System Monitoring Reference
1 Monitoring .............................................................................................................. 5
2 Summary Monitoring Data Format ....................................................................... 7
  2.1 The mpstats Command ..................................................................................... 7
  2.1.1 Quick Guide Example .................................................................................. 9
  2.2 Summary Data .................................................................................................. 10
    2.2.1 Buff Summary Data .................................................................................. 10
    2.2.2 Cache Summary Data .............................................................................. 11
    2.2.3 Cms Protocol Summary Data .................................................................. 13
      2.2.3.1 Cmsc Protocol Summary Data .............................................................. 13
      2.2.3.2 Cmsm Protocol Summary Data .............................................................. 14
      2.2.3.3 Cmss Protocol Summary Data .............................................................. 15
    2.2.4 Info Summary Data .................................................................................... 16
    2.2.5 Link Summary Data ................................................................................... 16
    2.2.6 Ofs Summary Data .................................................................................... 17
    2.2.7 Oss Summary Data ................................................................................... 18
    2.2.8 Poll Summary Data ................................................................................... 19
    2.2.9 Proc Summary Data ................................................................................... 19
    2.2.10 Pss Summary Data ................................................................................... 20
    2.2.11 Sched Summary Data .............................................................................. 21
    2.2.12 Sgen Summary Data .............................................................................. 21
    2.2.13 Xrootd Protocol Summary Data ............................................................... 22
  3 Detailed Monitoring Data Format ....................................................................... 25
    3.1 Event Monitoring Overview .......................................................................... 27
    3.2 The Common Packet Header ......................................................................... 28
    3.3 Monitor Map Message Format ....................................................................... 29
      3.3.1 Message Info Field .................................................................................. 30
    3.4 The “f” Stream (fstat) .................................................................................. 34
      3.4.1 Disc Event .............................................................................................. 37
      3.4.2 Open Event .............................................................................................. 37
      3.4.3 Close Event .............................................................................................. 39
      3.4.4 Xfr Event ................................................................................................. 41
    3.5 The “g” Stream (ccm and pfc) ...................................................................... 43
    3.6 The “r” Stream (redir) ................................................................................... 45
      3.6.1 Understanding Multiple Redirection Streams ......................................... 48
    3.7 The “t” Stream (files, io, and iov) ................................................................. 49
      3.7.1 Monitor Trace Message Format ............................................................... 50
  4 Document Change History .................................................................................. 55
1 Monitoring

**Xrootd** provides two types of monitoring: 1) summary monitoring and 2) detail monitoring. Summary monitoring is controlled by the `xrd.report` directive while detail monitoring is controlled by the `xrootd.monitor` directive. Both of these are documented in the “*Xrd/Xrootd Configuration Reference*”.

In order to provide real-time information with minimal impact, monitor data is sent as UDP messages. Each directive specifies what information is to be sent as well as the destinations. Because UDP is used, information is sent whether or not the receiving host is listening for the records. You should not activate monitoring if you do not activate the receiving end, as well.

Below is a graphic showing all of the available monitoring streams and their intended destinations.

Summary monitoring is suitable for providing a broad over-view of an xrootd cluster. The information is typically rendered by agents such as Ganglia or Mona Lisa, among others.
Monitoring

Detail monitoring is suitable for deep analysis of access and usage patterns of an xrootd cluster. Since such information is necessarily complex, specialized renderers must be used.

The following sections describe the data formats of each monitoring stream.
2 Summary Monitoring Data Format

The xrd.report directive specifies the parameters as well as the hosts that are to receive the summary information, Summary records are sent as UDP datagrams. Therefore, the information is sent whether or not the receiving host is enabled for the records. Summary information is formatted as an XML record and is described in the following sections. When dealing with XML formats you must:

1. Be insensitive to the XML tag order within a phrase, and
2. Ignore undocumented tags.

Normally, multiple xrootd servers transmit summary information to a collector (i.e., a process accepting messages on a specific port). In order to simplify the processing of summary information, a UDP multiplexing and XML parsing program, called mpxstats, is provided. This program accepts data on a selectable port, multiplexes received the datagrams into a single stream, and optionally parses the XML into either a CGI format or a flat key-value format. The output is sent to standard out for further processing.

2.1 The mpxstats Command

```bash
mpxstats [-f {cgi | flat | xml}] -p port [-s]
```

**Function**

Multiplex UDP datagrams into a single stream and optionally parse the data.

**Options & Parameters**

- `-f` Parses the received data into the specified format:
  - `cgi` Computer Gateway Interface
  - `flat` Simple keyword-value format
  - `xml` Original format (i.e., input is not parsed)

- `-p port` is the port to use for accepting UDP datagrams.

- `-s` includes the actual sender in `cgi` and `flat` format output.

**Defaults**

By default, xml output format is used. The UDP port must be specified.
Notes

1) The **cgi** and **flat** formats are based on the input the **xml** tags, without interpretation. Therefore, un-described tags may appear in the output and should be ignored.

2) The **cgi** format generally produces: “\texttt{var=value[&amp;var=value[. . .]]\n}”. Each \texttt{var} is based on an **xml** format item and the \texttt{value} is the item’s associated value. One new-line terminated string is generated for each UDP packet.

3) The **cgi** format is suitable for input to an **XrdOucEnv** class object which converts **cgi** strings into environment variable store. The class provides a simple value look-up scheme; much like \texttt{getenv()}.

4) The **flat** format generally produces: “\texttt{var value\n[var value\n[. . .]]\n}”. Each \texttt{var} is based on an **xml** format item and the \texttt{value} is the item’s associated value. Each \texttt{var-value} pair is a new-line terminated string. A null line is generated at the end for each UDP packet.

5) The flat format is suitable for input to Perl and Python scripts and can easily be used to construct \texttt{var-value} hashes for further processing.

6) The **mpxstats** program writes its output to standard out. Error messages are written to standard error.
2.1.1 Quick Guide Example

This picture illustrates the general scheme most installations use to gather summary statistics and insert them into their monitoring framework. Here a number of xrootd servers send their statistics to a collector machine listening at port 3333. The collector merges all of the data streams and sends a selection of the desired data to the monitoring infrastructure.

To implement such a scheme, follow these steps:

1. In the configuration file for each xrootd insert the following directive

   ```
   if exec xrootd
   xrd.report collector_host_name:3333 every 15 all -poll
   fi
   ```

   Where `collector_host_name` is the name of the machine that collects and formats the summary data. The `if/fi` construct only allows xrootd to report statistics as the `cmsd` does not currently report meaningful statistics.

2. Start the data multiplexing program and feed its output to a program or script that can inject the data into the monitoring infrastructure. For instance,

   ```
   mpxstats -f flat -p 3333 | send2monitor
   ```

   The `send2monitor` script is, of course, installation dependent. Below is a simple `perl` script that reads the statistical data from standard in, places it in a hash, and then calls a subroutine that can use the values in the hash to feed Ganglia.

   ```perl
   #!/bin/perl
   do {undef %StatsData;
     while (($Line = <STDIN>) ne "\n")
       (exit if !chomp($Line);
        ($Var,$Val) = split(' ', $Line);
        $StatsData{$Var} = $Val;
       }
     Ganglia(); # Inject data into the monitoring system
   } while(1);
   ```

   `send2monitor`: Place Data In a Hash Indexed By the Data’s Variable Name
2.2 Summary Data

```xml
<statistics
tod="int64" ver="chars" src="chars" tos="int64"
pgm="chars" ins="chars" pid="int" site="chars">
  ...
</statistics>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>char</td>
<td></td>
<td>The name of the host that sent the UDP packet.*</td>
</tr>
<tr>
<td>ins</td>
<td>char</td>
<td></td>
<td>The instance name specified via –n option (anon if none).</td>
</tr>
<tr>
<td>pgm</td>
<td>char</td>
<td></td>
<td>The name of the program.</td>
</tr>
<tr>
<td>pid</td>
<td>int</td>
<td>↔</td>
<td>The program’s process ID.</td>
</tr>
<tr>
<td>site</td>
<td>char</td>
<td></td>
<td>The specified site name.</td>
</tr>
<tr>
<td>src</td>
<td>char</td>
<td></td>
<td>Host and port reporting data, specified as “hostname:port”</td>
</tr>
<tr>
<td>tod</td>
<td>int64</td>
<td>↑</td>
<td>Unix time when statistics gathering started.</td>
</tr>
<tr>
<td>tos</td>
<td>int64</td>
<td>↑</td>
<td>Unix time when the program was started.</td>
</tr>
<tr>
<td>ver</td>
<td>char</td>
<td></td>
<td>The version name of the server’s code.</td>
</tr>
</tbody>
</table>

2.2.1 Buff Summary Data

```xml
<stats id="buff">
  <reqs>int</reqs><mem>int64</mem><buffs>int</buffs>
  <adj>int</adj>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>buff.adj</td>
<td>int</td>
<td>↑</td>
<td>Adjustments to the buffer profile.</td>
</tr>
<tr>
<td>buff.buffs</td>
<td>int</td>
<td>↔</td>
<td>Number of allocated buffers.</td>
</tr>
<tr>
<td>buff.mem</td>
<td>int64</td>
<td>↔</td>
<td>Bytes allocated to buffers.</td>
</tr>
<tr>
<td>buff.reqs</td>
<td>int</td>
<td>↑</td>
<td>Requests for a buffer.</td>
</tr>
</tbody>
</table>

