System Monitoring Reference

20-August-2020
Release 5.1.0+

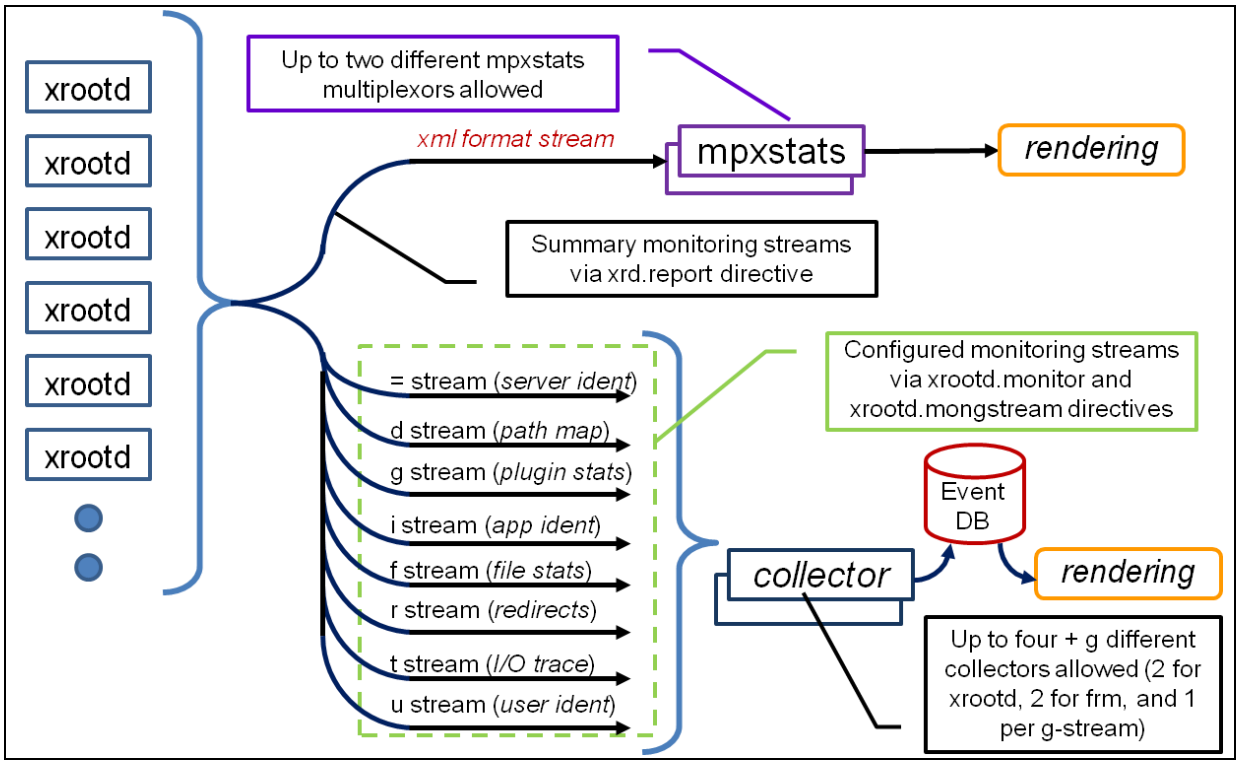
Andrew Hanushevsky

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** and **xrootd.mongstream** directives. All of these directives 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.

