRedir** flag. This option is meant to support replicated services in which there is a primary leader chosen by consensus.
11) Opaque data should be treated as truly opaque. The client should not inspect nor modify the data in any way.

2.4.8 kXR_status Response Format

```
kXR_char   streamid[2]
kXR_unt16  kXR_status
kXR_int32  resplen (should be >= 16)
kXR_unt32  crc32c
kXR_char   streamid[2]
kXR_char   requestid
kXR_char   resptype
kXR_char   reserved[4]
kXR_int32  dlen
kXR_char   info[resplen-16]
kXR_char   data[dlen]
```

Where:

streamid
binary identifier that is associated with this request stream corresponding to a previous request. It is repeated to allow for a quick integrity check of the *streamid* before doing more extensive checks.

resplen
binary length of the *response* portion of the message (i.e., the subsequent bytes not including any data portion).

crc32c **CRC32-C** as defined by the **IETF RFC 7143** standard (see the notes for details) of the *resplen*-**sizeof(***crc32c***)** bytes immediately after *crc32c*. This means that the *data* portion, if any, should *not* be included in the *cr32c* calculation.

requestid
identifier of the original request. The *requestid*+**kXR_1stRequest** should equal the original request code.

resptype
binary code identifying the response type. See the subsequent section for details.

dlen binary length of the *data* portion of the message, if any. If there is no data portion then *dlen* should be zero.

info optional additional response information whose contents should be interpreted in the context of the *requestid* and *resptype* codes. Refer to each corresponding request to see how to interpret the *info*, if present. The length should be calculated as *resplen*- **kXR_statusBodyLen** and should result in a value >= 0.

data 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 should be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client should appropriately pair responses with requests using the *streamid* value.
3) The *crc32c* should use the **CRC32-C** polynomial specified in the **IETF RFC 7143** standard. This corresponds to the polynomial **0x1edc6f41** or $x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+x^{11}+x^{10}+x^{9}+x^{8}+x^{6}+1$.
4) When **kXR_status** is received the client should perform an integrity check on the response, as follows:
    a. Verify that the two streamid values are identical, and
    b. calculate the **CRC32C** value of the response and verify that it matches the value sent by the server in *crc32c*.
5) When an integrity check fails, the only recourse is to close the connection and start with a new connection. The reason is that there is no way to know how much and what kind of data may be in transit, should any of the length fields be corrupted. Be aware that closing a connection with active requests causes those requests to be terminated.

2.4.8.1 Valid ResponseTypes

The *resptype* codes as defined in **struct ServerResponseStatus** are:

resptype	*datalen*	**Explanation**
kXR_FinalResult	>= 0	Request completed as indicated in the response.
kXR_PartialResult	>= 0	Request has partially completed as indicated.
kXR_ProgressInfo	== 0	Request is ongoing this is a progress report only.

Notes

1) The presence of *info* and *data* is determined by the particular request being performed. Refer to the requests returning **kXR_status** for details.
2) Sending requests using the same *streamid* when a **kXR_status** with a **PatrialResult** or **ProgressInfo** *restype* code has been returned may produce unpredictable results. A client should serialize all requests using the *streamid* until a **FinalResult** *restype* is returned by the request.
3) Currently, only **kXR_gpfile, kXR_pgread** and **kXR_pgwrite** return **kXR_status**. However, clients implementing this version of the protocol should be implemented to handle any request returning **kXR_status**.
4) Requests employing **kXR_status** should never return **kXR_ok** and **kXR_oksofar** as these are essentially subsumed by **kXR_status**. The use of other response types is allowed.
5) When **kXR_PartialResult** or **kXR_ProgressInfo** is received, the client should reset the wait timeout to its original value.

2.4.9 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
binary identifier that is associated with this request stream corresponding to a previous request.

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

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

infomsg
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 should be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client should 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.

2.4.10 kXR_waitresp Response Format

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

Where:

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

seconds
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 should be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client should appropriately pair responses with requests using the *streamid* value.
3) The client should wait the indicated number of seconds for the response. The response should 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 should 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.

2.5 Binary Definitions of Status, Error and Response Subcodes

2.5.1 Response Status Codes

Status Code	**Value**
kXR_ok	0
kXR_oksofar	4000
kXR_attn	4001
kXR_authmore	4002
kXR_error	4003
kXR_redirect	4004
kXR_status	4007
kXR_wait	4005
kXR_waitresp	4006

2.5.2 kXR_attn Subcodes

kXR_attn Subcode	**Value**
kXR_asyncms	5002
kXR_asynresp	5008

2.5.3 kXR_redirect Subcodes

kXR_redirect Subcode	**Value**
kXR_recoverWrts	0x00001000
kXR_collapseRedir	0x00002000

2.5.4 kXR_status Subcodes and Other Values

kXR_status subcode in XrdProto::	**Value**
kXR_FinalResult	0x00
kXR_PartialResult	0x01
kXR_ProgressInfo	0x02

kXR_status value in XrdProto::	**Value**
kXR_statusBodyLen	16

2.5.5 Error Codes

Error	Value	Corresponding POSIX errno Value
kXR_ArgInvalid	3000	EINVAL
kXR_ArgMissing	3001	EINVAL
kXR_ArgTooLong	3002	ENAMETOOLONG
kXR_FileLocked	3003	EDEADLK
kXR_FileNotOpen	3004	EBADF
kXR_FSError	3005	ENODEV
kXR_InvalidRequest	3006	EBADRQC
kXR_IOError	3007	EIO
kXR_NoMemory	3008	ENOMEM
kXR_NoSpace	3009	ENOSPC
kXR_NotAuthorized	3010	EACCES
kXR_NotFound	3011	ENOENT
kXR_ServerError	3012	EFAULT
kXR_Unsupported	3013	ENOTSUP
kXR_noserver	3014	ECONNREFUSED, EHOSTUNREACH, ENETUNREACH
kXR_NotFile	3015	ENOTBLK
kXR_isDirectory	3016	EISDIR
kXR_Cancelled	3017	ECANCELED
kXR_ItExists	3018	EEXIST
kXR_ChkSumErr	3019	EDOM
kXR_inProgress	3020	EINPROGRESS
kXR_overQuota	3021	EDQUOT
kXR_SigVerErr	3022	EILSEQ
kXR_DecryptErr	3023	ERANGE
kXR_Overloaded	3024	EUSERS
kXR_fsReadOnly	3025	EROFS
kXR_BadPayload	3026	EINVAL
kXR_AttrNotFound	3027	ENOATTR
kXR_TLSRequired	3028	EPROTOTYPE
kXR_noReplicas	3029	EADDRNOTAVAIL
kXR_AuthFailed	3030	EAUTH (preferable) or EBADE
kXR_Impossible	3031	EIDRM
kXR_Conflict	3032	ENOTTY
kXR_TooManyErrs	3033	ETOOMANYREFS
kXR_ReqTimedOut	3034	ETIMEDOUT
kXR_TimerExpired	3035	ETIME

3 Transport Layer Security (TLS) Support

The **XRootD** protocol supports **TLS** mode connections in two explcit ways:

1) client request using the **kXR_protocol** request, and
2) server request using **the kXR_protocol** response.

This mechanism provides several features:

- A single port can be used for **TLS** and non-**TLS** connections.
- The request channel can be split from the data channel using the **kXR_bind** request so that control information flows on a **TLS** connection while data flows on a non-**TLS** connection. Such an arrangement may significantly improve performance.
- The number of interactions can be reduced when a connection needs to use **TLS**.
- The server may independently enforce TLS requirements in for broad categories:
    - logins and all subsequent interactions,
    - all post-login interactions,
    - third party copy requests, and
    - data transfers.

Currently, once a connection switches to **TLS** mode it cannot switch back. This is not a protocol requirement but a practical side-effect of current **TLS** implementations that buffer an inditerminant amount of data making it problematic to deterministically switch modes. However, the **XRootD** protocol is sufficiently open to allow such switches if and when the **TLS** protocol can do so in the future.

A server is not required to support **TLS**. If it does, it should follow the protocol specifications described in the **kXR_protocol** and **kXR_bind** requests.

TLS may be considered a replacement for request signing in most circumstances. However, for certain workflows, request signing may offer better performance. Be ware, that **XRootD** request signing, as defined, does not protect data while **TLS**, when used for data, does so.

3.1 Client-Server interactions to unilaterally use TLS

- The client should connect to the server using a non-**TLS** connection and send the handshake packet.
- The client should then send a **kXR_protocol** request indicating that it wants to use **TLS**. For reduced latency, the handshake and the **kXR_protocol** request may be sent together.
- If the server supports **TLS** it should indicate in the **kXR_protocol** response that the connection will be switched to use **TLS** after the response is sent.
- The client should check if the server switched the connection to use **TLS** and do the same if so indicated.
- All communications from then on use **TLS**.

3.2 Client-Server interactions to use TLS only when required

- The client should connect to the server using a non-**TLS** connection and send the handshake packet.
- The client should then send a **kXR_protocol** request indicating that it is able to use **TLS**. For reduced latency, the handshake and the **kXR_protocol** request may be sent together. In the **kXR_protocol** request the client should also indicate the expected next operation (i.e. login, data transfer, or third party copy).
- If the server supports **TLS** it should indicate in the **kXR_protocol** response that the connection has been switched to use **TLS** if the client's subsequent operation requires **TLS**. Note that it is also possible for the server to indicate that **TLS** is required after the **kXR_login** request (i.e. login does not require **TLS**).
- The client should check if the server switched the connection to use **TLS** and do the same if so indicated. If the next reqest is a **kXR_login** and the server indicated that **TLS** is not required until after the login, the client should defer switching the connection to **TLS** until after the login and all authentication interactions (i.e. kXR_auth requests).

4 Server Request Format

4.1 kXR_auth Request

Purpose: Authenticate client's username to the server.

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

Where:

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

reserved
: area reserved for future use and should 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
: binary length of the supplied credentials, *cred*.

cred
: credentials used to provide authentication information.

kXR_auth

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 should 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.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_auth		3000	Perform authenication

4.2 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**	**kXR_ok**
kXR_char	*sessid*[16]	**kXR_int32**	**1**
kXR_int32	**0**	**kXR_char**	*pathid*

Where:

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

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

pathid 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 should be initialized to binary zero.
2) The *sessid* value should be treated as opaque data.
3) The socket issuing the **kXR_bind** request should 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) 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.
6) Bound sockets are meant to support parallel data transfer requests across wide-area networks. They are also meant to split control information from data allowing control to flow on a **TLS** connection while data flows on a non-**TLS** connection. See TLS Considerations for more information.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_bind		3024	Bind additional sockets to session

4.2.1 TLS Considerations

A server may indicate in the response to the **kXR_protocol** request that all data should flow across a **TLS** connection. The **kXR_protocol** request is normally sent by the client immediately after the handshake. If the the server's response indicates that **TLS** should be used for data then the connection to be bound should be set to TLS mode in order for the request to succeed. There are two ways to achieve this.

The client may record whether or not the bound connections should use **TLS**. If bound connections should use **TLS** the **kXR_bind** request should be prefixed by a **kXR_protocol** request indicating that the connection should be switched to **TLS**. To reduce latency, the **kXR_protocol** and **kXR_bind** requests should be sent together. This method is preferred.

Alternatively, the client may always send a **kXR_protocol** request ahead of the **kXR_bind** request indicating in the request that it is able to use **TLS** and the next request will be **kXR_bind**. If the server requires the data to use **TLS** it should respond that the connection will switch to using **TLS** after the **kXR_protocol** response is sent. If the connection was switched to use **TLS** the client should do the same and then send the **kXR_bind** request. Since this involves additional interactions, it is not the preferred method.

The client is also free to switch the connection to use **TLS** whether or not the server requires it to do so.

4.3 kXR_chkpoint Request

The **kXR_chkpoint** request allows a safe server-side modification of a file. When a file is modified, the server logs file modifications so that in the event of a failure the file can be restored to its original contents. The request consists of serveral subcode operations to provide complete control of checkpointing. In all cases the file being acted upon should be open in write mode.

The general sequence is:

- **kXR_chkpoint** with **kXR_ckpBegin** to establish a checkpoint.
- **kXR_chkpoint** with **kXR_ckpXeq** of a **kXR_pgwrite , kXR_trunc**, **kXR_write**, and **kXR_writev** request.
    - One ore more such operations may be executed.
- **kXR_chkpoint** with **kXR_ckpCommit** to commit the changes or **kXR_ckpRollback** to rollback the changes.

Loss of connectivity or a file close before a commit should cause a rollback to occur. Since the resiliency specifications of the **XRootD** protocol make it difficult, if not impossible, for an application to detect when connectivity has been lost and then re-established the **kXR_ckpXeq** subcode provides a safe way to execute modifications within a checkpoint context. Should the starting context become invalid, subsequent modifications should be rejected. This prevents a file from being left in a corrupted state due to partial updates.

An implementation should guarantee, barring media failures, that files can be successfully restored in any operational context (e.g. server failure). Should a checkpoint rollback fail, the file should either be made read/only or made inaccessible.

The total amount of data that may be recorded in a checkpoint should be limited. The protocol specifies that the minimum data limit is **kXR_ckpMinMax** defined as 10,485,760 (10MB). Implementations may allow for larger checkpoints. Note that the limit applies to data exclusive of any overhead in order to provide implementation consistency.

The following sections desctribe the various **kXR_chkpoint** subcodes in detail.

4.3.1 kXR_ckpBegin, kXR_ckpCommit, and kXR_ckpRollback Subcodes

Purpose: Create, delete or restore a checkpoint.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_chkpoint** **kXR_char** *fhandle*[4] **kXR_char** *reserved*[11] **kXR_char** *opcode* **kXR_int32 0**	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32 0**

Where:

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

fhandle
file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the checkpoint request. The file should be opened in write mode.

opcode checkpoint operation wanted:
kXR_ckpBegin - Create a checkpoint.
kXR_ckpCommit - Delete an existing checkpoint to commit changes.
kXR_ckpRollback - Restore file data and delete the checkpoint.

