Proxy Storage Services (Caching, Non-Caching, & Server-less Caching) Configuration Reference

7 June 2022
Release 5.5.0 and above

Andrew Hanushevsky (SLAC)
Alja Mrak-Tadel (UCSD)

©2003-2022 by the Board of Trustees of the Leland Stanford, Jr., University
All Rights Reserved
Produced under contract DE-AC02-76-SFO0515 with the Department of Energy
This code is open-sourced under a GNU Lesser General Public license.
For LGPL terms and conditions see http://www.gnu.org/licenses/

1 Introduction 5
1.1 Direct Mode Proxies 7
1.2 Forwarding Mode Proxies 7
1.2.1 Forwarding File Paths 8
1.2.2 Forwarding Object IDs 9
1.3 Combination Mode Proxies 10
1.4 Minimal Sample Configuration Files 11
2 Security Considerations 13
2.1 Privacy and Integrity 14
3 Automating Proxy Service Selection 15
4 Standard Proxy Service 17
4.1 Simple Proxy Server 18
4.2 Proxy Cluster 21
4.3 Configuration 25
4.3.1 Required Directives 25
4.3.1.1 Specifying the origin as a URL 26
4.3.2 Data Caching Directives 27
4.3.2.1 Direct Cache Access 27
4.3.3 Debugging Directives 28
4.3.4 Path Processing Directives 29
4.3.4.1 Reproxying Third Party Copy 30
4.3.5 Networking Directives 31
4.3.6 Security Directives 33
4.3.6.1 Persona restrictions and limitations 34
4.3.7 Tuning Directives 35
5 Disk Caching Proxy 38
5.1 Configuration 40
5.1.1 Disk Caching S3-type Objects 47
5.1.2 Cache Integrity Options 49
5.2 Disk Caching Proxy Clusters 51
5.3 Handling Cache Overloads 53
6 Memory Caching Proxy 55
6.1 Configuration 55
7 POSIX Server-less Caching 59
7.1 Server-less Disk Caching 60
7.2 Server-less Memory Caching 61
7.3 Esoteric Directives 61

7.4 Defining Virtual Mount Points 62
7.4.1 Examples 63
8 The netchk Utility 65
9 Document Change History 67

1 Introduction

This document describes **XRootD** configuration directives for the **P**roxy **S**torage **S**ervice (**pss**) components.

Configuration directives for each component come from a configuration file. The **XRootD** structure requires that all components read their directives from the same configuration file. This is the configuration file specified when **xrootd** was started (see the **xrootd -c** command line option). This is possible because each component is identified by a unique 3-letter prefix. This allows a common configuration file to be used for the whole system.

xrd (protocol driver)
xroot and http (protocol)
ofs (file system plug-in)
pss (proxy storage plug-in)

The particular components that need to be configured are the file system plug-in (**ofs**) and the proxy storage plug-in (**pss**). The relationship between **xrootd** and the plug-ins is shown on the left. The protocol driver (**xrd**) runs the **xroot** protocol which, in turn, utilizes the file system plug-in that, itself, relies on the proxy system plug-in. This is collectively called **xrootd**, the executable that encapsulates all of the components.

The prefixes documented in this manual are listed in the following table. The **all** prefix is used in instances where a directive applies to more than one component. *Records that do not start with a recognized identifier are ignored*. This includes blank record and comment lines (i.e., lines starting with a pound sign, #).

Prefix	Component
`ofs`	**O**pen **F**ile **S**ystem coordinating **acc**, **cms**, **oss** & **pss** components
`pfc`	Proxy File Cache (i.e. specialized proxy plug-in)
`pss`	**P**roxy **S**ystem **S**ervice (i.e., specialized **oss** plug-in)
`all`	Applies the directive to the above components.

Refer to the manual "**Configuration File Syntax**" on how to specify and use conditional directives and set variables. These features are indispensable for complex configuration files usually encountered in large installations.

When you are not running a clustered set of proxy servers, you need to specify the **ofs.osslib** directive. The plug-in is encapsulated in **libXrdPss.so** and is installed in the standard location. If you are clustering proxy servers (i.e. you specified the all.role directive indicating you are clustering proxies, **libXrdPss.so** is automatically loaded as the **ofs.osslib** plug-in.

To properly configure an **XRootD** proxy you need to review **xrd**, **xrootd**, and **ofs** directives and specify the ones that are relevant to your installation. The **xrd** and **xrootd** directives can be found in the "**Xrd/XRootd Configuration Reference**" while the **ofs** directives can be found in the "**Open File System & Open Storage System Configuration Reference**". If you are configuring a proxy cluster then you will need to specify certain **cms** directives. These can be found in the "**Cluster Management Service Configuration Reference**".

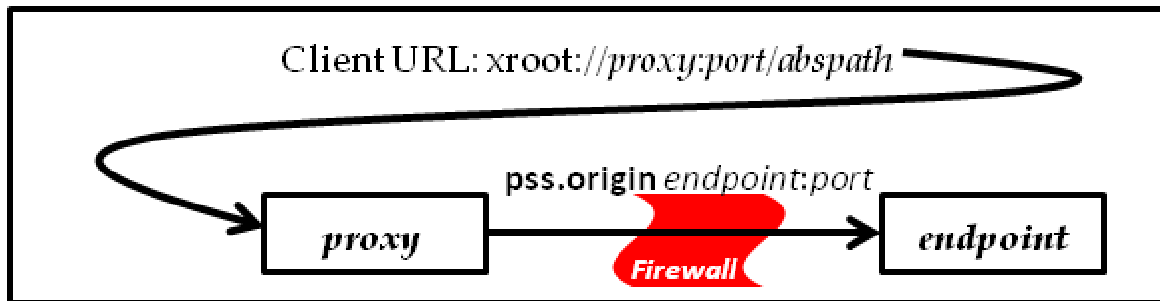
The following section describes the standard proxy service. This proxy service is primarily intended to be used for **LAN** access to bridge firewalls for remote clients. While it includes a memory caching component that can improve performance for **WAN** access in limited scenarios, it is not intended to be used as a true **WAN** proxy. In a subsequent section the disk caching proxy is described. This proxy variant is intended to improve **WAN** access as it is able to cache files or parts of files on disk at a remote location. Caching files or file segments may minimize **WAN** traffic depending on the specific workload.

Hence, the standard proxy service is intended to be used as a local proxy while the disk caching proxy is intended to be used as a remote proxy.

A proxy service, whether or not it is a disk caching proxy, can be configured in direct mode or forwarding mode. In direct mode the client is limited to initially contacting a particular endpoint. In forwarding mode, the client is free to specify the actual endpoint that is to be used for the initial contact.

1.1 Direct Mode Proxies

Direct mode proxies are the most common kind of proxies and are suitable to directly fronting an **XRootD** cluster. The **pss.origin** directive specifies the initial point of contact. The following diagram illustrates a direct mode proxy.

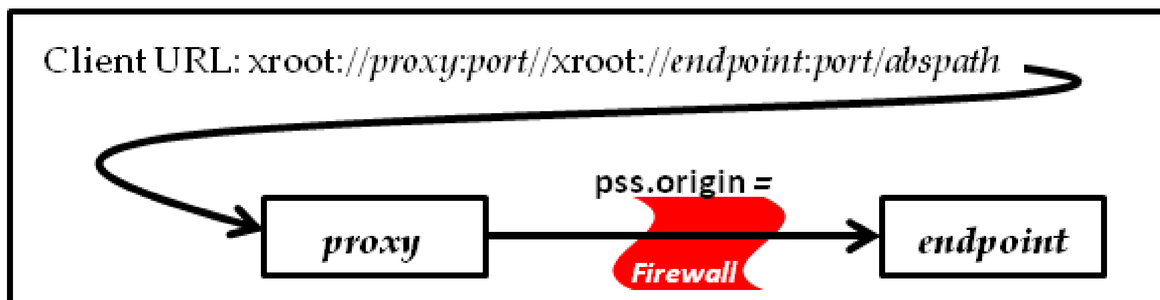


In the diagram the client present a URL that identifies the proxy server or proxy cluster. The proxy then contacts the data origin, a particular endpoint, on behalf of the client and performs the requested operation. The endpoint can be an actual server or a cluster redirector. This mode is suitable for incoming connections to a fire walled cluster or outgoing connections to a particular data source.