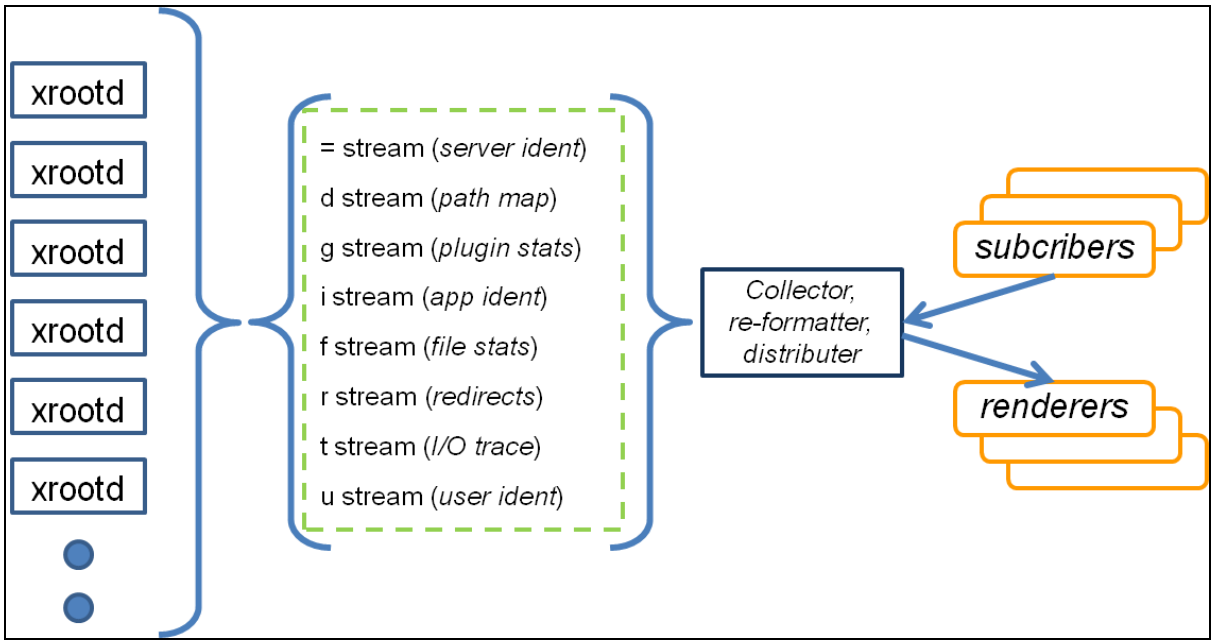
Below is a graphic showing all of the available monitoring streams and their intended destinations.



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.

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 **XRootD** monitoring architecture is highly suited for publish-subscribe environments; as shown below and typified by Apache Kafka or Spark.



Because **XRootD** monitoring data uses a common compact format it is easy to collect and cross-reference. A collector would reconstruct the streams to contain relevant data for each type of subscriber in the desired format (e.g. JSON). A collector could also push preset data streams to known renderers like dash boards.

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

```
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.

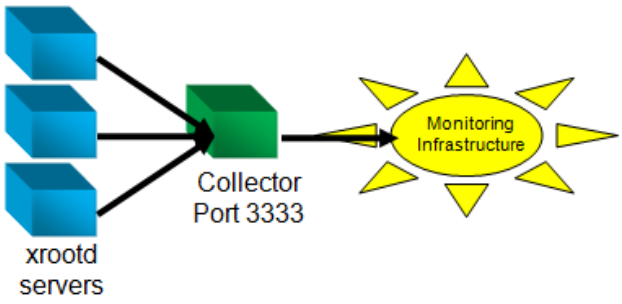
Defaults

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: "*var*=*value*[**&***var*=*value*[. . .]]**\n**". Each *var* is based on an **xml** format item and the *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 getenv().
4) The **flat** format generally produces: "*var value***\n**[*var value***\n**[. . .]]**\n**". Each *var* is based on an **xml** format item and the *value* is the item's associated value. Each *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 *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 statistic.
2. Start the data multiplexing program and feed its output to 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 a hash, and then calls a subroutine that can use the values in the hash to feed Ganglia.

```
#!/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

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

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

2.2.1 Buff Summary Data

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

Variable	Type	S	Explanation Of Value
buff.adj	int	↑	Adjustments to the buffer profile.
buff.buffs	int	↔	Number of allocated buffers.
buff.mem	int64	↔	Bytes allocated to buffers.
buff.reqs	int	↑	Requests for a buffer.

* This information is provided by the Operating System's recvfrom() function, not the data stream. It is present only when the **–s mpxstats** option has been specified.

2.2.2 Cache Summary Data

```
<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<cnt>int64</cnt></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>
```

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

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.

Role ID	Corresponding role directive	Role ID	Corresponding role directive
E	peer	PR	proxy supervisor
EM	peer manager	PS	proxy server
M	manager	R	supervisor
MM	meta manager	S	server
PM	proxy manager		

2.2.3.1 Cmsc Protocol Summary Data

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

Variable	Type	S	Explanation Of Value
cmsc.role	char		Role identification for reporter (see table above).

2.2.3.2 Cmsm Protocol Summary Data

```
<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>int<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>
```

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

[†] This tag is only present for MM roles (meta manager) and if requested via the **cms.repstats** directive.