* This information is provided by the Operating System’s recvfrom() function, not the data stream. It is present only when the –s mpstats option has been specified.
2.2.2 Cache Summary Data

```xml
<stats id="cache" type="type">
  <prerd>
    <in>int64</in><hits>int64</hits><miss>int64</miss>
  </prerd>
  <rd>
    <in>int64</in><out>int64</out>
    <hits>int64</hits><miss>int64</miss>
  </rd>
  <pass>int64</pass>
  <wr><out>int64</out><updt>int64</updt></wr>
  <saved>int64</saved>
  <purge>int64</purge>
  <files>
    <opened>int64</opened>
    <closed>int64</closed>
    <new>int64</new>
  </files>
  <store><size>int64</size><used>int64</used><min>int64</min><max>int64</max></store>
  <mem><size>int64</size><used>int64</used><wq>int64</wq></mem>
  <opcl>
    <odefer>int64</odefer><defero>int64</defero>
    <cdefer>int64</cdefer><clost>int64</clost>
  </opcl>
</stats>
```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>char</td>
<td></td>
<td>Type of cache (i.e. pfc or rmc)</td>
</tr>
<tr>
<td>prerd.in</td>
<td>int</td>
<td>↑</td>
<td>Bytes read into the cache via pre-read mechanism.</td>
</tr>
<tr>
<td>prerd.hits</td>
<td>int</td>
<td>↑</td>
<td>Number of pre-read pages that were wanted</td>
</tr>
<tr>
<td>prerd.miss</td>
<td>int64</td>
<td>↑</td>
<td>Number of pre-read pages that were not wanted.</td>
</tr>
<tr>
<td>rd.in</td>
<td>int</td>
<td>↑</td>
<td>Bytes read into the cache via demand.</td>
</tr>
<tr>
<td>rd.out</td>
<td></td>
<td>↑</td>
<td>Bytes delivered out of the cache to satisfy requests.</td>
</tr>
<tr>
<td>rd.hits</td>
<td></td>
<td>↑</td>
<td>Number of times wanted data was in the cache.</td>
</tr>
<tr>
<td>rd.miss</td>
<td></td>
<td>↑</td>
<td>Number of times wanted data was not in the cache.</td>
</tr>
<tr>
<td>pass</td>
<td></td>
<td>↑</td>
<td>Number of bytes read but not cached.</td>
</tr>
<tr>
<td>pass.cnt</td>
<td></td>
<td>↑</td>
<td>Number of times requested data bypassed the cache.</td>
</tr>
<tr>
<td>wr.out</td>
<td></td>
<td>↑</td>
<td>Bytes written out of the cache.</td>
</tr>
<tr>
<td>wr.updt</td>
<td></td>
<td>↑</td>
<td>Bytes written into the cache.</td>
</tr>
<tr>
<td>saved</td>
<td></td>
<td>↑</td>
<td>Bytes written from memory to storage.</td>
</tr>
<tr>
<td>purge</td>
<td></td>
<td>↑</td>
<td>Bytes purged from storage.</td>
</tr>
<tr>
<td>files.opened</td>
<td></td>
<td>↑</td>
<td>Number of cache files opened.</td>
</tr>
<tr>
<td>files.closed</td>
<td></td>
<td>↑</td>
<td>Number of cache files closed.</td>
</tr>
<tr>
<td>files.new</td>
<td></td>
<td>↑</td>
<td>Number of cache files that were created.</td>
</tr>
<tr>
<td>store.size</td>
<td></td>
<td></td>
<td>The size of cache storage in bytes.</td>
</tr>
<tr>
<td>store.used</td>
<td></td>
<td>↔</td>
<td>Storage bytes in use.</td>
</tr>
<tr>
<td>store.min</td>
<td></td>
<td>↓</td>
<td>The minimum number of storage bytes in use.</td>
</tr>
<tr>
<td>store.max</td>
<td></td>
<td>↑</td>
<td>The maximum number of storage bytes in use.</td>
</tr>
<tr>
<td>mem.size</td>
<td></td>
<td></td>
<td>The size of the cache memory in bytes.</td>
</tr>
<tr>
<td>mem.used</td>
<td></td>
<td>↔</td>
<td>Memory bytes in use.</td>
</tr>
<tr>
<td>mem.wq</td>
<td></td>
<td>↔</td>
<td>Bytes currently in the memory write queue.</td>
</tr>
<tr>
<td>opcl.odefer</td>
<td></td>
<td>↑</td>
<td>Number of deferred open requests.</td>
</tr>
<tr>
<td>opcl.defero</td>
<td></td>
<td>↑</td>
<td>Number deferred opens that were actually opened.</td>
</tr>
<tr>
<td>opcl.cdefer</td>
<td></td>
<td>↑</td>
<td>Number of deferred close requests.</td>
</tr>
<tr>
<td>Opcl.clost</td>
<td></td>
<td>↑</td>
<td>Number of uncompleted close requests.</td>
</tr>
</tbody>
</table>
2.2.3 Cms Protocol Summary Data

The cms protocol has three distinct sub-protocols:
- Client identified by the tag id cmsc,
- Manager identified by the tag id cmsm, and
- Server identified by the tag id cmss.

Each provides different summary statistics as each sub-protocol performs different actions. Even within each sub-protocol, the reporting entity may have a distinct role that also affects which information is actually reported. The following table lists the role identifiers (role ID) reported in the “role” tag.

<table>
<thead>
<tr>
<th>Role ID</th>
<th>Corresponding role directive</th>
<th>Role ID</th>
<th>Corresponding role directive</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>peer</td>
<td>PR</td>
<td>proxy supervisor</td>
</tr>
<tr>
<td>EM</td>
<td>peer manager</td>
<td>PS</td>
<td>proxy server</td>
</tr>
<tr>
<td>M</td>
<td>manager</td>
<td>R</td>
<td>supervisor</td>
</tr>
<tr>
<td>MM</td>
<td>meta manager</td>
<td>S</td>
<td>server</td>
</tr>
<tr>
<td>PM</td>
<td>proxy manager</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2.3.1 Cmsc Protocol Summary Data

```xml
<stats id="cmsc">
  <role>chars</role>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmsc.role</td>
<td>char</td>
<td>S</td>
<td>Role identification for reporter (see table above).</td>
</tr>
</tbody>
</table>
2.2.3.2 Cmsm Protocol Summary Data

```xml
<stats id="cmsm">
  <role>chars</role>
  <sel><t>int64</t><r>int64</r><w>int64</w></sel>
  <node>int
    <stats id="i">
      <host>chars</host>
      <role>chars</role><run>chars</run>
      <ref><r>int</r><w>int</w></ref>
      [<shr><use>int</use></shr>]
    </stats> • • •
  </node>
  [<frq>
    <add>int64<pb>int64</pb></add>
    <rsp>int64<m>int64</m></rsp>
    <lf>int64</lf><ls>int64</ls>
    <rf>int64</rf><rs>int64</rs>
  </frq>]
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmsm.role</td>
<td>char</td>
<td></td>
<td>Role identification for reporter (see table above).</td>
</tr>
<tr>
<td>cmsm.sel.t</td>
<td>int64</td>
<td>↑</td>
<td>Number of node selections.</td>
</tr>
<tr>
<td>cmsm.sel.r</td>
<td>int64</td>
<td>↑</td>
<td>Number of node selections for read access.</td>
</tr>
<tr>
<td>cmsm.sel.w</td>
<td>int64</td>
<td>↑</td>
<td>Number of node selections for write access.</td>
</tr>
<tr>
<td>cmsm.node</td>
<td>int</td>
<td>↔</td>
<td>Number of subsequent node stats (0 &lt;= i &lt; n).</td>
</tr>
<tr>
<td>cmsm.node.i.host</td>
<td>char</td>
<td></td>
<td>DNS name of host or IPV6 address.</td>
</tr>
<tr>
<td>cmsm.node.i.role</td>
<td>char</td>
<td></td>
<td>Role identification for host (see table above).</td>
</tr>
<tr>
<td>cmsm.node.i.run</td>
<td>char</td>
<td></td>
<td>Run status as a sequence of characters:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a – active    d – disabled   n - nostaging</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>o – offline   w - writable</td>
</tr>
<tr>
<td>cmsm.node.i.ref.r</td>
<td>int</td>
<td>↑</td>
<td>Number of times selected for read access.</td>
</tr>
<tr>
<td>cmsm.node.i.ref.w</td>
<td>int</td>
<td>↑</td>
<td>Number of times selected for write access.</td>
</tr>
<tr>
<td>cmsm.node.i.shr</td>
<td>int</td>
<td>↑</td>
<td>Desired share of requests†, if so configured.</td>
</tr>
<tr>
<td>cmsm.node.i.shr.use</td>
<td>int64</td>
<td>↑</td>
<td>Number of times share was exhausted.</td>
</tr>
</tbody>
</table>

† This tag is only present for MM roles (meta manager) and if requested via the `cms.repstats` directive.
### Variable Types and Explanations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmsm.frq.add</td>
<td>int64</td>
<td>↑</td>
<td>Additions to the fast response queue (frq)(^1).</td>
</tr>
<tr>
<td>cmsm.frq.add.d</td>
<td>int64</td>
<td>↑</td>
<td>Additions that were duplicates.</td>
</tr>
<tr>
<td>cmsm.frq.rsp</td>
<td>int64</td>
<td>↑</td>
<td>Responses received.</td>
</tr>
<tr>
<td>cmsm.frq.rsp.m</td>
<td>int64</td>
<td>↑</td>
<td>Multiple responses were fielded.</td>
</tr>
<tr>
<td>cmsm.frq.lf</td>
<td>int64</td>
<td>↑</td>
<td>Lookups dispatched that required no wait.</td>
</tr>
<tr>
<td>cmsm.frq.ls</td>
<td>int64</td>
<td>↑</td>
<td>Lookups dispatched that required a full wait.</td>
</tr>
<tr>
<td>cmsm.frq.rf</td>
<td>int64</td>
<td>↑</td>
<td>Redirects dispatched that required no wait.</td>
</tr>
<tr>
<td>cmsm.frq.rs</td>
<td>int64</td>
<td>↑</td>
<td>Redirects dispatched that required a full wait.</td>
</tr>
</tbody>
</table>

\(^1\) This tag is only present if requested via the \texttt{cms.repstats} directive.

### 2.2.3.3 Cmss Protocol Summary Data

```xml
<stats id="cmss">
  <role>chars</role>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmss.role</td>
<td>char</td>
<td></td>
<td>Role identification for reporter (see table above).</td>
</tr>
</tbody>
</table>
Monitoring

2.2.4 Info Summary Data

```xml
<stats id="info">
  <host>chars</host><port>int</port><name>chars</name>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>info.host</td>
<td>char</td>
<td></td>
<td>Hostname that generated the information.</td>
</tr>
<tr>
<td>info.name</td>
<td>char</td>
<td></td>
<td>Instance name specified via --n option (anon if none).</td>
</tr>
<tr>
<td>info.port</td>
<td>int</td>
<td>↔</td>
<td>Port used for server requests.</td>
</tr>
</tbody>
</table>

2.2.5 Link Summary Data

```xml
<stats id="link">
  <num>int</num><maxn>int</maxn><tot>int64</tot>
  <in>int64</in><out>int64</out><ctime>int64</ctime>
  <tmo>int</tmo><stall>int</stall><sfps>int</sfps>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>link.ctime</td>
<td>int64</td>
<td>↑</td>
<td>Cumulative number of connect seconds. ( ctime/tot ) gives the average session time per connection.</td>
</tr>
<tr>
<td>link.in</td>
<td>int64</td>
<td>↑</td>
<td>Bytes received.</td>
</tr>
<tr>
<td>link.maxn</td>
<td>int</td>
<td>↑</td>
<td>Maximum number of simultaneous connections.</td>
</tr>
<tr>
<td>link.num</td>
<td>int</td>
<td>↔</td>
<td>Current connections.</td>
</tr>
<tr>
<td>link.out</td>
<td>int64</td>
<td>↑</td>
<td>Bytes sent.</td>
</tr>
<tr>
<td>link.sfps</td>
<td>int</td>
<td>↑</td>
<td>Partial sendfile() operations.</td>
</tr>
<tr>
<td>link.stall</td>
<td>int</td>
<td>↑</td>
<td>Number of times partial data was received.</td>
</tr>
<tr>
<td>link.tmo</td>
<td>int</td>
<td>↑</td>
<td>Read request timeouts.</td>
</tr>
<tr>
<td>link.tot</td>
<td>int64</td>
<td>↑</td>
<td>Connections since start-up.</td>
</tr>
</tbody>
</table>

\(^8\) The info tag is deprecated and normally does not get included as this information is present in the header tag. It is documented here for backwards compatibility.
2.2.6  Ofs Summary Data

