The \texttt{xrootd} Protocol
Version 2.7.0

Andrew Hanushevsky
Stanford Linear Accelerator Center
25-Jan-2007
1 Contents

1 Contents.................................................................................................................................3
2 Request/Response Protocol......................................................................................................5
  2.1 Format of Client-Server Initial Handshake........................................................................5
  2.2 Data Serialization................................................................................................................6
  2.3 Client Request Format.........................................................................................................9
    2.3.1 Valid Client Requests.....................................................................................................10
    2.3.2 Valid Client Paths.........................................................................................................11
    2.3.3 Client Recovery From Server Failures........................................................................11
    2.3.4 Client Recovery From File Location Failures..............................................................12
  2.4 Server Response Format.....................................................................................................13
    2.4.1 Valid Server Response Status Codes............................................................................14
    2.4.2 Server kXR_attn Response Format.............................................................................15
    2.4.2.1 Server kXR_attn Response for kXR_asyncab Client Action....................................17
    2.4.2.2 Server kXR_attn Response for kXR_asyncdi Client Action....................................18
    2.4.2.3 Server kXR_attn Response for kXR_asyncgo Client Action....................................19
    2.4.2.4 Server kXR_attn Response for kXR_asyncms Client Action....................................20
    2.4.2.5 Server kXR_attn Response for kXR_asyncrd Client Action....................................21
    2.4.2.6 Server kXR_attn Response for kXR_asyncresp Client Action..................................23
    2.4.2.7 Server kXR_attn Response for kXR_asyncwt Client Action....................................24
    2.4.3 Server kXR_authmore Response Format..................................................................25
    2.4.4 Server kXR_error Response Format............................................................................26
    2.4.4.1 Server kXR_error Sub-Codes..................................................................................27
    2.4.5 Server kXR_ok Response Format.............................................................................28
    2.4.6 Server kXR_oksofar Response Format.......................................................................29
    2.4.7 Server kXR_redirect Response Format......................................................................30
    2.4.8 Server kXR_wait Response Format...........................................................................32
    2.4.9 Server kXR_waitresp Response Format....................................................................33
  3 Detailed Protocol Specifications..............................................................................................35
    3.1 kXR_admin Request........................................................................................................35
    3.2 kXR_auth Request..........................................................................................................36
    3.3 kXR_bind Request..........................................................................................................37
    3.4 kXR_chmod Request......................................................................................................38
    3.5 kXR_close Request..........................................................................................................39
    3.6 kXR_dirlist Request........................................................................................................41
    3.7 kXR_endsess Request.....................................................................................................43
    3.8 kXR_getfile Request........................................................................................................45
    3.8.1 Multi-Stream File Retrieval......................................................................................46
    3.9 kXR_login Request..........................................................................................................47
    3.10 kXR_mkdir Request.......................................................................................................51
    3.11 kXR_open Request.........................................................................................................53
    3.12 kXR_open Request.........................................................................................................55
    3.12.1 Passing Opaque Information......................................................................................58
    3.13 kXR_ping Request.........................................................................................................59
    3.14 kXR_prepare Request....................................................................................................61
    3.15 kXR_protocol Request..................................................................................................63
    3.16 kXR_putfile Request......................................................................................................64
    3.16.1 Multi-Stream File Storage.........................................................................................65
    3.17 kXR_query Request........................................................................................................67
    3.17.1 KXR_query Checksum Cancellation Request............................................................69
    3.17.2 KXR_query Checksum Request................................................................................71
    3.17.3 KXR_query Configuration Request............................................................................73
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.17.4</td>
<td>KXR_query Statistics Request</td>
<td>75</td>
</tr>
<tr>
<td>3.18</td>
<td>kXR_read Request</td>
<td>85</td>
</tr>
<tr>
<td>3.19</td>
<td>kXR_readv Request</td>
<td>89</td>
</tr>
<tr>
<td>3.20</td>
<td>kXR_rm Request</td>
<td>91</td>
</tr>
<tr>
<td>3.21</td>
<td>kXR_rmdir Request</td>
<td>92</td>
</tr>
<tr>
<td>3.22</td>
<td>kXR_set Request</td>
<td>93</td>
</tr>
<tr>
<td>3.22.1</td>
<td>Valid kXR_Set Values</td>
<td>94</td>
</tr>
<tr>
<td>3.23</td>
<td>kXR_stat Request</td>
<td>95</td>
</tr>
<tr>
<td>3.24</td>
<td>kXR_statx Request</td>
<td>97</td>
</tr>
<tr>
<td>3.25</td>
<td>kXR_sync Request</td>
<td>99</td>
</tr>
<tr>
<td>3.26</td>
<td>kXR_unbind Request</td>
<td>100</td>
</tr>
<tr>
<td>3.27</td>
<td>kXR_write Request</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>Local Socket Administrative Protocol</td>
<td>103</td>
</tr>
<tr>
<td>4.1</td>
<td>Initiating an Administrative Session</td>
<td>103</td>
</tr>
<tr>
<td>4.2</td>
<td>General Request Format</td>
<td>103</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Request Target Format</td>
<td>104</td>
</tr>
<tr>
<td>4.2.1.1</td>
<td>Connection name format</td>
<td>104</td>
</tr>
<tr>
<td>4.3</td>
<td>General Response Format</td>
<td>105</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Error Response Format</td>
<td>105</td>
</tr>
<tr>
<td>4.4</td>
<td>Abort request for kXR_asyncab Client Action</td>
<td>106</td>
</tr>
<tr>
<td>4.5</td>
<td>Close request</td>
<td>107</td>
</tr>
<tr>
<td>4.6</td>
<td>cj request</td>
<td>108</td>
</tr>
<tr>
<td>4.7</td>
<td>Cont request for kXR_asyncgo Client Action</td>
<td>109</td>
</tr>
<tr>
<td>4.8</td>
<td>Disc request for kXR_asyncdi Client Action</td>
<td>110</td>
</tr>
<tr>
<td>4.9</td>
<td>Login request (mandatory)</td>
<td>111</td>
</tr>
<tr>
<td>4.10</td>
<td>Lsc request</td>
<td>112</td>
</tr>
<tr>
<td>4.11</td>
<td>Lsd request</td>
<td>113</td>
</tr>
<tr>
<td>4.12</td>
<td>Lsj request</td>
<td>115</td>
</tr>
<tr>
<td>4.13</td>
<td>Msg request for kXR_asyncms Client Action</td>
<td>116</td>
</tr>
<tr>
<td>4.14</td>
<td>Pause request for kXR_asyncwt Client Action</td>
<td>117</td>
</tr>
<tr>
<td>4.15</td>
<td>Redirect request for kXR_asyncrd Client Action</td>
<td>118</td>
</tr>
<tr>
<td>5</td>
<td>Document Change History</td>
<td>119</td>
</tr>
</tbody>
</table>
2 Request/Response Protocol

2.1 Format of Client-Server Initial Handshake

When a client first connects to the XRootd server, it must perform a special handshake. This handshake will determine whether the client is communicating with an XRootd server or a rootd server.

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

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_int32</td>
<td>0</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>0</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>0</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>4 (network byte order)</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>2012 (network byte order)</td>
</tr>
</tbody>
</table>
```

The first twelve bytes are zero. The next eight bytes correspond to a standard rootd server protocol request (i.e., kROOTD_PROTOCOL). Both, rootd and XRootd, servers will respond, as follows:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>msglen: kXR_int32</td>
<td>8</td>
</tr>
<tr>
<td>msgtype: kXR_int32</td>
<td>2012</td>
</tr>
<tr>
<td>msgval: kXR_int32</td>
<td>pval</td>
</tr>
<tr>
<td>XRootd Response</td>
<td></td>
</tr>
<tr>
<td>streamid: kXR_char</td>
<td>smid[2]</td>
</tr>
<tr>
<td>status: kXR_uint16</td>
<td>0</td>
</tr>
<tr>
<td>msglen: kXR_int32</td>
<td>rlen</td>
</tr>
<tr>
<td>msgval1: kXR_int32</td>
<td>pval</td>
</tr>
<tr>
<td>msgval2: kXR_int32</td>
<td>flag</td>
</tr>
</tbody>
</table>
```

Where:

- `smid` is the initial streamid. The `smid` for the initial response is always two null characters (i.e., '\0');
- `rlen` is the binary response length (e.g., 8 for the indicated response).
- `pval` is the binary protocol version number.
- `flag` is additional bit-encoded information about the server; as follows:
  - `kXR_DataServer` - This is a data server.
  - `KXR_LBalancer` - This is a load-balancing server.
Notes

1) All binary fields are transmitted in network byte order using an explicit length. The \texttt{kXR\_char} and \texttt{kXR\_unt16} data types are treated as \texttt{unsigned} values. All reserved fields must be initialized to binary zero.

2) The first four bytes of the reply determine whether a client is communicating with \texttt{rootd} (has a value of 8) or \texttt{XRootd} (has a value of 0).

3) All twenty bytes must be received by the server at one time. All known TCP implementations will guarantee that the first message is sent intact if all twenty bytes are sent in a single system call. Using multiple system calls for the first message may cause unpredictable results.