Variable	Type	S	Explanation Of Value
cmsm.frq.add	int64	↑	Additions to the fast response queue (frq)[‡].
cmsm.frq.add.d	int64	↑	Additions that were duplicates.
cmsm.frq.rsp	int64	↑	Responses received.
cmsm.frq.rsp.m	int64	↑	Multiple responses were fielded.
cmsm.frq.lf	int64	↑	Lookups dispatched that required no wait.
cmsm.frq.ls	int64	↑	Lookups dispatched that required a full wait.
cmsm.frq.rf	int64	↑	Redirects dispatched that required no wait.
cmsm.frq.rs	int64	↑	Redirects dispatched that required a full wait.

2.2.3.3 Cmss Protocol Summary Data

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

Variable	Type	S	Explanation Of Value
cmss.role	char		Role identification for reporter (see table above).

[‡] This tag is only present if requested via the **cms.repstats** directive.

2.2.4 Info Summary Data

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

Variable	Type	S	Explanation Of Value[§]
info.host	char		Hostname that generated the information.
info.name	char		Instance name specified via **–n** option (anon if none).
info.port	int	↔	Port used for server requests.

2.2.5 Link Summary Data

```
<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>
```

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

[§] 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

```
<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>
```

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

2.2.7 Oss Summary Data

```
<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>
```

Variable	Type	S	Explanation Of Value
oss.paths	int	↔	Number of subsequent paths stats (0 <= *i* < n).
oss.paths.*i*.free	int64	↔	Kilobytes available.
oss.paths.*i*.ifr	int64	↔	Number of free inodes.
oss.paths.*i*.ino	int64	↔	Number of inodes.
oss.paths.*i*.lp	char		The minimally reduced logical file system path.
oss.paths.*i*.rp	char		The minimally reduced real file system path.
oss.paths.*i*.tot	int64	↔	Kilobytes allocated.
oss.space	int		Number of subsequent space stats (0 <= *i* < n).
oss.space.*i*.free	int64	↔	Kilobytes available.
oss.space.*i*.fsn	int	↔	Number of file system extents.
oss.space.*i*.maxf	int64	↔	Max kilobytes available in a filesystem extent.
oss.space.*i*.name	char		Name for the space.
oss.space.*i*.qta	int64	↔	Total space quota[**], if supported.
oss.space.*i*.tot	int64	↔	Kilobytes allocated.
oss.space.*i*.usg	int64	↔	Usage associated with space name, if supported.

[**] This tag may be missing if quotas have not been configured.

2.2.8 Poll Summary Data

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

Variable	Type	S	Explanation Of Value
poll.att	int	↔	File descriptors attached for polling.
poll.en	int	↑	Poll enable operations.
poll.ev	int	↑	Polling events.
poll.int	int	↑	Unsolicited polling events.

2.2.9 Proc Summary Data

```
<stats id="proc">
     <usr><s>int</s><u>int</u></usr>
     <sys><s>int</s><u>int</u></sys>
</stats>
```

Variable	Type	S	Explanation Of Values Reported by getrusage()
proc.sys.s	int	↑	Seconds of system-time.
proc.sys.u	int	↔	Microseconds of system-time.
proc.usr.s	int	↑	Seconds of user-time.
proc.usr.u	int	↔	Microseconds of user-time.

2.2.10 Pss Summary Data

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

Variable	Type	S	Explanation Of Value
pss.open	int	↑	Number of remotes file opens.
pss.open.errs	int	↑	Number of opens that failed.
pss.close	int	↑	Number of remote file closes.
pss.close.errs	int	↑	Number of closes that failed.

2.2.11 Sched Summary Data

```
<stats 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>
</stats>
```

Variable	Type	S	Explanation Of Value
sched.idle	int	↔	Number of scheduler threads waiting for work.
sched.inq	int	↔	Number of jobs that are currently in the run-queue.[††]
sched.jobs	int	↑	Jobs requiring a thread.
sched.maxinq	int	↑	Longest run-queue length
sched.tcr	int	↑	Thread creations.
sched.tde	int	↑	Thread destructions.
sched.threads	int	↑	Number of current scheduler threads.
sched.tlimr	int	↑	Number of times the thread limit was reached.

2.2.12 Sgen Summary Data

```
<stats id="sgen"><as>0</as><et>0</et><toe>toe</toe></stats>
```

Variable	Type	S	Explanation Of Value
sgen.as	int		One if data was asynchronously gathered, 0 otherwise.
sgen.et	int64	↔	Elapsed milliseconds from start to completion of statistics.
sgen.toe	int64	↑	Unix time when statistics gathering ended.

[††] The number of active requests is represented by (sched.threads – sched.idle + sched.inq).