Direct mode proxies can access files (i.e. paths that start with a slash) or data object (i.e. identifiers that do not start with a slash) depending on the specified **all.export** directives. See the "**Xrd/XRootd Configuration Reference**" for more information.

1.2 Forwarding Mode Proxies

Forwarding mode proxies are specialized proxies that allow the client to specify the actual endpoint to be used for a request and suitable in cases where the client determines the actual endpoint that must be used. The **pss.origin** directive specifies that the endpoint will be supplied by the client, as shown below.



In the diagram the client prefixes the destination URL with the proxy location. The proxy server then contacts the specified endpoint on behalf of the client. The client is free to specify any valid endpoint. This mode is most suitable for outgoing connections and poses certain security issues that are described in a subsequent section. A valid destination URL must specify the **xroot** or **root** protocol element.

1.2.1 Forwarding File Paths

When configuring a forwarding proxy you should realize that from the proxy's perspective the URL suffix supplied by the client is initially taken as an actual path. This means that you

a) must export at least some portion of the URL suffix, and
b) can perform authorization operations on the specified URL (see the "**Security Configuration Reference**" for more information).

Exports of the form

all.export /xroot:/ *options*
all.export /root:/ *options*

satisfy the export requirement (see the **OFS** reference manual for a description of the **all.export** directive). You can specify longer export names in order to restrict the endpoints that can be contacted using the forwarding proxy. However, since multiple slashes are compressed before comparing the URL against the exported path list, you must not specify successive slashes. For example, say you wanted to restrict forwarding to only the host **foobar:1094**, then the exports would be specified as

all.export /xroot:/foobar:1094/
all.export /root:/foobar:1094/

Even though the client would actually suffix "/xroot://foobar:1094/" to the proxy's URL location. The same applies to entries placed in the authorization file. All double slashes must be removed in order for authorization rules to be matched. In both cases, the client's destination must exactly match the export specification. As a side note, error messages in the proxy's log file will show URL portions of the path without multiple slashes and should not cause any alarm.

A better alternative is to use the **pss.permit** directive to restrict the set of target destinations. This allows client to specify the target host in a variety of syntactic ways since the permit applies to the actual resolved destination IP address. In this case you only need to export the protocol segment of the destination URL.

1.2.2 Forwarding Object IDs

To exporting object IDs (i.e. object names without a leading slash) via a forwarding proxy, the configuration file must contain one of the following two export directives:

all.export *

or **all.export *?**

See the "**Xrd/XRootD Configuration Reference**" for more information on using object IDs. Since object IDs do not start with a slash; hence, cannot be confused with a file path, the specified client destination URL must appear like an object ID (i.e. it may not start with a slash). Using an absolute file path URL to forward an object ID converts the object ID to a file path and access will likely fail.

Additionally, you can neither use the authorization framework nor the **all.export** directive to restrict forwarding by destination. Instead you must use the **pss.permit** directive. This is because object IDs are arbitrary and are only recognized as such in certain restricted contexts.

The following table compares forwarding an absolute path to an Object ID to a server named "foobar:4096" via "myproxy:1094" (notice the missing double slashes for myobject before the second "xroot" and the pathname, myobject).

Access Target	**Client URL Specification**
/my/data	xroot://myproxy:1094//xroot://foobar:4096//my/data
myobject	xroot://myproxy:1094/xroot://foobar:4096/myobject

A forwarding proxy server is capable of accepting file paths as well as object IDs as long as the correct exports are specified.

1.3 Combination Mode Proxies

A proxy server can be configured in direct more as well as forwarding mode. This is known as a combination mode proxy and is done via a special form of the **pss.origin** directive. For instance, specifying

pss.orgin = *host***:***port*

Allows a client to forward a connection via a URL type of path or connect to a particular destination (i.e. *host*:*port*) if the path is not a URL. The exports must allow such combinations. For instance, the trio of

all.export /xroot:/ *options*
all.export /root:/ *options*
all.export /atlas *options*

indicates that **xroot:** and **root:** URLs should forward the request to the client-specified destination while paths starting with "/atlas" connect to the destination specified in the **pss.origin** directive.

In practice, if it is sufficient for the destination host to enforce its exports the whole specification can collapse to a single specification

all.export / *options*

as the proxy recognizes a URL if it begins with "xrootd:/" or "root:/" and treats anything else as a regular path.

1.4 Minimal Sample Configuration Files

```
# Specify that we are a direct mode proxy fronting the host
# data.stanford.edu (server or redirector)
#
pss.origin data.stanford.edu

# The export allows access to any path via proxy as the
# origin host will enforce its own exports
#
all.export /

# We need to load the proxy plugin for this to actually work
#
ofs.osslib libXrdPss.so
```

Minimal Direct Mode Proxy Configuration

```
# Specify that we are a forwarding proxy
#
pss.origin =

# The export allows xroot and root type URL's destinations
#
all.export /xroot:/
all.export /root:/

# We need to load the proxy plugin for this to actually work
#
ofs.osslib libXrdPss.so
```

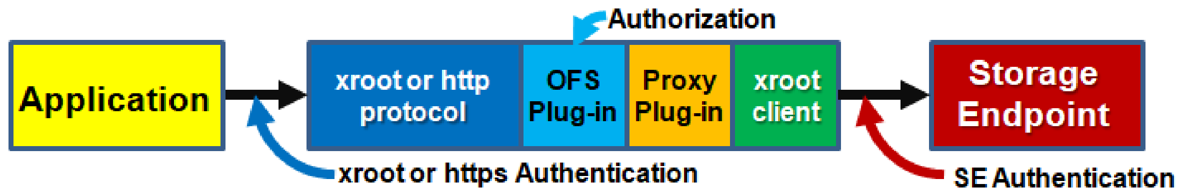
Minimal Forwarding Mode Proxy Configuration

```
# Specify that we are a forwarding proxy for URL type of paths
# and a direct mode proxy for any other type of path
#
pss.origin = data.stanford.edu

# The export allows any path to be specified. However, paths
# starting with "/xroot:/" or "/root:/" are treated as URLs
# and are forwarded. Anything else is accessed via
# data.stanford.edu
#
all.export /

# We need to load the proxy plugin for this to actually work
#
ofs.osslib libXrdPss.so
```

Minimal Combination Mode Proxy Configuration

2 Security Considerations

Security is a combination of authentication and authorization. In the **XRootD** proxy there are three aspects to consider, as shown below.



The application first authenticates with the **xrootd** running **xroot** or **http** protocol. This part needs to be configured. For **xroot** protocol, you need to specify the **xrootd.seclib** directive as well as certain **sec** prefixed directives. The **https** protocol automatically attempts to use **x509** authentication. Usually, authorization is required as well to limit what the application can do relative to a file path. This is handled by the **OFS** (Open File System) plug-in and is controlled by the **ofs.authorize** and **ofs.authlib** directives as well as any need **acc** directives. The **xrootd** directives can be found in the **Xrd/XRootD Reference**, the **ofs** directives in the **Open File System Reference**, and **sec** and **acc** directives in the **Security Reference**.

Control is handed off to proxy plug-in that eventually calls the imbedded **xroot** client to communicate to the desired storage endpoint. The storage endpoint may also require authentication. At this stage, no configuration is needed as the storage endpoint dictates how authentication happens. However, the client must be capable of providing the necessary credentials. In the **XRootD** case there are two main (and some minor) authentication schemes:

a) **x509** (either **GSI** or proxy **https**) which requires that the client supply a proxy certificate for exclusive use by the server (e.g. a robot certificate).
b) **SSS** (Simple Shared Secret) which requires the client have access to a shared private key that is used to authenticate with the endpoint.