2.2 Data Serialization

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

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

<table>
<thead>
<tr>
<th>XRootd Type</th>
<th>Sign</th>
<th>Bit Length</th>
<th>Bit Alignment</th>
<th>Typical Host Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char8</td>
<td>unsigned</td>
<td>8</td>
<td>8</td>
<td>unsigned char</td>
</tr>
<tr>
<td>kXR_unt16</td>
<td>unsigned</td>
<td>16</td>
<td>16</td>
<td>unsigned short</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>signed</td>
<td>32</td>
<td>32</td>
<td>long(^1)</td>
</tr>
<tr>
<td>kXR_int64</td>
<td>signed</td>
<td>64</td>
<td>64</td>
<td>long long</td>
</tr>
</tbody>
</table>

Table 1: XRootd Protocol Data Types

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

\(^1\) As of this writing, the long type has taken on several meanings for 64-bit architectures. Some machines define a long to be 64-bits and int 32-bits while some others reverse the definition.
unsigned long long htonl(unsigned long long x)
{ unsigned long long ret_val;
    if (isLittleEndian())
        { *( (unsigned long *)(&ret_val) + 1) =
            htonl(*( (unsigned long *)(&x)));
            *( ((unsigned long *)(&ret_val))) =
                htonl(*((unsigned long *)(&x)+1));
        } else {
            *( (unsigned long *)(&ret_val)) =
                htonl(*( (unsigned long *)(&x)));
            *( ((unsigned long *)(&ret_val)) + 1) =
                htonl(*((unsigned long *)(&x)+1));
        }
    return ret_val;
};

Figure 1: POSIX Host to Network Byte Order Serialization

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

Figure 2: POSIX Network to Host Byte Order Serialization
More compact and efficient, though OS restricted (i.e., Solaris and Linux), versions of 64-bit network byte ordering routines are given in the following figure.

```c
#if defined(__sparc) || __BYTE_ORDER==_BIG_ENDIAN
#else
#endif
#define htonl(x) __bswap_64(x)
#endif
#define ntohl(x) __bswap_64(x)
```

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_unr16  requestid
kXR_char  parms[16]
kXR_int32  dlen
kXR_char  data[dlen]
```

Where:

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

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

**parms**
- are parameters specific to the `requestid`.

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

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

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The `kXR_char` and `kXR_int16` data types are treated as *unsigned* values. All reserved fields must be initialized to binary zero.

2) All XRootd client requests consist of a standard 24-byte fixed length message. The 24-byte header may then be optionally followed by request specific data.

3) Stream id’s are arbitrary and are assigned by the client. Typically these id’s correspond to logical connections multiplexed over a physical connection established to a particular server.
4) The client may send any number of requests to the same server. The order in which requests are performed is undefined. Therefore, each request should have a different streamid so that returned results may be paired up with associated requests.

5) Requests sent by a client over a single physical connection may be processed in an arbitrary order. Therefore the client is responsible for serializing requests, as needed.

2.3.1 Valid Client Requests

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_admin</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>args</td>
</tr>
<tr>
<td>kXR_auth</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>authtype, authinfo</td>
</tr>
<tr>
<td>KXR_bind</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>sessid</td>
</tr>
<tr>
<td>kXR_chmod</td>
<td>y</td>
<td>y</td>
<td>yes</td>
<td>mode, path</td>
</tr>
<tr>
<td>kXR_close</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>fdnum</td>
</tr>
<tr>
<td>KXR_dirlist</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>path</td>
</tr>
<tr>
<td>KXR_endsess</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>sessid</td>
</tr>
<tr>
<td>kXR_getfile</td>
<td>y</td>
<td>y</td>
<td>y*</td>
<td>path</td>
</tr>
<tr>
<td>kXR_login</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>userid, token</td>
</tr>
<tr>
<td>kXR_ls</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>options path</td>
</tr>
<tr>
<td>kXR_mkdir</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>mode, path</td>
</tr>
<tr>
<td>kXR_mv</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>old_name, new_name</td>
</tr>
<tr>
<td>kXR_open</td>
<td>y</td>
<td>y</td>
<td>y*</td>
<td>mode, flags, path</td>
</tr>
<tr>
<td>kXR_ping</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>kXR_prepare</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>paths</td>
</tr>
<tr>
<td>kXR_protocol</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>kXR_putfile</td>
<td>y</td>
<td>y</td>
<td>y*</td>
<td>mode, flags, path</td>
</tr>
<tr>
<td>kXR_query</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>args</td>
</tr>
<tr>
<td>kXR_read</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>fdnum, length, offset</td>
</tr>
<tr>
<td>kXR_rmdir</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>path</td>
</tr>
<tr>
<td>kXR_rmdir</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>path</td>
</tr>
<tr>
<td>kXR_set</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>info</td>
</tr>
<tr>
<td>kXR_stat</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>path</td>
</tr>
<tr>
<td>kXR_statx</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>pathlist</td>
</tr>
<tr>
<td>kXR_write</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>fdnum, length, offset, data</td>
</tr>
</tbody>
</table>

Table 2: Valid Client Requests

*Only these requests may redirect a client from an XRootd server to a rootd server.*
2.3.2 Valid Client Paths

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

Path names are restricted to the following set of characters:

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

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

2.3.3 Client Recovery From Server Failures

A server failure should be recognized when the server unexpectedly closes its TCP/IP connection. Should this happen, the client may recover all operations by treating the termination of the connection as a redirection request (see page 30) to the initial XRootd server for all streams associated with the closed TCP/IP connections.

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.
2.3.4 Client Recovery From File Location Failures

A file location failure should be recognized when a server returns \texttt{kXR\_error} status code with a \texttt{kXR\_NotFound} error code and is either a load balancing server or the target of a redirect. The recovery steps are:

1. The client should contact the last load balancer used.
2. For \texttt{kXR\_Open} requests, the request should be re-issued with the \texttt{kXR\_refresh} option set. For other requests, no recovery is currently possible.
3. If the same result is encountered again, the client should consider the file missing and not attempt any further recovery actions.
2.4 Server Response Format

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

```
kXR_char streamid[2]
kXR_uint16 status
kXR_int32 dlen
kXR_char data[dlen]
```

Where:

- `streamid` is the binary identifier that is associated with this request stream corresponding to a previous request.
- `status` is the binary status code indicating how the request completed. The next section describes possible status codes.
- `dlen` is the binary length of the `data` portion of the message. If no data is present, then the value is zero.
- `data` are data specific to the `requestid`. Not all responses have associated data. If the response does have data, the length of this field is recorded in the `dlen` field.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The `kXR_char` and `kXR_uint16` data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the `streamid` value.
3) Unsolicited responses are server requests for client configuration changes to make better use of the overall system. Since these responses do not correspond to any request, the `streamid` value has no meaning.
4) Unsolicited responses must be immediately acted upon. They should not be paired with any previous request.
2.4.1 Valid Server Response Status Codes

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

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

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

<table>
<thead>
<tr>
<th>kXR_char</th>
<th>pad[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_un16</td>
<td>kXR_attn</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>plen</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>actnum</td>
</tr>
<tr>
<td>kXR_char</td>
<td>parms[plen-4]</td>
</tr>
</tbody>
</table>

Where:

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

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

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

- kXR_asyncav - The file or file(s) the client previously requested to be prepared are now available.
- kXR_asyncab - The client should immediately disconnect (i.e., close the socket connection) from the server and abort further execution.
- kXR_asyncdi - The client should immediately disconnect (i.e., close the socket connection) from the server. Parameters indicate when a reconnect may be attempted.
- kXR_asyncgo - The client may start sending requests. This code is sent to cancel the effects of a previous kXR_asyncwt code.
- kXR_asyncmsg - The client should send the indicated message to the console. The parameters contain the message text.
- kXR_asyncrd - The client should immediately disconnect (i.e., close the socket connection) and reconnect to the indicated server.
- kXR_asyncresp - The client should use the response data in the message to complete the request associated with the indicated streamid.
- kXR_asyncnav - The file or file(s) the client previously requested to be prepared cannot be made available.
- kXR_asyncwt - The client should hold off sending any new requests until the indicated amount of time has passed or until receiving a kXR_asyncgo action code.
**parms** is the parameter data, if any, that is to steer client action.

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The `kXR_char` and `kXR_und16` data types are treated as `unsigned` values. All reserved fields must be initialized to binary zero.

2) Servers use the `kXR_attn` response code to optimize overall system performance and to notify clients of any impending events. All responses except for `kXR_asynresp`, do not correspond to any client request and should not be paired up with any request.

3) When `kXR_attn` is received, the client must perform the requested action and indicated by the `actnum` value.
2.4.2.1 Server kXR_attn Response for kXR_asyncab Client Action

```
kXR_char      pad[2]
kXR_uint16    kXR_attn
kXR_int32     mlen
kXR_int32     kXR_asyncab
kXR_char      msg[mlen-4]
```

Where:

- `mlen` is the binary length of the following action code and message.
- `msg` is the message to be sent to the terminal. The `mlen` value, less four, indicates the length of the message. The ending null byte (‘\0’) is transmitted and included in the message length.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The `kXR_char` and `kXR_uint16` data types are treated as unsigned values. All reserved fields must be initialized to binary zero.

2) Servers use the `kXR_attn` response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.

3) When `kXR_attn` is received with the `kXR_asyncab` action code, the client should close all physical connections, write the message (msg), if any, to standard error, and terminate execution.
2.4.2.2 Server kXR_attn Response for kXR_asyncdi Client Action

<table>
<thead>
<tr>
<th>kXR_char</th>
<th>pad[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16</td>
<td>kXR_attn</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>12</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>kXR_asyncdi</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>wsec</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>msec</td>
</tr>
</tbody>
</table>

Where:

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

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

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.

2) Servers use the kXR_attn response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.

3) When kXR_attn is received with the kXR_asyncdi action code, the client should close the physical connection, wait \( wsec \) seconds, and attempt to reconnect to the server.

4) If a server reconnect fails, the client should wait either an additional \( wsec \) seconds or some other predetermined time and try again. If \( msec \) seconds have gone since the initial wait and the client has not reconnected to the server, a reconnect failure should be declared.

5) When a reconnect failure is declared, the client may either terminate the program or perform an internal redirection to a load-balancing server.

6) A reconnect is essentially a delayed redirect to the same server. The actions that must be carried out when reconnecting are identical to those that must be performed when reconnecting to a different server. Refer to the description of the kXR_asyncrd action for the set steps that the client must take to successfully reconnect.
2.4.2.3 Server kXR_attn Response for kXR_asyncgo Client Action