2.2.13 Xrootd Protocol Summary Data

```
<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>
```

Variable	Type	S	Explanation Of Value
xrootd.num	int	↑	Number of times the protocol was selected.
xrootd.aio.max	int	↑	Maximum simultaneous async I/O requests.
xrootd.aio.num	int64	↑	Async I/O requests processed.
xrootd.aio.rej	int64	↑	Async I/O requests converted to sync I/O.
xrootd.dly	int	↑	Number of requests that ended with a delay.
xrootd.err	int	↑	Number of requests that ended with an error.
xrootd.ops.getf	int	↑	Getfile requests.
xrootd.ops.misc	int	↑	Number of “other” requests.
xrootd.ops.open	int	↑	File open requests.
xrootd.ops.pr	int64	↑	Pre-read requests.
xrootd.ops.putf	int	↑	Putfile requests.
xrootd.ops.rf	int	↑	Cache refresh requests.
xrootd.ops.rd	int64	↑	Read requests.
xrootd.ops.rs	int64	↑	Readv segments.
xrootd.ops.rv	int64	↑	Readv requests.
xrootd.ops.sync	int	↑	Sync requests.
xrootd.ops.wr	int64	↑	Write requests.

Variable	Type	S	Explanation Of Value
xrootd.rdr	int64	↑	Number of requests that were redirected.
xrootd.lgn.num	int	↑	Number of login attempts.
xrootd.lgn.af	int	↑	Number of authentication failures.
xrootd.lgn.au	int	↑	Number of successful authenticated logins.
xrootd.lgn.ua	int	↑	Number of successful un-authentication logins.

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 **F**ile **R**esidency **M**anager (**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** and **xrootd.mongstream** directives, as follows:

- **f**-stream summarizes file access events; enabled by the **fstat** event option.
- **g-**stream summarizes various plug-in events; enabled by the **ccm**, **pfc**, and **tcpmon** event options.
- **r**-stream details client redirections; enabled by the **redir** event option.
- **t**-stream details file access events; enabled by the **files**, **io**, and **iov** event 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 **F**ile **R**esidency **M**anager (**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**-, **g**-, **r**-, and **t**-streams 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.

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-s**tream into useful information. The **fstat lfn** option provides a similar feature but includes the information directly in the **f-s**tream. It is rare to enable the "**t**" and **f-s**treams together.

iv. Each type of **g-s**tream can also generate "**d**" and "**i**" mapping records. The mapping records are sent to the receiving host assigned to the particular **g-s**tream.

v. 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.

vi. 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.

vii. Monitor records are formatted as structured binary records. All numeric fields within the record are sent in network byte order. However, it is possible to specify alternate non-binary formats for **g-s**treams as these streams are generated by plug-ins which may or may not be part of the **XRootD** core. See the **xrootd.mongstream** directive for additional information as well as **g-s**treams details in this document.

viii. 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-s**tream datagrams are always bracketed by window timing marks.

ix. The **r-s**tream (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.

x. Definitions of the structures and symbols described in the following sections can be found in the "**XrdXrootdMonData.hh**" file.

3.2 Common Packet Header

The following figure describes the common header in each UDP packet sent by **xrootd** or the **FRM** daemon.

```
struct XrdXrootdMonHeader
       {kXR_char   code;    // = | d | f | g | i | p | r | t | u | x
        kXR_char   pseq;    // packet sequence
        kXR_unt16  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.

3.2.1 Alternative Packet Header (g-Stream)

As mentioned earlier, **g-s**treams can request that the packet header be sent as a **CGI** query string or a **JSON** object. Both are text-only formats. Minimally, the packet always starts with the information shown below:

```
CGI:  dflthdr[srchdr]…

dflthdr:  code=code&pseq=pseq&stod=stod&sid=sid

srchdr:   sitehdr | hosthdr | insthdr | fullhdr

sitehdr:  &src.site=sname
hosthdr:  sitehdr&src.host=hname
insthdr:  hosthdr&src.port=port&src.inst=iname
fullhdr:  insthdr&src.pgm=pname&src.ver=ver
```