Neither of these schemes requires any configuration. They only require that the certificate or key be placed in a well known location that is only accessible by the user id associated with the running server.

Proxy servers never forward client credentials to the endpoint that they contact. Instead, they use their own credentials to establish authenticity with the **xrootd** server. This works well for incoming connections as the proxy can validate the client before performing any actions on behalf of the client.

However, outgoing connections pose a problem; especially for forwarding proxies which are meant to be used for outgoing connections. Validating a client using internal site rules does not necessarily capture any specific access restrictions in effect outside the site. For instance, a site may authorize an incoming client but that client might not be actually authorized by the final endpoint. If the proxy itself is authorized by the final endpoint then, in effect, the client gets access to resources that would otherwise be prohibited if the client made direct contact with the outside endpoint. Consequently, great care must be exercised when providing resources using a forwarding proxy.

Communications within a site are easier to manage. For instance it is possible to configure the proxy server to pass the identity to another server using **SSS** authentication between the proxy server and the internal endpoint. That way, a simple authentication scheme can be used within a site with the full fledged complex identity of the client is passed along to the final endpoint.

2.1 Privacy and Integrity

While the **xroot** protocol can be configured to provide client-server communication integrity; neither the **xroot** nor **http** protocols provide privacy (**http** is incapable of providing integrity). If either is required then communications must use Transport Layer Security (**TLS**). For application to proxy server communications, **TLS** needs to be configured using the **xrd.tls** and **xrd.tlca** directives (see the **Xrd/XRootD Reference**). These can also be used to configure **http** to use **TLS**. **TLS** is used when the client uses **xroots** or **https** protocols in place of **xroot** or **http**, respectively. In the **xroot** case, the client may continue to use **xroot** protocol but the server is free to force the client to switch to **xroots** when needed. The same considerations apply when the proxy server communicates with the storage endpoint since the proxy is merely a client communicating on behalf of the application.

3 Automating Proxy Service Selection

One of the biggest challenges to force clients to use a proxy server without requiring users to change the URL's they normally would use. The **XRootD** framework allows you to use an **XRootD** client plug-in to accomplish this in a transparent way.

A supplied **XRootD** client plug-in, **libXrdClProxyPlugin.so**, can be used to tunnel traffic through an **XRootD** proxy server. The proxy endpoint is specified via the environment variable **XROOT_PROXY**. To enable this plug-in the environment variable **XRD_PLUGIN** needs to specify the path to the **libXrdClProxyPlugin.so** library. An example is shown below.

```
XRD_PLUGIN=/usr/lib64/libXrdClProxyPlugin.so
XROOT_PROXY=root://esvm000:2010//
xrdcp -f -d 1 root://bigdata.cern.ch//tmp/file1.dat /tmp/dump
[1.812kB/1.812kB][100%][=======================================][1.812kB/s]
```

This will first redirect the client to the **XRootD** server at esvm000:2010, presumably a forwarding proxy, will fetch the data from the **XRootD** server at bigdata.cern.ch:1094 (i.e. using the default port).

You can also specify a list of exclusion domains, for which the original URL will not be modified even if the plug-in is enabled. An example is shown below.

```
XRD_PLUGIN=/usr/lib64/libXrdClProxyPlugin.so
XROOT_PROXY=root://esvm000.cern.ch:2010//
XROOT_PROXY_EXCL_DOMAINS="some.domain, some.other.domain, cern.ch "
xrdcp -f -d 1 root://esvm000.cern.ch//tmp/file1.dat /tmp/dump
```

This will not redirect the traffic since the original URL "root://esmv000.cern.ch//" contains the "cern.ch" domain which is in the list of excluded domains. This is useful when a proxy server is only needed for out of domain access.

Proxy services can be configured to automatically be in effect via a client configuration file. The **XRootD** client looks for plug-in configurations in three places:

- Global: /etc/xrootd/client.plugins.d/
- User: ~/.xrootd/client.plugins.d/
- Application: **XRD_PLUGINCONFDIR** envar setting.

The plug-in manager will first search for global configuration files. The global settings may be overridden by user configuration files. Global and user settings may be overridden by configuration files found in a directory pointed to by the **XRD_PLUGINCONFDIR** environmental variable, if set. Overrides are based on the name of the configuration file (i.e. only one configuration of the same name is processed).

Be aware that the client plug-in manager only processes files that end with "**.conf**".

All plug-in configuration files specify key-value settings that control plug-in processing. A proxy service plug-in configuration file contents for the previous **xrdcp** example using environmental variables is shown below.

```
url = *
lib = /usr/lib64/libXrdClProxyPlugin.so
enable = true

xroot_proxy = root://esvm000.cern.ch:2010//
xroot_proxy_excl_domains = some.domain,some.other.domain,cern.ch
```

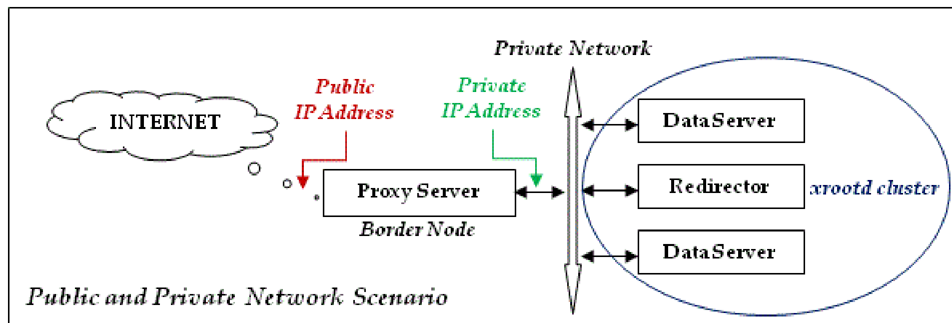
Placing the above configuration in the global directory forces all **XRootD** client applications to transparently forward all requests to esvm000.cern.ch:2010 which will then proxy the requests as needed.

The client plug-in manager silently ignores any configuration files that are missing the "**url**", "**lib**", or "**enable**" keywords. You must turn on debugging to see which configuration files are actually processed.

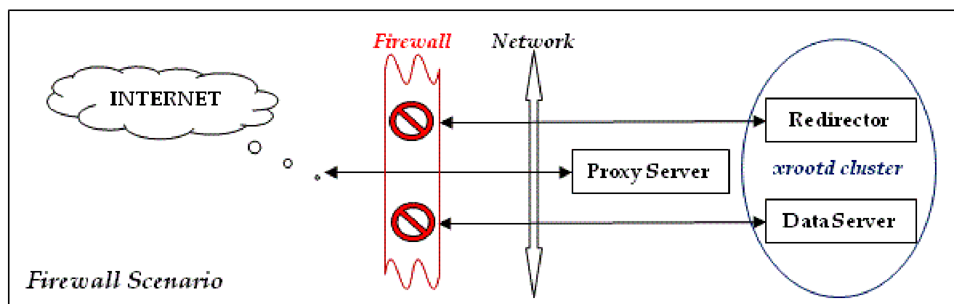
The automatic proxy selection described in this section is only available starting in release 4.7.0.

4 Standard Proxy Service

The **ofs** layer can provide a proxy service by using a special **oss** plug-in, **libXrdPss.so**. A proxy service is required if wish to provide public internet access to data served by **xrootd** servers that can only access a private network or sit behind a firewall. You do not need to setup a proxy service if all of your **xrootd** servers are accessible via the public network.



As shown in the diagram to the left, a proxy server must have access to the public as well as private or firewalled network. Typically, machines with this capability are called border nodes.



A border node is a computer that either has access to the public as well as the private network using two distinct IP addresses or one that is allowed to access the public network through a firewall using a single local IP address.

The particular scenario in effect is a local option determined by your particular network setup. The biggest challenge is to properly setup the network configuration; a task best left to networking personnel.