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.
3) The *fhandle* should refere to a file opened for writing. If it does not, the request should fail.
4) The file should be opened in update mode without the **POSC** option. Creation of a checkpoint should fail if this is not the case.
5) Once a checkpoint is establish a new one should not be allowed until the existing checkpoint is committed or rolled back.

6) Should the client loose connectivity to the server, all outstanding checkpoints should be restored.
7) Should the client close a file with an outstanding checkpoint, the checkpoint should be restored.
8) Upon server restart, all outstanding checkpoints should be restored.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_chkpoint		3012	Checkpoint file data.
	opcode		
	kXR_ckpBegin	0x00	Create a new checkpoint.
	kXR_ckpCommit	0x01	Delete the current checkpoint.
	kXR_ckpRollback	0x03	Restore the current checkpoint.

4.3.2 kXR_ckpQuery Subcode

Purpose: Create and execute a checkpointed operation on an open file.

Request	Normal Response
kXR_char *streamid*[2]	**kXR_char** *streamid*[2]
kXR_unt16 kXR_chkpoint	**kXR_unt16 kXR_ok**
kXR_char *fhandle*[4]	**kXR_int32 8**
kXR_char *reserved*[11]	**kXR_int32** *limit*
kXR_char kXR_ckpQuery	**kXR_int32** *used*
kXR_int32 0	

Where:

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

fhandle
: file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the checkpoint request. The file should be opened for write mode.

limit the maximum number of data bytes that may be recorded in a checkpoint.

used the number of data bytes already recorded in the checkpoint.

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.
3) The *fhandle* should refere to a file opened for writing. If it does not, the request should fail.
4) When the client closes a file with an outstanding checkpoint, the checkpoint should be deleted.

kXR_chkpoint

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_chkpoint		3012	Checkpoint file data.
	opcode		
	kXR_ckpQuery	0x02	Query checkpoint limit and usage.

4.3.3 kXR_ckpXeq Subcode

Purpose: Modify a file within a checkpoint context.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_chkpoint** **kXR_char** *reserved*[15] **kXR_char kXR_ckpXeq** **kXR_int32 24** *request*	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32 0**

Where:

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

fhandle
: file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the checkpoint request. The file should be opened in write mode.

request
: a fully formed *fhandle* oriented **kXR_pgwrite**, **kXR_truncate**, **kXR_write**, or **kXR_writev** 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 should be initialized to binary zero.
2) A checkpoint size is limited. The **kXR_ckpQuery** subcode may be used to determine the limit.
3) The *streamid* in *request* should match the *streamid* in the **kXR_chkpoint** request. An error should result if that is not the case.
4) The protocol does not specify fileset checkpoints (i.e. a checkpoint for a related group of files). Consequently, if *request* is a **kXR_writev** the file handles in each date segment should refer to the same file (i.e. have identical file handles). An error should result if this is not the case.

5) Should an error occur during a **kXR_ckpXeq**, the implementation is free to close the client connection to discard any outstanding data. In such an event, a **kXR_ckpRollback** should be implicitly performed. If data is drained and the connection left open, the checkpoint should be left intact, allowing the client to delete or restore the checkpoint.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_chkpoint		3012	Checkpoint file data.
	opcode		
	kXR_ckpXeq	0x04	Execute a request within a checkpoint context.

4.4 kXR_chmod Request

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

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16** **kXR_chmod** **kXR_char** *reserved*[14] **kXR_int16** *mode* **kXR_int32** *plen* **kXR_char** *path*[*plen*]	**kXR_char** *streamid*[2] **kXR_unt16** **kXR_ok** **kXR_int32** **0**

Where:

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

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

mode 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 binary length of the supplied path, *path*.

path path whose mode is to be set. It may be suffixed with CGI 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 should be initialized to binary zero.
2) No **umask** is applied to the specified mode.

kXR_clone

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_chmod		3002	Change directory or file permissions
	mode		
	kXR_ur	0x01 00	Owner readable
	kXR_uw	0x00 80	Owner writable
	kXR_ux	0x00 40	Owner searchable (directories)
	kXR_gr	0x00 20	Group readable
	kXR_gw	0x00 10	Group writable
	kXR_gx	0x00 08	Group searchable (directories)
	kXR_or	0x00 04	Other readable
	kXR_ow	0x00 02	Other writable
	kXR_ox	0x00 01	Other searchable (directories)

4.5 kXR_clone Request

Purpose: Clone parts of one or more files into another file.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_clone** **kXR_char** *fhandle*[4] **kXR_char** *reserved*[12] **kXR_int32** *alen* *alen* > 0: **clone_list**[*n*]	**kXR_char** *streamid*[2] **kXR_unt16** *status* **kXR_int32 0**
n = *alen*/16 ***with no remainder***	

clone_list	
kXR_char *srcfh*[4] **kXR_char** *rsvd*[4] **kXR_unt64** *srcoffs* **kXR_unt64** *srclen* **kXR_unt64** *dstoffs*	

Where:

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

status ending status of this request. Only the following status code indicate a normal ending:
kXR_ok - The operation has been completed without error.

fhandle
file handle value supplied by the successful response to the associated **kXR_open** request that is to be used as the destination file for the clone request. The file should be opened to allow writing.

alen binary length of the arguments that follow the request header. These arguments specify a **clone_list**. An *alen* less than the size of the **clone_list** should ignore the operation and return success. An *alen* which is not an integral multiple of the size of the **clone_list** should return an error (i.e. invalid argument).

kXR_clone

clone_list

is a vector of one ore more 32-byte elements that specify the files and their data that should be cloned into the destination file specified by *fhandle*. The variables are:

srcfh -the file handle of a file opened for at least reading that is to be used as the data source for the cloning operation.

srcoffs - the offset into the source file pointing to the start of the data to be cloned. See the notes for possible restrictions.

srclen - the numbert of bytes starting at srcoffs that should be cloned. If *srclen* is zero the data to the end of the file is cloned. See the notes for possible restrictions.

dstoffs - the offset in the destination file referenced by *fhandle* where the cloned data should appear. See the notes for possible restrictions.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_unt***xx* data types are treated as **unsigned** values. All reserved fields should be initialized to binary zero.
2) Support for kXR_clone is dependent on the implementation as well as the target file system. Generally, the most restrictive approach should yield the best results. These are:
    a. *srcoffs* should be aligned on the fundamental block size of the source file system; typically a multiple of 4096.
    b. *srclen* should not go past the end of the file when added to *srcoffs*. Additionally, *srclen* should also be a multiple of the fundamental blocksize.
    c. *dstoffs* should be aligned on the fundamental block size of the source file system.
    d. All source files should reside in the same filesystem partition where the destination file resides.
3) The *fhandle* and *srcfh* values should be treated as opaque data.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_clone		3032	Clone one or more files

4.6 kXR_close Request

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

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

Where:

streamid

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

reserved

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

fhandle

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_close		3003	Close an open file

4.7 kXR_dirlist Request

Purpose: Enumerate the contents of a directory.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16** **kXR_dirlist** **kXR_char** *reserved*[15] **kXR_char** *options* **kXR_int32** *plen* **kXR_char** *path*[*plen*]	**kXR_char** *streamid*[2] **kXR_unt16** **kXR_ok** **kXR_int32** *dlen* **kXR_char** $Dirname_0$**\n** • • • **kXR_char** $dirname_n$ **kXR_char** 0
Normal Response w/ kXR_dcksm	**Normal Response w/ kXR_dstat**
kXR_char *streamid*[2] **kXR_unt16** **kXR_ok** **kXR_int32** *dlen* **kXR_char** "**.\n**" **kXR_char** "**0 0 0 0\n**" **kXR_char** $dirname_0$**\n** **kXR_char** $statinfo_0$ **kXR_char** [*ctype*:*csval*]$_0$**\n** • • • **kXR_char** $dirname_n$**\n** **kXR_char** $statinfo_n$ **kXR_char** [*ctype*:*csval*]$_n$ **kXR_char** 0	**kXR_char** *streamid*[2] **kXR_unt16** **kXR_ok** **kXR_int32** *dlen* **kXR_char** "**.\n**" **kXR_char** "**0 0 0 0\n**" **kXR_char** $dirname_0$**\n** **kXR_char** $statinfo_0$**\n** • • • **kXR_char** $dirname_n$**\n** **kXR_char** $statinfo_n$ **kXR_char** 0

Where:

streamid

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

options

optionally, one or more of the following:

kXR_dstat - return stat information with each entry (protocol version 3+).

kXR_dcksm - return stat information and checksum with each entry (protocol version 4+).

reserved
area reserved for future use and should be initialized to null characters (i.e. '\0').

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

path path of a directory whose entries are to be listed. It may be suffixed with **CGI** information.

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

dirname
entry in the directory whose listing was requested.

statinfo
the **kXR_stat** information for the preceeding *dirname*. Refer to **kXR_stat** for details. The *statinfo* is only returned when **kXR_dcksm** or **kXR_dstat** is set and the server implements the protocol version indicated in *options*.

ctype the name of the checksum algorithm.

csval the checksum value as a hexadecimal ASCII text string.

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 should be initialized to binary zero.
2) A directory may have multiple entries and the response contains all of the entries.
3) Each directory entry should be suffixed by a new-line character; except for the last entry which should be suffixed by a null character. When **kXR_dstat** is specified, the last entry is the couplet "*dirname*\n*statinfo*".
4) When **kXR_dcksm** is specified, **kXR_dstat** should be assumed and the checksum information as "**[** *ctype***:***csval* **]**" (e.g. "**[adler32:d395bc71]**") appended to the corresponding *statinfo* separated by a single space. If there is no checksum value associated with the entry the *csval* should be set to the word **none** (e.g. "**[adler32:none]**").
5) If a file has more than one checksum associated with it and the request does not specify which checksum should be returned, then the default checksum should be retruned.

6) Similar to **kXR_query** with the **kXR_Qcksum** subcode, the directory path may contain the **cks.ctype CGI** element specifying the particular checksum that should be returned. Refer to **kXR_Qcksum** for details.
7) 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 should a directory name be split across a response packet.
8) The server should not return the entries "." and ".." except when **kXR_dstat** is specified, in which case only the "." entry is returned.
9) An empty directory should return the eight-byte triplet {*streamid*, 0, 0} unless **kXR_dstat** is specified; in which case "{*streamid*,0,8}.\n0 0 0\0" should be returned.
10) Clients should always check if the server supports **kXR_dstat**. If the option is supported, the first entry should be a *dot* entry followed the zero stat information.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_dirlist		3004	List a directory
	options		
	kXR_dcksm	0x04	Return checksum with entry
	kXR_dstat	0x02	Return stat information with entry
	kXR_online	0x01	Only list online entries

4.8 kXR_endsess Request

Purpose: Terminate a pre-existing session.

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

Where:

streamid

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

sessid

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 should be initialized to binary zero.
2) The *sessid* value should be treated as opaque data.
3) The socket issuing the **kXR_endsess** request should 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**.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_endsess		3023	End previous session

4.9 kXR_fattr Request

The **kXR_fattr** request code is used to delete, list, retrieve, and set file attributes (also known as extended attributes). This is accomplished using request subcodes. File attributes are specific to the file system being exported by the server. The exported file system has its own specific limits on the length of attribute names and the amount of data that may associated with a name. Some even limit the total amount of attribute data that may be associated with a file. Finally, not all file systems support extended attributes. The **kXR_query** request using the **kXR_QConfig** subcode with the **xattr** argument may be used to ascertain limits for any particular server.

The **kXR_fattr** request imposes its own limits on the maximum length of an attribute name (i.e. **kXR_faMaxNlen**, currently 248 bytes) and attributes value (i.e. **kXR_faMaxVlen**, currently 65536 bytes or 64K). Be aware that smaller limits may apply, depending on the underlying file system.

The **kXR_fattr** request supports deleting, retrieving, and setting multiple attributes with one request. However, the operation should not be considered atomic when multiple attributes are specified. A maximum **kXR_faMaxVars** (currently 16) attribute vales may be deleted, set, or retrieved per request.

For delete and retrieve requests, only attribute names are specified. For set requests, the attribute names are followed by the corresponding values (i.e. in 1-to-correspondence to the names) to be used for each attribute. Regardless of the subcode, the first string in each request is the path name of the file to which the request applies; which may be a null string.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_fattr		3020	Perform file attribute function
	kXR_fattrDel	0	Delete one or more attributes
	kXR_fattrGet	1	Get one or more attributes
	kXR_fattrList	2	List file attribute names
	kXR_fattrSet	3	Set one or more attributes
	isNew	0x01	Attribute must not exist
	aData	0x10	Include attribute value

4.9.1 Layout of *namevec*

Subsequent sections refer to *namevec* which is a vector whose elements are laid out as follows:

```
kXR_unt16   rc
kXR_char    name[]
kXR_char    0
```

Where:

rc as an argument is should be set to zero. In the response, it holds the status code associated with the attribute name. A status code not equal to kXR_ok indicates that the requested operation with respect to the attribute name was not completed.

name name of an attribute. The length of each *name*, excluding the null byte, should not be greater than **kXR_faMaxNlen**. Notice that the *name* is followed by a null byte. Attribute names are null terminated strings. These elements are concatenated together to produce a vector of names.

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.
2) There is no alignment requirement in the for the *namevec* element. That is, *namevec* elements should be streamed together irrespective of byte boundaries.
3) A *namevec* element should not be split across **kXR_oksofar** responses.

4.9.2 Layout of *valuvec*

Subsequent sections refer to *valuvec* which is a vector whose elements are laid out as follows:

```
kXR_int32   vlen
kXR_char    value[vlen]
```

Where:

vlen length of the subsequent *value*.

value value that the attribute is to have when issuing a **kXR_fattrSet** request or the actual value of the attribute when issuing a **kXR_fattrGet** 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.
2) There is no alignment requirement in the for the *valuvec* element. That is, *valuvec* elements should be streamed together irrespective of byte boundaries.
3) A *valuvec* element should not be split across **kXR_oksofar** responses.