```
kXR_char    pad[2]
kXR_uint16  kXR_attn
kXR_int32   4
kXR_int32   kXR_asyncgo
```

Notes
1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.
2) Servers use the kXR_attn response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.
3) When kXR_attn is received with the kXR_asyncgo action code, the client may resume sending requests to the server.
4) The kXR_asyncgo code is sent to cancel the effects of a previously sent kXR_asyncwct code. Therefore, if the client is still waiting for the kXR_asyncwct interval to expire, the interval should be cancelled.
### 2.4.2.4 Server kXR_attn Response for kXR_asyncms Client Action

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>pad[2]</td>
</tr>
<tr>
<td>kXR_uint16</td>
<td>kXR_attn</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>mlen</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>kXR_asyncms</td>
</tr>
<tr>
<td>kXR_char</td>
<td>msg[mlen-4]</td>
</tr>
</tbody>
</table>

Where:

- `mlen` is the binary length of the following action code and message.
- `msg` is the message to be sent to the terminal. The `mlen` value, less four, indicates the length of the message. The ending null byte (`\0`) is transmitted and included in the message length.

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The `kXR_char` and `kXR_uint16` data types are treated as unsigned values. All reserved fields must be initialized to binary zero.

2) Servers use the `kXR_attn` response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.

3) When `kXR_attn` is received with the `kXR_asyncms` action code, the client should simply write the indicated message to the terminal.
2.4.2.5 Server kXR_attn Response for kXR_asyncrd Client Action

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

Where:

- `plen` is the binary length of the parameter portion of the message (i.e., the subsequent bytes).
- `port` is the binary port number to which the client must connect. If the value is zero, the default XRootd port number must be used.
- `host` is the ASCII name of the to which the client must connect. The `host` does not end with a null (\0) byte.
- `token` is an optional ASCII token that, when present, must be delivered to the new host during the login phase. The `token`, if present, is separated from the `host` by a single question mark. The `token` does not end with a null (\0) byte.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The `kXR_char` and `kXR_unt16` data types are treated as unsigned values. All reserved fields must be initialized to binary zero.
2) Servers use the `kXR_attn` response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.
3) When `kXR_attn` is received with the `kXR_asyncrd` action code, the client should perform the following steps:
   a. Decompose the response to extract the port number, host name, and possible token value.
   b. Physically close the connection to the current host, regardless of type.
   c. Establish a new physical connection with the indicated host at the specified or default port number.
d. Perform the initial handshake, login with token (see \texttt{kXR\_login} description), and authentication (see \texttt{kXR\_auth} description).

e. Re-establish all open files, as needed. Previously opened files may be re-opened all at once or when a request attempts to use the file.

f. Re-issue any requests that were sent to the previous server but have not received a response.

4) Since \texttt{XRootd} allows multiple open files per physical connection, a \texttt{kXR\_asyncrd} response can become somewhat complicated to handle. The client can re-open files immediately after a new connection is made or can re-open files as they are needed. In either case, the client must:

g. Issue a \texttt{kXR\_open} request using the same file name and options as was originally used.

h. Use the returned file handle for all subsequent requests for that file (i.e., substitute the new \texttt{fhandle} for the old \texttt{fhandle}).

5) An \texttt{XRootd} server will never redirect a physical connection to a \texttt{rootd} server. This differs for logical connections where a logical connection may be so redirected.

6) After 256 redirect responses within 10 minutes on the same physical connection, the client should declare an internal system error since it is obvious that effective work is not being performed.
2.4.2.6 Server kXR_attn Response for kXR_asynresp Client Action

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

Where:

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

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

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

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

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

**Notes**

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

2) Servers use the `kXR_attn` response code to optimize overall system performance and to notify clients of any impending events.

3) Unlike other asynchronous events, this response is associated with a previous request and the response data must be used to complete that request.

4) The *rlen-dlen* is always 16.

5) When `kXR_attn` is received with the `kXR_asynresp` action code, the client should remove the request paired with `streamid` from wait state and complete it using the response data.
2.4.2.7 Server kXR_attn Response for kXR_asyncwt Client Action

```
  kXR_char   pad[2]
kXR_uint16  kXR_attn
kXR_int32   8
kXR_int32   kXR_asyncwt
kXR_int32   wsec
```

Where:

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

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_uint16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

2) Servers use the **kXR_attn** response code to optimize overall system performance and to notify clients of any impending events. This response does not correspond to any client request and should not be paired up with any request.

3) When **kXR_attn** is received with the **kXR_asyncwt** action code, the client should queue any new requests (i.e., not send new requests) until **wsec** seconds have elapsed.

4) While waiting, the client should still be receiving messages from the server. It is possible for the server to send additional unsolicited responses even after a **kXR_asyncwt** has been sent. For example, the server may send a **kXR_asyncgo** request to cancel the effects of the **kXR_asyncwt** request before the **wsec** interval has gone by.
2.4.3 Server kXR_authmore Response Format

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>streamid[2]</td>
</tr>
<tr>
<td>kXR_uint16</td>
<td>kXR_authmore</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>dlen</td>
</tr>
<tr>
<td>kXR_char</td>
<td>data[dlen]</td>
</tr>
</tbody>
</table>

Where:

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

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

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

**Notes**

5) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.

6) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the streamid value.

7) The kXR_authmore response code is issued only for those authentication schemes that require several handshakes in order to complete (e.g., .x500).

8) When an kXR_authmore response is received, the client must call the appropriate authentication continuation method and pass it *data*, if present. The output of the continuation method should be sent to the server using another kXR_auth request. This handshakes continues until either the continuation method fails or the server returns a status code of kXR_error or kXR_ok.
2.4.4 Server kXR_error Response Format

```
kXR_char  streamid[2]
kXR_uint16 kXR_error
kXR_int32  dlen
kXR_int32  errnum
kXR_char  errmsg[dlen-4]
```

Where:

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

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

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

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

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The `kXR_char` and `kXR_uint16` data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

2) Since the error message is null-terminated, `dlen` includes the null byte in it’s count of bytes that were sent.

3) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the `streamid` value.
2.4.4.1 Server kXR_error Sub-Codes

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

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_ArgInvalid</td>
<td>A request argument was not valid</td>
</tr>
<tr>
<td>kXR_ArgMissing</td>
<td>Required request argument was not provided</td>
</tr>
<tr>
<td>kXR_ArgTooLong</td>
<td>A request argument was too long (e.g., path)</td>
</tr>
<tr>
<td>kXR_Cancelled</td>
<td>The operation was cancelled by the administrator</td>
</tr>
<tr>
<td>kXR_FileLocked</td>
<td>File is locked, open request was rejected</td>
</tr>
<tr>
<td>kXR_FileNotOpen</td>
<td>File if not open for the request (e.g., read)</td>
</tr>
<tr>
<td>kXR_FSError</td>
<td>The file system indicated an error</td>
</tr>
<tr>
<td>kXR_InvalidRequest</td>
<td>The request code is invalid</td>
</tr>
<tr>
<td>kXR_IOError</td>
<td>An I/O error has occurred</td>
</tr>
<tr>
<td>kXR_isDirectory</td>
<td>Object being opened with <code>kXR_open</code> is a directory</td>
</tr>
<tr>
<td>kXR_NoMemory</td>
<td>Insufficient memory to complete the request</td>
</tr>
<tr>
<td>kXR_NoSpace</td>
<td>Insufficient disk space to write data</td>
</tr>
<tr>
<td>kXR_NotAuthorized</td>
<td>Client is not authorized for the request</td>
</tr>
<tr>
<td>kXR_NotFile</td>
<td>The object being opened with <code>kXR_open</code> is not a file.</td>
</tr>
<tr>
<td>kXR_NotFound</td>
<td>The requested file was not found</td>
</tr>
<tr>
<td>kXR_noserver</td>
<td>There are no servers available to process the request</td>
</tr>
<tr>
<td>kXR_ServerError</td>
<td>An internal server error has occurred</td>
</tr>
<tr>
<td>kXR Unsupported</td>
<td>The request is valid but not supported</td>
</tr>
</tbody>
</table>
2.4.5 Server kXR_ok Response Format

<table>
<thead>
<tr>
<th>kXR_char</th>
<th>streamid[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16</td>
<td>kXR_ok</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>dlen</td>
</tr>
<tr>
<td>kXR_char</td>
<td>data[dlen]</td>
</tr>
</tbody>
</table>

Where:

*streamid*

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

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

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

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the streamid value.

3) The kXR_ok response indicates that the request fully completed and no additional responses will be forthcoming.
2.4.6 Server kXR_oksofar Response Format

<table>
<thead>
<tr>
<th>kXR_char</th>
<th>streamid[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16</td>
<td>kXR_oksofar</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>dlen</td>
</tr>
<tr>
<td>kXR_char</td>
<td>data[dlen]</td>
</tr>
</tbody>
</table>

Where:

streamid

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

dlen

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

data

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

Notes

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

2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the streamid value.

3) The kXR_oksofar response indicates that the server is providing partial results and the client should be prepared to receive additional responses on the same stream. This response is primarily used when a read request would transmit more data than the internal server segment size. Refer to the kXR_getfile and kXR_read requests.

4) Sending requests using the same streamid when a kXR_oksofar status code has been returned may produced unpredictable results. A client must serialize all requests using the streamid in the presence of partial results.

5) Any status code other than kXR_oksofar indicates the end of transmission
2.4.7 Server kXR_redirect Response Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>streamid[2]</td>
</tr>
<tr>
<td>kXR_uint16</td>
<td>kXR_redirect</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>dlen</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>port</td>
</tr>
<tr>
<td>kXR_char</td>
<td>host[?opaque][?token][dlen-4]</td>
</tr>
</tbody>
</table>

Where:

- **streamid** is the binary identifier that is associated with this request stream corresponding to a previous request.
- **dlen** is the binary length of the *data* portion of the message (i.e., the subsequent bytes).
- **port** is the binary port number to which the client must connect. If the value is zero, the default *XRootd* port number must be used.
- **host** is the ASCII name of the to which the client must connect. The *host* does not end with a null (\0) byte.
- **opaque** is an optional ASCII token that, when present, must be delivered to the new host as opaque information added to the file name\(^2\) associated with the operation being redirected. The *opaque*, if present, is separated from the *host* by a single question mark. The *opaque* does not end with a null (\0) byte but may end with a question mark (see *token* below). Therefore, *opaque* may never contain a question mark.
- **token** is an optional ASCII token that, when present, must be delivered to the new host during the login phase. The *token*, if present, is separated from the *host* by a two question marks. The first question mark may be followed by *opaque* information. If none is present, another question mark immediately follows the first one. The *token* does not end with a null (\0) byte.

\(^2\) In the case of *kXR_mv*, two file names are present. The opaque information must be added to the second of the two file names.
Notes
1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_unt16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the streamid value.
3) After 256 redirect responses within 10 minutes on the same logical connection, the client should declare an internal system error since it is obvious that effective work is not being performed.
4) The client must be prepared to handle a redirect response at any time. A redirect response requires that the client
   i. Decompose the response to extract the port number, host name, and possible token value.
   j. Logically close the connection of the current host, if the current host is a data server. Otherwise, if this is the first load-balancing server encountered in the operation sequence, the connection should remain open since a load-balancing server always responds with a redirect.
   k. Establish a new logical connection with the indicated host at the specified or default port number.
   l. Perform the initial handshake, login with token (see kXR_login description), and authentication (see kXR_auth description).
   m. If the redirection occurred for a request using a file handle (i.e., fhandle) then a new file handle must be obtained.
      i. A kXR_open request must be issued using the same file name and options as was originally used.
      ii. The returned file handle must be used for the request that is to be re-issued as well as all subsequent requests relating to the file.
   n. Re-issue the request that was redirected.
5) An XRootd server may redirect a client to a rootd server only for certain types of requests. A client should declare an internal system error if the redirection rules in the previous table describing valid client requests is violated.
6) Opaque data must be treated as truly opaque. The client should not inspect nor modify the data in any way.
2.4.8 Server kXR_wait Response Format

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>streamid[2]</td>
</tr>
<tr>
<td>kXR_unt16</td>
<td>kXR_wait</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>dlen</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>seconds</td>
</tr>
<tr>
<td>kXR_char</td>
<td>infomsg[dlen-4]</td>
</tr>
</tbody>
</table>
```