```xml
<stats id="ofs">
  <role>chars</role><opr>int</opr><opw>int</opw>
  <opp>int</opp><ups>int</ups><han>int</han>
  <rdr>int</rdr><bxq>int</bxq><rep>int</rep>
  <err>int</err><dly>int</dly><sok>int</sok>
  <ser>int</ser>
  <tpc><grnt>int</grnt><deny>int</deny>
    <err>int</err><exp>int</exp></tpc>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ofs.bxq</td>
<td>int</td>
<td>↑</td>
<td>Background tasks processed.</td>
</tr>
<tr>
<td>ofs.dly</td>
<td>int</td>
<td>↑</td>
<td>Delays imposed.</td>
</tr>
<tr>
<td>ofs.err</td>
<td>int</td>
<td>↑</td>
<td>Errors encountered.</td>
</tr>
<tr>
<td>ofs.han</td>
<td>int</td>
<td>↔</td>
<td>Active file handles.</td>
</tr>
<tr>
<td>ofs.opp</td>
<td>int</td>
<td>↔</td>
<td>Files open in read/write POSC mode.</td>
</tr>
<tr>
<td>ofs.opr</td>
<td>int</td>
<td>↔</td>
<td>Files open in read-mode.</td>
</tr>
<tr>
<td>ofs.opw</td>
<td>int</td>
<td>↔</td>
<td>Files open in read/write mode.</td>
</tr>
<tr>
<td>ofs.rdr</td>
<td>int</td>
<td>↑</td>
<td>Redirects processed.</td>
</tr>
<tr>
<td>ofs.rep</td>
<td>int</td>
<td>↑</td>
<td>Background replies processed.</td>
</tr>
<tr>
<td>ofs.role</td>
<td>char</td>
<td>↑</td>
<td>Reporter’s role (e.g., manager, server, etc).</td>
</tr>
<tr>
<td>ofs.ser</td>
<td>int</td>
<td>↑</td>
<td>Events received that indicated failure.</td>
</tr>
<tr>
<td>ofs.sok</td>
<td>int</td>
<td>↑</td>
<td>Events received that indicated success.</td>
</tr>
<tr>
<td>ofs.ups</td>
<td>int</td>
<td>↑</td>
<td>Number of times a POSC mode file was un-persisted.</td>
</tr>
<tr>
<td>ofs.tpc.grnt</td>
<td>int</td>
<td>↑</td>
<td>Number of third party copies allowed.</td>
</tr>
<tr>
<td>ofs.tpc.deny</td>
<td>int</td>
<td>↑</td>
<td>Number of third party copies denied.</td>
</tr>
<tr>
<td>ofs.tpc.err</td>
<td>int</td>
<td>↑</td>
<td>Number of third party copies that failed.</td>
</tr>
<tr>
<td>ofs.tpc.exp</td>
<td>int</td>
<td>↑</td>
<td>Number of third party copies whose auth expired.</td>
</tr>
</tbody>
</table>
### 2.2.7 Oss Summary Data

```xml
<stats id="oss">
  <paths>int
    <stats id="i">
      <lp>"chars"</lp><rp>"chars"</rp>
      <tot>int64</tot><free>int64</free>
      <ino>int64</ino><ifr> int64</ifr>
    </stats> • • •
  </paths>
  <space>int
    <stats id="i">
      <name>chars</name>
      <tot>int64</tot><free>int64</free>
      <maxf>int64</maxf><fsn>int</fsn>
      <usg>int64</usg>[<qta>int64</qta>]
    </stats> • • •
  </space>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>oss.paths</td>
<td>int</td>
<td>↔</td>
<td>Number of subsequent paths stats (0 &lt;= i &lt; n).</td>
</tr>
<tr>
<td>oss.paths.i.free</td>
<td>int64</td>
<td>↔</td>
<td>Kilobytes available.</td>
</tr>
<tr>
<td>oss.paths.i.ifr</td>
<td>int64</td>
<td>↔</td>
<td>Number of free inodes.</td>
</tr>
<tr>
<td>oss.paths.i.ino</td>
<td>int64</td>
<td>↔</td>
<td>Number of inodes.</td>
</tr>
<tr>
<td>oss.paths.i.lp</td>
<td>char</td>
<td></td>
<td>The minimally reduced logical file system path.</td>
</tr>
<tr>
<td>oss.paths.i_rp</td>
<td>char</td>
<td></td>
<td>The minimally reduced real file system path.</td>
</tr>
<tr>
<td>oss.paths.i.tot</td>
<td>int64</td>
<td>↔</td>
<td>Kilobytes allocated.</td>
</tr>
<tr>
<td>oss.space</td>
<td>int</td>
<td>↔</td>
<td>Number of subsequent space stats (0 &lt;= i &lt; n).</td>
</tr>
<tr>
<td>oss.space.i.free</td>
<td>int64</td>
<td>↔</td>
<td>Kilobytes available.</td>
</tr>
<tr>
<td>oss.space.i.ifsn</td>
<td>int</td>
<td>↔</td>
<td>Number of file system extents.</td>
</tr>
<tr>
<td>oss.space.i.maxf</td>
<td>int64</td>
<td>↔</td>
<td>Max kilobytes available in a filesystem extent.</td>
</tr>
<tr>
<td>oss.space.i.name</td>
<td>char</td>
<td></td>
<td>Name for the space.</td>
</tr>
<tr>
<td>oss.space.i.qta</td>
<td>int64</td>
<td>↔</td>
<td>Total space quota*, if supported.</td>
</tr>
<tr>
<td>oss.space.i.tot</td>
<td>int64</td>
<td>↔</td>
<td>Kilobytes allocated.</td>
</tr>
<tr>
<td>oss.space.i.usg</td>
<td>int64</td>
<td>↔</td>
<td>Usage associated with space name, if supported.</td>
</tr>
</tbody>
</table>

** This tag may be missing if quotas have not been configured.
2.2.8 Poll Summary Data

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>poll.att</td>
<td>int</td>
<td>↔</td>
<td>File descriptors attached for polling.</td>
</tr>
<tr>
<td>poll.en</td>
<td>int</td>
<td>↑</td>
<td>Poll enable operations.</td>
</tr>
<tr>
<td>poll.ev</td>
<td>int</td>
<td>↑</td>
<td>Polling events.</td>
</tr>
<tr>
<td>poll.int</td>
<td>int</td>
<td>↑</td>
<td>Unsolicited polling events.</td>
</tr>
</tbody>
</table>

2.2.9 Proc Summary Data

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Values Reported by getrusage()</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc.sys.s</td>
<td>int</td>
<td>↑</td>
<td>Seconds of system-time.</td>
</tr>
<tr>
<td>proc.sys.u</td>
<td>int</td>
<td>↔</td>
<td>Microseconds of system-time.</td>
</tr>
<tr>
<td>proc.usr.s</td>
<td>int</td>
<td>↑</td>
<td>Seconds of user-time.</td>
</tr>
<tr>
<td>proc.usr.u</td>
<td>int</td>
<td>↔</td>
<td>Microseconds of user-time.</td>
</tr>
</tbody>
</table>
Monitoring

2.2.10 Pss Summary Data

```xml
<stats id="pss">
  <open>int64</open> <errs>int64</errs> </open>
  <close>int64</close> <errs>int64</errs> </close>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pss.open</td>
<td>int</td>
<td>↑</td>
<td>Number of remotes file opens.</td>
</tr>
<tr>
<td>pss.open.errs</td>
<td>int</td>
<td>↑</td>
<td>Number of opens that failed.</td>
</tr>
<tr>
<td>pss.close</td>
<td>int</td>
<td>↑</td>
<td>Number of remote file closes.</td>
</tr>
<tr>
<td>pss.close.errs</td>
<td>int</td>
<td>↑</td>
<td>Number of closes that failed.</td>
</tr>
</tbody>
</table>
2.2.11 Sched Summary Data

<sched id="sched">
  <jobs>int</jobs><inq>int</inq><maxinq>int</maxinq>
  <threads>int</threads><idle>int</idle><tcr>int</tcr>
  <tde>int</tde><tlimr>int</tlimr>
</sched>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched.idle</td>
<td>int</td>
<td></td>
<td>Number of scheduler threads waiting for work.</td>
</tr>
<tr>
<td>sched.inq</td>
<td>int</td>
<td></td>
<td>Number of jobs that are currently in the run-queue.††</td>
</tr>
<tr>
<td>sched.jobs</td>
<td>int</td>
<td>†</td>
<td>Jobs requiring a thread.</td>
</tr>
<tr>
<td>sched.maxinq</td>
<td>int</td>
<td>†</td>
<td>Longest run-queue length</td>
</tr>
<tr>
<td>sched.tcr</td>
<td>int</td>
<td>†</td>
<td>Thread creations.</td>
</tr>
<tr>
<td>sched.tde</td>
<td>int</td>
<td></td>
<td>Thread destructions.</td>
</tr>
<tr>
<td>sched.threads</td>
<td>int</td>
<td>†</td>
<td>Number of current scheduler threads.</td>
</tr>
<tr>
<td>sched.tlimr</td>
<td>int</td>
<td>†</td>
<td>Number of times the thread limit was reached.</td>
</tr>
</tbody>
</table>

2.2.12 Sgen Summary Data

<sched id="sgen">
  <as>0</as><et>0</et><toe>toe</toe>
</sched>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sgen.as</td>
<td>int</td>
<td></td>
<td>One if data was asynchronously gathered, 0 otherwise.</td>
</tr>
<tr>
<td>sgen.et</td>
<td>int64</td>
<td>↔</td>
<td>Elapsed milliseconds from start to completion of statistics.</td>
</tr>
<tr>
<td>sgen.toe</td>
<td>int64</td>
<td>†</td>
<td>Unix time when statistics gathering ended.</td>
</tr>
</tbody>
</table>