4.9.3 kXR_fattr Request – Delete Subcode

Purpose: Delete one or more file attributes.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_fattr**	**kXR_unt16**	**kXR_ok**
kXR_char	*fhandle*[4]	**kXR_int32**	*rlen*
kXR_char	**kXR_fattrDel**	**kXR_char**	*nerrs*
kXR_char	*nattr*	**kXR_char**	*nattr*
kXR_char	*options*	**kXR_char**	*namevec*[*nattr*]
kXR_char	*reserved*[9]		
kXR_int32	*alen*		
kXR_char	*path*		
kXR_char	*namevec*[*nattr*]		

Where:

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

fhandle
file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the request when no path is supplied (i.e, *path* is a null string). If a *path* is supplied, *fhandle* should be ignored.

nattr number of attribute names that follow. The value should be one or greater but no more than **kXR_faMaxVars**.

options
reserved for future options.

alen binary length of the arguments that follow the request header.

path null terminated path. The *path* may be suffixed with CGI information. If *path* is a null string (i.e. only contains a null byte) then *fhandle* should be used to identify the file to which this request applies.

namevec

a vector of null terminated attribute names. Each name in the vector is preceeded by two bytes of zero. The number of such names concatenated together should equal *nattr*. The length of each name, excluding the null byte, should not be greater than **kXR_faMaxNlen**. The *namevec* layout is described here.

rlen binary length of the response that follow the request header.

nerrs number of variables in *namevec* that could not be deleted. The two byte field preceeding the name contains a status code (i.e. *rc* in *namevec*). When it contains **kXR_OK** then variable was deleted. Otherwise, it should be the error code describing the error encountered when deleting the variable.

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.
3) The server should process the elements in the order specified.
4) There are no alignment requirements in the argument or respronse portions of the request.
5) Deletion of extended attributes should be restricted to clients with write access to the target file.

4.9.4 kXR_fattr Request – Get Subcode

Purpose: Retrieve one or more file attributes.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_fattr**	**kXR_unt16**	**kXR_ok**
kXR_char	*fhandle*[4]	**kXR_int32**	*rlen*
kXR_char	**kXR_fattrGet**	**kXR_char**	*nerrs*
kXR_char	*nattr*	**kXR_char**	*nattr*
kXR_char	*options*	**kXR_char**	*namevec*[*nattr*]
kXR_char	*reserved*[9]	**kXR_char**	*valuvec*[*nattr*]
kXR_int32	*alen*		
kXR_char	*path*		
kXR_char	*namevec*[*nattr*]		

Where:

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

fhandle
file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the request when no path is supplied (i.e, *path* is a null styring). If a *path* is supplied, *fhandle* should be ignored.

nattr number attribute names that follow. The value should be one or greater but no more than **kXR_faMaxVars**.

options
reserved for future options.

alen binary length of the arguments that follow the request header.

path null terminated path. The *path* may be suffixed with CGI information. If *path* is a null string (i.e. only contains a null byte) then *fhandle* should be used to identify the file to which this request applies.

namevec

is a vector of null terminated attribute names. Each name in the vector is preceeded by two bytes of zero. The number of such names concatenated together should equal *nattr*. The length of each name, excluding the null byte, should not be greater than **kXR_faMaxNlen**.

The *namevec* is echoed in the response. The two byte header in each name is replaced by the status code associated with retreiving the value (i.e. *rc* in *namevec*). The namevec layout is described here.

rlen binary length of the response that follows the request header.

nerrs number of variables in *namevec* that could not be retrieved. The two byte field preceeding the name contains a status code (i.e. *rc* in *namevec*). When it contains **kXR_ok** then variable's value was retrieved. Otherwise, it is the error code describing the error encountered when retrieving the variable.

valuvec

value corresponding to the specified attribute name. Values are returned in name specified order (i.e. there should be a 1-to-1 correspondene between *namevec* and *valuvec*). For attribute names that indicate an error the length for the corresponding value should be set to zero. If the attribute, in fact, has no associated value (i.e. it exists but the data is null) then the status code associated with the attribute name should be set to **kXR_ok**. The *valuvec* layout is described here.

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.
3) The server should process the elements in the order specified.
4) Only those variables that can be set via **kXR_fattr** should be returned.
5) There are no alignment requirements in the argument or respronse portions of the request.
6) Retreival of extended attributes should be restricted to clients with read access to the target file.

4.9.5 kXR_fattr Request – List Subcode

Purpose: List file attribute names.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_fattr**	**kXR_unt16**	*statok*
kXR_char	*fhandle*[4]	**kXR_int32**	*rlen*
kXR_char	**kXR_fattrList**	**kXR_char**	*names*[*rlen*]
kXR_char	*reserved*		
kXR_char	*options*	**Response**	**with ::adata set**
kXR_char	*reserved*[9]	**kXR_char**	*streamid*[2]
kXR_int32	*alen*	**kXR_unt16**	*statok*
kXR_char	*path*	**kXR_int32**	*rlen*
		kXR_char	{*name*
		kXR_int32	*vlen*
		kXR_char	*value*[*vlen*]
			}[]

Where:

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

fhandle
: file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the request when no path is supplied (i.e, *path* is a null string). If a *path* is supplied, *fhandle* should be ignored.

options
: **ClientFattrRequest::adata** include the attribute value in the response.

alen binary length of the arguments that follow the request header.

path null terminated path. The *path* may be suffixed with CGI information. If *path* is a null string (i.e. only contains a null byte) then *fhandle* should be used to identify the file to which this request applies. This is should also be the case when *alen* is zero.

statok is one of two status codes:

kXR_ok

indicates successful completion as a final response.

kXR_oksofar

indicates that a subsequent response should follow with more data. In either case, the response header is followed by one or more null terminated attribute names. Attribute names and optional subsequest values should not be split across response segements.

rlen binary length of the of the response data that follows.

names if *rlen* is not zero, then one or more null terminated attribute names forming a list of names (e.g. *name*\0[*name*\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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.
3) An attribute name should never be split across multiple responses
4) Only settable variables via **kXR_fattr** should be returned.
5) When **ClientFattrRequest::adata** is specified, attribute names whose value cannot be retrieved should not be returned.
6) There are no alignment requirements in the argument or resprone portions of the request.
7) Listing of extended attributes should be restricted to clients with read access to the target file.

4.9.6 kXR_fattr Request – Set Subcode

Purpose: Set one or more file attributes.

Request		**Normal Response**	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_fattr**	**kXR_unt16**	**kXR_ok**
kXR_char	*fhandle*[4]	**kXR_int32**	*rlen*
kXR_char	**kXR_fattrList**	**kXR_char**	*nerrs*
kXR_char	*nattr*	**kXR_char**	*nattr*
kXR_char	*options*	**kXR_char**	*namevec*[*nattr*]
kXR_char	*reserved*[9]		
kXR_int32	*alen*		
kXR_char	*path*		
kXR_char	*namevec*[*nattr*]		
kXR_char	*valuvec*[*nattr*]		

Where:

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

fhandle
file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the request when no path is supplied (i.e, *path* is a null string). If a *path* is supplied, *fhandle* should be ignored.

nattr number attribute name-value pairs that follow. The value should be one or greater but no more than **kXR_faMaxVars**.

options
is one of the following options:
isNew - the variable should only be set if it does not exist.

alen binary length of the arguments that follow the request header.

path null terminated path. The *path* may be suffixed with CGI information. If *path* is a null string (i.e. only contains a null byte) then *fhandle* should be used to identify the file to which this request applies.

namevec

is a vector of null terminated attribute names. Each name in the vector is preceeded by two bytes of zero. The number of such names concatenated together should equal *nattr*. The length of each name, excluding the null byte, should not be greater than **kXR_faMaxNlen**.

The *namevec* is echoed in the response. The two byte header in each name is replaced by the status code associated with setting the value (i.e. *rc* in *namevec*). The namevec layout is described here.

valuvec

is a vector of attribute values. Each value starts with a four byte length which may be zero to set an attribute without a corresponding value . The length should not be greater than **kXR_faMaxVlen**. The *valuvec* layout is described here.

nerrs number of variables in *namevec* that could not be set. The two byte field, *rc*, preceeding the name contains a status code. When it contains **kXR_ok** then variable's value was set. Otherwise, it should be the error code describing the error encountered when setting the variable.

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.
3) The server should process the elements in the order specified.
4) Attributes set via **kXR_fattr** should be placed in a separate internal namespace to avoid conflicts with other extended attributes.
5) There are no alignment requirements in the argument or resprone portions of the request.
6) Setting of extended attributes should be restricted to clients with write access to the target file.

4.10 kXR_gpfile Request

Purpose: Direct a server to get or put a complete file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_gpfile**	**kXR_unt16**	**kXR_waitresp**
kXR_unt16	*options*	**kXR_int32**	**4**
kXR_char	*sources*	**kXR_int32**	*seconds*
kXR_char	*streams*	**Async Attn**	**Status Update Response**
kXR_char	*reserved*[10]	**kXR_char**	*pad*[2]
kXR_unt16	*srclen*	**kXR_unt16**	**kXR_attn**
kXR_int32	*totlen*	**kXR_int32**	**18**
kXR_char	[*cstype*:*csval*\s]#	**kXR_unt32**	**kXR_asyncinfo**
kXR_char	*src*[*srclen*]	**kXR_char**	*streamid*[2]
kXR_char	**\s**	**kXR_unt16**	**kXR_gpfile**
kXR_char	*dst*[]	**kXR_int64**	*xfrbytes*
		kXR_char	*pctdone*
		kXR_char	*status*
#Should *only* be present when **kXR_gpfcsver** is specified in *options*.			

Where:

streamid

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

options

request options:

kXR_gpfcsver - verify that specified checksum matches.
kXR_gpfdlgid - use identity delegation for the retrieval otherwise token based authorization should be assumed.
kXR_gpfforce - remove any existing file with the same name prior to the retrieval.
kXR_gpfkeep - do not remove any partial file upon failure.
kXR_gpfhush - do not send status updates.
kXR_gpfPut - this is a request to put a file; otherwise, get the file.
kXR_gpftls - transfer the data using **TLS**.

kXR_gpfile

sources

the binary number of the maximum number of sources to use for the copy. A value of zero should use the default number of sources.

streams

the number of parallel streams to use for the retrieval specified in binary. A value of zero should use the default number of streams.

srclen binary length of the source URL argument, *src*.

totlen binary length of the arguments that follow.

cstype:*csval*

Specified the the checksum of *cstype* (e.g. adler32, crc32, md5, etc) and the corresponding value, *csval* that the file should have at the destination. The *cstype* should be a supported checksum algorithm and *csval* should be specified as an ASCII text hex string without the leading '0x'. A single space character should follow *csval*. A valid *csval* consists of an even number of characters whose length divided by two equals the algorithm's result in binary. The retrival should fail if the checksum of the transferred file does not equal the specified value.

src **URL** of the source file along with any **CGI** information releant to the source's location.

dst **URL** of the destination file along with any **CGI** information relevant to the destination's location. The URL should idenify the name (i.e. path), that the file is to have at the destination server (i.e. the server to which the request is directed). The path may include **CGI** information to modify file creation. Elements that can be specified in the **kXR_open** request may be specified in the **CGI**. See the notes on possible restrictions.

xfrbytes

the binary number of bytes that have been processed so far reported via an asynchronous **kXR_attn** plus **kXR_asyninfo** response.

pctdone

the binary number indicating the percentage, 0 to 100, of the of operation that has been completed reported via an asynchronous **kXR_attn** plus **kXR_asyninfo** response.

status the status of the retrieval request as one of:

kXR_gpfpend	- not started
kXR_gpfxfr	- transferring data
kXR_gpfver	- performing checksum verification

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 should be initialized to binary zero.
2) Support for **kXR_gpfile** may be determined from the **kXR_protocol** response.
3) It is up to the implementation whether or not a third party transfer is cancelled when network connectivity is lost to the client. Minimally, the desired action is for pending requests be removed from the transfer queue.
4) It is up to the implementation whether or not protocols beyond **file**, **xroot** and **xroots** are supported for the retrieval or sending of *src.* Ostensibly, the protocol specification allows the client to specify an arbitrary protocol to be used (e.g. **http**, **s3**, etc) in the *src* and *dst* **URLs**. Should a specified protocol not be supported the request should fail.
5) The **kXR_gpfile** request is primarily geared for token based authorization retrieval. However, it does allow delegated identity retrieval. An implementation should support token based authorization if it supports **kXR_gpfile**. Delegated identity retrieval is an optional extension. However, an error should be reported if the **kXR_gpfdlgid** option is set but not supported.
6) An implementation should assure that if a transfer fails for any reason whatsoever, the destination file is removed.
7) The **kXR_gpfkeep** is meant for debugging pruposes to allow failing transfer to be better diagnosed.
8) The client should handle a **kXR_auhmore** response to the the **kXR_gpfile** request. This may occur if the server needs to obtain delegated credentials to continue the request (e.g. **kXR_gpfdlgid** was specified). This is independent of any previous **kXR_authmore** response that the client may have handled (e.g. during a **kXR_login** request).
9) The general response to a successful **kXR_gpfile** request should be **kXR_waitresp**. This allows the retrieval to occur asynchronously to client execution with possible asynchronous status updates. When the request completes the client should receive a final response indicating success or failure.

10) Status updates should be handled by an asynchronous **kXR_attn** plus **kXR_asyninfo** response. The frequency is implementation dependent but typically should be spaced between 3 to 5 seconds.
11) The final response should be provided via an asynchronous **kXR_attn** plus **kXR_asynresp** response.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_gpfile		3005	n/a
	options		
	kXR_gpfcsver	0x0001	Check supplied ofr verification.
	kXR_gpfdlgid	0x0002	Use delegated identity.
	kXR_gpfforce	0x0004	Remove file at destination first.
	kXR_gpfkeep	0x0008	Keep file upon failure.
	kXR_gpfhush	0x0010	Do not send status updates.
	kXR_gpfPut	0x0020	Send the file to the destination.
	kXR_gpftls	0x0040	Send the data using TLS.
	status		
	kXR_gpfpend	0x00	Request is pending.
	kXR_gpfxfr	0x01	Request is transferring data.
	kXR_gpfver	0x02	Request is verifying the checksum.