Where:

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

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

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

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

Notes

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

2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the streamid value.

3) The client should wait the indicated number of seconds and retry the request.

4) Nothing prohibits the client from waiting for less time than the indicated number of seconds.
2.4.9 Server kXR_waitresp Response Format

```
kXR_char  streamid[2]
kXR_uint16 kXR_waitresp
kXR_int32 4
kXR_int32 seconds
```

Where:

streamid

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

seconds

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

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.
2) Since requests may be completed in any order, the ordering of responses is undefined. The client must appropriately pair responses with requests using the streamid value.
3) The client should wait the indicated number of seconds for the response. The response will be returned via an unsolicited response (kXR_attn with kXR_asynresp) at some later time which may be earlier than the time indicated in seconds. When the response arrives, the client must use the response data to complete the request that received the kXR_waitresp.
4) Nothing prohibits the client from waiting for different time than the indicated number of seconds. Generally, if no response is received after at least seconds have elapsed; the client should treat the condition as a fatal error.
3 Detailed Protocol Specifications

3.1 kXR_admin Request

Purpose: Perform an administrative function.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16 kXR_admin</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char reserved[16]</td>
<td>kXR_int32 ilen</td>
</tr>
<tr>
<td>kXR_int32 rlen</td>
<td>kXR_char resp[ilen]</td>
</tr>
<tr>
<td>kXR_char reqs[rlen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

streamid

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

reserved

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

rlen

is the binary length of the supplied request, reqs.

reqs

is the request.

ilen

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

resp

is the response to the administrative request.

Notes

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

2) The kXR_admin request is only valid for users who have successfully performed a kXR_login operation in an administrative role (i.e., logged in as administrator).

3) This request type is not currently supported. Use the local socket interface protocol to execute administrative requests.
3.2 kXR_auth Request

**Purpose:** Authenticate client’s username to the server.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_auth</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char reserved[16]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_int32 credlen</td>
<td></td>
</tr>
<tr>
<td>kXR_char cred[credlen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

- **streamid**
  - is the binary identifier that is associated with this request stream. This identifier will be echoed as kXR_int32 with any response to the request.

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

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

- **cred**
  - are the credentials used to provide authentication information.

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_uint16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

2) Authentication credentials may be supplied by many means. The common mechanism used by **XRootd** is to use the classes in the **libXrdSec.so** library. See the “Authentication & Access Control Configuration Reference” for more information.

3) Refer to the Authentication section of this document on how a client authenticates to an **XRootd** server.
3.3 kXRBind Request

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16 kXR_bind</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char sessid[16]</td>
<td>kXR_int32 1</td>
</tr>
<tr>
<td>kXR_int32 0</td>
<td>kXR_char pathid</td>
</tr>
</tbody>
</table>

Where:

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

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

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

Notes

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

2) The *sessid* value should be treated as opaque data.

3) The socket issuing the *kXR_bind* request must neither have a session id (i.e., be logged in) nor be already bound.

4) Once a socket is bound to a session, if may only supply data for *kXR_write* requests or receive responses for *kXR_read* and *kXR_readv* requests.

5) Should the client close a bound socket, the client should issue a *kXR_unbind* request specifying the *pathid* of the socket that was just closed. Failure to do so may cause future *kXR_bind* requests to fail.

6) Each login session is limited to the number of bound sockets. Use the *kXR_Qconfig* sub-request code of *kXR_query* to determine the maximum number of sockets that can be bound to a login session.

7) Bound sockets are meant to support parallel data transfer requests across wide-area networks.
3.4 kXR_chmod Request

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_chmod</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char reserved[14]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_uint16 mode</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td></td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

mode is the access mode to be set for path. The access mode is an “or’d” combination of the following values:

<table>
<thead>
<tr>
<th>Access</th>
<th>Readable</th>
<th>Writeable</th>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>kXR_ur</td>
<td>kXR_uw</td>
<td>not supported</td>
</tr>
<tr>
<td>Group</td>
<td>kXR_gr</td>
<td>kXR_gw</td>
<td>not supported</td>
</tr>
<tr>
<td>Other</td>
<td>kXR_or</td>
<td>kXR_ow</td>
<td>not supported</td>
</tr>
</tbody>
</table>

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

path is the path whose mode is to be set.

Notes
1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.
2) No umask is applied to the specified mode.
3.5 kXR_close Request

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16 kXR_close</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char fhandle[4]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_char reserved[12]</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 0</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

**Notes**

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

9) The fhandle value should be treated as opaque data.
3.6 kXR_dirlist Request

Purpose: Enumerate the contents of a directory.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_un16 kXR_dirlist</td>
<td>kXR_un16 0</td>
</tr>
<tr>
<td>kXR_char reserved[16]</td>
<td>kXR_int32 dlen</td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td>kXR_char dirname\n</td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>kXR_char 0</td>
</tr>
</tbody>
</table>

Where:

streamid

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

reserved

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

plen

is the binary length of the supplied path, path.

path

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

dlen

is the binary length of the data that follows dlen.

dirent

is an entry in the directory whose listing was requested.

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_un16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.

2) A directory may have multiple entries and the response contains all of the entries.

3) Each directory entry is suffixed with a new-line character; except for the last entry which is suffixed by a null character.
4) Since more entries may exist than is possible to send at one time, the kXR\_oksofar protocol may be used to segment the response. Under no circumstances will a directory name be split across a response packet.
5) The server does not return the entries “.” And “..”.
6) An empty directory will return the eight-byte triplet \{streamid, 0, 0\}. 
3.7 kXR_endsess Request

**Purpose:** Terminate a pre-existing session.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>streamid[2]</td>
</tr>
<tr>
<td>kXR_unt16</td>
<td>kXR_endsess</td>
</tr>
<tr>
<td>kXR_char</td>
<td>sessid[16]</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>0</td>
</tr>
</tbody>
</table>

Where:

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

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

**Notes**

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

2) The sessid value should be treated as opaque data.

3) The socket issuing the kXR_endsess request must be logged in and, optionally, authenticated.

4) If the sessid is all binary zeroes, the current session is terminated.

5) The server verifies that the process presenting the sessid actually received it on a previous kXR_login.
3.8 kXR_getfile Request

**Purpose:** Retrieve a complete file.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_getfile</td>
<td>kXR_uint16 status</td>
</tr>
<tr>
<td>kXR_int32 options</td>
<td>kXR_int32 dlen</td>
</tr>
<tr>
<td>kXR_char reserved[8]</td>
<td>kXR_int64 offset</td>
</tr>
<tr>
<td>kXR_int32 buffsz</td>
<td>kXR_char data[dlen-8]</td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td></td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

**streamid**

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

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

- **kXR_ok** - All of the data has been transmitted with error.
- **kXR_oksofar** - Partial data has been transmitted without error; additional data should be expected on this stream.

**options**

is a bit vector representing the options that are to apply to the file transfer. The valid set of options are:

- **kXR_md5file** - Compute and transmit an MD5 checksum for the file.
- **kXR_md5block** - Compute and transmit an MD5 checksum for each block.

**reserved**

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

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

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

**path** is the path of the file to be retrieved.
**kXR_getfile**

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

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

*data* is the data associated with the file.

**Notes**

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

2) Since a file may be much larger than the allowable buffer size, the file is sent in *buffsz* segments until the whole file is sent. This is accomplished using the *kXR_oksofar* status code. Each subsequent data segment is transmitted using a `{*streamid*, *status*, *dlen*, *offset*, *data*}` response. The last segment is indicated by a *kXR_ok*, if no error occurred.

3) Any status code other than *kXR_oksofar* indicates the end of transmission.

4) Sending requests using the same *streamid* when a *kXR_oksofar* status code has been returned may produced unpredictable results. A client must serialize all requests using the *streamid* in the presence of partial results.

5) When a 16-byte MD5 checksum is requested, it is transmitted either after the complete file is transferred or after each block, as specified by the options. An MD5 checksum will have a *dlen* of 24 and an *offset* of negative one (i.e., -1).

6) MD5 block checksums are always sent on the same TCP/IP connection that was used to send the block.