```
JSON: {dflthdr[,src{srchdr}]…}

dflthdr:  "code":"code","pseq":pseq,"stod":stod,"sid":sid

srchdr:   sitehdr | hosthdr | insthdr | fullhdr

sitehdr:  "site":"sname"
hosthdr:  sitehdr,"host":"hname"
insthdr:  hosthdr,"port":port, "inst":"iname"
fullhdr:  insthdr,"pgm":"pname","ver":"ver"
```

Where:

dflthdr is the default header. It contains

code Identifies the packet and is one of:
- **=** – server identification sent by **xrootd**
- **d** – **dictid** for a path (**xrootd** only)
- **g** – general events such as file cache information (**xrootd** only)
- **i** – **dictid** for information (**xrootd** only)

pseq packet sequence number that ranges from 0 to 999.

stod server's start time in Unix seconds.

sid server's fingerprint.

srchdr lists the attributes of the server producing the message. The list of included attributes is configurable and options correspond to the tag names. The *srchdr* is optional in most cases. When it exists is contains one or more of the following:

sname site name (**sitehdr, hosthdr, insthdr, or fullhdr** option).
hname host name or IP address (**hosthdr, insthdr, or fullhdr** option).
port port number (**insthdr or fullhdr** option).
iname instance name (**insthdr or fullhdr** option).
pname program name (**fullhdr** option).
ver version string (**fullhdr** option).

Notes

1) Additional data may be contained in the packet depending on its code, as indicated by the triple dot, and is described in each relevant "code" section.
2) The server's fingerprint is a **SHA3-512** digest of the server's site name, host name, port number, instance name, and program name. The digest is convoluted with a **CRC32C** checksum of the same information to produce a practically unique 48-bit number. Hence, *sid* is for **S**erver **ID** and can be used as a shorthand to cross reference monitoring records with a particular server. The details of the server are specified in the "**=**" map record and some or all of the information may also be contained in other **g-s**tream records, depending on the configuration.

3.3 Monitor Map Message Format

```
struct XrdXrootdMonMap
       {XrdXrootdMonHeader hdr;
        kXR_unt32          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. Since **UDP** packets may arrive out of order it is possible to receive a map record with a **dictid** that was used in a previous packet. To avoid this problem buffer a small number of packets and order them by packet sequence number before processing. Alternatively, hold the packet that has an undefined **dictid**, with a suitable timeout, until the matching map record arrives.