†† The number of active requests is represented by (sched.threads – sched.idle + sched.inq).
2.2.13 Xrootd Protocol Summary Data

```xml
<stats id="xrootd">
  <num>int</num>
  <ops>
    <open>int</open><rf>int</rf><rd>int64</rd>
    <pr>int64</pr><rv>int64</rv><rs>int64</rs>
    <wr>int64</wr><sync>int</sync>
    <getf>int</getf><putf>int</putf><misc>int</misc>
  </ops>
  <aio>
    <num>int64</num><max>int</max><rej>int64</rej>
  </aio>
  <err>int</err><rdr>int64</rdr><dly>int</dly>
  <lgn>
    <num>int</num><af>int</af><au>int</au><ua>int</ua>
  </lgn>
</stats>
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>S</th>
<th>Explanation Of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>xrootd.num</td>
<td>int</td>
<td>↑</td>
<td>Number of times the protocol was selected.</td>
</tr>
<tr>
<td>xrootd.aio.max</td>
<td>int</td>
<td>↑</td>
<td>Maximum simultaneous async I/O requests.</td>
</tr>
<tr>
<td>xrootd.aio.num</td>
<td>int64</td>
<td>↑</td>
<td>Async I/O requests processed.</td>
</tr>
<tr>
<td>xrootd.aio.rej</td>
<td>int64</td>
<td>↑</td>
<td>Async I/O requests converted to sync I/O.</td>
</tr>
<tr>
<td>xrootd.dly</td>
<td>int</td>
<td>↑</td>
<td>Number of requests that ended with a delay.</td>
</tr>
<tr>
<td>xrootd.err</td>
<td>int</td>
<td>↑</td>
<td>Number of requests that ended with an error.</td>
</tr>
<tr>
<td>xrootd.ops.getf</td>
<td>int</td>
<td>↑</td>
<td>Getfile requests.</td>
</tr>
<tr>
<td>xrootd.ops.misc</td>
<td>int</td>
<td>↑</td>
<td>Number of “other” requests.</td>
</tr>
<tr>
<td>xrootd.ops.open</td>
<td>int</td>
<td>↑</td>
<td>File open requests.</td>
</tr>
<tr>
<td>xrootd.ops.pr</td>
<td>int64</td>
<td>↑</td>
<td>Pre-read requests.</td>
</tr>
<tr>
<td>xrootd.ops.putf</td>
<td>int</td>
<td>↑</td>
<td>Putfile requests.</td>
</tr>
<tr>
<td>xrootd.ops.rf</td>
<td>int</td>
<td>↑</td>
<td>Cache refresh requests.</td>
</tr>
<tr>
<td>xrootd.ops.rd</td>
<td>int64</td>
<td>↑</td>
<td>Read requests.</td>
</tr>
<tr>
<td>xrootd.ops.rs</td>
<td>int64</td>
<td>↑</td>
<td>Readv segments.</td>
</tr>
<tr>
<td>xrootd.ops.rv</td>
<td>int64</td>
<td>↑</td>
<td>Readv requests.</td>
</tr>
<tr>
<td>xrootd.ops.sync</td>
<td>int</td>
<td>↑</td>
<td>Sync requests.</td>
</tr>
<tr>
<td>xrootd.ops.wr</td>
<td>int64</td>
<td>↑</td>
<td>Write requests.</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>S</td>
<td>Explanation Of Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>---</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>xrootd.rdr</td>
<td>int64</td>
<td>↑</td>
<td>Number of requests that were redirected.</td>
</tr>
<tr>
<td>xrootd.lgn.num</td>
<td>int</td>
<td>↑</td>
<td>Number of login attempts.</td>
</tr>
<tr>
<td>xrootd.lgn.af</td>
<td>int</td>
<td>↑</td>
<td>Number of authentication failures.</td>
</tr>
<tr>
<td>xrootd.lgn.au</td>
<td>int</td>
<td>↑</td>
<td>Number of successful authenticated logins.</td>
</tr>
<tr>
<td>xrootd.lgn.ua</td>
<td>int</td>
<td>↑</td>
<td>Number of successful un-authentication logins.</td>
</tr>
</tbody>
</table>
3 Detailed Monitoring Data Format

The `xrootd.monitor` directive specifies the monitor parameters as well as the hosts that are to receive the monitoring information. A similar directive, `frm.all.monitor` provides monitor parameters for the File Residency Manager (FRM). Monitor records are sent as UDP datagrams. Therefore, the information is sent whether or not the receiving host is enabled for the records.

Four main streams available from `xrootd` and are enabled using the `xrootd.monitor` directive, as follows:

- "f‘" stream summarizes file access events; enabled by the `fstat` option.
- "g“ stream summarizes various events; enabled by the `pfc` option.
- "r‘" stream details client redirections; enabled by the `redir` option.
- "t‘" stream details file access events; enabled by the `files`, `io`, and `iov` options.

The above three streams are continuous in that multiple events are contain in each information UDP packet. Other `xrootd` streams contain a single event per packet and provide information necessary to relate the events contained in the continuous streams. These are:

- “=“ stream provides server identification; enabled by the `ident` option.
- “d“ stream provides the identifier assigned to a user and file path; enabled by the `files` option.
- “i“ stream provides client supplied information; enabled by the `info` option.
- “u“ stream provides client login information; enabled by the `auth` and `user` options.

Finally, there are two streams available from the File Residency Manager (FRM) and are enabled using the `frm.monitor` directive. These are:

- “p“ stream provides information about file purge events from; enabled by the `purge` option.
- “x“ stream provides information on files copied into and out of the server; enabled by the `migr` and `stage` options.

Each stream is independent in that event types are not mixed together in any UDP packet. That is, a “d“ stream only contains events related to that stream. Streams other than “f‘”, “r‘”, and “t‘” are grouped under the rubric of map messages. They contain only one event per UDP packet and are described in the “Monitor Map Message Format” section. The “f‘”, “r‘”, and “t‘” streams are sufficiently complicated to deserve separate treatment.
Monitoring
3.1 Event Monitoring Overview

i. When the server starts up it sends a identification message to each stream receiver (i.e. ‘=’ record). This message may be periodically repeated, depending on the specified configuration.

ii. Each time a client logs in or authenticates, the system assigns the client a unique dictionary ID (dictid). The mapping between the dictionary ID and the client generates a separate monitor record that is sent to the destination host. This dictid is used in subsequent records that refer to the client. This only occurs if the auth or user option is specified on the xrootd.monitor directive.

iii. Each time a client opens a particular file, the system assigns the client/file-path combination a unique dictionary ID (dictid). The mapping between the dictionary ID and the client/file-path pair generates a separate monitor record that is sent to the destination host. This dictid is used in subsequent records that refer to the client’s use of the particular file. This only occurs if the files option is specified on the xrootd.monitor directive. It is meant to expedite translating the “t” stream into useful information. The fstat lfn option provides a similar feature but includes the information directly in the “f” stream. It is rare to enable the “t” and “f” streams together.

iv. Each time a client injects application information into the monitoring stream, the system assigns the information a unique dictionary ID (dictid). The mapping between the dictionary ID and the client/application pair generates a separate monitor record that is sent to the destination host. The dictid is also returned to the client to help cross reference client activities with the server. This occurs only when the info option is specified on the xrootd.monitor directive.

v. The dictid is used to compress out redundant information. Every event that is associated with a particular mapping uses the dictid for that mapping in the actual monitoring stream. Thus, it is critical for the receiver to maintain the mapping.

vi. Monitor records are formatted as structured binary records. All numeric fields within the record are sent in network byte order.

vii. Each datagram is self-consistent. That is, information is never logically split across data-grams. Mapping requests are always fully contained within a datagram. The “f” and “t” stream datagrams are always bracketed by window timing marks.

viii. The “r” stream (redirect events)) contain only a single timing mark, ostensibly to supply the server’s identification. However, each event is time stamped with a resolution equal to the timing window.
ix. Definitions of the structures and symbols described in the following sections can be found in the “XrdXrootdMonData.hh” file.

3.2 The Common Packet Header
The following figure describes the common header in each UDP packet sent by xrootd or the FRM.

```c
struct XrdXrootdMonHeader {
    kXR_char   code;   // = d f g i p r t u x
    kXR_char   pseq;   // packet sequence
    kXR_uint16  plen;  // packet length
    kXR_int32   stod;  // Unix time at Server start
};
```

Header for Each Monitor Message Data-gram

Actual information structures follow the header in the same data-gram. The code identifies the stream, as follows:

- `=` – server identification sent by xrootd or the FRM
- `d` – `dictid` of a user/path combination (xrootd only)
- `f` – file access events (xrootd only)
- `g` – general events such as file cache information (xrootd only)
- `i` – `dictid` of a user/information combination (xrootd only)
- `p` – file purge event (FRM only)
- `r` – client redirect events (xrootd only)
- `t` – a file or I/O request trace (xrootd only)
- `u` – `dictid` of the user login name and authentication (xrootd only)
- `x` – file transfer event (FRM only)

The stream code, also called the record type, is placed in the header’s `code` variable. The `pseq` variable is an ascending, wrapping, packet sequence number, whose value ranges from 0 to 255. This provides a gross mechanism to order packets. I/O event timing marks and file and redirect time stamps within the packet provide more accurate information. The `plen` variable contains the packet’s length. This value can be used to verify that the system’s reported length equals the intended length. The `stod`, defined as Unix time, is the time when the server was started. Thus, each `stod/dictid` and `stod/hostid` combination is unique across all time.
All binary information in a packet is formatted in network byte order and must be converted to host order in order to be meaningful.

### 3.3 Monitor Map Message Format