7) An empty file will return the eight-byte triplet `{*streamid*, 0, 0}`.

8) Empty files will not transmit MD5 checksums, even when so requested.

9) This request type is not currently supported.

### 3.8.1 Multi-Stream File Retrieval

*To be written.*
3.9 kXR_login Request

Purpose: Initialize a server connection.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response (server &lt; 2.4.0</th>
<th>client &lt; 1.0)</th>
<th>Normal Response (server &gt;= 2.4.0 &amp; client &gt; 0.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_login</td>
<td>kXR_uint16 0</td>
<td>kXR_char sec[slen]</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 pid</td>
<td>kXR_int32 slen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXR_char username[8]</td>
<td>kXR_char reserved[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXR_char capver[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXR_char role[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXR_int32 tlen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXR_char token[tlen]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

*pid* is the process number associated with this connection.

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

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

- 0b00vvvvvv - client only supports synchronous responses
- 0b10vvvvvv - (kXR_asycap) client supports asynchronous responses
**kXR_getfile**

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

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

- **token** is the token supplied by the previous redirection response that has initiated this login request.

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

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

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

**Notes**

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

2) If no security information is returned (i.e., *slen* is zero), the XRootd server does not require that the client authenticate.

3) If security information is returned, then the client must create the security context allowed by the security information, obtain credentials, and send them using an `kXR_auth` request.

4) Authentication must occur prior to any operation that requires authentication. See the table on page 10 for a list of requests that must be authenticated.

5) Logging in as an administrator suppresses any redirection attempts and limits the request set to `kXR_auth` and `kXR_admin`.

6) A subsequent `kXR_auth` request may revert the login into a normal user login should `xrootd` find that the authenticated user cannot assume the role of administrator.

7) Logging in as a normal user prohibits the use of the `kXR_admin` request.

8) Sending a `kXR_login` request on a previously authenticated connection destroys the authentication context; requiring that the connection be re-authenticated.

9) The sessid is used in `kXR_bind` and `kXR_endsess` requests,
10) Opaque information must be treated as truly opaque. The client must not inspect nor modify opaque information in any way.
3.10 kXR_mkdir Request

**Purpose:** Create a directory.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>kXR_char</td>
</tr>
<tr>
<td>kXR_unt16</td>
<td>kXR_unt16</td>
</tr>
<tr>
<td>kXR_mkdir</td>
<td>0</td>
</tr>
<tr>
<td>kXR_char</td>
<td>kXR_int32</td>
</tr>
<tr>
<td>options</td>
<td>0</td>
</tr>
<tr>
<td>kXR_char</td>
<td>reserved[13]</td>
</tr>
<tr>
<td>mode</td>
<td></td>
</tr>
<tr>
<td>kXR_int32</td>
<td>plen</td>
</tr>
<tr>
<td>path[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

- **kXR_mkpath** - create directory path if it does not already exist

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

<table>
<thead>
<tr>
<th>Access</th>
<th>Readable</th>
<th>Writeable</th>
<th>Searchable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>kXR_ur</td>
<td>kXR_uw</td>
<td>kXR_ux</td>
</tr>
<tr>
<td>Group</td>
<td>kXR_gr</td>
<td>kXR_gw</td>
<td>kXR_gx</td>
</tr>
<tr>
<td>Other</td>
<td>kXR_or</td>
<td><em>not supported</em></td>
<td>kXR_ox</td>
</tr>
</tbody>
</table>

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

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

Notes

1) All binary fields are transmitted in network byte order using an explicit length. The \texttt{kXR\_char} and \texttt{kXR\_unt16} data types are treated as \texttt{unsigned} values. All reserved fields must be initialized to binary zero.

2) When a directory path is created, as requested by the \texttt{kXR\_mkpath} option, the directory permission specified in \textit{mode} are propagated along the newly created path.

3) No \texttt{umask} applies to the specified mode.
3.11 kXR_mv Request

**Purpose:** Rename a directory or file.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>streamid[2]</td>
</tr>
<tr>
<td>kXR_unt16</td>
<td>kXR_mv</td>
</tr>
<tr>
<td>kXR_char</td>
<td>reserved[16]</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>plen</td>
</tr>
<tr>
<td>kXR_char</td>
<td>path[plen]</td>
</tr>
<tr>
<td>kXR_char</td>
<td>streamid[2]</td>
</tr>
<tr>
<td>kXR_unt16</td>
<td>0</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>0</td>
</tr>
</tbody>
</table>

Where:

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

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

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

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

**Notes**

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

2) Renames across file systems are not supported.
3.12 kXR_open Request

Purpose: Open a file or a communications path.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_open</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_uint16 mode</td>
<td>kXR_int32 resplen</td>
</tr>
<tr>
<td>kXR_uint16 options</td>
<td>kXR_char fhandle[4]</td>
</tr>
<tr>
<td>kXR_char reserved[12]</td>
<td>[kXR_int32 cpsize]</td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td>[kXR_char cptype[4]]</td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td>[kXR_char info[resplen-12]]</td>
</tr>
</tbody>
</table>

Where:

streamid

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

mode is the advisory mode in which path is to be opened. The mode is an “or’d” combination of the following values:

<table>
<thead>
<tr>
<th>Access</th>
<th>Readable</th>
<th>Writeable</th>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>kXR_ur</td>
<td>kXR_uw</td>
<td>kXR_ux</td>
</tr>
<tr>
<td>Group</td>
<td>kXR_gr</td>
<td>kXR_gw</td>
<td>kXR_gx</td>
</tr>
<tr>
<td>Other</td>
<td>kXR_or</td>
<td>not supported</td>
<td>kXR_ox</td>
</tr>
</tbody>
</table>

options are the options to apply when path is opened. The options are an “or’d” combination of the following values:

- kXR_async - open the file for asynchronous i/o (see notes)
- kXR_compress - open a file even when compressed (see notes)
- kXR_delete - open a new file, deleting any existing file
- kXR_force - ignore file usage rules
- kXR_mkpath - create directory path if it does not already exist
- kXR_new - open a new file only if it does not already exist
- kXR_open_apnd - open only for appending
- kXR_open_read - open only for reading
- kXR_open_updt - open for reading and writing
- kXR_refresh - update cached information on the file’s location (see notes)
- kXR_retstat - return file status information in the response.
**kXR_prepare**

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

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

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

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

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

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

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

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

**Notes**

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

2) Open fails if the path designates a directory.

3) No umask applies to the specified mode.
4) The **kXR_async** option tells the server to overlap file i/o with network requests as much as possible for this file. For instance, read requests may be done in parallel with other read requests sent on the same link. This option is only useful if the client is able to issue multiple requests (i.e., is not serializing the requests-response stream).

5) While the **kXR_async** option applies to write operations, as well. Server-side asynchronous opportunities are far more limited. The client needs to perform appropriate multiplexing of write requests with other requests to gain improved parallelism.

6) The **kXR_async** option imposes additional overhead on the server and should only be specified when the client can take advantage of request-response parallelism.

7) The **kXR_refresh** option imposes additional overhead on the server because it requires that the server obtain the most current information on the file’s location before attempting to process the open request. This option should only be used as part of the error recovery process outlined in section “Client Recovery From File Location Failures”.

8) The **kXR_refresh** option is ignored by any server not functioning as a primary redirecting server.

9) When a directory path is created, as requested by the **kXR_mkpath** option, the directory permission of 0775 (i.e., rwxrwxr-x) are propagated along the newly created path.

10) Only files may be opened using the **kXR_open** request code.

11) The **kXR_restat** option is meant to eliminate an additional server request for file status information for applications that always need such information.
3.12.1 Passing Opaque Information

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

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

Where:

- **layer**
  - is the layer to which the directive is sent. Valid layer names are:
    - **ofs** — the logical file system layer
    - **oss** — the physical storage system layer.

- **directive**
  - is the name of the specific directive

- **arg**
  - are directive-specific arguments.

**Notes**

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

**Example**

```
&ooss.cgroup=index&oofs.snotify=120,msg,0,imserv,xyzzy
```
3.13 kXR_ping Request

**Purpose:** Determine if the server is alive.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16 kXR_ping</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char reserved[16]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_int32 0</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

**Notes**

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

2) Use the kXR_ping request to see if the server is running.
3.14 kXR_prepare Request

**Purpose:** Prepare one or more files for access.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_prepare</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char options</td>
<td>kXR_int32 rlen</td>
</tr>
<tr>
<td>kXR_char prty</td>
<td>kXR_char resp[rlen]</td>
</tr>
<tr>
<td>kXR_int32 reserved[14]</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td></td>
</tr>
<tr>
<td>kXR_char plist[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

- **options** are the options to apply to each *path*. The notes explain how these options can be used. The *options* are an “or’d” combination of the following:
  - kXR_cancel - cancel a prepare request
  - kXR_notify - send a message when the file has been processed
  - kXR_noerrs - do not send notification of preparation errors
  - kXR_stage - stage the file to disk if it is not online
  - kXR_wmode - the file will be accessed for modification

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

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

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

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

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

resp is the response to request.

Notes
1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_unlt16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.
2) The kXR_prepare request attempts to make the indicated files available for access. This may require that the files be brought in from a Mass Storage device.
3) The kXR_prepare request always executes asynchronously. Therefore, unless there are obvious errors in the request, a successful status code is immediately returned.
4) The system makes no guarantees that the files will be made available for access ahead of a future kXR_open request. Hence, the kXR_prepare request is treated as merely a hint.
5) The kXR_prepare request should normally be directed to a load-balancing server should one be present.
6) The when the prepare request has been accepted in the presence of the kXR_stage option, the server returns a request locator (i.e., locatorid) as the normal response. This locatorid should be treated as an opaque ASCII text string. The locatorid can be used to cancel the request at some future time and to pair up asynchronous messages with requests when kXR_notify has been set.
3.15 kXR_protocol Request

**Purpose:** Obtain the protocol version number and type of server.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>streamid[2]</td>
</tr>
<tr>
<td>kXR Uint16</td>
<td>kXR_protocol</td>
</tr>
<tr>
<td>kXR_char</td>
<td>reserved[16]</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>0</td>
</tr>
</tbody>
</table>

Where:

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

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

- **pval** is the binary protocol version number.