Two IP address scenarios require static routes to be established within the border machine so that the local **XRootD** cluster is always found using the private network while internet traffic uses the public network. Single IP address scenarios (i.e. firewalled networks) require that special firewall rules be established to allow internet access to the proxy server on the **XRootD** port while disallowing internet access to the local **XRootD** cluster.

Once routes or firewall rules have been established, you should use the **netchk** utility, found in the **XRootD utils** directory, to see if you have access from an external public node all the way to the redirector and each data server node. This verifies that network routing or firewall rules have been correctly established. In order to use **netchk**, **perl** must be installed on each machine and you must be able to do an **ssh** login to each one of them. The utility is described in the last subsection of this section.

The following sections describe how to setup a simple proxy server as well as a proxy server cluster. Additionally, various tuning options are also described. These options are independent of whether you setup a simple proxy or a proxy cluster.

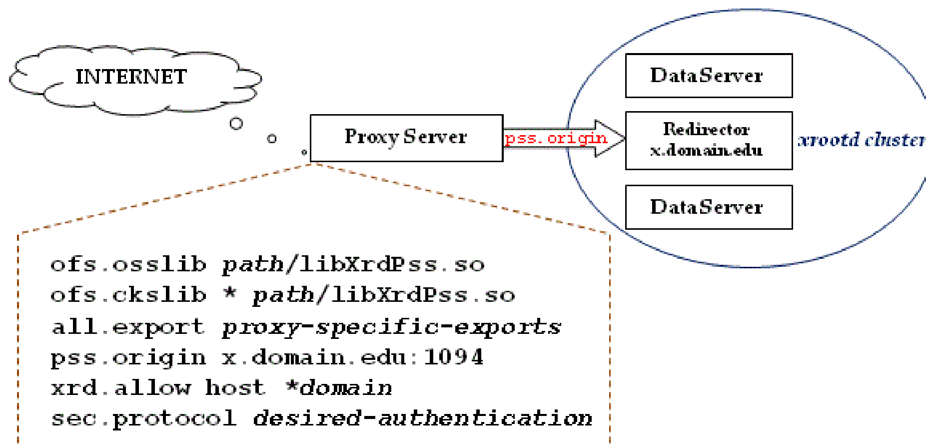
4.1 Simple Proxy Server

The simple proxy server is configured as a non-clustered **XRootD** server. This means you neither specify the **all.role** nor the **all.manager** directives documented in the "**Cluster Management Service Configuration Reference**". Other directives may be specified at your discretion.

You control which paths are publically available using the **all.export** directive that is specific to the proxy server (i.e. the proxy will only allow access to the specified paths). This also means that data servers can provide read-write access to particular paths while the proxy server can only provide read-only access to some or all such paths. The same holds true for the **stage** attribute. You can prohibit staging files via the proxy by exporting the files via the proxy with the **nostage** attribute.

The **xrd.allow** directive can be used to limit which range of hosts can use the proxy server; while the **sec.protocol** directive can be used to limit access to the proxy server to clients that can only be properly authenticated. These directives do not differ from those you would use in a standard **XRootD** cluster and documentation can be found the **XRootD** and security reference manuals.

Very few directives are needed to setup a simple proxy server. A sample configuration file matching the illustrated cluster setup is shown below. Directives shown in **red** are ***required***.



The **ofs.osslib** directive tells the **ofs** layer to use the plug-in that changes a regular **XRootD** server into a proxy server. Normally, the plug-in is named "libXrdPss.so". You must specify the location of this shared library plug-in. To support checksum calculations you must also indicate that the checksum manager is a proxy as well. This is done loading the proxy library via the **ofs.ckslib** directive.

The **all.export** directive restricts the proxy server to those paths you want to be publically accessible. Most likely, you will also designate these paths as read-only and nostage regardless of how they are locally served within the cluster.

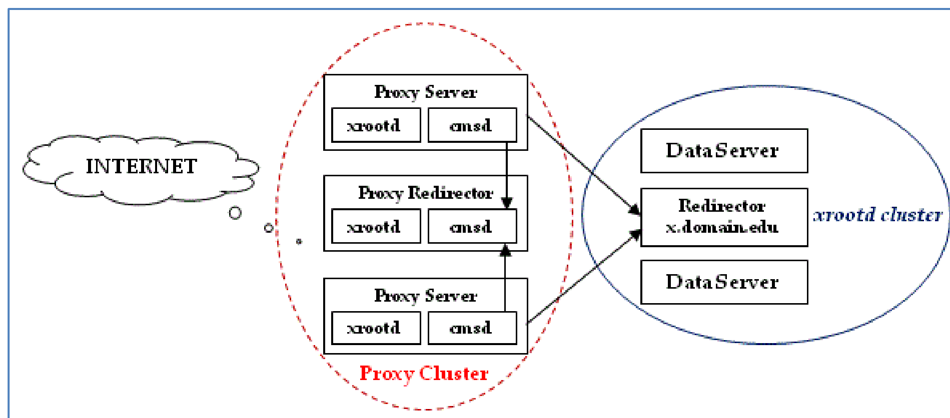
The **pss.origin** directive tells the plug-in where the XRootD cluster's redirector is located (in the example it is x.domain.edu). The port number must match the port number used by your local clients (in the example this is 1094). Indeed, the proxy server is, in fact, just another local client that is capable of serving data across the public internet.

The directives, **xrd.allow** and **sec.protocol**, are used to limit who can use the proxy server. The **xrd.allow** directive can restrict access by domain when you use the asterisk notation. The **sec.protocol** directive can restrict access to authenticated clients. While neither is required, your particular security requirements may force you to specify one or both of them.

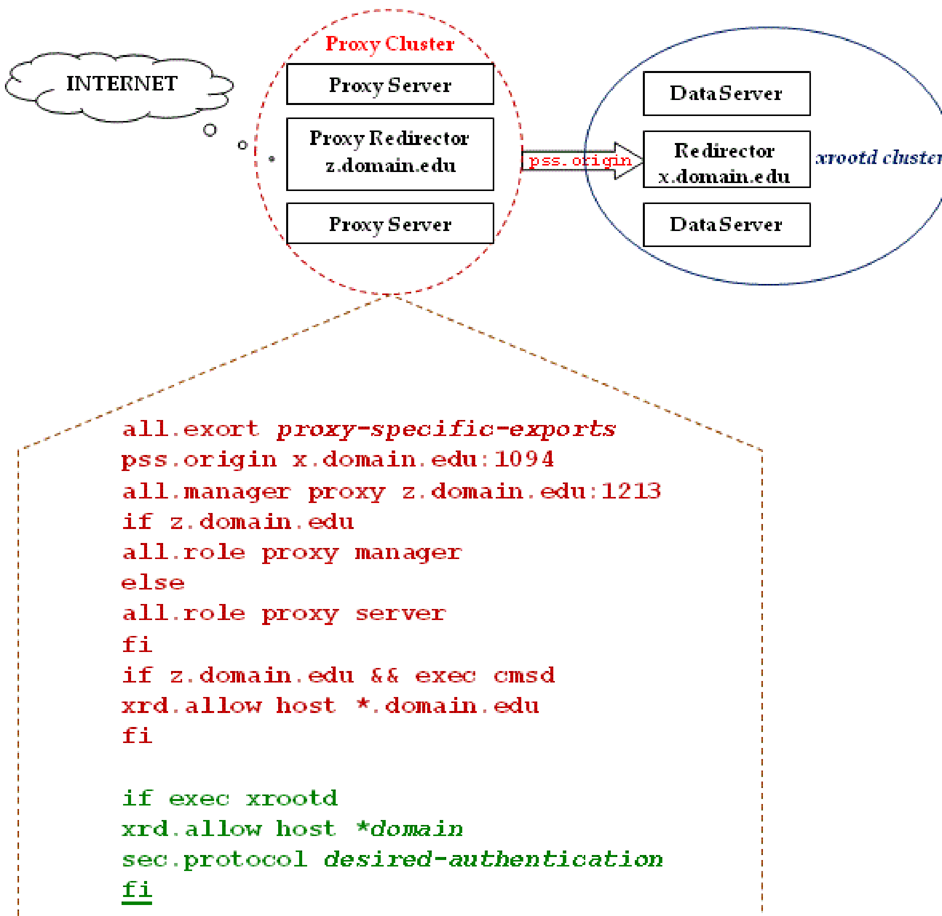
4.2 Proxy Cluster

A proxy cluster consists of two or more simple proxy servers arranged as an **XRootD** cluster. While one could use a load-balancing DNS to bridge together multiple proxy servers, using the **XRootD** cluster management infrastructure to do this provides not only true load balancing but also fail-over. That is, should a proxy server become unresponsive an **XRootD** client will automatically switch to another working proxy server. This also allows you to take down proxy servers for maintenance without worrying if they are currently in use, as long as one proxy server remains.