4.11 kXR_locate Request

Purpose: Locate a file.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16** **kXR_locate** **kXR_int16** *options* **kXR_char** *reserved*[14] **kXR_int32** *plen* **kXR_char** *path*[*plen*]	**kXR_char** *streamid*[2] **kXR_unt16** **kXR_ok** **kXR_int32** *rlen* **kXR_char** *info*[*rlen*]

Where:

streamid

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

options

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

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

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

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

path path of the file to be located. CGI 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*

rlen byte length of the response that follows

info 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
IPv4 portion of the IPV6 node address, for IPV4 environments. Otherwise, a true IPV6 address is returned.

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

ppppp 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 should 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 should 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.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_locate		3027	Perform location operation
	options		
	kXR_compress	0x00 01	Return unique hosts
	kXR_nowait	0x20 00	Return immediate information
	kXR_prefname	0x01 00	Preferentially return DNS names
	kXR_refresh	0x00 80	Refresh cached information

4.12 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**	**kXR_ok**
kXR_int32	*pid*	**kXR_int32**	*slen*
kXR_char	*username*[8]	**kXR_char**	*sec*[*slen*]
kXR_char	*reserved*		**server >= 2.4.0 & client >= 1.0**
kXR_char	*ability*	**kXR_char**	*streamid*[2]
kXR_char	*capver*	**kXR_unt16**	**kXR_ok**
kXR_char	*reserved*	**kXR_int32**	*slen*+16
kXR_int32	*tlen*	**kXR_char**	*sessid*[*16*]
kXR_char	*token*[*tlen*]	**kXR_char**	*sec*[*slen*]

Where:

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

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

pid process number associated with this connection.

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

ability 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**.
0b00001000	the client is dual-stacked and supports IPv4 and IPv4 connections.This bit is also identified as **kXR_hasipv64**.
0b00010000	the client only supports IPv4 connections.This bit is also identified as **kXR_onlyprv4**.
0b00100000	the client only supports IPv6 connections.This bit is also identified as **kXR_onlyprv6**.
0b01000000	the client only supports local file access.This bit is also identified as **kXR_lclfile**.
0b10000000	the client supports redirect flags in the **kXR_redirect** response.This bit is also identified as **kXR_redirflags**.

capver

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

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

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

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

sessid 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 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 should 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 should create the security context allowed by the security information, obtain credentials, and send them using the **kXR_auth** request.
4) Authentication should occur prior to any operation that requires authentication. See the table on page 13 for a list of requests that should be authenticated.
5) 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.
6) Sending a **kXR_login** request on a previously authenticated connection destroys the authentication context; requiring that the connection be re-authenticated.
7) The sessid is used in **kXR_bind** and **kXR_endsess** requests.
8) When the client indicates **kXR_lclfile** along with **kXR_fullurl** then the client should accept redirects to a local file the via file:// protocol indicator.
9) The **kXR_redirflags** should be only used in conjunction with the **kXR_redirect** server response.
10) Opaque information should be treated as truly opaque. The client should not inspect nor modify opaque information in any way.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_login		3007	Perform server login
	ability		
	kXR_fullurl	0x01	Accepts full URL redirect
	kXR_hasipv64	0x08	IPv4 and IPv6 capable
	kXR_multipr	0x03	Accepts non-root protocol redirects
	kXR_nothing	0x00	No special abilities
	kXR_onlyprv4	0x10	Only accepts private IPv4 addresses
	kXR_onlyprv6	0x20	Only accepts private IPv6 addresses
	kXR_lclfile	0x40	Supports local file access.
	kXR_redirflags	0x80	Supports kXR_redirect flags.
	capver		
	kXR_asyncap	0x80	Supports asynchronous responses
	kXR_vermask	0x3f	Mask to isolate kXR_ver*nnn*
	kXR_ver000	0x00	Predates 2005 protocol
	kXR_ver001	0x01	Implements original 2005 protocol
	kXR_ver002	0x02	Implements above + async responses
	kXR_ver003	0x03	Implements above + 2011 extensions
	kXR_ver004	0x04	Implements above + request signing
	kXR_ver005	0x05	Implements above + TLS

4.12.1 Additional Login CGI Tokens

The following table lists additional CGI tokens that may be passed to further identify the client. They are passed in the *token* argument.

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

4.13 kXR_mkdir Request

Purpose: Create a directory.

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

Where:

streamid

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

reserved

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

options

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

kXR_mkdirpath - create directory path if it does not already exist

mode 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

kXR_mkdir

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

path path of the of the directory to be created. The *path* may be suffixed with CGI 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 should be initialized to binary zero.
2) When a directory path is created, as requested by the **kXR_mkdirpath** option, the directory permission specified in *mode* are propagated along the newly created path.
3) No **umask** applies to the specified mode.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_mkdir		3008	Create a directory
	mode		
	kXR_ur	0x01 00	Owner readable
	kXR_uw	0x00 80	Owner writable
	kXR_ux	0x00 40	Owner searchable (directories)
	kXR_gr	0x00 20	Group readable
	kXR_gw	0x00 10	Group writable
	kXR_gx	0x00 08	Group searchable (directories)
	kXR_or	0x00 04	Other readable
	kXR_ow	0x00 02	Other writable (not allowed)
	kXR_ox	0x00 01	Other searchable (directories)
	options		
	kXR_mkdirpath	0x01	Create missing directories in *path*

4.14 kXR_mv Request

Purpose: Rename a directory or file.

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

Where:

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

reserved
: area reserved for future use and should 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 binary length of the supplied old and new paths, *paths*.

path 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. Each *path* string may be suffixed with CGI 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 should 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 the first component to allow paths to have embedded spaces. When *arg1len* is non-zero then the *paths+arg1len* should 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.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_mv		3009	Rename directory or file

4.15 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**	**kXR_ok**
kXR_unt16	*mode*	**kXR_int32**	*rlen*
kXR_unt16	*options*	**kXR_char**	*fhandle*[4]
kXR_unt16	*optiont*		***optional addition***
kXR_char	*reserved*[6]	**kXR_int32**	*cpsize*
kXR_char	*fhtemplt*[4]	**kXR_char**	*cptype*[4]
kXR_int32	*plen*	**kXR_char**	*info*[*resplen*-12]
kXR_char	*path*[*plen*]		

Where:

streamid

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

mode 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

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_open_apnd - open only for appending
kXR_open_read - open only for reading
kXR_open_updt - open for reading and writing

kXR_open

kXR_open_wrto	- open for wrinting only
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)

optiont

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

kXR_dup	- the opened file should have the same contents as the one referenced by *fhtemplt* (see notes).
kXR_samefs	- the opened file should be in the same file system as The one referenced by *fhtemplt* (see notes).

reserved

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

fhtemplt

is the file handle returned by a previous **kXR_open** request to a file to be used as the template file when **kXR_dup** or **kXR_samefs** is set in *optiont*. If neither option is set then this field should be set to null characters.

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

path path of the file to be opened. The *path* may be suffixed with CGI information to provide additional information necessary to properly process the request. See the following section on CGI information for more information.

resplen

byte length of the response that follows. At least four bytes should be returned.

fhandle

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

cpsize compression page size. The *cpsize* field is returned when the **kXR_compress** or **kXR_retstat** have been specified. Subsequent reads should be equal to this value and read offsets should 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 name of the compression algorithm used to compress the file (e.g. lz4). 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 should use the returned algorithm to decompress each *cpsize* worth of data data.

info 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 should 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.
14) The **kXR_dup** option allows the initial contents of a file to be a duplicate of the file referenced in *fhtemplt* (i.e. the template file). This option may imply **kXR_samefs**, depending on the implementation.
15) The **kXR_samefs** allows you to open a new file in the same file system where the file referenced by *fhtemplt* resides.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_open		3010	Open a file
	mode		
	kXR_ur	0x01 00	Owner readable
	kXR_uw	0x00 80	Owner writable
	kXR_ux	0x00 40	Owner searchable (directories)
	kXR_gr	0x00 20	Group readable
	kXR_gw	0x00 10	Group writable
	kXR_gx	0x00 08	Group searchable (directories)
	kXR_or	0x00 04	Other readable
	kXR_ow	0x00 02	Other writable
	kXR_ox	0x00 01	Other searchable (directories)

	options		
	kXR_async	0x00 40	Allow asynchronous I/O
	kXR_compress	0x00 01	Open without inflating files
	kXR_delete	0x00 02	Delete any existing file
	kXR_force	0x00 04	Disregard locking rules
	kXR_mkpath	0x01 00	Create any missing directories
	kXR_new	0x00 08	Create a new file
	kXR_open_apnd	0x02 00	Open only for appending
	kXR_open_read	0x00 10	Open only for reading
	kXR_open_updt	0x00 20	Open for reading and writing
	kXR_open_wrto	0x80 00	Open only for writing
	kXR_posc	0x10 00	Persist on successful close
	kXR_refresh	0x00 80	Refresh cached information
	kXR_replica	0x08 00	Open for replication
	kXR_retstat	0x04 00	Return file stat information
	kXR_seqio	0x40 00	Open for sequential I/O

	optiont		
	kXR_dup	0x00 01	Create duplicate file
	kXR_samefs	0x00 02	Create file in a particular filesys

4.15.1 Additional Open CGI Tokens

The **kXR_open** request allows a client to pass CGI information to properly steer the open. The information may or may not be acted upon, depending on the server's capabilities. The following table lists the defined CGI tokens.

Token	Token Value
`ofs.posc`	When set to a value of 1 requests "persist on successful close" processing. This is historical as the **kXR_posc** option should be preferentially used.
`oss.asize`	The mber of bytes to reserve for a new file.
`oss.cgroup`	The desired space name (a.k.a space token).

Notes

1) Unrecognized CGI tokens should be ignored.
2) Invalid arguments to a recognized CGI token should result in the termination of the request.

Example

/tmp/foo&oss.cgroup=index

4.16 kXR_ping Request

Purpose: Determine if the server is alive.

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

Where:

streamid

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

reserved

area reserved for future use and should 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 should be initialized to binary zero.
2) Use the **kXR_ping** request to see if the server is running.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_ping		3011	Send keep alive

4.17 kXR_pgread Request

Purpose: Read one or more integrity protected data pages from an open file.

Request			Normal Response	
kXR_char	*streamid*[2]		**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_pgread**		**kXR_unt16**	**kXR_status**
kXR_char	*fhandle*[4]		**kXR_int32**	*resplen*
kXR_int64	*offset*		**kXR_unt32**	*crc32c*
kXR_int32	*rlen*		**kXR_char**	*streamid*[2]
kXR_int32	*alen*		**kXR_char**	*pgrid*
			kXR_char	*pgrtype*
			kXR_char	*reserved*[4]
Arguments	when *alen* > 0		**kXR_int32**	*dlen*
kXR_char	*pathid*	*alen*>0	**kXR_int64**	*offset*
kXR_char	*reqflags*	*alen*=2	**kXR_char**	*data*[*dlen*]

Where:

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

offset binary offset from which the data is to be read. For best performance, the *offset* should be an integral multiple of the page size. However, unaligned offsets are allowed but require special data framing as described in a subsequent section. In the response, it is the file offset from which the returned data was read.

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

alen binary length of the arguments that follow the request header.

pathid
: when *alen* is > 0, this is a path identifier returned by **kXR_bind**. The response data is sent via this path, if possible. If *pathid* is not specified or is zero, the login stream should be used to deliver the response. When *pathid* is set to **kXR_AnyPath** then the server can use any bound path to return the response.

kXR_pgread

reqflags

when *alen* >= 2, these are request flags, as follows:

kXR_pgRetry - request is a retry of a previous request.

resplen

binary length of the response that follows excluding the *data* portion.

crc32c **CRC32-C** as defined by the **IETF RFC 7143** standard (see the **kXR_status** response for details) of the *resplen*-**sizeof(***crc32c***)** bytes immediately after *crc32c*. This means that the *data* portion, if any, should *not* be included in the *cr32c* calculation.

pgrid response signature and should be equal to **kXR_pgread**-**kXR_1stRequest**.

pgrtype

indicates the type of status being reported. Only the following type codes are allowed relative to namespace **XrdProto**:

kXR_FinalResult - All of the data has been transmitted.

kXR_PartialResult - Partial data has been transmitted; additional data should be expected on this stream.

datalen

binary length of the of the data, *data*, that was *actually* read plus associated checksums.

data data that was read. Each page or page segment should be preceeded by a 4 byte **CRC23C** checksum. The first page may be shorter than a full page if the offset is not page aligned. The last page may be shorter than a full page if it is the last one in the file being read and is incomplete.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char**, **kXR_unt16** and **kXR_unt32** data types are treated as **unsigned** values. All reserved fields should be initialized to binary zero.
2) Support for **kXR_pgread** may be determined from the **kXR_protocol** response and testing for the presence of the **kXR_suppgrw** flag.
3) The *fhandle* value should be treated as opaque data.
4) The **kXR_pgPageSZ** defines the page size (currently 4096 bytes).