```
struct XrdXrootdMonMap
{
    XrdXrootdMonHeader hdr;
    kXR_uint32 dictid;
    char info[];
};
```

A map message ‘=’, ‘d’, ‘i’, ‘p’, ‘u’ or ‘x’ in hdr.code is generated when a client:
- user logs in (type ‘u’),
- purges a file (type ‘p’),
- transfers a file (type ‘x’),
- opens a file (type ‘d’), and
- associates information with the session (type ‘i’).

For each record other than ‘=’, ‘p’ and ‘x’, xrootd generates a unique dictionary ID and assigns it to the user/information, user/path combination, or user/authinfo. This identifier is called a **dictid**.

The **MonMap** record describes this mapping. It starts with a standard header. Following the header is the binary dictionary ID, **dictid**. This ID is unique within the server’s boot-session. That is, every time the server is restarted, the **dictid** value is reset to zero. For ‘=’, ‘p’ and ‘x’ records, the **dictid** is always zero.

In order to maintain unique **dictid**’s across multiple servers so that the **dictid** can be used as a database key, you must combine the **dictid** with the sending server’s host name or IP address, port number or instance name (if multiple servers are running on the same host), and boot time.

The **dictid** is referenced in the continuous streams (i.e. “f”, “g”, “r”, and “t”) to avoid repeating rather lengthy information in each event record. Therefore, it is necessary to collect this information in order to report events relative to specific file names and users.
3.3.1 Message Info Field

<table>
<thead>
<tr>
<th>Code</th>
<th>Contents of info</th>
<th>Code</th>
<th>Contents of info</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>userid\nsrvinfo</td>
<td>p</td>
<td>userid\npmsginfo</td>
</tr>
<tr>
<td>d</td>
<td>userid\npath</td>
<td>u</td>
<td>userid[\nauthinfo]</td>
</tr>
<tr>
<td>i</td>
<td>userid\nnappinfo</td>
<td>x</td>
<td>userid\nxfrinfo</td>
</tr>
</tbody>
</table>

userid: prot/user.pid:pid@host

authinfo: [&p=ap&n=[dn]&h=[hn]&o=[on]&r=[rn]&g=[gn]&m=[info]][loginfo]

loginfo: &x=[xeqname] &y=[minfo] &I={4|6}

prginfo: xfn\n&tod=tod&sz=bytes&at=at&ct=ct&mt=mt&fn=x

srvinfo: &pgm=prog&ver=vname&inst=iname&port=pnum&site=site

xfrinfo: lfn\n&tod=tod&sz=bytes&tm=sec&op=op&rc=rc[&pd=data]

Where:

ap    The authentication protocol name used to authenticate the client.

appinfo
Un-interpreted application or plug-in supplied information.

at    Is the file’s access time in Unix seconds.

bytes The size of the migrated, purged, or staged file in bytes.

call Is the file’s creation time in Unix seconds.

data  Is optional program monitoring data returned by the transfer command.

dn    The client’s distinguished name as reported by ap. If no name is present, the variable data is null.

gn    The client’s group names in a space-separated list. If no groups are present, the tag variable data is null.
The client’s host’s name as reported by \textit{ap}. If no host name is present, the variable data is null.

\textit{host} The host name, or IP address, where the user’s request originated.

\textit{iname} The server’s instance name as specified with the \texttt{-n} command line option. If no instance name was specified, \textit{“anon”} is reported as the instance name.

\textit{loginfo} is arbitrary monitoring information specific to the protocol being used at login time. If no information is present, the tag variable data is null.

\textit{lfn} The logical name of the transferred file.

\textit{minfo} The contents of the \texttt{XRD_MONINFO} client-side environmental variable.

\textit{mt} Is the file’s modification time in Unix seconds.

\textit{op} Is the character operation code for a file transfer event, as follows:
- \texttt{0} - Unknown operation, this usually indicates a logic error.
- \texttt{1} - File was copied into the server by client request.
- \texttt{2} - File was copied out of the server by migration system request.
- \texttt{3} - Same as \texttt{2} but the file was removed after migration.
- \texttt{4} - File was copied out of the server by client request.
- \texttt{5} - Same as \texttt{4} but the file was removed after the copy completed.
- \texttt{6} - File was copied into the server by staging system request.

\textit{on} The client’s organization name as reported by \textit{ap}. If no organization is present, the tag variable data is null.

\textit{path} The full path name of the file being opened.

\textit{pid} The user’s process number that issued the request.

\textit{pnum} The server’s main port number.

\textit{prog} The name of the server’s executable program.

\textit{prot} The communication protocol being used by the client (e.g., xroot, http, etc).
The return code. If the request was successful, the return code is zero. Otherwise, the request failed. For failing stage requests, the bytes is also zero.

The client’s role name as reported by prot. If no role name is present, the variable data is null.

The number of seconds it took to migrate or stage in the file (i.e., the time between the start of the request to the time the request completed).

The server’s identification processing the connection to user:pid at host. The identification is constructed as “getpid()<<16|pnum” resulting in a unique number when combined with the server’s start time and host name.

The server’s designated site name.

The Unix seconds, as returned by time(), when the record was produced.

The Unix username of the user as reported by the client (i.e. unverified) or the plug-in identifier. You can distinguish between the two as plug-ins always report a pid of zero. The user name is the plug-in component name (e.g. pfc).

The name of the executable program the client is running with the path removed.

The logical or physical name of the file that was purged. The “fn” tag indicated the name’s type (see the description of the x value).

The server’s version identification string.

Is the letter ‘l’ if xfn is a logical file name (LFN) or ‘p’ if it is a physical file name (PFN). Normally, x should always be ‘l’. See the notes for exceptions.

The client’s network mode: 4 for IPv4 and 6 for IPv6.

Notes

1) The sid The ‘d’, ‘i’ and ‘u’ records with authinfo contain two ASCII text strings, separated by a new-line (\n) character.

2) The ‘p’ and ‘x’ records contain three ASCII text strings, separated by a new-line (\n) character.
3) The “fn” tag in the ‘p’ record should normally have a value of ‘l’. When an error occurs translating the physical file name to its logical counterpart, the physical name is reported and the tag value is set to ‘p’. Reporting of physical names should be treated as an error and is likely due to a misbehaving name-to-name plug-in.

4) The ‘u’ records do not end with a new-line character if authinfo was not requested when configuring monitoring.

5) The server’s site name is arbitrary and optional and may be specified on the command line or in the configuration file.

6) Mapping packets can be sent at any time. Interspersed with the mapping packets are file, redirect, and trace packets.

7) The server identification packet, ‘=’, may be sent on a periodic basis. See the ident option on the xrootd.monitor and frm.all.monitor directives.
3.4 The “f” Stream (fstat)

The “f” stream is enabled using the `fstat` option on the `xrootd.monitor` directive. It contains multiple types of variable length structures, each describing a particular event. However, it is always framed in the same way:

- The packet starts with the standard header (`XrdXrootdMonHeader`),
- followed by the UNIX time of the first event entry in the packet, encapsulated in an `XrdXrootdMonFileTOD` structure, and
- is followed by one or more variable length structures detailing specific events.

The following diagram shows the packet structure.

```
struct XrdXrootdMonHeader;
struct XrdXrootdMonFileTOD;
*  
* 1 or more of XrdXrootdMonFileCLS, XrdXrootdMonFileIO,  
*       and XrdXrootdMonFileOPN  
*
```

Because the structures are variable length, each one starts (i.e. contains as its first member) a standard header structure, `XrdXrootdMonFileHdr`, that not only details the type of structure but also the length of the structure. It is always followed by the `XrdXrootdMonFileTOD` structure, as follows:

```
struct XrdXrootdMonFileHdr
{
char      recType;  // Identifies type of structure
char      recFlag;  // Structure specific flags
short     recSize;  // Size of this structure in bytes
union     {kXR_unt32 fileID;   // dictid if recType != isTime
           kXR_unt32 userID;   // dictid if recType == isDisc
           short     nRecs[2]; // isTime: nRecs[0] == isXfr recs
                        //              nRecs[1] == total recs
        }
};
```

You must use the length `XrdXrootdMonFileHdr::recSize` to skip to the next structure in the packet as members may be added causing the structure to change in length. The `recSize` member will always have the correct size of the enclosing structure.
The recType member identifies the type of structure. The value comes from the recTval enum defined in the structure but not shown in the graphic. The following table summarizes the possible values (recType values, when used, should be preceded by “XrdXrootdMonFileHdr::”).

<table>
<thead>
<tr>
<th>XrdXrootdMonFileHdr::recType</th>
<th>Structure</th>
<th>Encompassing Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>isClose</td>
<td>XrdXrootdMonFileCLS</td>
<td></td>
</tr>
<tr>
<td>isDisc</td>
<td>XrdXrootdMonFileDSC</td>
<td></td>
</tr>
<tr>
<td>isOpen</td>
<td>XrdXrootdMonFileOPN</td>
<td></td>
</tr>
<tr>
<td>isTime</td>
<td>XrdXrootdMonFileTOD</td>
<td></td>
</tr>
<tr>
<td>isXFR</td>
<td>XrdXrootdMonFileXFR</td>
<td></td>
</tr>
</tbody>
</table>

The recFlag member contains structure-specific flags which are discussed along with each structure. The recSize member contains the actual size of the structure. After converting it to host byte order, it must be used to find the start of the next structure in the packet. Since each structure starts with XrdXrootdMonFileHdr, it is easy to make a determination as to which actual structure the header should be cast to.

When recType is neither isDisc nor isTime then fileID in the structure contains the dictid assigned to the file associated with the structure. When recType is isDisc then userID in the structure contains the dictid assigned to the disconnecting user. Otherwise, nRecs should be used as it contains the number of records in the packet and can be used to distribute events across the reporting time interval.

As previously noted, XrdXrootdMonHeader is always the first structure in the packet and is always followed by the XrdXrootdMonFileTOD structure, as follows:

```c
struct XrdXrootdMonFileTOD
{
    XrdXrootdMonFileHdr Hdr;   // recType == isTime
    int                 tBeg; // time(0) of following record
    int                 tEnd; // time(0) when packet was sent
    kXR_int64           sID;  // Server identifier lower 48 bits
};
```

The tbeg value is the Unix time when the following record was added to the packet and tEnd is the Unix time when the packet was sent. Recall that the Hdr contains the number of subsequent records in the packet in the nRecs field in this record type.
Monitoring

The server’s identifier appears in each \texttt{XrdXrootdMonFileTOD} entry in the \texttt{sID} member. This is identical to the \texttt{sid} in the \texttt{userid} in map entries. It is encoded in the lower 48 bits of the first 8 bytes and always appears once after the header. You can extract the \texttt{sid} with the following expression

\[
\texttt{ntohl64(sID) \& XROOTD_MON_SIDMASK}
\]

Definitions of the structures and symbols described here can be found in the “\texttt{XrdXrootdMonData.hh}” file.

The following table lists all possible values in the header \texttt{recFlag} member for the TOD structure. The values are defined in \texttt{XrdXrootdMonFileHdr::recFval} enum. However, they must be tested individually using a bitwise “and” operator. The \texttt{recFlag} values, when used, should be preceded by “\texttt{XrdXrootdMonFileHdr::}”.

<table>
<thead>
<tr>
<th>\texttt{XrdXrootdMonFileHdr::recFlag}</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasSID</td>
<td>The \texttt{sID} member is present</td>
</tr>
</tbody>
</table>

If you are using the \texttt{sID} member, you should test if the \texttt{hasSID} flag is set. Old records did not have this member.
3.4.1 Disc Event
When a client disconnects from the server, an **isDisc** record is placed in the f-stream. This record consists of nothing more than the header and identifies the disconnecting user. It is always the last record generated by the user.

```c
struct XrdXrootdMonFileDSC {
  XrdXrootdMonFileHdr Hdr;   // recType == isDisc
};
```

The `tbeg` value is the Unix time when the following record was added to the packet and `tEnd` is the Unix time when the packet was sent. Recall that the `Hdr` contains the number of subsequent records in the packet in the `nRecs` field in this record type.

3.4.2 Open Event
Information regarding a file open event is shown below.