The mechanics of setting up a proxy cluster is largely the same as setting up a regular **XRootD** cluster. However, different parameters are specified for the **all.role** and **all.manager** directives. Additionally, unlike a simple proxy server you must also run a **cmsd** daemon everywhere you run an **XRootD** proxy server and you must configure a proxy redirector. The initial point of contact for external clients is the proxy redirector. This is shown in the adjacent diagram.



Setting up a proxy cluster is a minor variation of setting up a standard **XRootD** cluster. For proxy servers you need to identify the proxy manager and tell each proxy server that it is part of a proxy cluster. This allows the proxy servers to cluster around the proxy manager and the proxy manager then knows how to redirect clients to a working proxy server. A sample configuration file follows. You should compare it to one for a simple **XRootD** data cluster.



All red directives are required. The **all.export** and **pss.origin** directives are the same as for simple proxy servers. These tell the **XRootD**'s what to expor, and where the **XRootD** data cluster resides, respectively.

The **all.manager** directive tells each proxy server who the proxy redirector is and what port the **cmsd** is using (here we arbitrarily chose 1213).

In the **if-fi** construct roles are assigned to each server. Since z.domain.edu is the redirector, its **role** is **proxy manager**. All other nodes in the proxy cluster have a **role** of **proxy server**. This also causes the ofs layer to automatically load **libXrdPss.so**, by default, as the **ckslib** and the **osslib**; making the **ofs.ckslib** and **ofs.osslib** directives unnecessary. If the defaults are inappropriate, specify these directives to override the default.

Since we only want true proxy servers to be allowed to connect to the **cmsd** running on the proxy redirector node, the second **if-fi** construct specifies that only those nodes are allowed to connect. This is especially important because the proxy redirector is publically accessible and cmsd connections should be restricted to a known set of hosts. The particular example uses a generic specification. You may wish to make "allow" more specific. For instance, if proxy servers have a **DNS** name of "proxy*nn*.domain.edu", where *nn* is a sequence number, then it would be better to specify "**xrd.allow host proxy*.domain.edu**".

The green directives, **xrd.allow** and **sec.protocol**, are used to limit who can use the proxy servers. The **xrd.allow** directive can restrict access by domain when you use the asterisk notation. The **sec.protocol** directive can restrict access to authenticated clients. While neither is required, your particular security requirements may force you to specify one or both of them. You should especially note that these directives *apply only* to the **XRootD** and not the **cmsd**. This is why they have been qualified by an **if-fi** clause that applies them only to the **XRootD** proxy servers.

The default load distribution is to round-robin requests across all available proxy servers. You can use the **cms.sched** directive for the proxy redirector along with the **cms.perf** directive for proxy servers to do load balancing on actual load. These directives are described in the **cmsd** reference manual.

When configuring a proxy cluster, all proxy servers must either be in direct mode, forwarding mode, or combination mode. Any attempt to mix modes will lead to unpredictable result.

If you are using disk-caching proxies, a different type of cluster configuration must be used. Refer to the description of disk caching proxy clusters for details.

4.3 Configuration

The proxy server plug-in is based on the **XRootD POSIX** library which, in turn, uses the **XRootD client** library. Each of these components has specific options that can be specified in the same configuration file used for the proxy setup. Specifically, the following options are processed by the proxy

- **all.export**
- **oss.defaults**

Proxy specific directives and options are described in the following sections.

4.3.1 Required Directives

```
pss.origin  dest | =[plist] [dest]   required directive

dest:     host[:port | port] | url
url:      prot://host[:port]
prot:     http | https | root | roots | xroot | xroots
plist:    prot[,prot[,…]]
```

Where:

origin specifies the location of the redirector or server for which the server being configured is to proxy. ***This directive is required***. You may specify this is a *host* and optional *port* or a *url*. See the section that describes when you should use a **URL** format.

host The DNS name or IP address of the redirector or server being proxied.

port The TCP port number or service name of the redirector or server associated with *host*. The port may be specified with an adjacent colon or space separation. The default is 1094.

= Configures the proxy in forwarding mode where the client supplies the actual endpoint. If a *host* and optional *port* follows the equals sign then this endpoint is used when the client has not specified a **root** or **xroot** URL as an endpoint. Otherwise, the client receives a "not supported" error. The equal sign may be followed by a list of *additional* acceptable protocols. The default set includes **root**, **roots**, **xroot**, and **xroots**. These protocols are always acceptable.

4.3.1.1 Specifying the origin as a URL

Normally, the proxy uses **xroot** protocol (equivalent to **root** protocol) to communicate with the origin. If the communications must be encrypted (i.e. **TLS**) or you wish to use **http** protocol, the origin must be specified in **URL** format so that correct protocol is used. For **TLS** with **xroot** protocol; specify **xroots** or **roots**. For unencrypted **http** protocol, specify **http**. For **http** using **TLS**, specify **https**.

Be aware that using **http** or **https** as the origin protocol requires that you configure the embedded **xroot** client that the proxy uses to automatically load an **http** plug-in so that it can use **http** protocol. The proxy server will not function unless this is done.

4.3.2 Data Caching Directives

```
pss.cache       see memory caching section

pss.cachelib    see disk caching section

pss.ccmlib      see disk caching section

pss.dca [group | world] [recheck {tm | off}]
```

Where:

dca [recheck {*tm* | **off**}]

Enables direct cache access if the proxy server is configured to be a disk caching server. Direct cache access is only available for clients that can be redirected to an actual file (i.e. release 4.8.0 or above). The parameters are:

group the file mode is set to group read access prior to redirecting the client to directly access the file. This is the default.

world the file mode is set to world read access prior to redirecting the client to directly access the file.

recheck *tm* The number of seconds between checks whether or not the file is fully cached. The client is redirected to the file on the next read should the file be fully cached. This is only done should the client support redirects on a read request (i.e. release 4.9.0 or above). Specify for *tm* the number of seconds between checks. You may also suffix *tm* with **h**, **m**, or **s** to indicate **h**ours, **m**inutes, or **s**econds, respectively.

recheck off Turns off periodic checking for a fully cached file. This is the default. The client is only redirected to the file upon opening it should it be fully cached

4.3.2.1 Direct Cache Access

The proxy system service, when configured to be a disk caching proxy, supports direct cache access should the file be fully cached. This only makes sense when the cache is directly available to clients using the service. This is normally the case when the cache is located on a distributed file system (e.g. **GPFS**, **Lustre**, etc) and all nodes running client applications have the file system mounted at the same mount-point as the proxy service. Direct cache access is a feature of the **P**roxy **S**torage **S**ervice and must be enabled using the **pss.dca** directive.

Direct cache access can provide a significant performance enhancement, especially if the file system uses **RDMA** to transfer data to the client. Even without **RDMA**, eliminating the server between the client and the data can substantially increase performance.

Direct cache access is only available to clients that support redirection to a file on open (i.e. release 4.8.0). The secondary mode of redirecting the client to a file during read operations is only supported for release 4.9.0 clients. Should a client not support the required redirection level, the Proxy Storage System selectively inhibits redirecting the client and continues to provide the appropriate level of access.