5) If more data is requested than the file contains, the last page may be smaller than the actual page size.
6) Since a read may request more data than the allowable internal buffer size, the data may be sent in fixed-sized segments until the request is satisfied. This is accomplished using the **kXR_status** subcode **kXR_PartialResult**. Any number of these status subcodes may be transmitted. However, the final result should be transmitted using the **kXR_FinalResult** subcode. For details, see the description of **kXR_status**.
7) The server may return a **kXR_FinalResult** with a data length of zero. The *offset* in the response should be the offset at which the read would have occurred. This may occur for implementation-specific reasons. However, if the *offset* is beyond the end of the file this should always occur.
8) The **kXR_pgread** request should never return **kXR_ok** or **kXR_oksofar** status codes as these are subsumed in the **kXR_status** response.
9) Sending requests using the same *streamid* when a **kXR_PartialResult** subcode has been returned may produced unpredictable results unless unique offsets are tracked. A client should serialize all requests using the *streamid* in the presence of partial results.
10) To provide strong integrity, requests should use a **TLS** connection. Data responses, however, are returned on the socket associated with *pathid* may or may not use **TLS**. Checksums in the response provide the integrity so **TLS** should generally be used only when privacy is required or to protect againt an intervening malicious agent.
11) To maximize performance, the client should request that data be delivered on a unecrypted bound socket. If the socket is not using **TLS**, the client should verify
    a. the *crc32c* checksum in the returned response is correct,
    b. the *pgrid* signature is the expected response, and
    c. the **CRC32C** checksum that preceeds each page matches the checksum calculated for the subsequent data.
12) If the socket is using **TLS** then only *pgrid* needs to be verified.

kXR_pgread

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_pgread		3030	Read pages from a file
---	**---**	---	---
	kXR_AnyPath	0xff	Use any bound path.
	kXR_pgPageSZ	4096	
	kXR_pgUnitSZ	4100	kXR_pgPageSZ + sizeof(kXR_unt32)
	kXR_pgRetry	0x01	Request is a retry.
	kXR_1stRequest	3000	First request code.

4.17.1 Error recovery

4.17.1.1 Client

1. When a checksum error is detected in the response header none of the data in the header can be trusted. This includes the indicated data length which means the client cannot determine the actual amount of data the server is returning. The only viable course of action is to close the stream socket, recreate it if it is needed, and resubmit the request.

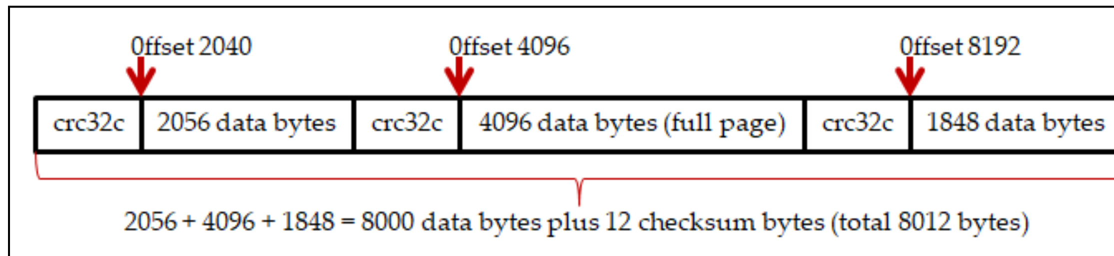
2. When a checksum error is detected in the returned data page the client should request a replacement for the page in error. If on the second attempt the checksums do not match it is likely that the data is corrupted on the device and any additional retries are likely to be ineffective. Additionally, the client should set the **kXR_pgRetry** flag in *flags* when resuesting a replacement page or page segment to enable additional checks should he server support them. The client should only request a **kXR_pgRetry** replacement for a single page or page segment. Requesting more than one page or crossing a page boundary in a **kXR_pgRetry** request is undefined and the server may treat the request as if the flag were not set.

4.17.1.2 Server

If a request indicates **kXR_pgRetry** the server should attempt to verify that media corruption did not occur if at all possible. If media corruption did occur and cannot be corrected, a checksum error should be returned. Otherwise, the request may be treated as a normal **kXR_pgread** request.

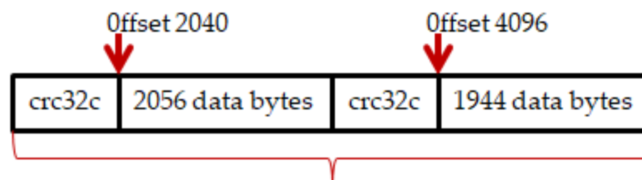
4.17.2 Unaligned reads

The **kXR_pgread** request allows unaligned offsets (i.e. the read does not start on a page boundary). By extension, a read need not read a multiple of the page size. This is defined as a special case of a short segment, followed by full page segments, followed by the ending segment. As such, the returned data reflects this definition as shown in the following example.



Example 1: Read 8000 bytes at offset 2040

The server should never allow a data segment shorter than a page to cross a page boundary. This allows the server to maintain the highest possible performance when handling full pages. A degenerate case arises when an unaligned **kXR_pgread** reads less than a page worth of data but the data crosses a page boundary. The returned result is shown below.



Example 2: Read 4000 bytes at offset 2040

It follows that unaligned reads that do not cross a page boundary should be returned as a single segment with a single checksum.

4.17.3 Backward Compatability

The **kXR_pgread** request is meant to be used for new operations that require full checksumming of the data being read. It may not be supported by all servers. In order to provide backward compatability the client-side implementation should perform the following steps if the server does not specify the **kXR_suppgrw** flag in the **kXR_protocol** response (i.e. does not support **kXR_pgread**):

- Use a standard **kXR_read** request to satisfy the request, and
- If the connection is using **TLS**, optionally generate the checksums for each page to be returned to the application.
- Otherwise, indicate to the application that while the data was read, no checksums are available.

It is up to the application to decide the subsequent course of action.

4.18 kXR_pgwrite Request

Purpose: Write one or more integrity protected data pages to an open file.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_pgwrite**	**kXR_unt16**	**kXR_status**
kXR_char	*fhandle*[4]	**kXR_int32**	*resplen*
kXR_int64	*offset*	**kXR_unt32**	*crc32c*
kXR_char	*pathid*	**kXR_char**	*streamid*[2]
kXR_char	*reqflags*	**kXR_char**	*pgwid*
kXR_char	*reserved*[2]	**kXR_char**	**kXR_FinalResult**
kXR_int32	*dlen*	**kXR_char**	*reserved*[4]
kXR_char	*data*[*dlen*]	**kXR_int32**	*elen*
		kXR_int64	*offset*
		Appended when *elen* **> 0**	
		kXR_unt32	*csecrc*
		kXR_int16	*dlfirst*
		kXR_int16	*dllast*
		kXR_int64	*boffs*[*bnum*]
Where: *bnum* = (*elen* **– 8**) / **sizeof(kXR_int64)**			

Where:

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

offset binary offset at which the data is to be written. For highest performance, the *offset* should be an integral multiple of the page size. Unaligned offsets are allowed but require special data framing as described in the following section. The *offset* in the response should match the *offset* in the request relative to the *streamid* used.

pathid the path identifier returned by **kXR_bind**. The *data* should be sent via this stream. If *pathid* is zero, the login stream should be used.

reqflags
: request flags, as follows:
kXR_pgRetry - request is a retry of a previous request.

dlen binary length of the data plus checksums sent.

data data to be written. Each page or page segment should be preceeded by a 4 byte **CRC32C** checksum.

resplen
binary length of the response that follows.

crc32c **CRC32-C** as defined by the **IETF RFC 7143** standard (see the **kXR_status** response for details) of the *resplen*-**sizeof(***crc32c***)** bytes immediately after *crc32c*.

pgwid response signature and should be equal to **kXR_pgwrite-kXR_1stRequest**.

pgwtype
indicates the type of status being reported. Only the following type code is allowed relative to namespace **XrdProto**:
kXR_FinalResult - All of the data has been received possibly with some checksum errors (see notes).

csecrc **CRC32-C** as defined by the **IETF RFC 7143** standard (see the **kXR_status** response for details) of the *elen*-**sizeof(***csecrc***)** bytes immediately after *csecrc*. This is the checksum of the appended extension.

dlfirst is the data length associated with the first *boffs* entry. This is the number of bytes that need to be resent at *boffs*[0].

dllast is the data length associated with the last *boffs* entry. This is the number of bytes that need to be resent at *boffs*[(`elen`**−8**)/**sizeof(kXR_int64)-1**].

boffs binary offset of each page that was sent whose checksum did not match. The total number should be calculated as shown. The result should be non-negative, have no remainder.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char**, **kXR_unt16** and **kXR_unt32** data types are treated as **unsigned** values. All reserved fields should be initialized to binary zero.

2) Support for **kXR_pgwrite** may be determined from the **kXR_protocol** response and testing for the presence of the **kXR_suppgrw** flag.
3) The *fhandle* value should be treated as opaque data.
4) The **kXR_pgPageSZ** defines the page size (currently 4096 bytes).
5) When *pathid* equals zero the client should use the login stream to send
   a. the request header immediately followed by
   b. exactly *dlen* bytes of data including checksums.
6) When *pathid* is not zero, then the client should
   a. send the request header on the login stream and
   b. send exactly *dlen* bytes of data including checksums on the stream indentified by *pathid*.
7) To provide strong integrity requests should be sent using a **TLS** connection. To provide performance, the data should be sent via a non-**TLS** bound path.
8) A request should be considered invalid if it attempts to write less than a single data byte either in the first and only segment or the last segment.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_pgwrite		3026	Write pages to a file
---	**---**	---	---
	kXR_1stRequest	3000	First request code.
	kXR_pgMaxEos	256	Maximum uncorrected errors
	kXR_pgMaxEpr	64	Maximum errors per request
	kXR_pgPageSZ	4096	
	kXR_pgRetry	0x01	Request is a retry.
	kXR_pgUnitSZ	4100	kXR_pgPageSZ + sizeof(kXR_unt32)

4.18.1 Error recovery

4.18.1.1 Client

When a client receives the response to **kXR_pgwrite** and the response contains one or more offsets associated with pages or page segments whose checksum did not match, the client should perform the following steps:

1. For each *offset*, resend the page or page segment with the **kXR_pgRetry** flag set in *flags*. Unless this flag is set, the write request is not treated as a correction and should result in an error should any uncorrected checksum errors when the file is closed.
2. For the first *offset* in the list, use the *dlfirst* length as the the amount of data that needs to be resent.
3. For the last *offset* in the list, use the *dllast* length as the amount of data that needs to be resent.
4. For all offsets except the first and last, use **kXR_pgPageSZ** as the amount of data that needs to be resent.
5. For each offset the resent data should be in a separate request. The client should not combine adjacent offsets. Violating this restriction should result in an error and rejection of the request.

4.18.1.2 Server

When the server detects a page or page segment in error due to a checksum error, it should perform the following steps:

1. Record the offset to be sent back to the client when the request is completed.
2. Limit the number of checksum errors in a single request. The lower bound for the limit is defined by **kXR_pgMaxEpr**. A server may implement a higher limit. When the limit is exceeded the request should fail with a **kXR_TooManyErrs** eror code.
3. Record the offset and length of the page or page segment in error with the associated file. The set of offset-length pairs defines the areas that should be corrected before the file is closed.
4. Limit the number of outstanding checksum errors in the file. The lower bound for the limit is defined by **kXR_pgMaxEos**. A server may implement a higher limit. When the limit is exceeded the request should fail with a **kXR_TooManyErrs** eror code.

When the server receives a **kXR_pgwrite** request with the **kXR_pgRetry** flag set it should perform the following actions:

1. Reject the request if it attempts to write more than one page or page segment.
2. Determine if the request is actually applying a correction based on the recorded offset-length pairs or areas that need correction recorded with the file. If the request's offset-length is not in the list, the request should be treated as a regular request. The server may log this as a client error.
3. Upon successfully correcting the data in error, remove the associated offset-length pair from he list of areas that need to be corrected.

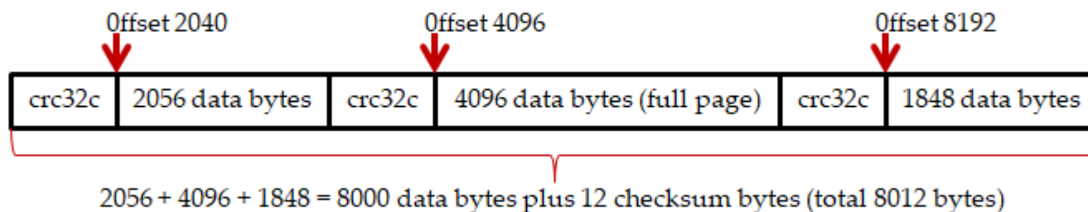
When the server receives a **kXR_close** request for a file that was subject to **kXR_pgwrite** requests it should perform the following actions:

1. Return a **kXR_ChkSumErr** error if the file still has any outstanding checksum errors (i.e. uncorrected errors).
2. Treat the file as if the client connection was lost. That is, perform a forced close so that the underlying system is aware that file closure is not being done upon request and if any recovery mechanism is in place (e.g. **POSC** or checkpointing) they are to be applied to the file to recover from the error.

It is unspecified what the server should do with a page in error. It is up to the implementation to appropriately dispose of pages in error.

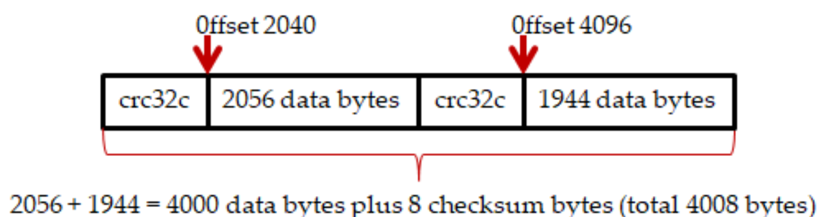
4.18.2 Unaligned writes

The **kXR_pgwrite** request allows unaligned offsets (i.e. the write does not start on a page boundary). By extension, a write need not write a multiple of the page size. This is defined as a special case of a short segment, followed by full page segments, followed by the ending segment. As such, the supplied data should reflect this definition as shown in the following example.



Example 1: Write 8000 bytes at offset 2040

The server should never allow a data segment shorter than a page to cross a page boundary. This allows the server to maintain the highest possible performance when handling full pages. A degenerate case arises when an unaligned **kXR_pgrwrite** writes less than a page worth of data but the data crosses a page boundary. The data that should be supplied is shown below.