```c
struct XrdXrootdMonFileLFN {
  kXR_unt32 user;     // dictid for the user
  char     lfn[1032];  // Variable length!
};
struct XrdXrootdMonFileOPN {
  XrdXrootdMonFileHdr Hdr;      // recType == isOpen
  long long fsz;             // file size at open
  XrdXrootdMonFileLFN ufn;     // OPTIONAL
};
```

Open events insert the variable length structure `XrdXrootdMonFileOPN` into the “f” stream. The structure is variable because it may or may not contain the `XrdXrootdMonFileLFN` structure. If the structure exists then `recFlag` indicates this. The structure is included if the lfn option is specified on the `xrootd.monitor` directive. The reason this is optional is because a “d” map record is sent if the `files` option is specified as well. In this case, there is no reason to duplicate the information.
If `XrdXrootdMonFileLFN` is present it is variable in size. The structure merely defines the maximum size of the `lfn` which makes it convenient to use functions like `strcpy()` without the compiler warning that the copy exceeds the length of the buffer. Since the string defined in `ufn.lfn` is guaranteed to end with a null byte, all string functions can be used on this array.

Preceding the array is the `dictid` assigned to the client that performed the open in `ufn.user`. This `dictid` is reported in the “u” stream (i.e. “u” map info) when the client initiates a session. This is enabled with the `auth` or `user` options on the `xrootd.monitor` directive. If neither has been selected, the `dictid` is reported as zero (i.e. unassigned).

The following table lists all possible values in the header `recFlag` member for the open structure. The values are defined in `XrdXrootdMonFileHdr::recFval` enum. However, they must be tested individually using a bitwise “and” operator. The `recFlag` values, when used, should be preceded by “`XrdXrootdMonFileHdr::`”.

<table>
<thead>
<tr>
<th><code>XrdXrootdMonFileHdr::recFlag</code></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasLFN</td>
<td><code>XrdXrootdMonFileLFN</code> present</td>
</tr>
<tr>
<td>hasRW</td>
<td>File opened for reads &amp; writes</td>
</tr>
</tbody>
</table>
3.4.3 Close Event

The close structure describing a close event is shown below.

```c
struct XrdXrootdMonStatOPS
{
    int     read;     // Number of read() calls
    int     readv;    // Number of readv() calls
    int     write;    // Number of write() calls
    short   rsMin;    // Smallest readv() segment count
    short   rsMax;    // Largest readv() segment count
    long long rsegs; // Number of readv() segments
    int     rdMin;    // Smallest read() request size
    int     rdMax;    // Largest read() request size
    int     rvMin;    // Smallest readv() request size
    int     rvMax;    // Largest readv() request size
    int     wrMin;    // Smallest write() request size
    int     wrMax;    // Largest write() request size
};

union XrdXrootdMonDouble
{
    long long dlong;
    double dreal;
};

struct XrdXrootdMonStatSDV
{
    XrdXrootdMonDouble read; // Sum(all read requests)^2 (bytes)
    XrdXrootdMonDouble readv; // Sum(all readv requests)^2 (bytes)
    XrdXrootdMonDouble rsegs; // Sum(all readv segments)^2 (count)
    XrdXrootdMonDouble write; // Sum(all write requests)^2 (bytes)
};

struct XrdXrootdMonStatXFR
{
    long long read;     // Bytes read from file using read()
    long long readv;    // Bytes read from file using readv()
    long long write;    // Bytes written to file
};

struct XrdXrootdMonFileCLS     // Variable Length!
{
    XrdXrootdMonFileHdr Hdr;     // Always present
    XrdXrootdMonStatXFR Xfr;     // Always present
    XrdXrootdMonStatOPS Ops;     // OPTIONAL
    XrdXrootdMonStatSSQ Ssq;     // OPTIONAL
};
```
When a file is closed, a XrdXrootdMonFileCLS structure is inserted into the “f” stream. It is variable in length because certain statistics are optional. The recFlag bits in the XrdXrootdMonFileHdr record indicate which structures are present. Additionally, the flag indicates whether or not the client actually closed the file. The values are defined in XrdXrootdMonFileHdr::recFval enum. However, they must be tested individually using a bitwise “and” operator. The following table summarizes the possible values (recFlag values, when used, should be preceded by “XrdXrootdMonFileHdr::”).

<table>
<thead>
<tr>
<th>XrdXrootdMonFileHdr::recFval</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>forced</td>
<td>Disconnect prior to close</td>
</tr>
<tr>
<td>hasOPS</td>
<td>XrdXrootdMonFileOPS present</td>
</tr>
<tr>
<td>hasSSQ</td>
<td>XrdXrootdMonFileSSQ present</td>
</tr>
</tbody>
</table>

The XrdXrootdMonFileOPS structure is inserted when the ops option is specified in the xrootd.monitor directive. It is important to note that the minimum and maximum values for readv requests represent bytes for a complete quest request (i.e. sum of all segments). It is not the minimum and maximum of any individual segment.

The XrdXrootdMonFileSSQ structure is inserted when the fstat ssq option is specified. The counts can be used to compute the standard deviation for read and write request sizes using the formulae show below. Normally, ssq implies ops because standard deviation cannot be computed without the operation counts.

\[ S = \sqrt{\frac{\sum fx^2 - \left( \frac{\sum fx}{n} \right)^2}{n}} \]

The sum of squares count is reported in network byte order using the IEEE 754 floating point format. The counts are not available on platforms that do not support the IEEE 754 format.
3.4.4 Xfr Event

The “f” stream may contain transfer events when the `fstat xfr` option is specified. These events detail in-progress data transfers for currently open files. One such event is produced for each open file that has had I/O activity since the last report. Because it is time driven, files opened during the reporting may or may not be included in the event stream. However, if they are still open they are included during the next reporting interval. In all cases, an open event always precedes a transfer event for that file. Xfr events for a file can never appear after the file’s close event entry.

The following details the transfer event data structure.

```c
struct XrdXrootdMonStatXFR
{
    long long  read;  // Bytes using read()
    long long  readv; // Bytes using readv()
    long long  write; // Bytes using write()
};

struct XrdXrootdMonFileXFR
{
    XrdXrootdMonFileHdr  Hdr;  // recType == isXfr
    XrdXrootdMonStatXFR Xfr;   // Current bytes so far
};
```
3.5 The “g” Stream (ccm and pfc)

The “g” stream is enabled using the \texttt{pfc} option on the \texttt{xrootd.monitor} directive. It contains multiple types of variable length \texttt{ASCII} newline separated text fields, each describing a particular event. However, it is always framed in the same way:

- The packet starts with the standard header (\texttt{XrdXrootdMonHeader}).
- It is followed by temporal information which contains the UNIX time of the first event entry followed by the UNIX time of the last event entry. The time stamps are followed by an encoded a plug-in and server identifier.
- The temporal information is followed by one or more variable length \texttt{ASCII} newline separated text fields detailing specific events. The last field always ends with a null byte.

The following diagram shows the packet structure.

```c
struct XrdXrootdMonGS;
{
    struct XrdXrootdMonMonHeader hdr;
    Int       tBeg; // UNIX time of first entry
    Int       tEnd; // UNIX time of last entry
    kXR_int64 sID; // Provider identification
};
```

The content of each text string is specific to the plug-in that generates the information. The plug-in chooses which format to use (e.g. CGI, JSON, xml, etc).

The \texttt{sID} identifies who generated the information. In host byte order, the first eight bits contain the provider’s identification while the last 48 bits contain the server’s identification. The table below lists possible providers:

<table>
<thead>
<tr>
<th>Provider (i.e. plug-in)</th>
<th>Contents of 1\textsuperscript{st} 8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Context Manager (ccm)</td>
<td>XROOTD_MON_GSCCM</td>
</tr>
<tr>
<td>Proxy File Cache (pfc)</td>
<td>XROOTD_MON_GSPFC</td>
</tr>
</tbody>
</table>
You can extract the provider’s identifier with the following expression

\[(\text{ntohl}(\text{sID}) \gg XROOTD\_MON\_PIDSHFT) \& XROOTD\_MON\_PIDMASK)\]

You can extract the server’s identifier with the following expression

\[\text{ntohl}(\text{sID}) \& XROOTD\_MON\_SIDMASK\]

Definitions of the structures and symbols described here can be found in the “\text{XrdXrootdMonData.hh}” file.

While the g stream can be fed by many different information providers no UDP packet will every contain information from more than a single provider. However, the packets may be intermixed and you will need to separate the streams and sequentially order the packets using the sID, the time stamps, and the packet sequence number. Be aware that each provider’s g stream uses its own packet sequence. Hence, each provider’s stream must be ordered independently.

Since information providers define the format of the data format contained in the UDP packet, the actual contents is described in the manual associated with the provider, as follows:

<table>
<thead>
<tr>
<th>Provider ID</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>XROOTD_MON_GSCCM</td>
<td>This plug-in is not core component.</td>
</tr>
<tr>
<td>XROOTD_MON_GSPFC</td>
<td>Proxy Storage Services Reference</td>
</tr>
</tbody>
</table>
3.6 The “r” Stream (redir)

```c
struct XrdXrootdMonRedir
{
    union {
        kXR_int32 Window;
        struct {
            kXR_char Type;
            kXR_char Dent;
            kXR_int16 Port;
        } rdr;
    } arg0;

    union {
        kXR_unt32 dictid;
        kXR_int32 Window; } arg1;
};

struct XrdXrootdMonBurr
{
    XrdXrootdMonHeader hdr;
    union {
        kXR_int64 sID;
        kXR_char sXX[8];
    };