To maintain secure access, clients using direct access should share a group with the user running the proxy cache service. This allows the clients and the proxy service to access the file via a group read access. If this is not possible, you may use the **world** option, though this would allow anyone to read the file from the cache.

4.3.3 Debugging Directives

```
pss.debug

pss.trace {all | on | debug}
```

Where:

debug
Enables debug output in the proxy layer.

trace Enables debug output in the **POSIX** layer. Each option enables the tracing of events up to a certain level:

all informational events.
on warning events.
debug error events.

4.3.4 Path Processing Directives

```
pss.localroot path

pss.namelib [-lfncache[src[+]]] [-lfn2pfn] path [parms]

pss.reproxy
```

Where:

localroot

Configures the default name- to-name mapping using the specified path. The action is identical to that of **oss.localroot** but is only used by the proxy to map file names prior to requesting action from the origin.

namelib

specifies the name- to-name mapping plug-in library. This directive works just like the **oss.namelib** directive but only applies to the proxy server. When neither the **-lfncache** nor **-pfn2lfn** options are specified, then **-pfn2lfn** is assumed. Otherwise, you must specify *all* the required options:

-lfncache	Calls the **namelib** plugin's **pfn2lfn()** method to convert the incoming physical filename to a logical one. That filename is used to track the file in any configured cache. This implies that different physical filenames may map onto the same file as long as they have the same logical name.
-lfncachesrc	Performs the same action as **-lfncache** but also passes a single **CGI** element indicate the data source in **URL** format as **src=***protocol***://***host*[**:***port*]
-lfncachesrc+	Performs the same action as **-lfncache** but also appends all of the **CGI** elements that are present on the **URL** to be used.
-lfn2pfn	Calls the **namelib** plugin's **lfn2pfn()** method to convert the incoming logical filename to a physical one. That filename is used to access the file at the specified **origin**.

reproxy

This directive configures the proxy server for third party copy targeted towards writable origins. This is a very specialized feature and works only with specific storage system (e.g. EOS) and is explained in the next section.

4.3.4.1 Reproxying Third Party Copy

The proxy server is capable of handling third party copy requests (**TPC**). This allows data to flow through a firewall directly from a source to destination. In most cases this works with no special accommodation for the target endpoint. In some cases, however, certain endpoints require that the proxy be aware of the actual server creating the file. This is required when the proxy is used to also track the progress of the copy. Certain storage system, like **EOS**, track file creation at the final endpoint and when file status is requested from the cluster redirector, the returned information always shows zero length until the file is closed. Since externally is appears the copy is making no progress the copy operation, in most cases, is terminated.

The mechanism used by the transfer agent to communicate the actual endpoint to the proxy server is:

1) When the transfer agent starts the environmental variable **XRD_CPTARGET** is set to contain a local path that the proxy server expects to find a symlink that contains the **URL** of the final endpoint.
2) The transfer agent should create such a symlink once it determines the final endpoint.
3) Whenever an **fstat()** request is processed for the destination file the proxy server checks for a symlink whose path matches **XRD_CPTARGET** given to the transfer agent.
4) If the symlink exists, it is read and the symlink deleted. He contents of the symlink is used to issue a **stat()** request to get the state of the file. The result is returned to **fstat()** issuer.
5) Should the final endpoint change, the transfer agent should create a new symlink with the correct **URL**.

Currently, only **xrdcp**, version 5.3.0 or higher, supports this mechanism.

4.3.5 Networking Directives

```
pss.inetmode {v4 | v6}
```

Where:

inetmode

Specifies which network stack to use when connecting to other servers. The default is determined by the **-I** command line option; which defaults to **v6**. Options are:

v4 Use the IPV4 network stack to connect to servers. This means only servers reachable by IPV4 addresses may be used.

v6 Use the IPV6 network stack to connect to servers. This means only servers reachable by IPV4 mapped addresses or IPV6 addresses may be used. This is the least restrictive option.

Notes

1) The **inetmode** option allows you to create network protocol stack bridges. Fr instance, if the proxy accepts **IPV6** addresses and **inetmode v4** is specified; the proxy becomes a bridge between incoming **IPV6** addresses and an **IPV4** network.

4.3.6 Security Directives

```
pss.permit [/] [*] hspec

pss.persona {client | server} [[non]strict] [[no]verify]
```

Where:

permit

specifies an allowed destination that a client can use via a forwarding mode proxy. If no permit directives are specified, no restrictions apply. Permits, by default, apply to path and objectid URLs.

/ The permit applies only to path-type URLS.

* The permit applies only to objectid-type URLS.

hspec The DNS host name allowed as a target connection. Substitute for *name* a host name or IP address. A host name may contain a single asterisk anywhere in the name. This lets you allow a range of hosts should the names follow a regular pattern or if you wish to allow connections to all hosts in a particular domain. IP addresses may be specified in IPV4 format (i.e. "a.b.c.d") or in IPV6 format (i.e. "[x:x:x:x:x:x]").

persona

specifies how credentials are to be handled by the proxy when authenticating with the **sss** (**S**imple **S**hared **S**ecret) authentication protocol. The next section describes the restrictions and limitations of this directive.

client the proxy should connect using the client's credentials (client persona).

server the proxy should connect using its own credentials (server persona). This is the default.

strict The specified persona must always be used. This is option is only meaningful for client personas.

nonstrict

The specified persona should always be used for but operations except tests for file existence (i.e. **stat()** calls). For **stat()** calls the server's persona is used in order to significantly reduce the latency of the operation. This option is only meaningful for client personas.

verify Perform mutual authentication with the origin server during **sss** authentication. This is the default for client personas.

noverify
does not perform mutual authentication with the origin server during **sss** authentication. This is the default for server personas.

4.3.6.1 Persona restrictions and limitations

The persona capabilities enabled by the **pss.persona** directive are not designed for generalized credential forwarding or even delegation. They are specifically designed to assist front-backend configurations in applying the correct authorization requirements. The proxy, at the front-end, performs the required authentication and then recreates the client's identity in a backend server so that it appears that the client actually authenticated with the backend. The backend can then perform required authorization using those credentials. This requires that the proxy and backend servers authenticate with the **sss** (**S**imple **S**hared **S**ecret) authentication protocol. Refer to the *Security Reference* on how to configure the **sss** protocol to successfully enable this style of credential forwarding.

Client personas cannot be enabled for certain types of proxy server configurations. Doing so causes proxy initialization to fail. These restrictions are:

- servers configured to use a cache (disk or memory), and
- servers configured to only forward client requests (i.e. have no origin server).

You may enable client personas on a server that can forward requests if it can also handle non-forwarded requests (i.e. has an origin server). However, the client persona is only applied to interactions with the origin server. Forwarded requests use the server's persona.

4.3.7 Tuning Directives

```
pss.ciosync ssec msec

pss.config {[streams snum] [workers number]}

pss.setopt ConnectTimeout seconds
pss.setopt DataServerConn_ttl seconds
pss.setopt DebugLevel {0 | 1 | 2 | 3}
pss.setopt ParallelEvtLoop number
pss.setopt ParStreamsPerPhyConn number
pss.setopt ReconnectWait seconds
pss.setopt RedirectLimit number
pss.setopt RedirectorConn_ttl seconds
pss.setopt RequestTimeout seconds
pss.setopt TransactionTimeout seconds
pss.setopt WorkerThreads number
```

Where:

ciosync *ssec msec*

Configures the parameters for cache I/O synchronization when a file is closed. All outstanding I/O for a file being closed must completed within msec, otherwise the file object is orphaned to prevent the possibility of a crash. The default is 30 seconds for *ssec* and 180 seconds for *msec*. This directive is only applicable to a proxy configured with a cache.The parameters are:

ssec The number of seconds between attempts to synchronize outstanding I/O with a file close request. The minimum value is 10 seconds. You may suffix the value with **s**, **m**, or **h** for seconds (the default), minutes, or hours, respectively.

msec The maximum number of seconds that I/O may be outstanding before closure of the file is abandoned and the file object is orphaned. The value must be greater than or equal to *ssec* times two. You may suffix the value with **s**, **m**, or **h** for seconds (the default), minutes, or hours, respectively

config

Configures various parameters affecting the fuse interface. The defaults are:

streams 512 workers 16

Otherwise, the following option may be specified:

streams The *number* of parallel streams (i.e. connections) to be used when processing a stat() call. The default is 512. You may specify a maximum of 8192.

workers The *number* of parallel threads to be used when collecting information from data servers or issuing requests to multiple data servers due to a single request. The default is 16.

setopt

Configures the **XRootD** client used by the proxy top communicate with the origin. Each **setopt** directive allows a single configuration argument. Specify one or more of the following, as needed.