Example 2: Write 4000 bytes at offset 2040

It follows that unaligned writes that do not cross a page boundary should be supplied as a single segment with a single checksum.

4.18.3 Backward Compatability

The **kXR_pgwrite** request is meant to be used for new operations that require full checksumming of the data being written. It may not be supported by all servers. In order to provide backward compatability the client-side implementation should use a standard **kXR_write** request if the application indicates that it wants to write data with checksums and the server did not specify the **kXR_suppgrw** flag in the **kXR_protocol** response.

4.19 kXR_prepare Request

Purpose: Prepare one or more files for access.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_prepare** **kXR_char** *options* **kXR_char** *prty* **kXR_unt16** *port* **kXR_unt16** *optionX* **kXR_char** *reserved*[10] **kXR_int32** *plen* **kXR_char** *plist*[*plen*]	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32** *rlen* **kXR_char** *resp*[*rlen*]

Where:

streamid

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

options

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

optionX

extended options to apply to each *path*. The *options* are an "or'd" combination of the following:

kXR_evict - the file is no longer needed.

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

kXR_prepare

port 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
area reserved for future use and should be initialized to null (i.e., '\0').

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

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

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

resp 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 should 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.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_prepare		3021	Prepare for future file access
	options		
	kXR_cancel	0x01	Cancel previous prepare request
	kXR_notify	0x02	Send stage completion notification
	kXR_noerrs	0x04	Suppress error notifications
	kXR_stage	0x08	Stage in missing files
	kXR_wmode	0x10	Prepare for writing
	kXR_coloc	0x20	Colocate all specified files
	kXR_fresh	0x40	Update file access time
	options		
	kXR_evict	0x0001	File is no longer needed.

4.20 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**	**kXR_ok**
kXR_int32	*clientpv*	**kXR_int32**	*dlen*
kXR_char	*options*	**kXR_int32**	*pval*
kXR_char	*expect*	**kXR_int32**	*flags*
kXR_char	*reserved*[10]	[	***Bind Preferences***
kXR_int32	**0**	**kXR_char**	**'B'**
		kXR_char	*rsvd*
		kXR_unt16	*bplen*
		kXR_char	*bindprefs*[*bplen*]
		] [	***Security Requirements***
		kXR_char	**'S'**
		kXR_char	*rsvd*
		kXR_char	*secver*
		kXR_char	*secopt*
		kXR_char	*seclvl*
		kXR_char	*secvsz* | **0**
		[	***Optional Overrides***
		kXR_char	**{***reqidx*
		kXR_char	*reqlvl***}[***secvsz***]**
		]]	
dlen: **8** [+ **4** + *bplen*] [+ **6** + *secvsz****2**]			

Where:

streamid

binary identifier that is associated with this request stream. This identifier should 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.

expect specifies what the client intends to do next and is optional. The presence of an *expect* value allows a server to unilaterally transition the connection to **TLS** if the subsequent intended operation requires **TLS**. This avoids an additional roundtrip between the client and server. The expectations are encoded bits and the server should use the **kXR_ExpMask** to isolate the bits. An invalid expectation setting should be treated as **kXR_ExpNone** (i.e. no expectation). The expect value should be treated as an indication of a single possible future action that may be taken by the client.
Single valid values are:

kXR_ExpNone	no intentions.
kXR_ExpBind	a **kXR_bind** request should be expected.
kXR_ExpLogin	a **kXR_login** should be expected.
kXR_ExpGPF	a **kXR_gpfile** should be expected.
kXR_ExpGPFA	an anonymous **kXR_gpfile** should be expected.
kXR_ExpTPC	a third party copy should be expected.

options

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

kXR_ableTLS	client is **TLS** capable.
kXR_bifreqs	return **kXR_bind** preferences, if any.
kXR_secreqs	return protocol security signing requirements.
kXR_wantTLS	client wants to transition the connection to **TLS**.

reserved

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

pval binary protocol version number the server is using.

flags 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_attrCache - Has the cache attribute (e.g. caching proxy).
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.

kXR_anongpf - Allows anonymous **kXR_gpfile** requests.
kXR_supgpf - Supports the **kXR_gpfile** request.
kXR_suppgrw - Supports **kXR_pgread** & **kXR_pgwrite** requests.
kXR_supposc - Supports persist on successful close option.

kXR_haveTLS - Supports **TLS** connections.
kXR_gotoTLS - Server has transitoned connection to use **TLS**.
kXR_tlsData - Data must be sent over a **TLS** connection.
kXR_tlsLogin - Login must use a **TLS** connection.
kXR_tlsGPF - **kXR_gpfile** must use a **TLS** connection.
kXR_tlsGPFA - anonymous **kXR_gpfile** must use a **TLS** connection.
kXR_tlsSess - Connection transitions to **TLS** after login.
kXR_tlsTPC - Third party copy must use a **TLS** connection.

Bind Preferences

If the server supports **kXR_bifreqs** and the information is present, bind preferences should be returned with *bindprefs* null padded so to make the complete package a multiple of 8 bytes.

B the ASCII character **B** (0x42).
rsvd a reserved byte set to zero.
bplen length of the following *bindprefs* field.
bindprefs
a comma separated list of *hostname*:*port*, *ipv4addr*:*port*, or [*ipv6addr*]:*port* bind targets. The ASCII text string should end with a

null byte. If more than one target is present, the client should round-robbin across the addresses for each **kXR_bind** request.

Security Requirements

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

S the ASCII character **S** (0x53).

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 doubltes 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_signIgnore 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 should 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 should 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 should 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.
10) Protocol version 5.0.0 introduced a mechanism to create bound data channels to alternate server interfaces. Pre 5.0.0 servers should never return bind information when requested to do so. Servers that have no bind preferences should not return any bind preference information when requested to do so. When bind preferences have not been returned the client should assume that no preferences exist.
11) The **kXR_protocol** request should be used to transition a normal connection to one that uses **TLS**. The client requests such a transition by setting the **kXR_wantTLS** flag in *options*. If the option is set and the server supports **TLS**, it should transition the connection to use **TLS** after the response is sent. The server may also unilaterally transition the connection to use **TLS** after sending the response based on the expect flags which indicate what the client intends to do next. When a connection will transition to use **TLS** immediately afer the response is sent, the **kXR_gotoTLS** should be set in the response to indicate the client should also transition to **TLS**. Th flag should not be set if the conection will not transition to **TLS** mode (e.g. the connection is already in **TLS** mode).
12) Clients should indicate whether or not they can use **TLS** on the connection by setting the **kXR_ableTLS** option. Should only the **kXR_wantTLS** option be set, severs should assume the **kXR_ableTLS** option is also in effect.
13) If the server does not support **TLS**, none of the **TLS** related flags should be set in *flags*. Otherwise, settings are left up to the implementation, except as noted below.
14) The **kXR_tlsLogin** and **kXR_tlsSess** flags are mutually exlusive. Should both be erroneously set, the client should ignore the **kXR_tlsSess** setting.
15) The **kXR_tlsGPFA** flag may be set without **kXR_tlsLogin** or **kXR_tlsSess** being set as anonymous **kXR_gpfile** does not require a login or session.
16) If the server supports **kXR_gpfile** requests, the **kXR_supgpf** flag should be set. The client should ignore the **kXR_anongpf** flag if **kXR_supgpf** flag is not set.
17) If the **kXR_tlsTPC** flag is set, the server should also set the **kXR_tlsLogin** or **kXR_tlsSess** flag as well. In the erroneous absence of either flag, the client should assume **kXR_tlsSess**.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_protocol		3006	Ascertain server's protocol
	clientpv		Client's protocol version
		0x00 00 02 45	Protocol version 2.4.5 (2005)
		0x00 00 02 96	Protocol version 2.9.6 (2010)
		0x00 00 02 99	Protocol version 2.9.9 (2011)
		0x00 00 03 00	Protocol version 3.0.0 (2013)
		0x00 00 03 10	Protocol version 3.1.0 (2016)
		0x00 00 04 00	Protocol version 4.0.0 (2018)
		0x00 00 05 00	Protocol version 5.0.0 (2020)
		0x00 00 05 11	Protocol version 5.1.1 (2022)
	expect		
	kXR_expMask	0x0f	Mask to isolate *expect* enum encoding
	kXR_ExpNone	0x00	No expectations
	kXR_ExpBind	0x01	Expect a kXR_bind
	kXR_ExpGPF	0x02	Expect kXR_gpfile
	kXR_ExpGPFA	0x05	Expect anonymous kXR_gpfile
	kXR_ExpLogin	0x03	Expect a kXR_login
	kXR_ExpTPC	0x04	Expect a third party copy request
	options		
	kXR_ableTLS	0x02	Client is TLS capable
	kXR_bifreqs	0x08	Return kXR_bind preferences
	kXR_secreqs	0x01	Return security requirements
	kXR_wantTLS	0x04	Client wants to transition to TLS
Response	*flags*		
	kXR_DataServer	0x00 00 00 01	Node is a data server
	kXR_LBalServer	0x00 00 00 00	Node is not a data server
	kXR_isManager	0x00 00 00 02	Node has a manager role
	kXR_isServer	0x00 00 00 01	Node has a server role
	kXR_attrMeta	0x00 00 01 00	Node has a meta role attribute
	kXR_attrProxy	0x00 00 02 00	Node has a proxy role attribute
	kXR_attrSuper	0x00 00 04 00	Node has a supervisor role attribute
	kXR_anongpf	0x00 80 00 00	Allows anonymous kXR_gpfile
	kXR_supgpf	0x00 40 00 00	Supports kXR_gpfile
	kXR_suppgrw	0x00 20 00 00	Supports kXR_pgread & kXR_pgwrite
	kXR_supposc	0x00 10 00 00	Supports kXR_posc open option
	kXR_haveTLS	0x80 00 00 00	Supports TLS connections
	kXR_gotoTLS	0x40 00 00 00	Connection will transition to TLS mode
	kXR_tlsAny	0x1f 00 00 00	Mask to isolate requirement flags
	kXR_tlsData	0x01 00 00 00	All data requires a TLS connection
	kXR_tlsGPF	0x02 00 00 00	kXR_gpfile requires TLS
	kXR_tlsGPFA	0x20 00 00 00	Anonymous kXR_gpfile requires TLS
	kXR_tlsLogin	0x04 00 00 00	kXR_login requires a TLS connection

kXR_protocol

Response	Modifiers	Value	Explanation
	kXR_tlsSess	0x08 00 00 00	Connection transition to TLS after login
	kXR_tlsTPC	0x10 00 00 00	TPC requests require a TLS connection
	pval	see *clientpv*	Server's protocol version
	reqlvl		
	kXR_signIgnore	0x00	Signature is not needed
	kXR_signLikely	0x01	Signature needed when modifying
	kXR_signNeeded	0x02	Signature need in all cases
	seclvl		
	kXR_secCompatible	0x01	
	kXR_secStandard	0x02	
	kXR_secIntense	0x03	
	kXR_secPedantic	0x04	
	secopt		
	kXR_secOData	0x01	Write data must be signed
	kXR_secOFRCE	0x02	Sign requests even if unencrypted

4.20.1 Client's expect setting & Server's TLS Requirement Response

	Client's Expect Setting vs Server's TLS Requirement Flag Settings					
Client's Expect Setting	**tlsData**	**tlsLogin or tlsSess**	**tlsGPF**	**tlsGPFA**	**tlsTPC**	**gotoTLS**
none	Only if required	Only if required	Only if required	Only if required	Only if required	no
ExpBind	Only if required	*immaterial*	*immateral*	*immaterial*	*immaterial*	Only if tlsData
ExpGPF	Only if required	Only if required	Only if required	*immaterial*	Only if required	Only if tlsLogin
ExpGPFA	*immaterial*	*immaterial*	*immaterial*	Only if required	*immaterial*	Only if tlsGPFA
ExpLogin	Only if required	Only if required	Only if required	*immaterial*	Only if required	Only if tlsLogin
ExpTPC	Only if required	Only if tlsTPC	Only if required	*immaterial*	Only if required	Only if tlsLogin

Flag	Explanation of Client's kXR_Exp*xxx* Setting
none	When the expect value is zero the client is not declaring any future action and the server should have no expectations. The server's flag settings should cover the TLS requirements for all possible subsequent client requests.
ExpBind	Client plans to issue a kXR_bind request (note: kXR_login is disallowed).
ExpGPF	Client plans to issue a kXR_gpfile request *after* a kXR_login request.
ExpGPFA	Client plans to issue a kXR_gpfile request *without* a kXR_login request.
ExpLogin	Client plans to issue a kXR_login request.
ExpTPC	Client plans to issue a kXR_open request that initiates a third-party copy *after* a kXR_login request.

Flag	Client's Interpretation of Server's kXR_*xxx* Flag Settings
gotoTLS	Connection should convert to using TLS as the server has already converted its side of the connection. The server should set this flag in the response
tlsData	When this flag is set then data transmission (i.e. reads and writes) should be encrypted via a TLS connection. This applies to the session connection as well as any connection bound to the session connection using kXR_bind. When not set the session connection may be forced to use TLS due to other flag settings but bound connections need not use TLS.
tlsGPF	Connection should be using TLS *prior* to a kXR_gpfile request.
tlsGPFA	Connection should be using TLS *prior* to an anonymous kXR_gpfile request.
tlsLogin	Connection should be converted to TLS *prior* to a kXR_login request.
tlsSess	Connection should be using TLS for *any* request issued *after* the kXR_login request.
tlsTPC	Connection should be using TLS prior to issuing a kXR_open request that results in a third-party copy. While converting a connection to TLS could be delayed until the actual kXR_open, the protocol requires that the connection be converted *prior* kXR_login (tlsLogn) or *post* kXR_login (tlsSess) and the correct flag should be set.