3.3.1 Message Info Field

```
Code Contents of info          Code Contents of info
=    userid\nsrvinfo           p    userid\nprginfo
d    userid\npath              u    userid[\nauthinfo]
i    userid\nappinfo           x    userid\nxfrinfo

userid:   prot/user.pid:sid@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:  &site=sname&port=pnum&inst=iname&pgm=prog&ver=vname

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

Where:

Token	**Explanation**
ap	Authentication protocol name used to authenticate the client.
appinfo	Un-interpreted application or plug-in supplied information.
at	File's access time in Unix seconds.
bytes	Size of the migrated, purged, or staged file in bytes.
ct	File's creation time in Unix seconds.
data	Optional program monitoring data returned by the transfer command.
dn	Client's distinguished name as reported by *ap*. If no name is present, the variable data is null.
gn	Client's group names in a space-separated list. If no groups are present, the tag variable data is null.
hn	Client's host's name as reported by *ap*. If no host name is present, the variable data is null.
host	Host name, or IP address, where the user's request originated.
iname	Server's instance name as specified with he **–n** command line option. If no instance name was specified, "**anon**" is reported as the instance name.
lfn	Logical name of the transferred file.
loginfo	Arbitrary monitoring information specific to the protocol being used at login time. If no information is present, the tag variable data is null.
minfo	Contents of the **XRD_MONINFO** client-side environmental variable.

Token	**Explanation**
mt	File's modification time in Unix seconds.
on	Client's organization name as reported by *ap*. If no organization is present, the tag variable data is null.
op	The character operation code for a file transfer event, as follows: **0** - Unknown operation, this usually indicates a logic error. **1** - File was copied into the server by client request. **2** - File was copied out of the server by migration system request. **3** - Same as **2** but the file was removed after migration. **4** - File was copied out of the server by client request. **5** - Same as **4** but the file was removed after the copy completed. **6** – File was copied into the server by staging system request.
path	Full path name of the file being opened.
pid	User's process number that issued the request.
pnum	Server's main port number.
prog	Name of the server's executable program.
prot	Communication protocol being used by the client (e.g., xroot, http, etc).
rc	The return code. If the request was successful, the it is zero. Otherwise, the request failed. For failing stage requests, the *bytes* is also zero.
rn	Client's role name as reported by *prot*. If no role name is present, the variable data is null.
sec	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).
sid	Server's fingerprint for the connection to *user***:***pid* at *host*.
sname	Server's designated site name.
tod	The Unix seconds, as returned by time(), when the record was produced.
user	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**).
vname	Server's version identification string.
x	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.
xeqname	Name of the executable program the client is running with the path removed.
xfn	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).
4|**6**	Client's network mode: **4** for IPv4 and **6** for IPv6.

Notes

1) The sid The **d** , **i** and **u** messages with *authinfo* contain two ASCII text strings, separated by a new-line (**\n**) character.
2) The **p** and **x** messages contain three ASCII text strings, separated by a new-line (**\n**) character.
3) The "**fn**" tag in the **p** message 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** messages 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.

3.3.2 Alternative Monitor Map Messages (g-Stream)

g-streams can be configured to send packet headers as a **CGI** query string or a **JSON** object. Both are text-only formats. This only affects the **=**, **d**, and **i** map records as these are the only ones that the **g-s**tream can create. Each message starts with the alternative packet default header described earlier (i.e. *dflthdr*, *fullhdr*, and *srchdr*). Additional tokens are added to the header, as shown below.

CGI: ***dflthdr****fullhdr* ***dflthdr:*** **code==**&**pseq=***pseq***&stod=***stod***&sid=***sid***
JSON: {*dflthdr*,src{*fullhdr*}} ***dflthdr*:** **"code":"=","pseq":***pseq***,"stod":***stod***,"sid":***sid***
ident ("=") message

CGI: ***dflthdr***[*srchdr*]**&gs.type=***type***&did=***did***&data=***data* ***dflthdr:*** **code=***code*&**pseq=***pseq***&stod=***stod***&sid=***sid***
JSON: {*dflthdr*[,src{*srchdr*}],*mapinfo*} ***dflthdr*:** **"code":"***code***","pseq":***pseq***,"stod":***stod***,"sid":***sid* *mapinfo:* **"gs":{"type":***type***},"did":***did***,"data":"***data***"**
"d" or "i" messages

Where:

Token	**Explanation**
code	Identifies the packet and is one of: **d** – **dictid** for a path (**xrootd** only) **i** – **dictid** for information (**xrootd** only)
type	The **g**-stream generating the message and is one of: **C** – generated by the **pfc** plug-in (i.e. cache) **M** – generated by the **ccm** plug-in (i.e. cache context management) **T** – generated by the **tcpmon** plug-in
dictid	Numeric dictionary identifier which is unique within a server instance.
data	Data associated with *dictid*.

3.4 The f-stream (fstat)

The **f-s**tream 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 X**rdXrootdMonFileTOD** 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::**").

XrdXrootdMonFileHdr::recType	**Structure Encompassing Header**
isClose	XrdXrootdMonFileCLS
isDisc	XrdXrootdMonFileDSC
isOpen	XrdXrootdMonFileOPN
isTime	XrdXrootdMonFileTOD
isXFR	XrdXrootdMonFileXFR

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:

```
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.

The server's identifier appears in each **XrdXrootdMonFileTOD** entry in the **sID** member. 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 expression

```
ntohll(sID) & XROOTD_MON_SIDMASK
```

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

The following table lists all possible values in the header **recFlag** member for the TOD 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::**".

XrdXrootdMonFileHdr::recFlag	**Meaning**
hasSID	The sID member is present

If you are using the **sID** member, you should test if the **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.

```
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.

```
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-s**tream. 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 message 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::**".

XrdXrootdMonFileHdr::recFlag	**Meaning**
hasLFN	XrdXroodMonFileLFN present
hasRW	File opened for reads & writes

3.4.3 Close Event

The close structure describing a close event is shown below.

```
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-s**tream. 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::**").

XrdXrootdMonFileHdr::recFval	**Meaning**
forced	Disconnect prior to close
hasOPS	XrdXroodMonFileOPS present
hasSSQ	XrdXroodMonFileSSQ present

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.

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

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-s**tream 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.

```
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, pfc, and tcpmon)

The **g-s**tream is enabled using the **ccm**, **pfc,** or **tcpmon** option on the **xrootd.monitor** or **xrootd.mongstream** directive. It contains multiple types of variable length **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 (binary, **CGI**, or **JSON**).
- 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 **ASCII** newline separated text fields detailing specific events. The last field always ends with a null byte.