ConnectTimeout

The number of *seconds* to wait for a server connection to complete. The default is 120 seconds.

DataServerConn_ttl

The number of *seconds* to keep an idle data server socket open. The default is 1200 (i.e. 20 minutes).

DebugLevel

The level of debugging. **0,** the default, turns off debug output; increasing values produce more detail.

ParallelEvtLoop

The *number* of parallel event loops to run to field socket interrupts. The default is **3**. Consider increasing this value for servers with a very high transaction rate.

ParStreamsPerPhyConn

The *number* of parallel TCP streams to use for data server I/O. You can specify a number from **0** to **15**. The default is **2**.

ReconnectWait

The number of *seconds* to wait between server reconnects in the presence of a fatal error. The default is 1800.

RedirectLimit

The maximum number of sequential (i.e. without a break) redirects to other servers that may occur before a fatal error is declared. The default is 16.

RedirectorConn_ttl

The number of *seconds* to keep an idle redirector socket open. The default is 3600 (i.e. 60 minutes).

RequestTimeout

The maximum number of seconds a request can be active without a response before an error is declared. The default is 300 seconds.

TransactionTimeout

The maximum number of *seconds* of wait-time that can be imposed by the server before an error is declared. The default is 28800 (i.e. 8 hours).

WorkerThreads

The maximum *number* of responses that can be processed in parallel. Each response is handled in a separate thread. The default is 64.

Notes

1) The proxy also accepts environmental variables that control the underlying **XRootD** client. Some of these environmental variables offer more extensive control over the client's behavior then the **setopt** directive allows. When an environmental variable is set, it over-rides the equivalent specification in the configuration file. The variables are described in the documentation for the **xrdcp** command.

5 Disk Caching Proxy

The disk caching proxy is almost identical to the standard proxy except that a disk is substituted for a memory cache. This makes the proxy more suitable for WAN access. The **libXrdPfc.so** shared library is a proxy server cache plug-in used for caching of data into local files. Two modes of operation are supported.

The first mode simply pre-fetches complete files and stores them on local disk. This implementation is suitable for optimization of access latency, especially when reading is not strictly sequential or when it is known in advance that a significant fraction of a file will be read, potentially several times. Of course, once parts of a file are downloaded, access speed is the same as it would be for local **XRootD** access.

The pre-fetching is initiated by the file open request, unless the file is already available in full. Pre-fetching proceeds sequentially, using a configurable block size (1 MB is the default). Client requests are served as soon as the data becomes available. If a client requests data from parts of the file that have not been pre-fetched yet the proxy puts this request to the beginning of its download queue so as to serve the client with minimal latency. Vector reads are also fully supported. If a file is closed before pre-fetching is complete, further downloading is also stopped. When downloading of the file is complete it could in principle be moved to local storage. Currently, however, there are no provisions in the proxy itself to coordinate this procedure.

A state information file is maintained in parallel with each cached file to store the block size used for the file and a bit-field of blocks that have been committed to disk; this allows for complete cache recovery in case of a forced restart. Information about all file-accesses through the proxy (open & close time, number of bytes read and number of requests) is also put into the state file to provide cache reclamation algorithms with ample details about file usage.

The second mode supports on-demand downloading of individual blocks of a file (block-size can also be passes as **cgi** information in the file-open request). The distinguishing feature of the second implementation is that it only downloads the requested fixed-size blocks of a file.

While the main motivation is to provide pre-fetching of **HDFS** blocks (typical size 64 or 128 MB) when they become unavailable at the local site, either permanently or temporarily due to server overload or other transient failures. As **HDFS** block size is a per-file property, it has to be passed to the proxy on per-file basis as a **cgi** parameter on the file's URL Each block is stored as a separate file, post-fixed by block size and its offset in the full file; this facilitates potential reinjection back into **HDFS** to heal or increase replication of a file-block. Extensions to the **HDFS** have been developed to allow for an immediate fallback to **XRootD** access when local **HDFS** storage fails to provide the requested block.

That said, the mere fact that only referenced blocks are cached makes the caching proxy a powerful tool for any kind of remote file system, irrespective of block size. This is especially true in workloads that reference of a fraction of the file. Because only relevant blocks are cached, disk space usage is minimized and overall **WAN** performance is improved especially when running multiple jobs that largely reference the same blocks.

When additional file replicas exist in a data-federation, the remote data can be used to supplement local storage, to improve its robustness, and to provide a means for healing of local files. In particular, our intention is to avoid any local file replication of rarely-used, non-custodial data.

Unlike the full-file pre-fetching version, the partial-file proxy does not begin pre-fetching any data until a read request is actually received. At that point a check is made if the blocks required to fulfill the request exist on disk and, if they don't, they get queued for pre-fetching in whole. The client request is served as soon as the data becomes available.

5.1 Configuration

The disk caching proxy is implemented as a specialized caching plug-in that supplants the optional memory caching provided by the standard proxy server. Disk caching proxy specific directives and options are described below.

```
pfc.acchistorysize maxcount

pfc.blocksize bytes[k|m]

pss.cachelib path [libopts]          required directive

pss.ccmlib path [libopts]

pss.cschk [[no]cache] [[no]net] [off] [[no]tls]
          [uvkeep n[d|h|m|s]|lru}]

pfc.decisionlib path [libopts]

pfc.diskusage lowWatermark[k|m|g|t] highWatermark[k|m|g|t]
              [files base[k|m|g|t] nom[k|m|g|t] max[k|m|g|t]]
              [{purgeinterval | sleep} purgeitvl[h|m|s]]
              [purgecoldfiles age{d|h|m|s} period]

pfc.flush {blocks | bytes[k|m|g]}

pfc.hdfsmode [hdfsbsize bytes[k|m]]

pfc.osslib path [libopts]

pfc.prefetch numPrefetchingBlocksPerFile

pfc.ram bytes[m|g]

pfc.spaces data metadata

pfc.trace {none | error | warning | info | debug | dump}

pfc.user username

pfc.writequeue maxblks nthreads
```

Required Directive

cachelib *path* [*libopts*]

specifies the path of he shared library than contains the disk caching proxy plug-in. It may be followed by options, if any. Currently, there are no options.

Optional Directives

pfc.acchistorysize *maxcount*

Specifies the number of access records kept for each data file. Specify for *maxcount* a value between 20 and 200, inclusive. The default is 20.

blocksize *bytes*[**k** | **m**]

sets the block size used by proxy. Specify for *bytes* the size of a block. The quantity may be suffixed by **k** or **m** to indicate **k**ilo- or **m**egabytes, respectively. All read requests, including prefetching are rounded up into requests of this size and aligned to block boundaries. Read requests that cross one or more block boundaries are issued as separate read requests. This is also the size that gets written to disk in a single operation. The default is 128k.