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

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.

2) The client should not rely on the response data length being 8. In the future, additional information may be returned.
3.16 kXR_putfile Request

**Purpose:** Store a complete file.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_putfile</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_int32 options</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_char reserved[8]</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 buffsz</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td></td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 dlen</td>
<td></td>
</tr>
<tr>
<td>kXR_int64 offset</td>
<td></td>
</tr>
<tr>
<td>kXR_char data[dlen-8]</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 0</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

*options* is a bit vector representing the options that are to apply to the file transfer. The valid set of options are:
- **kXR_delete** - deleting any existing file
- **kXR_force** - ignore file usage rules
- **kXR_md5file** - Compute and transmit an MD5 checksum for the file.
- **KXR_md5block** - Compute and transmit an MD5 checksum for each block.
- **kXR_new** - create a new file only if it does not already exist

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

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

*plen* is the binary length of the supplied path, *path*.
path is the path of the file to be stored.

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

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

data is the data associated with the file.

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

3.16.1 Multi-Stream File Storage

To be written.
3.17 kXR_query Request

Purpose: Obtain server information.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_query</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char reserved[14]</td>
<td>kXR_int32 len</td>
</tr>
<tr>
<td>kXR_uint16 reqcode</td>
<td>kXR_char info[len]</td>
</tr>
<tr>
<td>kXR_int32 alen</td>
<td></td>
</tr>
<tr>
<td>kXR_char args[alen]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Delayed Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char streamid[2]</td>
</tr>
<tr>
<td>kXR_uint16 kXR_waitresp</td>
</tr>
<tr>
<td>kXR_int32 4</td>
</tr>
<tr>
<td>kXR_int32 seconds</td>
</tr>
</tbody>
</table>

Where:

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

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

- **reqcode**
  - is the binary code indicating the specific query being made. Valid codes are:
    - kXR_Qconfig: Query server configuration
    - kXR_Qckscan: Query file checksum cancellation
    - kXR_Qcksum: Query file checksum
    - kXR_Qprep: Query prepare status
    - kXR_Qstats: Query server statistics

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

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

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

---

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

info is the requested information.

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

Notes

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

2) The kXR_waitresp response is not an error response but merely indicates that the response may take approximately seconds of time to deliver and will be reported using the unsolicited response mechanism (i.e., kXR_attn with kXR_asynresp). Refer to the description of each server response for detailed handling information.

3) A delayed response appears in protocol version 2.5.0 or higher. Earlier protocol versions did not use the delayed response mechanism.
3.17.1 KXR_query Checksum Cancellation Request

**Purpose:** Obtain server information.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_query</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char reserved[14]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_uint16 kXR_Qckscan</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td></td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

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

**Notes**

1. All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_uint16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

2. Only check sums requested by the current client may be cancelled.
3.17.2 KXR_query Checksum Request

**Purpose:** Obtain server information.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_query</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char reserved[14]</td>
<td>kXR_int32 ilen</td>
</tr>
<tr>
<td>kXR_uint16 kXR_Qcksum</td>
<td>kXR_char info[ilen]</td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td></td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

- **path**
  is the path of the file to be stored.

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

- **info**
  is the requested information.

**Notes**

3) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_uint16 data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

4) Each installation determines the type of checksum that will be returned. The algorithmic name of the checksum precedes the checksum value.
kXR_Qstats

Returned Response

The general format for the kXR_Qcksum response is:

```
csname csvalue
```

Where:

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

* **csvalue**
  
is the checksum name as a hexadecimal ASCII text string. The format is dependent on the algorithm used to compute the checksum.
3.17.3 KXR_query Configuration Request

**Purpose:** Obtain server information.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16 kXR_query</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char reserved[14]</td>
<td>kXR_int32 ilen</td>
</tr>
<tr>
<td>kXR_unt16 kXR_Qconfig</td>
<td>kXR_char info[ilen]</td>
</tr>
<tr>
<td>kXR_int32 qlen</td>
<td></td>
</tr>
<tr>
<td>kXR_char qry[qlen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

- **qry** are the space-separated names of the variables to be returned. Current variables that may be queried are:
  - **bind_max** - maximum number of sockets that may be bound to login session.
  - **pio_max** - maximum number of requests that may be queued on a bound socket before the session stream must wait.
  - **readv_iot_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.

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

- **info** is the requested information.
**kXR_Qstats**

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

**Returned Response**

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

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

Where:

*Cvalue*

is the corresponding value associated with the queried variable. If the variable has no value then the name of the variable is returned as Cvalue.
3.17.4 KXR_query Statistics Request

**Purpose:** Obtain server information.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_un16 kXR_query</td>
<td>kXR_un16 0</td>
</tr>
<tr>
<td>kXR_char reserved[14]</td>
<td>kXR_int32 ilen</td>
</tr>
<tr>
<td>kXR_un16 kXR_QStats</td>
<td>kXR_char info[ilen]</td>
</tr>
<tr>
<td>kXR_int32 alen</td>
<td></td>
</tr>
<tr>
<td>kXR_char args[alen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

- **args** is an optional list of letters, each indicating the statistical components to be returned. Valid letters are:
  - **a** - Return all statistics (default)
  - **p** - Protocol statistics
  - **b** - Buffer usage statistics
  - **s** - Scheduling statistics
  - **d** - Device polling statistics
  - **u** - Usage statistics
  - **i** - Server identification
  - **z** - Synchronized statistics
  - **l** - Connection statistics

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

- **info** is the requested information.

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_un16 data types are treated as *unsigned* values. All reserved fields must be initialized to binary zero.

2) Statistical information is returned as an XML text string. The XML schema is outlined below.
kXR_Qstats

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

Returned Response

The general XML schema for the kXR_Qstats response is:

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

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

Where:

- **time** is the Unix time() value of when the statistics were generated.
- **vers** is the xrootd version identification string.
- **sect** is the section name assigned to the statistical information, as follows:
  - **id** arg **Information**
    - **buff** b - Buffer usage statistics.
    - **info** i - Server identification.
    - **link** l - Connection (i.e., link) statistics.
    - **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. Only one section, corresponding to each requested letter code, is returned.
3) The letter a (the default) returns all sections.
4) The details on each section are provided below
Buffer (b) Statistics

```
<stats id="buff">
    <reqs>req</reqs>
    <mem>mem</mem>
    <bufs>bufs</bufs>
    <adj>adj</adj>
</stats>
```

Where:

- `req` number of requests for a buffer.
- `mem` number of bytes allocated to buffers.
- `bufs` number of allocated buffers.
- `adj` number of times an adjustment has been made to the buffer profile.

Server (i) Identification

```
<stats id="info">
    <host>host</host>
    <port>port</port>
</stats>
```

Where:

- `host` the name of the host generating the statistics.
- `port` port number being used to accept new connections.
Connection (1) Statistics

```xml
<stats id="link">
  <num>num</num>
  <maxn>maxn</maxn>
  <tot>tot</tot>
  <in>in</in>
  <out>out</out>
  <ctime>ctime</ctime>
  <tmo>tmo</tmo>
  <stall>stall</stall>
</stats>
```

Where:

- `num` number of current connections.
- `maxn` maximum number of simultaneous connections.
- `num` total number of connections.
- `in` total number of bytes received.
- `out` total number of bytes sent.
- `ctime` cumulative number of connect seconds. `ctime/tot` gives the average session time per connection.
- `tmo` number of times a connection timed out waiting for a new request.
- `stalls` number of connection timeouts where some but not all data was received within the timeout window.
Device Polling (d) Statistics

```
<stats id="poll">
  <att>att</att>
  <en>en</en>
  <ev>ev</ev>
  <int>int</int>
</stats>
```

Where:

- **att**: total number of file descriptors attached for polling.
- **en**: number of times a file descriptor was enabled for polling.
- **ev**: number of polling events.
- **int**: number of polling interrupts received.

Process Usage (u) Statistics.

```
<stats id="proc">
  <pid>pid</pid>
  <utime>
    <s>uts</s>
    <u>utu</u>
  </utime>
  <stime>
    <s>sts</s>
    <u>stu</u>
  </stime>
  <maxrss>maxrss</maxrss>
  <majflt>majflt</majflt>
  <nswap>nswap</nswap>
  <inblock>inblock</inblock>
  <oublock>oublock</oublock>
  <msgsnd>msgsnd</msgsnd>
  <msgrcv>msgrcv</msgrcv>
  <signals>signals</signals>
</stats>
```
kXR_Qstats

Where:

- **pid** is the process id of the server.
- **uts** seconds of user-time used.
- **utu** micro-seconds of user time used.
- **sts** seconds of system time used.
- **stu** micro-seconds of system time used.
- **maxrss** maximum resident working set size as reported by `getrusage()` in `ru_maxrss`.
- **majflt** number of major page faults as reported by `getrusage()` in `ru_majflt`.
- **nswap** number of times the processes was swapped out as reported by `rusage()` in `ru_nswap`.
- **inblock** number of block input operations as reported by `getrusage()` in `ru_inblock`.
- **outblock** number of block output operations as reported by `getrusage()` in `ru_outblock`.
- **msgsnd** number of messages sent as reported by `getrusage()` in `ru_msgsnd`.
- **msgrcv** number of messages received as reported by `getrusage()` in `ru_msgrcv`.
- **nsignals** number of signals received as reported by `getrusage()` in `ru_nsignals`.
Protocol (p) Statistics for rootd

```xml
<stats id="rootd">
  <num>num</num>
</stats>
```

Where:

- **num**  number of times rootd protocol was used.