    XrdXrootdMonRedir info[];
};
```

The **MonRedir** record is highly encoded and repeated as often as possible in a single datagram, as shown in the **MonBurr** structure. Each instance of info represents a server identification record‡‡, a redirect record, or a window timing mark. All binary data appears in network byte order. The **info[]**.arg0.Type character identifies the type of information the entry contains. The character is bit encoded and should be tested for the proper bit values to determine the type of record, as follows:

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XROOTD_MON_REDTIME</td>
<td>0x00</td>
<td>Window timing mark§§</td>
</tr>
<tr>
<td>XROOTD_MON_REDIRECT</td>
<td>0x8x</td>
<td>Redirect event generated by <strong>cmsd</strong></td>
</tr>
<tr>
<td>XROOTD_MON_REDDLOCAL</td>
<td>0x9x</td>
<td>Redirect event generated by <strong>xrootd</strong></td>
</tr>
<tr>
<td>XROOTD_MON_REDSID</td>
<td>0xf0</td>
<td>Server identification</td>
</tr>
</tbody>
</table>

**XROOTD_MON_REDIRECT** and **XROOTD_MON_REDDLOCAL** entries are variable length but always occupy and integral multiple of 8 characters (i.e., are padded out to always end on an 8-byte boundary). The **“info[]**.arg0.Dent” indicates

‡‡ The server identification record always appears after the header and is never repeated in the packet.

§§ Window timing marks are indicated when the high order bit is not set.
Monitoring

how many 8-byte words, less one, in the record. All other records occupy exactly eight bytes and the “Dent” field is used for other purposes.

Additionally, the low order four bits of the XROOD_MON_REDIRECT and XROOTD_MON_REDLOCAL entry codes are modified by inserting the operation code in the last four bits of the symbol value. You can obtain the operation that caused the redirect by looking at the last four bits and comparing it to the following symbols.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XROOTD_MON_CHMOD</td>
<td>0x01</td>
<td>Change file mode.</td>
</tr>
<tr>
<td>XROOTD_MON_LOCATE</td>
<td>0x02</td>
<td>Locate file or directory.</td>
</tr>
<tr>
<td>XROOTD_MON_OPENDIR</td>
<td>0x03</td>
<td>Open directory for reading.</td>
</tr>
<tr>
<td>XROOTD_MON_OPENC</td>
<td>0x04</td>
<td>Open file for creation.</td>
</tr>
<tr>
<td>XROOTD_MON_OPENR</td>
<td>0x05</td>
<td>Open file for reading.</td>
</tr>
<tr>
<td>XROOTD_MON_OPENW</td>
<td>0x06</td>
<td>Open file for writing.</td>
</tr>
<tr>
<td>XROOTD_MON_MKDIR</td>
<td>0x07</td>
<td>Create a directory or path.</td>
</tr>
<tr>
<td>XROOTD_MON_MINV</td>
<td>0x08</td>
<td>Rename a file or directory.</td>
</tr>
<tr>
<td>XROOTD_MON_PREP</td>
<td>0x09</td>
<td>Prepare request.</td>
</tr>
<tr>
<td>XROOTD_MON_QUERY</td>
<td>0x0a</td>
<td>Query information request.</td>
</tr>
<tr>
<td>XROOTD_MON_COLS</td>
<td>0x0b</td>
<td>Remove a file.</td>
</tr>
<tr>
<td>XROOTD_MON_RMDIR</td>
<td>0x0c</td>
<td>Remove a directory.</td>
</tr>
<tr>
<td>XROOTD_MON_STAT</td>
<td>0x0d</td>
<td>Stat a file or directory.</td>
</tr>
<tr>
<td>XROOTD_MON_TRUNC</td>
<td>0x0e</td>
<td>Truncate a file.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents For Redirect Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>info[].arg0.Type</td>
<td>XROOTD_MON_REDIRECT or XROOTD_MON_REDLOCAL plus modifier. See modifier table for operation being performed.</td>
</tr>
<tr>
<td>info[].Dent</td>
<td>Number of 8-byte entries used by this entry less 1.</td>
</tr>
<tr>
<td>info[].Port</td>
<td>The server’s port number to which the client is being redirected to.</td>
</tr>
<tr>
<td>info[].dictid</td>
<td>The client’s dictionary ID (‘u’ map message).</td>
</tr>
<tr>
<td>info[].arg1+4</td>
<td>The server’s name and path being accessed. It always ends with a null byte.</td>
</tr>
</tbody>
</table>
Monitoring

The server’s “name” and target path follow the eight byte entry. The number of eight byte words occupied by this information is recorded in “info[arg0].Dent”. This always appears as a null terminated character string with the following format:

```
[servername]:pathname

servername: dnsname | ipv4address | [ipv6address]
```

When the server’s name is not present in the record, it means that the client has been directed to a physical file on the client’s host whose physical name is `pathname` (i.e. the `pfn`). Otherwise, `pathname` is the logical file name (i.e. `lfn`) used by the client.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents For Window Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>info[arg0].Type</td>
<td>XROOTD_MON_REDTIME</td>
</tr>
<tr>
<td>info[arg0].Window</td>
<td>Window size in the low order 24 bits.</td>
</tr>
<tr>
<td>info[arg1].Window</td>
<td>Unix time of the new window.</td>
</tr>
</tbody>
</table>

Since each datagram is self-consistent, a window entry will always appear before any redirect entries (i.e. the first entry after the server identification entry) with the last entry being another window entry. Additional window entries may be placed within the message should redirect requests cross window boundaries within the same data-gram. Because request timing is variable, window start and end times are rarely adjacent. That is, a window may end at time x but the new window may start at a time that is many windows away from the end time. This is because `xrootd` compresses adjacent empty windows.

To obtain the end time of a window you must add `info[arg0].Window` (low order 24 bits) to the previous `info[arg1].Window`. While this should not be done for the first window entry, it should be done for all subsequent window entries.

A window entry may also be forced should the buffer fill or the connection is closed before the window actually ends. In this case, the window may be substantially smaller than configured window size. The receiver should not count that each window will be the same size. When this happens, the `info[last-1].arg1.Window` value will be the same as the `info[last].arg1.Window` value. The receiver should internally time-stamp each entry using an appropriate distribution curve within the reported window.
Monitoring

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents For Server Identification Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>info[].arg0&gt;Type</td>
<td>XROOTD_MON_REDSID</td>
</tr>
<tr>
<td>info[]</td>
<td>Server’s identification in the low order 48 bits.</td>
</tr>
</tbody>
</table>

The server’s identifier appears in each XROOTD_MON_REDSID entry. This is identical to the sid in the userid in map entries. It is encoded in the lower 48 bits of the first 8 bytes and always appears once after the header. You can extract the sid with the following statement:

\[ \text{ntohl}(\text{sid} \& \text{XROOTD_MON_SIDMASK}) \]

Definitions of the structures and symbols described here can be found in the “XrdXrootdMonData.hh” file.

3.6.1 Understanding Multiple Redirection Streams

In order to maximize parallelism, xrootd maintains several redirection monitoring streams, assigning each request to the first available stream. The number of streams may be specified on the xrootd.monitor directive. The default is 3 streams.

Because multiple streams exist, event data ordering is non-deterministic within the monitoring window. That is, it is impossible to tell the order of a specific sequence of requests within a window once the streams are merged.
3.7 The “t” Stream (files, io, and iov)

The “t” stream is produced when the files, io, or iov options are used on the xrootd.monitor directive. The information contained in this stream is a virtual superset of the information in the “f” stream. This is because all values in the “f” stream can be derived from the “t” stream. However, the “t” stream also provides the ability to obtain insights on data access pattern information that is not available elsewhere. However, this comes at a substantial cost since all data seeks are reported. This results in a substantial amount of monitoring information and about 7% degradation in server performance. In almost all cases, the “f” stream provides sufficient monitoring information.

In order to maintain a low overhead, each connection collects its own I/O event data in a local buffer and sends the data when the buffer is full or when the connection is closed. Non-I/O events (e.g., open, close, etc) are collected globally in a separate stream while redirect events are collected globally as one or more separate streams. A stream buffer is sent when it is full or when the specified timeout occurs. I/O and non-I/O events may be intermixed when the configuration specifies a particular recipient for such a combination. Low overhead is also maintained by not time-stamping each event. That is, the information is collected within a statistical window. While the order of events is maintained, it is impossible to tell precisely when the event actually happened within this window. The receiver should uniformly distribute the events across the window.

Since each connection maintains its own local buffer of I/O events, multiple datagrams may be sent with disparate, possibly overlapping, windows. The receiver must merge all of these windows into a uniform coherent time stream. This is possible because precise times are always given for the start and end of the window in which the events were collected. Care should be taken to appropriately order the packets, as UDP packets can arrive in any order. To assist in ordering packets, each packet carries a time-stamp as well as a sequence number so that the receiver can easily order packets as well as discover if any packets were lost due to network congestion.

The next page illustrates one a typical “t” stream. It also illustrates the physical and logical sequence of packets from a single server. You should note that the window start and end times do not correlate with the packet send time. Hence, packet reordering is typically necessary to get a linear view of time.
Monitoring

Packet physical order:
- Packet 0: t=4 seq=2 window=tod+a:tod+b <I/O requests>
- Packet 1: t=4 seq=1 window=tod+x:tod+y <I/O requests>
- Packet 2: t=1 seq=0 window=tod+j:tod+k <I/O requests>
- Packet 3: t=5 seq=3 window=tod+d:tod+e <I/O requests>

Packet logical order:
- Packet 2: t=1 seq=0 window=tod+j:tod+k <I/O requests>
- Packet 1: t=4 seq=1 window=tod+x:tod+y <I/O requests>
- Packet 0: t=4 seq=2 window=tod+a:tod+b <I/O requests>
- Packet 3: t=5 seq=3 window=tod+d:tod+e <I/O requests>

Window logical order:
- Packet 0: t=4 seq=2 window=tod+a:tod+b <I/O requests>
- Packet 3: t=5 seq=3 window=tod+d:tod+f <I/O requests>
- Packet 2: t=1 seq=0 window=tod+j:tod+k <I/O requests>
- Packet 1: t=4 seq=1 window=tod+x:tod+y <I/O requests>

In order to maximize the amount of information that can be stored in a single data-gram as well as to minimize redundancy, a dense encoding scheme is used. The messages are described in the following sections.

### 3.7.1 Monitor Trace Message Format