Setting	Explanation Server Setting Decision
immaterial	The server may set the column heading flag if **TLS** required but the client should not use the information for the planned request or any future request on the connection.
Only if required	The server should set the column heading flag if it is required to meet the site's security policy.
Only if tlsData	The server should set gotoTLS if it set the tlsData and the client set ExpBind. After the response, the server should convert its side of the connection to TLS.
Only if tlsGPFA	The server should set gotoTLS if it set the tlsGPFA and the client set ExpGPFA. After the response, the server should convert its side of the connection to TLS.
Only if tlsLogin	The server should set gotoTLS if it set the tlsLogin and the client set ExpGPF, ExpLogin, or ExpTPC. After the response, the server should convert its side of the connection to TLS.
Only if tlsTPC	The server should set tlsLogin or tlsSess, but not both, if it set tlsTPC and the client set ExpTLS. The choice on which flag to set is dictated by the site's security policy.
Only if tlsGPFA	Client plans to issue a kXR_open request that initiates a third-party copy *after* a kXR_login request.

4.20.2 Protocol Security Requirements vs 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_auth	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore
kXR_bind	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded
kXR_chkpoint	kXR_signNeeded	kXR_signNeeded	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_dirlist	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_endsess	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded
kXR_fattr	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded	kXR_signNeeded
kXR_gpfile	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_pgread	kXR_signIgnore	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded
kXR_pgwrite	kXR_signIgnore	kXR_signIgnore	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_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_write	kXR_signIgnore	kXR_signIgnore	kXR_signNeeded	kXR_signNeeded

kXR_protocol

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.

4.21 kXR_query Request

Purpose: Obtain server information.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_query** **kXR_unt16** *reqcode* **kXR_char** *reserved*[2] **kXR_char** *fhandle*[4] **kXR_char** *reserved*[8] **kXR_int32** *alen* **kXR_char** *args*[*alen*]	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32** *ilen* **kXR_char** *info*[*ilen*]

Where:

streamid

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

reserved

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

fhandle

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

reqcode

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_QFinfo	Query file information
kXR_QFSinfo	Query filesystem information
kXR_Qopaque	Query implementation-dependent information
kXR_Qopaquf	Query implementation-dependent information
kXR_Qopaqug	Query implementation-dependent information
kXR_QPrep	Query prepare status
kXR_Qspace	Query server logical space statistics
kXR_Qstats	Query server statistics

kXR_query

kXR_Qvisa	Query file visa attributes
kXR_Qxattr	Query file extended attributes

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

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

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

info 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 should 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 **CGI** 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 should 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.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_query		3001	Return information
	reqcode		
	kXR_QStats	0x00 01	Return statistics
	kXR_QPrep	0x00 02	Return prepare status
	kXR_Qcksum	0x00 03	Return checksum
	kXR_qxattr	0x00 04	Return extended space attributes
	kXR_qspace	0x00 05	Return space information
	kXR_qckscan	0x00 06	Return checksum cancellation info
	kXR_Qconfig	0x00 07	Return configuration information
	kXR_Qvisa	0x00 08	Return visa status
	kXR_QFinfo	0x0009	Return file information
	kXR_QFSinfo	0x000A	Return filesystem information
	kXR_Qopaque	0x00 10	Return implementation information #1
	kXR_Qopaquf	0x00 20	Return implementation information #2
	kXR_Qopaqug	0x00 40	Return implementation information #3

4.21.1 KXR_query Checksum Cancellation Request

Purpose: Obtain server information.

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

Where:

streamid

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

reserved

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

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

path path of the file whose check sum is to be cancelled. The *path* may be suffixed with CGI 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 should be initialized to binary zero.
2) Only check sums requested by the current client may be cancelled.

4.21.2 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 kXR_ok**
kXR_unt16 kXR_Qcksum	**kXR_int32** *ilen*
kXR_char *reserved*[14]	**kXR_char** *info*[*ilen*]
kXR_int32 *plen*	
kXR_char *path*[*plen*]	

Where:

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

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

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

path path of the file whose checksum is to be returned. The *path* may be suffixed with CGI information.

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

info 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 should 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.

kXR_Qcksum

Returned Response

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

```
csname csvalue
```

Where:

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

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

4.21.2.1 Additional Query Checksum CGI Tokens

The **kXR_Qcksum** request allows a client to pass CGI information to select a particular checksum should the server support multiple checksums. The information may or may not be acted upon, depending on the server's capabilities. CGI information is passed by suffixing the *path* with a question mark (**?**) and then coding the cgi information as shown below:

```
path?cks.cktype=arg
```

Where:

arg name of the desired checksum.

Notes

1) Invalid values or arguments to a recognized directive normally result in termination of the request.

Example

/tmp/foo?cks.cktype=md5

4.21.3 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**	**kXR_ok**
kXR_unt16	**kXR_Qconfig**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*qlen*		
kXR_char	*qry*[*qlen*]		
qry: **cmsd** | **xrootd** | *vars*			

Where:

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

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

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

qry configuration variables to be displayed:

cmsd	the effective **cmsd** server configuration.
xrootd	the effective **xrootd** server configuration.
vars	other information as described below.

vars 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.
fattr	the **kXR_fattr** request limits. The response, if supported, should contain three tokens: *"maxattrs maxnlen maxvlen"* *maxattrs* - maximum number of bundled requests *maxnlen* - maximum length for a name *maxvlen* - maximum length for attribute data
pio_max	maximum number of requests that may be queued on a bound socket before the session stream must wait.
proxy	the proxy setting for a proxy server.
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'.
start	the UNIX time the server was started.
tls_port	the port number to connect to for an TLS encrypted connection.
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.
tpcdlg	a space-separated list of authentication protocol names that can be used for TPC delegation, if delegation is actually supported.
version	version identification string (implementation dependent).
vnid	the specified virtual network identifier, if any.
window	socket buffer size (i.e., window) for the default port.
xattrs	user settable extended attribute limits.

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

info 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 should 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

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

kXR_Qconfig

4.21.3.1 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*
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 status of the connection:
- **c** - the endpoint is connected.
- d - the endpoint is disconnected
- s - the endpoint is connected but is in a suspended state.

4.21.3.2 Format for Query Config proxy

```
[=][host:port | /path]
```

Where:
= indicates a forwarding proxy (i.e. client can specify the origin).

host:*port*
is the origin endpoint.

/path indicates that the server is proxying a filesyste mounted at the given path.

4.21.3.3 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**

4.21.3.4 Format for Query Config xattrs

The general format for the **kXR_Qconfig xattrs** response consists of two numbers as follows:

```
maxnsz maxvsz
```

Where:

maxnsz
    maximum length a client can specify for an attribute name. If the value is zero, then client settable extended attributes are not allowed.

maxvsz
    maximum length a client can specify for an attribute value.

4.21.4 KXR_query Opaque Request

Purpose: Obtain implementation-dependent server information.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_query** **kXR_unt16** *querycode* **kXR_char** *reserved*[2] **kXR_char** *fhandle* **kXR_char** *reserved*[8] **kXR_int32** *qlen* **kXR_char** *qry*[*qlen*]	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32** *ilen* **kXR_char** *info*[*ilen*]
querycode: **kXR_Qopaque** | **kXR_Qopaquf** | **kXR_Qopaqug**	

Where:

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

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

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

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

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

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

info 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 should be initialized to binary zero.
2) These subcodes provide a mechanism to use special implementation specific features. Use of these subcodes is not portable.

4.21.5 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**	**kXR_ok**
kXR_unt16	**kXR_Qspace**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*slen*		
kXR_char	*sname*[*slen*]		

Where:

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

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

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

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

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

info 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 should 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.

4.21.6 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**	**kXR_ok**
kXR_unt16	**kXR_QStats**	**kXR_int32**	*ilen*
kXR_char	*reserved*[14]	**kXR_char**	*info*[*ilen*]
kXR_int32	*alen*		
kXR_char	*args*[*alen*]		

Where:

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

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

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

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

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

vers **XRootD** version identification string.

setc 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.
Cms	**p**	- Cluster Management Services
info	**i**	- Server identification.
link	**l**	- Connection (i.e., link) statistics.
ofs	**p**	- Open File System layer
oss	**p**	- 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.

4.21.7 KXR_query Visa Request

Purpose: Obtain server information.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_query** **kXR_unt16 kXR_QVisa** **kXR_char** *reserved*[2] **kXR_char** *fhandle* **kXR_char** *reserved2*[8] **kXR_int32 0**	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32** *ilen* **kXR_char** *info*[*ilen*]

Where:

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

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

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

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

info 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 should be initialized to binary zero.
2) The response to the **kXR_Qvisa** request is documented in the Bandwidth Manager Configuration reference.

4.21.8 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** **kXR_ok**
kXR_unt16 **kXR_QXattr**	**kXR_int32** *ilen*
kXR_char *reserved*[14]	**kXR_char** *info*[*ilen*]
kXR_int32 *plen*	
kXR_char *path*[*plen*]	

Where:

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

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

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

path path of the file whose extended attributes are to be returned. The *path* may be suffixed with CGI information.

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

info 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 should 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) The **kXR_Qxattr** request bears no relationship to the **kXR_fattr** request.

4.22 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*		
alen > 0:	**read_args**		

read_args		**readahead_list**	
kXR_char	*pathid*	**kXR_char**	*fhandle2*[4]
alen > 8:		**kXR_int32**	*rlen2*
kXR_char	*reserved*[7]	**kXR_int64**	*roffset2*
	readahead_list[*n*]		
n = (*alen*-8)/16 ***with no remainder***			

Where:

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

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

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

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

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

fhandle2
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 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 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 binary length of the of the data, *data*, that was *actually* read.

data data that was read.

Notes

4) 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 should be initialized to binary zero.
5) If more data is requested than the file contains, the total of all *dlen*'s should be less than *rlen*.
6) Reading past the end of file with a valid offset should return a *dlen* of 0.
7) The *fhandle* value should be treated as opaque data.
8) 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.
9) Any status code other than **kXR_oksofar** indicates the end of transmission.
10) Sending requests using the same *streamid* when a **kXR_oksofar** status code has been returned may produced unpredictable results. A client should serialize all requests using the *streamid* in the presence of partial results.

11) 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.
12) 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.
13) 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.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_read		3013	Read a file

4.23 kXR_readv Request

Purpose: Read data from one or more open files.

Request		Normal Response	
kXR_char	*streamid*[2]	**kXR_char**	*streamid*[2]
kXR_unt16	**kXR_readv**	**kXR_unt16**	*status*
kXR_char	*reserved*[15]	**kXR_int32**	*dlen*
kXR_char	*pathid*	**kXR_char**	*data*[*dlen*]
kXR_int32	*alen*		
	read_list[*n*]		
n = *alen*/16 ***with no remainder***			

read_list		
kXR_char	*fhandle*[4]	
kXR_int32	*rlen*	
kXR_int64	*offset*	

Where:

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

status 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 binary length of the arguments that follow the request header. These arguments may include the *pathid* and read request list, **struct read_list**. The maximum allowed value for *alen* is 16384. This allows up to 1024 read segments.

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

fhandle

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 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 binary offset from which the data is to be read.

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

data 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 should 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) The **struct read_list** is equivalent to **struct readahead_list**.
4) If an element requests more data than the file contains, the read request should fail. It is undefined whether the failure should occur before any bytes are transmitted to the client.
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 should serialize all requests using the *streamid* in the presence of partial results.
9) The server may return the read elements in any order.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_readv		3025	Read file segments

4.24 kXR_rm Request

Purpose: Remove a file.

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

Where:

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

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

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

path path of the of the file to be removed. The *path* may be suffixed with CGI 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 should be initialized to binary zero.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_rm		3014	Remove a file

4.25 kXR_rmdir Request

Purpose: Remove a directory.

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

Where:

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

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

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

path path of the of the directory to be removed. The *path* may be suffixed with CGI 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 should be initialized to binary zero.
2) The directory must be empty (i.e., no entries other than "." And "..").

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_rmdir		3015	Remove a directory

4.26 kXR_set Request

Purpose: Set server information.

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

Where:

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

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

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

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

data value to set.

resp 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 should be initialized to binary zero.
2) Set processing takes a command-like string in the *data* field. The following documents valid set arguments.

kXR_set

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_set		3018	Set monitoring information

4.26.1 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 should 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

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.

4.27 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*		
char	*crypto*		
kXR_char	*reserved*[3]		
kXR_int32	*dlen*		
kXR_char	*Sigdata*[*dlen*]		

Where:

streamid

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

expectrid

requestid of the subsequent request.

version

the 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 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.

kXR_sigver

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

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

sigdata
the signature for 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 should 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.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_sigver		3029	Present signature for following request
	crypto		
	kXR_rsaKey	0x80	RSA key encrypts the hash
	kXR_SHA256	0x01	SHA-256 used for the hash
	flags		
	kXR_nodata	0x01	Data payload is not signed

4.27.1 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[3];

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.

[3] 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.

4.27.2 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[4] 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.

[4] 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).

4.28 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**	**kXR_ok**
kXR_char	*opts*	**kXR_int32**	*ilen*
kXR_char	*reserved*[7]	**kXR_char**	*info*[*ilen*]
kXR_unt32	*wants*		
kXR_char	*fhandle*[4]		
kXR_int32	*plen*		
kXR_char	*path*[*plen*]		

Default info:
id size flags mtime [*ctime atime mode owner group* [**/n***cgi*]][5]**\0**

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

Where:

streamid

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

opts stat processing options:

kXR_vfs - return virtual file system information for the specified *path*

reserved

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

wants a bit mask indicating the additional information to be included in the *cgi* portion of the response. See the notes for details.

plen binary length of the supplied path, *path*. If *plen* is zero then *fhandle* should 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 or when **kXR_vfs** is specified. In the latter case, a *path* argument is required.

[5] Information in brackets is only returned for servers implementing version 4.0.0 of the protocol.

path path whose status information is to be returned. The *path* may be suffixed with CGI information.