The following diagram shows the packet structure.

```
struct XrdXrootdMonGS;
      {struct XrdXrootdMonMonHeader hdr;
       int       tBeg; // UNIX time of first entry
       int       tEnd; // UNIX time of last  entry
       kXR_int64 sID;  // Provider identification
      }
*
* 1 or more newline separated ASCII text strings with the last
* such string ending with a null byte.
*
```

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 **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 fingerprint. The table below lists possible providers:

Provider (i.e. plug-in)	**Contents of 1st 8 bits**
Cache Context Manager (ccm)	XROOTD_MON_GSCCM
Proxy File Cache (pfc)	XROOTD_MON_GSPFC
TCP connection monitor	XROOTD_MON_GSTCP

You can extract the provider's identifier with the following expression

```
(ntohll(sID) >> XROOTD_MON_PIDSHFT) & XROOTD_MON_PIDMASK
```

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

```
ntohll(sID) & XROOTD_MON_SIDMASK
```

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

While the **g-s**tream can be fed by many different information providers no **UDP** packet will ever 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:

Provider ID	**Manual**
XROOTD_MON_GSCCM	*This plug-in is not a core component.*
XROOTD_MON_GSPFC	Proxy Storage Services Reference
XROOTD_MON_GSTCP	*This plug-in is not a core component.*

3.5.1 Alternative g-Stream Headers

g-streams can be configured to send packet headers as a **CGI** query string or a **JSON** object. Both are text-only formats. When configured, the stream's payload is prefixed by the following header (*dflthdr* and *srchdr* are described in alternative default header section). The g-stream can also generate text-only versions of the ident and map messages. Be sure to review those.

CGI: ***dflthdr***`[`*srchdr*`]`**&gs.type=***type***&gs.tbeg=***tbeg***&gs.tend=***tend***\n** ***dflthdr:*** **code=g&pseq=***pseq***&stod=***stod***&sid=***sid*
JSON: {*dflthdr*`[`,"src":{*srchdr*}`]`,*gsinfo*}\n ***dflthdr*:** **"code":"g","pseq":***pseq***,"stod":***stod***,"sid":***sid* *gsinfo:* **"gs":{"type":***type***,"tbeg":***tbeg***,"tend":***tend***}**
Alternative Header for g-stream Payload

Where:

Token	**Explanation**
type	The **g**-stream generating the message and is one of: **C** – generated by the **pfc** plug-in (i.e. cache) **M** – generated by the **ccm** plug-in (i.e. cache context management) **T** – generated by the **tcpmon** plug-in
tbeg	UNIX time of first payload entry.
tend	UNIX time of last payload entry.

The packet payload (i.e. data following the header's newline character) consists of one or more newline separated ASCII text strings with the last such string ending with a null byte.

3.6 The r-stream (redir)

```
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:

Definition	Value	Meaning
XROOTD_MON_REDTIME	0x00	Window timing mark[§§]
XROOTD_MON_REDIRECT	0x8x	Redirect event generated by **cmsd**
XROOTD_MON_REDLOCAL	0x9x	Redirect event generated by **xrootd**
XROOTD_MON_REDSID	0xf0	Server identification

XROOD_MON_REDIRECT and **XROOTD_MON_REDLOCAL** 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.

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.

Definition	Value	Meaning
XROOTD_MON_CHMOD	0x01	Change file mode.
XROOTD_MON_LOCATE	0x02	Locate file or directory.
XROOTD_MON_OPENDIR	0x03	Open director for reading.
XROOTD_MON_OPENC	0x04	Open file for creation.
XROOTD_MON_OPENR	0x05	Open file for reading.
XROOTD_MON_OPENW	0x06	Open file for writing.
XROOTD_MON_MKDIR	0x07	Create a directory or path.
XROOTD_MON_MV	0x08	Rename a file or directory.
XROOTD_MON_PREP	0x09	Prepare request.
XROOTD_MON_QUERY	0x0a	Query information request.
XROOTD_MON_RM	0x0b	Remove a file.
XROOTD_MON_RMDIR	0x0c	Remove a directory.
XROOTD_MON_STAT	0x0d	Stat a file or directory.
XROOTD_MON_TRUNC	0x0e	Truncate a file.

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

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.

Field	Contents For Window Entries
`info[].arg0.Type`	XROOTD_MON_REDTIME
`info[].arg0.Window`	Window size in the low order 24 bits.
`info[].arg1.Window`	Unix time of the new window.

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 he 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.

Field	Contents For Server Identification Entries
`info[].arg0.Type`	XROOTD_MON_REDSID
`info[]`	Server's identification in the low order 48 bits.

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

```
ntohll(sID & 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-s**tream 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 he information in the **f-s**tream. This is because all values in the **f-s**tream can be derived from the **t-s**tream. However, the **t-s**tream 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-s**tream 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-s**tream. 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.