```c
struct XrdXrootdMonTrace
{
    union {kXR_int64  val;
        kXR_char   id[8];
        kXR_int16  sVal[4];
        kXR_unt32  rTot[2];
    }               arg0;
    union {kXR_int32  buflen;
        kXR_unt32  HostID;
        kXR_unt32  wTot;
        kXR_int32  Window;
    }               arg1;
    union {kXR_unt32  dictid;
        kXR_int32  Window;
    }               arg2;
};

struct XrdXrootdMonBuff
{
    XrdXrootdMonHeader hdr;
    XrdXrootdMonTrace  info[];
};
```
The **MonTrace** record is highly encoded and repeated as often as possible in a single datagram, as shown in the **MonBuff** structure. Each instance of info represents a read, write, open, close request, application id, or a window timing mark. All binary data is appears in network byte order. The `info[].arg0.id[0]` character identifies the type of information the entry contains, as follows:

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XROOTD_MON_OPEN</td>
<td>0x80</td>
<td>File has been opened</td>
</tr>
<tr>
<td>XROOTD_MON_READV</td>
<td>0x90</td>
<td>Details for a kXR_readv request</td>
</tr>
<tr>
<td>XROOTD_MON_READU</td>
<td>0x91</td>
<td>Unpacked details for kXR_readv</td>
</tr>
<tr>
<td>XROOTD_MON_APPID</td>
<td>0xa0</td>
<td>Application provided marker</td>
</tr>
<tr>
<td>XROOTD_MON_CLOSE</td>
<td>0xc0</td>
<td>File has been closed</td>
</tr>
<tr>
<td>XROOTD_MON_DISC</td>
<td>0xd0</td>
<td>Client has disconnected</td>
</tr>
<tr>
<td>XROOTD_MON_WINDOW</td>
<td>0xe0</td>
<td>Window timing mark</td>
</tr>
<tr>
<td>---</td>
<td>&lt;=0x7f</td>
<td>Read or write request</td>
</tr>
</tbody>
</table>

Some records contain additional flags. The definition and meaning of these flags is described below:

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XROOTD_MON_FORCED</td>
<td>0x01</td>
<td>Entry due to forced disconnect.</td>
</tr>
<tr>
<td>XROOTD_MON_BOUNDP</td>
<td>0x02</td>
<td>Entry for a bound path.</td>
</tr>
</tbody>
</table>

Entries in the I/O and non-I/O event streams are always of fixed size (i.e., 16 characters). The following fields are used for each type of record:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents For Appid Request Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>info[].arg0.id[0]</td>
<td>XROOTD_MON_APPID</td>
</tr>
<tr>
<td>info[].arg0.id[1...3]</td>
<td>Reserved.</td>
</tr>
<tr>
<td>info[].arg0.id[4...15]</td>
<td>Up to 12 characters of application identification.</td>
</tr>
</tbody>
</table>

***Indicates that if the high order bit is zero, then this is a read/write request.
### Field Contents For Close Request Entries

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info[].arg0.id[0]</code></td>
<td>XROOTD_MON_CLOSE</td>
</tr>
<tr>
<td><code>info[].arg0.id[1]</code></td>
<td>Number of bits <code>Info[].arg0.rTot[1]</code> has been right shifted to fit into a 32-bit unsigned int.</td>
</tr>
<tr>
<td><code>info[].arg0.id[2]</code></td>
<td>Number of bits <code>Info[].arg1.wTot</code> has been right shifted to fit into a 32-bit unsigned int.</td>
</tr>
<tr>
<td><code>info[].arg0.id[3]</code></td>
<td>Reserved.</td>
</tr>
<tr>
<td><code>info[].arg0.rTot[1]</code></td>
<td>Scaled number of bytes read from the file.</td>
</tr>
<tr>
<td><code>info[].arg1.wTot</code></td>
<td>Scaled number of bytes written to the file.</td>
</tr>
<tr>
<td><code>info[].arg2.dictid</code></td>
<td>The file path’s dictionary ID (‘d’ map message).</td>
</tr>
</tbody>
</table>

### Field Contents For Disconnect Request Entries

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info[].arg0.id[0]</code></td>
<td>XROOTD_MON_DISC</td>
</tr>
<tr>
<td><code>info[].arg0.id[1]</code></td>
<td>May contain XROOTD_MON_BOUNDP and XROOTD_MON_FORCED</td>
</tr>
<tr>
<td><code>info[].arg0.id[2...7]</code></td>
<td>Reserved.</td>
</tr>
<tr>
<td><code>info[].arg1.buflen</code></td>
<td>Number of seconds that client was connected.</td>
</tr>
<tr>
<td><code>info[].arg2.dictid</code></td>
<td>The client’s dictionary ID (‘u’ map message).</td>
</tr>
</tbody>
</table>

### Field Contents For Open Request Entries

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info[].arg0.id[0]</code></td>
<td>XROOTD_MON_OPEN</td>
</tr>
<tr>
<td><code>info[].arg0.id[1...7]</code></td>
<td>Size of the file in bytes.</td>
</tr>
<tr>
<td><code>info[].arg1</code></td>
<td>Reserved.</td>
</tr>
<tr>
<td><code>info[].arg2.dictid</code></td>
<td>The file path’s dictionary ID (‘d’ map message).</td>
</tr>
</tbody>
</table>

### Field Contents For Read/Write Request Entry

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info[].arg0.val</code></td>
<td>Read or write offset (see below).</td>
</tr>
<tr>
<td><code>info[].arg1.buflen</code></td>
<td>Length of the read when non-negative. When negative, this is the length of a write request.</td>
</tr>
<tr>
<td><code>info[].arg2.dictid</code></td>
<td>The file path’s dictionary ID (‘d’ map message).</td>
</tr>
</tbody>
</table>

### Field Contents For Readv Request Entry

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info[].arg0.id[0]</code></td>
<td>XROOTD_MON_READU</td>
</tr>
<tr>
<td><code>info[].arg0.id[1]</code></td>
<td><code>readv</code> request identifier</td>
</tr>
<tr>
<td><code>info[].arg0.sVal[1]</code></td>
<td>Number of elements in the <code>readv</code> vector</td>
</tr>
<tr>
<td><code>Info[].arg0.rTot[1]</code></td>
<td>Reserved.</td>
</tr>
<tr>
<td><code>info[].arg1.buflen</code></td>
<td>Length of the read.</td>
</tr>
<tr>
<td><code>info[].arg2.dictid</code></td>
<td>The file path’s dictionary ID (‘d’ map message).</td>
</tr>
</tbody>
</table>
Since each datagram is self-consistent, a trace message will always start and end with a window entry. Additional window entries may be placed within the record should requests cross window boundaries within the same data-gram. Because request timing is variable, window start and end times are rarely adjacent. That is, a window may end at time x but the new window may start at a time that is many windows away from the end time. This is because xrootd compresses adjacent empty windows.

A window entry may also be forced should the buffer fill or the connection is closed before the window actually ends. In this case, the window may be substantially smaller than configured window size. The receiver should not count that each window will be the same size. The receiver should internally time-stamp each entry using an appropriate distribution curve within the reported window.

The only difference between XROOTD_MON_READU and XROOTD_MON_READV entries is that the XROOTD_MON_READU indicates that individual read entries described by XROOTD_MON_READU follow the entry. The number of read entries equals the number specified in info[].arg0.sVal[1]. That is, the reads in the readv request vector are unpacked and presented as individual reads. This happens with the iov option is specified on the xrootd.monitor directive. This allows you to associate read entries with a particular readv request.
Monitoring

A single `readv` request may generate multiple `XROOTD_MON_READU` or `XROOTD_MON_READV` records. A record is generated whenever the vector switches from reading one file to another file. The information in the record then pertains to a file identified by the `dictid`. Multiple entries associated with a single `readv` request will always have the same request identifier placed in `info[].arg0.id[1]`. Request identifiers cycle every 256 `readv` requests.

An `XROOTD_MON_READV` entry may be followed by multiple read entries. This happens with the `iov` option was specified on the `xrootd.monitor` directive. The read entries detail each element in the `readv` vector. The preceding `XROOTD_MON_READV` entry indicates how many read entries follow. This allows you to associate read entries with a particular `readv` request. Request identifiers cycle every 256 `readv` requests.

The server’s identifier appears in each `XROOTD_MON_WINDOW` entry. This is identical to the `sid` in the `userid` in map entries. It is encoded in the lower 48 bits of the first 8 bytes. You can extract the `sid` with the following statement

```c
ntohl(info[].arg0.val & XROOTD_MON_SIDMASK)
```
4 Document Change History

14 July 2009
• This manual was introduced.

16 July 2009
• Added example on mpstats.

17 February 2010
• Correct the xrd.report directive example.
• Move toe id from statistics end-tag to the sgen part.

24 May 2011
• Correct description of the Monitor Map Message. Specifically, remove the ‘v’ record and expand on the ‘u’; record.

14 June 2011
• Describe the cms protocol summary report information.
• Indicate which int and int64 values are increasing or variable.

29 June 2011
• Add dly, err and rdr statistics to the xrootd protocol summary data.
• Indicate that aio.num, aio.rej, ops.pr, ops.rd and ops.wr values are actually int64 in size in the xrootd protocol summary data.

5 October 2011
• Describe the &g and &m fields in the authinfo monitor record.

--------- Release 3.1.0

10 October 2011
• Document the new XROOTD_MON_READV monitor record.
• Document the new ‘m’, and ‘p’ map records.
• Describe additional information added to the ‘s’ map record.
Monitoring

25 October 2011
- Describe the migration (‘m’), staging (‘s’), and purging (‘p’) monitor records.
- Describe the new option on the xrootd.monitor directive that enables migration and purging monitor records.
- Describe the xpd option on the frm.xfr.copypcmd directive that allows the transfer script to add monitoring information to the migration and staging records.
- Document <lgn> tag in the xrootd summary statistics.
- Describe the XROOTD_MON_BOUNDP and the XROOTD_MON_FORCED flags that may appear in the disconnect record.

3 November 2011
- Document the new XROOTD_MON_REDIRECT and XROOTD_MON_REDFHOST monitor records.

3 December 2011
- Document the new server identification map (‘=’) record.
- Combine the migration (‘m’) and staging (‘s’) map records under a single transfer (‘x’) map record.

13 December 2011
- Add more descriptive information about redirection events.
- Document changes in the XROOTD_MON_REDIRECT and XROOTD_MON_REDFHOST monitor record.

14 January 2012
- Re-implement the redirection monitoring data to make it easier to process. The record format and codes have completely changed since the last issue of this document.

------------- Release 3.1.1

24 April 2012
- Document third party copy statistics in the ofs summary record.
- Correct types in the redirect monitoring section.
Monitoring

---------------  Release 3.2.0
---------------  Release 3.2.1
---------------  Release 3.2.2
---------------  Release 3.2.3
---------------  Release 3.2.4

22 September 2012
- Document the “f” stream (fstat option).
- Document the xrootd.ops.rs and xrootd.ops.rv counters in the xrootd summary data.
- General re-ordering of the manual to improve comprehension.

---------------  Release 3.2.5

22 October 2012
- Document the site name information is the summary record as well as in the server’s identification record.

26 October 2012
- Document the XROOTD.MON_READU detailed entry.

---------------  Release 3.2.6
---------------  Release 3.2.7

15 December 2012
- Document the revised format of the f-stream (fstat option).

15 January 2013
- Document the isDisc f-stream (fstat option) record type.

17 June 2014
- Document the &x and &y cgi tags in the user identification detailed map record.

21 April 2016
- Document the sID member in XrdXrootdMonFileTOD structure.
- Add admonition to always use the XrdXrootdMonFileHdr::recSize to skip to the next record as structure lengths may change.
Monitoring

---------- Release 4.0.0

7 July 2014
- Correct mistakes in the user identification record.
- Document the fact that the user identification also includes the communication protocol (started in R4).

17 September 2018
- Document the “I” tag in the loginfo portion of the “u” mapping record.

---------- Release 5.0.0

1 May 2019
- Document the “g” stream.

2 December 2019
- Document the “cache” and “pss” summary statistics.