Scheduler (s) Statistics

```xml
<stats id="sched">
  <jobs>jobs</jobs>
  <inq>inq</inq>
  <maxinq>maxinq</maxinq>
  <threads>threads</threads>
  <tcr>tcr</tcr>
  <tde>tde</tde>
  <tlimr>tlimr</tlimr>
</stats>
```

Where:

- **jobs**  total number of jobs requiring thread assistance.
- **inq**  number of jobs that are currently in the run-queue.
- **maxinq**  the longest run-queue length.
- **threads**  the number of current threads.
- **idle**  number of threads waiting for work.
- **tcr**  number of times a thread was created.
- **tdr**  number of times a thread was destroyed.
- **tlimr**  number of times the thread limit was reached.
Protocol (p) Statistics for xrootd

```xml
<stats id="xrootd">
  <ops>
    <num>uses</num>
    <open>open</open>
    <rf>refs</rf>
    <rd>rd</rd>
    <pr>pr</pr>
    <wr>wr</wr>
    <sync>sync</sync>
    <getf>getf</getf>
    <putf>putf</putf>
    <misc>misc</misc>
  </ops>
  <aio>
    <num>num</num>
    <max>max</max>
    <rej>rej</rej>
  </aio>
</stats>
```

Where:

- **uses**  total number of times the protocol was used for a connection.
- **open**  total number of file open requests.
- **refs**  total number of cache refresh requests.
- **rd**  total number of read requests.
- **pr**  total number of pre-read requests.
- **wr**  total number of write requests.
- **sync**  total number of sync requests.
- **getf**  total number of getfile requests.
- **putf**  total number of putfile requests.
- **misc**  total number of miscellaneous (i.e., other) requests.
\(num\quad\text{total number of asynchronous requests processed.}\)

\(max\quad\text{maximum number of simultaneous asynchronous requests.}\)

\(rej\quad\text{total number times an asynchronous request was converted to a synchronous request.}\)
3.18 kXR_read Request

Purpose: Read data from an open file.

```
<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>kXR_char streamid[2]</td>
</tr>
<tr>
<td>kXR_uint16</td>
<td>kXR_uint16 status</td>
</tr>
<tr>
<td>kXR_char fhandle[4]</td>
<td>kXR_int32 dlen</td>
</tr>
<tr>
<td>kXR_int64 offset</td>
<td>kXR_char data[dlen]</td>
</tr>
<tr>
<td>kXR_int32 rlen</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 alen</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Where:

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

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

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

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

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

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

pathid is the pathid returned by kXR_bind. The response data is sent to this path.

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

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

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

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

data is the data that was read.

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

8) The `kXR_read` request allows you to also schedule the pre-reading of data that you will ask for in the very near future. Pre-reading data may substantially speed up the execution because data will be available in memory when it is actually asked for. On the other hand, requesting data that you will not need will simply cause a general slow-down of the complete system.

9) The pre-read request is considered only a hint. The system may or may not honor the pre-read request, depending on the current system load.

10) To schedule a pre-read without actually reading any data, issue a `kXR_read` request with `rlen` and `offset` set to zero and `readahead_list` filled out to reflect what data should be pre-read.
### 3.19 kXR_readv Request

**Purpose:** Read data from one or more open files.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_int16 kXR_readv</td>
<td>kXR_int16 status</td>
</tr>
<tr>
<td>kXR_char reserved1[16]</td>
<td>kXR_int32 dlen</td>
</tr>
<tr>
<td>kXR_int32 alen</td>
<td>kXR_char data[dlen]</td>
</tr>
</tbody>
</table>

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

struct read_args  
  { kXR_char pathid;  
    kXR_char reserved2[7];  
    read_list rvec[(dlen-8)/16];  
  };
```

Where:

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

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

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

* **pathid** is the *pathid* returned by *kXR_bind*. The response data is sent to this path.
**kXR_read**

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

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

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

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

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

**Notes**

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

2) Each `read_list` element represents a read request. All of the read requests are aggregated into one or more responses. Read data is always prefixed by its corresponding `read_list` element. However, the `rlen` value in the element indicates the actual amount of data that was read.

3) If an element requests more data than the file contains, the returned `rlen` will be smaller than the `rlen` in the request element.

4) Reading past the end of file with a valid offset will return a request element whose `rlen` is 0 with no data following.

5) The `fhandle` value should be treated as opaque data.

6) Since a read may request more data than the allowable internal buffer size, the data is sent in fixed-sized segments until the request is satisfied. This is accomplished using the `kXR_oksofar` status code. Each subsequent data segment is transmitted using a `{streamid, status, dlen, data}` response. The last segment is indicated by a `kXR_ok`, if no error occurred.

7) Any status code other than `kXR_oksofar` indicates the end of transmission.

8) Sending requests using the same `streamid` when a `kXR_oksofar` status code has been returned may produce unpredictable results. A client must serialize all requests using the `streamid` in the presence of partial results.

9) The server may return the read elements in any order.
3.20 kXR_rm Request

Purpose: Remove a file.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16 kXR_rm</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char reserved[16]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td></td>
</tr>
<tr>
<td>kXR_char path[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

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

Notes

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

3.21 kXR_rmdir Request

Purpose: Remove a directory.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>kXR_char</td>
</tr>
<tr>
<td>kXR_unt16</td>
<td>kXR_unt16</td>
</tr>
<tr>
<td>kXR_rmdir</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char</td>
<td>kXR_char</td>
</tr>
<tr>
<td>reserved[16]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>plen</td>
</tr>
<tr>
<td>plen</td>
<td>path[plen]</td>
</tr>
</tbody>
</table>

Where:

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

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

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

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

Notes

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

2) The directory must be empty (i.e., no entries other than “.” And “..”).
3.22 kXR_set Request

**Purpose:** Set server information.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_char</td>
<td>kXR_char</td>
</tr>
<tr>
<td>kXR_int16</td>
<td>kXR_set</td>
</tr>
<tr>
<td>kXR_char</td>
<td>kXR_unt16</td>
</tr>
<tr>
<td>reserved[16]</td>
<td>0</td>
</tr>
<tr>
<td>kXR_int32</td>
<td>n</td>
</tr>
<tr>
<td>dlen</td>
<td>resp[n]</td>
</tr>
<tr>
<td>data[dlen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

- **data**
  - is the value to set.

- **resp**
  - is the response value to the specific set requested.

**Notes**

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

2) Set processing takes a command-like string in the **data** field. The following documents valid set arguments.
kXR_read

3.22.1 Valid kXR_Set Values

\[
\text{msg \hspace{0.5em} msg} \\
\text{monitor \{off \mid on\} [appid] \mid info [info]}
\]

Where:

\text{msg \hspace{0.5em} msg}
includes \text{msg} in the server’s log. This request is meant to be used to identify the start and stop if certain application processes for rudimentary monitoring purposes. Up to 80 characters will be recorded.

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

\text{Response}
is the unique four-character identification value that has been assigned to the \text{info} value.

\text{Notes}
1) Monitoring is enabled using the \text{xrootd.monitor} configuration directive. When monitoring is not enabled, the monitor set requests are ignored.
2) Use the returned identification value to tag future records in order to correlate related information.
### 3.23 `kXR_stat` Request

**Purpose:** Obtain status information for a path.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kXR_char</code></td>
<td><code>streamid[2]</code></td>
</tr>
<tr>
<td><code>kXR_int16</code></td>
<td><code>kXR_stat</code></td>
</tr>
<tr>
<td><code>kXR_char</code></td>
<td><code>reserved[16]</code></td>
</tr>
<tr>
<td><code>kXR_int32</code></td>
<td><code>plen</code></td>
</tr>
<tr>
<td><code>kXR_char</code></td>
<td><code>path[plen]</code></td>
</tr>
<tr>
<td></td>
<td><code>kXR_char</code> <code>streamid[2]</code></td>
</tr>
<tr>
<td></td>
<td><code>kXR_int16</code> 0</td>
</tr>
<tr>
<td></td>
<td><code>kXR_int32</code> <code>ilen</code></td>
</tr>
<tr>
<td></td>
<td><code>kXR_char</code> <code>info[ilen]</code></td>
</tr>
</tbody>
</table>

Where:

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

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

- `plen` is the binary length of the supplied path, `path`.

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

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

- `info` is the information about the requested path. If the path does not exist, the null-terminated string “-1 -1 -1 -1” is returned.

- `id` is the OS-dependent identifier assigned to this entry. Uniqueness is not guaranteed.

- `size` is the decimal size of the data associated with the path whose information is being returned. The size may represent a number up to $2^{64}-1$ (i.e., a long long).
flags identifies the entry’s attributes as a decimal encoded 32-bit string. The entry should be assumed to be a regular file unless one or more of the following bits are set.
- kXR_xset - Either an executable file or a searchable directory.
- kXR_isDir - This is a directory.
- kXR_other - This neither a file nor a directory.
- kXR_offline - For files, the file is not online (i.e., on disk).
- kXR_readable - Read access allowed.
- kXR_writable - Write access allowed.