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 datagram 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

```
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:

Definition	Value	Meaning
XROOTD_MON_OPEN	0x80	File has been opened
XROOTD_MON_READV	0x90	Details for a kXR_readv request
XROOTD_MON_READU	0x91	Unpacked details for kXR_readv
XROOTD_MON_APPID	0xa0	Application provided marker
XROOTD_MON_CLOSE	0xc0	File has been closed
XROOTD_MON_DISC	0xd0	Client has disconnected
XROOTD_MON_WINDOW	0xe0	Window timing mark
----	<=0x7f***	Read or write request

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

Definition	Value	Meaning
XROOTD_MON_FORCED	0x01	Entry due to forced disconnect.
XROOTD_MON_BOUNDP	0x02	Entry for a bound path.

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:

Field	Contents For Appid Request Entries
info[].arg0.id[0]	XROOTD_MON_APPID
info[].arg0.id[1…3]	Reserved.
info[].arg0.id[4…15]	Up to 12 characters of application identification.

***Indicates that if the high order bit is zero, then this is a read/write request.

Field	Contents For Close Request Entries
info[].arg0.id[0]	XROOTD_MON_CLOSE
info[].arg0.id[1]	Number of bits Info[].arg0.rTot[1] has been right shifted to fit into a 32-bit unsigned int.
info[].arg0.id[2]	Number of bits Info[].arg1.wTot has been right shifted to fit into a 32-bit unsigned int.
info[].arg0.id[3]	Reserved.
info[].arg0.rTot[1]	Scaled number of bytes read from the file.
info[].arg1.wTot	Scaled number of bytes written to the file.
info[].arg2.dictid	The file path's dictionary ID ('**d**' map message).

Field	Contents For Disconnect Request Entries
info[].arg0.id[0]	XROOTD_MON_DISC
info[].arg0.id[1]	May contain XROOTD_MON_BOUNDP and XROOTD_MON_FORCED
info[].arg0.id[2…7]	Reserved.
info[].arg1.buflen	Number of seconds that client was connected.
info[].arg2.dictid	The client's dictionary ID ('**u**' map message).

Field	Contents For Open Request Entries
info[].arg0.id[0]	XROOTD_MON_OPEN
info[].arg0.id[1…7]	Size of the file in bytes.
info[].arg1	Reserved.
info[].arg2.dictid	The file path's dictionary ID ('**d**' map message).

Field	Contents For Read/Write Request Entry
info[].arg0.val	Read or write offset (see below).
info[].arg1.bufflen	Length of the read when non-negative. When negative, this is the length of a write request.
info[].arg2.dictid	The file path's dictionary ID ('**d**' map message).

Field	Contents For Readv Request Entry
info[].arg0.id[0]	XROOTD_MON_READU
info[].arg0.id[1]	**readv** request identifier
info[].arg0.sVal[1]	Number of elements in the **readv** vector
Info[].arg0.rTot[1]	Reserved.
info[].arg1.bufflen	Length of the read.
info[].arg2.dictid	The file path's dictionary ID ('**d**' map message).

Field	Contents For Readv Request Entry
`info[].arg0.id[0]`	XROOTD_MON_READV |
`info[].arg0.id[1]`	**readv** request identifier
`info[].arg0.sVal[1]`	Number of elements in the **readv** vector
`Info[].arg0.rTot[1]`	Reserved.
`info[].arg1.bufflen`	Length of the read.
`info[].arg2.dictid`	The file path's dictionary ID ('**d**' map message).

Field	Contents For Window Entry
`info[].arg0.id[0]`	XROOTD_MON_WINDOW
`info[].arg0.id[1]`	Reserved.
`info[].arg0.val`	Server identifier in the low order 48 bits.
`info[].arg1.Window`	Unix time of when the previous window ended.
`info[].arg2.Window`	Unix time of when this window has started.

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.

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

```
ntohll(info[].arg0.val & XROOTD_MON_SIDMASK)
```

4 Document Change History

14 July 2009

- This manual was introduced.

16 July 2009

- Added example on **mpxstats**.

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.

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.copycmd** 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_REDHOST** 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_REDHOST** 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.

---------- 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-s**tream (**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.

---------- 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-s**tream.

2 December 2019

- Document the "**cache**" and "**pss**" summary statistics.

20 August 2020

- Document the **g**-stream and optional **CGI** and **JSON** headers.