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

Default Response

info information about the requested path. Note that information in brackets is returned only if the server supports extended stat information as defined in protocol version 4.0.0 or higher.

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

size 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.

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

ctime creation time in Unix time units (i.e., seconds since 00:00:00 UTC, January 1, 1970).

atime last access time in Unix time units (i.e., seconds since 00:00:00 UTC, January 1, 1970).

mode octal value of the entries mode bits (i.e. permissions and settings). A leading zero should always appear to indicate this is an octal value.

owner name of the owner associated with the entry. If the name cannot be determined, the owner's numeric identifier should be returned.

group name of the group associated with the entry. If the name cannot be determined, the group's numeric identifier should be returned.

cgi additional information, as requested by the *wants* field, that is returned in **CGI** format. This field may be pesent when the *wants* field has one or more bits set. If the wanted information is available, the **CGI** key-value pairs returned are as follows:

Wants requested	**CGI Returned**
kXR_Want_btime	**btime_s=***seconds***&btime_n=***nanoseconds*

Response for kXR_vfs

info location information about the requested path.

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

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

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

nstg number of nodes that can provide staging space.

fstg 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**, **kXR_unt16** and **kXR_unt32** data types are treated as **unsigned** values. All reserved fields should be initialized to binary zero.

2) The returned string is compatible to the format returned by the root method **Tsystem::GetPathInfo()**. However, the optional *cgi* information is unique to this protocol.
3) When a **CGI** string is returned it is prefixed by a space followed by a newline character.
4) **CGI** information returned generally corresponds to the information requested via the wants field. Possible requests are defined in the subsequent "**Binary Definitions**" table. Requested information should only be returned if the information is available. The protocol also allows that other information may be returned even when not requested.
5) **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.
6) **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.
7) **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.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_stat		3017	Get file information
	opts		
	kXR_vfs	0x01	Return file system information
	wants		
	kXR_Want_btime	0x00 00 00 01	File birth (i.e. creation) time.
Response	*flags*		
	kXR_xset	0x00 00 00 01	Execution/search bit set
	kXR_isDir	0x00 00 00 02	Item is a directory
	kXR_other	0x00 00 00 04	Item neither directory nor file
	kXR_offline	0x00 00 00 08	Item is not online
	kXR_readable	0x00 00 00 10	Item is readable
	kXR_writable	0x00 00 00 20	Item is writable
	kXR_poscpend	0x00 00 00 40	Item is pending a close
	kXR_bkpexist	0x00 00 00 80	Item has a backup

4.28.1 Additional Stat CGI Tokens

The **kXR_stat** request allows a client to pass CGI information to restrict information lookup to online space. The information may or may not be acted upon, depending on the server's capabilities. CGI information is passed by suffixing the *path* with a question mark (**?**) and then coding the cgi information as shown below:

```
path?oss.lcl=1
```

Example

/tmp/foo?oss.lcl=1

4.29 kXR_statx Request

Purpose: Obtain type information for one or more paths. This request is now deprecated and should not be used.

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

Where:

streamid

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

reserved

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

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

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

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

info 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 should 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.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_statx		3022	Get file information
Response	*flags*		
	kXR_xset	0x00 00 00 01	Execution/search bit set
	kXR_isDir	0x00 00 00 02	Item is a directory
	kXR_other	0x00 00 00 04	Item neither directory nor file
	kXR_offline	0x00 00 00 08	Item is not online
	kXR_readable	0x00 00 00 10	Item is readable
	kXR_writable	0x00 00 00 20	Item is writable
	kXR_poscpend	0x00 00 00 40	Item is pending a close
	kXR_bkpexist	0x00 00 00 80	Item has a backup

4.30 kXR_sync Request

Purpose: Commit all pending writes to an open file.

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

Where:

streamid

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

reserved

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

fhandle

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_sync		3016	Synchronize file to disk

4.31 kXR_truncate Request

Purpose: Truncate a file to a particular size.

Request (Opened File)	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_truncate** **kXR_char** *fhandle*[4] **kXR_int64** *size* **kXR_char** *reserved*[4] **kXR_int32 0**	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32 0**

Request (Closed File)	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_truncate** **kXR_char** *reserved*[4] **kXR_int64** *size* **kXR_char** *reserved*[4] **kXR_int32** *plen* **kXR_char** *path*[*plen*]	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32 0**

Where:

streamid

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

reserved

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

fhandle

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 binary size that the file is to have.

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

path path of the of the file to be truncated. The *path* may be suffixed with CGI information.

kXR_truncate

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_truncate		3028	Truncate a file

4.32 kXR_write Request

Purpose: Write data to an open file.

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

Where:

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

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

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

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

reserved
: area reserved for future use and should be initialized to null characters (i.e., ‘\0’).

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

data data to be written.

kXR_write

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 should be initialized to binary zero.
2) The *fhandle* value should be treated as opaque data.

Binary Definitions

Request	Modifiers	Value	Explanation
kXR_write		3019	Write into a file

4.33 kXR_writev Request

Purpose: Write data to one or more open files.

Request	Normal Response
kXR_char *streamid*[2] **kXR_unt16 kXR_writev** **kXR_char** *options* **kXR_char** *reserved*[15] **kXR_int32** *alen* **write_list**[*n*]	**kXR_char** *streamid*[2] **kXR_unt16 kXR_ok** **kXR_int32 0**
n = *alen*/**16 *with no remainder***	

write_list	
kXR_char *fhandle*[4] **kXR_int32** *wlen* **kXR_int64** *offset*	

Where:

streamid

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

options

following options or'd together:

kXR_doSync - perform an **fsync()** after data is written.

alen binary length of the arguments that follow the request header. The argument should be a write request list, **struct write_list**. The maximum allowed value for *alen* is 16384. This allows up to 1024 write segments.

fhandle

file handle value supplied by the successful response to the associated **kXR_open** request that is to be used for the write request. Each *fhandle* is treated separately allowing writes to occur to multiple files. Note that the *fhandle* causing an error is echoed back in an error response.

wlen binary amount of data that is to be written. Elements whose *wlen* is zero are ignored.

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

status ending status of this request. A **kXR_ok** status indicates successful completion. Any other status code indicates an error and a description follows.

dlen binary length of the of the error response data that follows. Note that the first four bytes are the binary filehandle that encountered the error.

msg null terminated message describing the error.

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 should be initialized to binary zero.
2) Each **write_list** element represents a write request. All of the write requests are aggregated into one response.
3) The *fhandle* value should be treated as opaque data.
4) Data to be written should follow the **write_list**. The total number of bytes that follow should equal the sum of all the individual *wlen* values.
5) The server should process the elements in the order specified.
6) In the presence of the **doSync** option, an **fsync()** should be performed after all the elements have been written to the file corresponding to *fhandle*. This may occur at the end of the **write_list** if no other file handles appear in the list or when encountering a new *fhandle*. When a new *fhandle* is encountered, an **fsync()** should be performed for the previous *fhandle* before processing the list for the new *fhandle*.
7) When an error occurs, it is unpredictable how many bytes have actually been written to any particular file specified in the **write_list**. Specifically, elements prior to the error may have been partially written while elements subsequent to the element causing the error are ignored.

Binary Definitions

Request	**Modifiers**	**Value**	**Explanation**
kXR_writev		3031	Write file segments
	options		
	kXR_doSync	0x01	Perform kXR_sync afterwards

5 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.

[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.

The format of the security token is:

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

Where:

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

protparms
optional protocol specific configuration parameters that should be supplied to the protocol's initialization routine when it is instantiated. The 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 should 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 should 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 should return some data that will be sent to the server. The data should 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 should 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 should 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 should 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 should 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 should be sent to the client who is expected to respond with a **kXR_auth** request. The data provided by the authentication method should 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 should be sent to the server in a new **kXR_auth** request or indicate failure. In this processing continues with step 3.

5.1 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.

5.2 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.

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**.**

Change History

1 Aug 2007

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

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**

Change History

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.

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

8 Jan 2018

- Document **kXR_fattr** request.
- Document kXR_**writev** request.
- Document the **kXR_query config fattr** request.
- Remove documentation for the **kXR_query config wan_port** and **wan_window** request (no longer useful).

17 Feb 2018

- Specify binary values for all protocol symbols.
- Remove kXR_unbind request code.
- Modify description of kXR_getfile and kXR_putfile.

18 May 2018

- Move local administrative protocol description to a separate document.

28 May 2018

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

20 Jun 2018

- Correct description of kXR_close request.
- Remove ambiguities from the kXR_dirlist request description.
- Correct description of kXR_mkdir request regarding making directory paths.
- Remove ambiguities from the kXR_open request description.
- Correct description of kXR_stat request.
- Indicate kXR_verifyw request is a placeholder.

17 Jul 2018

- Document the **ClientFattrRequest::aData** option for the **kXR_fattrList** request subcode.
- Remove all references to the **kXR_unbind** request code.
- Add caveats to the **kXR_verifyw** request code.

17 Jul 2018

- Document missing error codes.

24 Oct 2018

- Describe extended stat information returned by **kXR_dirlist**, **kXR_open** and **kXR_stat**.
- Describe **CGI** information in greater detail.

17 Dec 2018

- Describe **kXR_protocol** modifications hat support **TLS**, **kXR_getfile**, and **kXR_putfile**.
- Add new error code **kXR_TLSRequired**.

Change History

19 Apr 2019

- Describe **kXR_login** abilities **kXR_hasipv64**, **kXR_onlyprv4**, and **kXR_onlyprv6**.
- Document **kXR_FileLocked**, **kXR_NotAuthorized**, and **kXR_Unsupported** errors as retriable.

19 Apr 2019

- Add **kXR_tlsGPF** and **kXR_ExpGPF** flags to the **kXR_protocol** request.

27 Jul 2019

- Add **kXR_lclfile** flag to the **kXR_login** request.

14 Oct 2019

- Add **kXR_chkpoint**, **kXR_gpfile**, **kXR_pgread** and **kXR_pgwrite** request codes.
- Add **kXR_anongpf**, **kXR_supgpf**, **kXR_suppgrw**, and **kXR_supposc** flags to **kXR_protocol** response.
- Add **kXR_attn** + **kXR_asyninfo** response combination.
- Remove **kXR_admin** (now assigned to **kXR_fattr**), **kXR_decrypt** (now assigned to **kXR_pgread**), **kXR_getfile** (now assigned to **kXR_gpfile**), **kXR_putfile** (now assigned to **kXR_chkpoint**), and **kXR_verifyw** (now assigned to **kXR_pgwrite**) request codes.
- Remove **kXR_attn** subcodes: **kXR_asyncab**, **kXR_asyncav**, **kXR_asyncdi**, **kXR_asyncgo**, **kXR_asyncrd**, **kXR_asynunav**, and **kXR_asyncwt**.

14 Nov 2019

- Redo **kXR_gpfile**, **kXR_pgread** and **kXR_pgwrite** request codes.

14 Jan 2020

- Add missing variables to **kXR_query config** request**: cmsd**, **start**, **tpcdlg**, **tls_port**, **vnid**, and **xrootd**.

17 Jan 2020

- Add missing option, **kXR_evict**, to the **kXR_prepare** request.
- Document backward compatability requirements for **kXR_pgread** and **kXR_pgwrite** requests.

2 Apr 2020

- Document the **kXR_status** response code.
- Rearchitect the **kXR_pgread** and **kXR_pgwrite** requests.
- Update error recovery procedures for **kXR_noserver** and **kXR_SigVerErr** errors.
- Substitute **kXR_ItExists** error code for never used **kXR_ChkLenErr** error code and map it to **POSIX EEXIST** errno.
- Remap **kXR_InvalidRequest** from **EEXIST** to **EBADRQC** errno.
- Provide consistent errno mapping for **kXR_ServerError** to **EFAULT**.
- Provide consistent errno mapping for **kXR_Unsupported** to **ENOTSUP**.
- Provide consistent errno mapping for **kXR_FSError** to **ENODEV**.

15 Apr 2020

- Document the **kXR_AuthFailed** error response code.

2 May 2020

- Document the **kXR_ExpGPFA** and **kXR_tlsGPFA** values in he **kXR_protocol** request and response.

25 Aug 2020

- Correct size of **crc32** in **kXR_pgread** and **kXR_pgwrite**.

8 Oct 2020

- Fully specify the **kXR_chkpoint** request.
- Add the **kXR_Conflict** and **kXR_Impossible** error codes.

18 Oct 2020

- Document the **kXR_collapseRedir** and **kXR_recoverWrts** flags in the **kXR_redirect** response.
- Document the **kXR_redirflags** ability flag in the **kXR_login** request.

6 Jan 2021

- Document the **kXR_dcksm** flag in the **kXR_dirlist** request.

16 Mar 2021

- Rewrite the **kXR_pgread** and **kXR_pgwrite** requests.

13 Jun 2021

- Document the **kXR_ReqTimedOut** error code. Minor corrections to **kXR_chkpoint** request.

Change History

2 Aug 2021

- Correct **kXR_protocol** documentation with respect to bind preferences.
- Design this reference to be for protocol version 5.

7 Jun 2022

- Document the **kXR_dup** and **kXR_samefs** options as well as the *fhtemplt* field in the **kXR_open** request.

13 Oct 2022

- Document the format of host in the redirect response.

27 Jun 2023

- Document the **kXR_clone** request.
- Fix typographical error specifying **kXR_int31**.

4 Dec 2023

- Document the **kXR_TimerExpired** error code.
- Add missing description of the **kXR_open_wrto** option for **kXR_open.**

30 Jul 2024

- Document the **kXR_attrCache** setting in the **kXR_protocol** response.
- Document the **proxy** variable for **kXR_query** config.
- Document all errno values that map to **kXR_noserver**.

31 Jul 2025

- Document the **kXR_QFinfo** and **kXR_QFSinfo** options in the **kXR_query** request.
- Document the **kXR_directio** option in the **kXR_open** request.

18 Feb 2026

- Document the **kXR_stat** *wants* field and the corresponding response as a CGI string.