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

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

**Purpose:** Obtain type information for one or more paths.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_statx</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char reserved[16]</td>
<td>kXR_int32 ilen</td>
</tr>
<tr>
<td>kXR_int32 plen</td>
<td>kXR_char info[ilen]</td>
</tr>
<tr>
<td>kXR_char paths[plen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

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

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

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

- **flags**
  - identifies the entry’s attributes as a binary character. The entry should be assumed to be an immediately available regular file unless one or more of the following bits are set.
    - **kXR_xset** - Either an executable file or a searchable directory.
    - **kXR_isDir** - This is a directory.
    - **kXR_other** - This neither a file nor a directory, or does not exist.
    - **kXR_offline** - For files, the file is not online (i.e., on disk).
kXR_statx

Notes
1) All binary fields are transmitted in network byte order using an explicit length. The kXR_char and kXR_unst16 data types are treated as unsigned values. All reserved fields must be initialized to binary zero.
2) kXR_statx requests directed to a redirector (i.e., load balancer) referring to a non-file object may result in a non-deterministic response. That is, the response may indicate that the object does not exist when, in fact, it does exist but is not a file. Future versions may resolve the differences between redirectors and file servers.
3.25 kXR_sync Request

**Purpose:** Commit all pending writes to an open file.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kXR_unt16 kXR_sync</code></td>
<td><code>kXR_unt16 0</code></td>
</tr>
<tr>
<td><code>kXR_char fhandle[4]</code></td>
<td><code>kXR_int32 0</code></td>
</tr>
<tr>
<td><code>kXR_char reserved[12]</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>kXR_int32 0</code></td>
<td><code>0</code></td>
</tr>
</tbody>
</table>

Where:

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

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

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

**Notes**

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

2) The fhandle value should be treated as opaque data.
3.26 kXR_unbind Request

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_unt16 kXR_unbind</td>
<td>kXR_unt16 0</td>
</tr>
<tr>
<td>kXR_char pathid</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_char sessid[15]</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 0</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

3.27 kXR_write Request

**Purpose:** Write data to an open file.

<table>
<thead>
<tr>
<th>Request</th>
<th>Normal Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXR_uint16 kXR_write</td>
<td>kXR_uint16 0</td>
</tr>
<tr>
<td>kXR_char fhandle[4]</td>
<td>kXR_int32 0</td>
</tr>
<tr>
<td>kXR_int64 offset</td>
<td></td>
</tr>
<tr>
<td>kXR_char pathid</td>
<td></td>
</tr>
<tr>
<td>kXR_char reserved[3]</td>
<td></td>
</tr>
<tr>
<td>kXR_int32 dlen</td>
<td></td>
</tr>
<tr>
<td>kXR_char data[dlen]</td>
<td></td>
</tr>
</tbody>
</table>

Where:

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

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

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

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

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

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

- **data** is the data to be written.

**Notes**

1) All binary fields are transmitted in network byte order using an explicit length. The **kXR_char** and **kXR_uint16** data types are treated as **unsigned** values. All reserved fields must be initialized to binary zero.

2) The **fhandle** value should be treated as opaque data.
4 Local Socket Administrative Protocol

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

4.1 Initiating an Administrative Session

To successfully initiate an administrative session, you must

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

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

4.2 General Request Format

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

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

Where:

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

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

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

`args` are `command` specific arguments.
4.2.1 Request Target Format

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

Where:

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

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

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

Notes

1) The target specification allows you to route requests to particular connections. Each connection is identified in a uniform way, described in the following section.

2) In order to route a request to a single connection, the complete connection name must be specified.

3) The target is mandatory even if the request applies to all connections. In this case, specify a single asterisk for target.

4.2.1.1 Connection name format

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

Where:

- **user** is the Unix username of the user.

- **pid** is the user’s process number that issued the request.

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

- **host** is the host name, or IP address, where the user’s request originated.
4.3 General Response Format

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

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

Where:

- `reqid` is the text identifier token of the request that is associated with this response.
- `code` is the numeric code indicating success or failure of the request. Zero ("0") always indicates that the request succeeded. A non-zero value indicates that the request failed.
- `xmltoks` are additional XML response elements that are specific to the request.

4.3.1 Error Response Format

All error responses are structured as follows:

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

Where:

- `reqid` is the text identifier token of the request that is associated with this response.
- `code` is a non-zero numeric code indicating failure.
- `msg` is a message explaining the failure.
4.4 Abort request for kXR_asyncab Client Action

Purpose: Send a kXR_attn/kXR_asyncab unsolicited response.

**Request Format**

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

**Normal Response Format**

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

Where:

- `reqid` is the request-response association text identifier token.
- `target` is the pattern that identifies the connections to be aborted.
- `num` is the number of kXR_attn/kXR_asyncab unsolicited responses that were sent.
- `msg` is optional message text to be sent to applicable clients.

**Notes**

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

**Purpose:** Close client connections.

**Request Format**

```
reqid close target
```

**Normal Response Format**

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

Where:

- `reqid` is the request-response association text identifier token.
- `target` is the pattern that identifies the connections to be closed.
- `num` is the number of connections that were closed.

**Notes**

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

**Purpose:** Cancel background job.

### Request Format

```
reqid cj job key
```

### Normal Response Format

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

Where:

- `reqid` is the request-response association text identifier token.
- `job` is the type of background job to be cancelled.
- `key` is the key that identifies the particular background job to be cancelled.
- `num` is the number of connections that were closed.

### Notes

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

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

### Request Format

```
reqid cont target
```

### Normal Response Format

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

Where:

- `reqid` is the request-response association text identifier token.
- `target` is the pattern that identifies the connections to be resumed.
- `num` is the number of kXR_attn/kXR_asyncgo unsolicited responses that were sent.

### Notes

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

Purpose: Send a kXR_attn/kXR_asyncdi unsolicited response.

Request Format

\[ reqid \text{ disc} \text{ target} \text{ wsec} \text{ msec} \]

Normal Response Format

\[ <\text{resp id="reqid"}>\text{rc}0</\text{rc}><\text{num}>num</\text{num}></\text{resp}>\]

Where:

- \text{reqid} is the request-response association text identifier token.
- \text{target} is the pattern that identifies the connections to be disconnected.
- \text{wsec} is the number of seconds the client should wait before attempting to reconnect to the server.
- \text{msec} is the maximum number of seconds the client should wait before declaring reconnect failure.
- \text{num} is the number of kXR_attn/kXR_asyncdi unsolicited responses that were sent.

Notes

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

**Purpose:** Create an administrative session.

<table>
<thead>
<tr>
<th>Request Format</th>
</tr>
</thead>
</table>
| `reqid login adminid
` |

<table>
<thead>
<tr>
<th>Normal Response Format</th>
</tr>
</thead>
</table>
| `<resp id="reqid"><rc>0</rc><v>vers</v></resp>
` |

Where:

- `reqid` is the request-response association text identifier token.
- `adminid` is a string that uniquely identifies the administrative client instance. Typically, this would be of the form: “`<username>.<pid>`”. Where,
  - `<username>` is the unix name of the process uid
  - `<pid>` is the process number as a ASCII string
- `vers` is the protocol version number being used.

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

Purpose: List client connections.

Request Format

```
reqid lsc target
```

Normal Response Format

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

Where:

- `reqid` is the request-response association text identifier token.
- `target` is the pattern that identifies the connections to be listed.
- `conn` is a space separated list of connections. If no connections exist that match `target`, the list is empty. Otherwise, each connection has the form of:
  
  `user.pid:fd@host`

Where:

- `user` is the Unix username of the user.
- `pid` is the user’s process number.
- `fd` is the server’s file descriptor number associated with the connection to `user:pid` at `host`.
- `host` is the host name, or IP address, of the user’s machine.
4.11 Lsd request

**Purpose:** List detailed client connections.

### Request Format

```
reqid lsd target
```

### Normal Response Format

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

Where:

- **reqid** is the request-response association text identifier token.
- **target** is the pattern that identifies the connections to be listed.
- **role** is the role assumed by the client connect. The ‘a’ is used to designate an administrative role and ‘u’ as a regular user role.
- **ctime** is the server-local Unix time that existed when the connection was established (i.e., connect time).
- **vers** is the client’s version number that was specified during login. Refer to the description of the login request for information on how to interpret the version number.
- **mon** the monitoring status of this connection. The letter ‘f’ is used to indicate file-level monitoring and ‘i’ as I/O-level monitoring. If no monitoring is in effect, the string is empty.
**cname** is the name of the connection as "user.pid:fd@host" (see lsc).

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

**nfiles** number of files the connection has open.

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

**pcnt** the number of pre-read requests.

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

**wcnt** the number of write requests.

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

**rcnt** the number of read requests.

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

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

**prot** the protocol name used for authentication.

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

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

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

**role** the client’s role name as reported by prot. If no role name is present, the tag data is null.
4.12 Lsj request

Purpose: List jobs.

Request Format

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

Normal Response Format

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

Where:

- **reqid** is the request-response association text identifier token.
- ***** lists all background jobs.
- **checksum** lists only check sum jobs.
- **jtype** is the job type. Possible types are:
  - checksum - job requested via `kXR_query` - `kXR_Qcksum`
  - other types
- **jkey** is the key unique identifying the job. Applicable keys are:
  - checksum - the logical filename of the file being check summed
- **stat** is the job status. Possible values are:
  - a - actively executing
  - d - completed
  - w - waiting for resources
  - u - unknown status
- **conn** is a space separated list of client connections that initiated the job. If no connections exist for the job, the list is empty. See the **lsc** request for the definition of conn.
4.13 Msg request for kXR_asycns Client Action

Purpose: Send a kXR_attn/kXR_asyncdms unsolicited response.

**Request Format**

```
reqid msg target [msg]\n```

**Normal Response Format**

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

Where:

- `reqid` is the request-response association text identifier token.
- `target` is the pattern that identifies the connections to be sent messages.
- `msg` is optional message text to be sent to applicable clients.
- `num` is the number of kXR_attn/kXR_asyncdms unsolicited responses that were sent.

**Notes**

1) When a client receives an unsolicited kXR_attn/kXR_asyncdms response, it prints the message to the console and continues normal execution. If the message is null, the client ignores the kXR_attn/kXR_asyncdms unsolicited response.
4.14 Pause request for kXR_asyncwt Client Action

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

### Request Format

```
reqid pause target wsec
```

### Normal Response Format

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

Where:

- `reqid` is the request-response association text identifier token.
- `target` is the pattern that identifies the connections to be disconnected.
- `wsec` is the number of seconds the client should wait before resuming communications with the server.
- `num` is the number of `kXR_attn/kXR_asyncwt` unsolicited responses that were sent.

### Notes

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

Purpose: Send a kXR_attn/kXR_asyncrd unsolicited response.

**Request Format**

```
reqid redirect target host[?token]:port[?token]
```

**Normal Response Format**

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

Where:

- **reqid** is the request-response association text identifier token.
- **target** is the pattern that identifies the connections to be sent redirections.
- **host** is the DNS name or IP address of the target host.
- **token** is the optional token that the client must transmit to the target host upon connection. The token may be specified after host or after port.
- **port** is the port number that the client must use when connecting to host.
- **num** is the number of kXR_attn/kXR_asyncrd unsolicited responses that were sent.

**Notes**

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