Be aware that when client requests are processed using asynchronous I/O, the incoming request is broken into segment sized pieces to allow the data to be streamed from the origin. The default segment size is 64k. The only adverse effect of this mismatch is that the hit/miss statistics for a particular read will not match the expected result. Setting the **xrootd** segment size to match the default block size will degrade the transfer rate, especially if the origin is behind a firewall. Consider adjusting the block size to match the segment size if you need to eliminate the hit/miss aberration.

ccmlib *path* [*libopts*]

specifies the location of the shared library that contains the cache context manager plugin provides custom cache semantics. Specify for *path* the location of the **ccm** plugin shared library and, optionally, plugin specific options for *libopts*.

cschk { [[**no**]**cache**] [[**no**]**net**] [**off**] [[**no**]**tls**] [**uvkeep** {*n*[**d** | **h** | **m** | **s**]} | **lru**}}

specifies data integrity options. The cache has the capability of verifying that the correct data was read from the network (data in transit integrity) as well as the correct data was read from the permanent cache (data at rest integrity). Refer to the following section on a detailed explanation of these options. The options are:

[**no**]**cache**	do [**not**] perform cache checksum integrity checking; **nocache** is the default.
[**no**]**net**	do [**not**] perform network checksum integrity checking; **net** is the default.
[**no**]**tls**	do [**not**] use **TLS** as a substitute for network checksum integrity checking when a server supports **TLS** but does not support data checksums; **tls** is the default.
off	synonym for the combination **nocache nonet**.
uvkeep	specifies how long cached files with unverified checksums are to be kept. Specify for *n* the number of **d**ays, **h**our, **m**inutes, or **s**econds (the default) after which such cached data is to be deleted. A value of **0** never records unverified checksums and their associated data to permanent media (i.e. such data is served only via a memory cache). Specify **lru**, the default, to use the standard mechanism for deleting cache data files.

decisionlib *path* [*libopts*]

specifies the location of the shared library that contains the cache decision plugin which determines whether or not to cache a file. All files are cached if a plugin is not specified. Specify for *path* the location of the decision plugin shared library and, optionally, plugin specific options for *libopts*.

diskusage

specifies physical storage management properties. The parameters and options are (see the notes for additional caveats):

lowWatermark[**k** | **m** | **g** | **t**] *highWatermark*[**k** | **m** | **g** | **t**]

specifies the total usage on the file system or set of disks configured with the **oss.space** directive that triggers purging of cached files (*highWatermark*) and the total usage at which the purging is stopped (*lowWatermark*). The watermarks must be specified as a decimal fraction of total available space (default values are 0.90 and 0.95) unless the values are suffixed by **k**, **m**, **g**, or **t**. In which case, they must be absolute sizes in **k** (kilo-), **m** (mega-), **g** (giga-), or **t** (tera-) bytes, respectively.

files *base*[**k**|**m**|**g**|**t**] *nom*[**k**|**m**|**g**|**t**] *max*[**k**|**m**|**g**|**t**]

Specifies limits for actual usage of disk space by the data files owned by the configured cache instance. When files reach *max*, they are purged down to *nom*. If purging due to total disk usage requires further removal of files, they will be purged down to *base* if required for reaching of *lowWatermark* disk usage. Specify absolute sizes suffixed by **k** (kilo-), **m** (mega-)**, g** (giga-), or **t** (tera-) bytes. Default: limits are not set and file usage limits are not checked nor enforced.

{**purgeinterval** | **sleep**} *purgeitvl*[**h**|**m**|**s**]

specifies the interval between subsequent disk usage checks. Specify for *purgeitvl* a number optionally suffixed by **h** for hours, **m** for minutes, or **s** for seconds. The resulting value must be between 60 and 3600 seconds. The default is 300s. The keyword **sleep** is a deprecated synonym.

purgecoldfiles *age*{**d**|**h**|**m**|**s**} *period*

specifies the time since last access at which a file is unconditionally purged. This is useful for cache instances that share disk space and are only used sporadically so it makes sense to clear the cache between peak usages. Specify for *age* a number optionally suffixed by **h** for hours, **m** for minutes, or **s** for seconds. The resulting value must be between 1h and 360d, inclusive. The *period* argument specifies how many purge cycles need to occur before an age check if performed. This avoids redundant scanning of meta-data when other purge conditions (disk and file-usage based ones) are not met. There is no default and cold files are not purged unless this parameter is specified.

flush {*blocks* | *bytes*[**k**|**m**|**g**]}

forces a **sync** operation on all of the data and meta-data files after the number of specified *blocks* or *bytes* have been written to memory. The *bytes* value may be suffixed by **k**, **m**, or m to indicate **k**ilo-, **m**ega-, or **g**igabytes, respectively. The default is 2000 blocks.

hdfsmode [**hdfsbsize** *bytes* [**k**|**m**]]

enables storage of file fragments as separate files. This is specifically used for storage healing in HDFS. Specifying *bytes* sets the default fragment size. Specify for *bytes* the size of a fragment. The quantity may be suffixed by **k** or **m** to indicate **k**ilo- or **m**egabytes, respectively. The default is 128MB.

osslib *path* [*libopts*]

specifies the location of the shared library that contains the storage system plugin that should be used for all cache operations (i.e. read, write, etc). Specify for *path* the shared library that contains the plugin and, optionally, plugin specific parameters for *libopts*. The default is to use standard **oss** plug-in for all file system functions. Refer to the "**Open File System & Open Storage System Configuration Reference**" on how to configure the standard **oss** plug-in.

prefetch *num*

maximum number of pre-fetching blocks per file. The default is 10. Set this value to 0 to disable pre-fetching.

ram *bytes*[m|**g**]

maximum allowed **RAM** usage for caching proxy buffers that still need to be written to disk. Beyond that point the cache will serve further read requests by forwarding them to the remote server (note, it still needs to allocate buffers for this). For clients, specify a value between 256m and 64g (default is 256m). For servers, specify a value between 1g and 256g (default is 1g).

spaces *data metadata*

spacifies the names of **oss** space name for data and the **oss** space name for metadata. The **oss.space** directive is used to create spaces. Default value of both is public.

trace {**none** | **error** | **warning** | **info** | **debug** | **dump**}

Set severity level of caching proxy log messages. The levels are listed in increasing verboseness. The default value is **warning**.

user *username*

specifies the user name to pass to the **osslib** plug-in when accessing the disk cache. Specify a valid Unix user name for *username*. Normally, this should be the user name of the proxy daemon.

writequeue *maxblks nthreads*

specifies how disk cache write-backs are to be handled. Specify for *maxblks* the maximum number of blocks to be taken off the pending write queue and written to disk in a single iteration of the disk-writer loop. Accepted values are between 1 and 1024. The default is 16 for a regular proxy server and 8 for a server-less proxy cache. Specify for *nthreads* the number of threads that perform writing of cache blocks to disk. Accepted values are between 1 and 64. The default is 4 for a regular proxy server and 1 for a server-less proxy cache.

Notes

1) All features of the standard Open Storage System are available to the disk caching proxy. Perhaps the most essential feature is logical volume management that allows the concatenation of disk partitions into a single logical volume. Thus, you can easily extend the disk cache. See the **oss.space** directive in the "**Open Storage System Configuration Reference**". There are many other **oss** options that may also be helpful in extending the features of the disk caching proxy.
2) If several caching servers are running on the same file system or set of disks configured with **oss.space** directive, they should all use the same *lowWatermark*, *highWatermark*, and *purgeinterval*. They can use different settings for the **files** section and can even oversubscribe the disk usage for *nom* usage as the *low/highWatermark* mechanism will kick in to ensure specified amount of disk remains free. The *baseline* values for all the instances should add up to less than *lowWatermark*.
3) The **tls** option is ignored unless **net** is in effect.
4) The options are cumulative across directive specifications.

Examples

a) Enable file prefetching:

```
pss.cachelib   libXrdPfc.so
pfc.ram        100g
pfc.blocksize  512k
pfc.prefetch   8
```

b) enable **hdfs** healing mode, with block size 64 MB:

```
pss.cachelib   libXrdPfc.so
pfc.ram        100g
pfc.hdfsmode   hdfsbsize 64m
```

5.1.1 Disk Caching S3-type Objects

The disk caching feature can be used to also cache **S3**-type data objects (e.g. **Ceph** objects). Such objects are identified by object ID and do not start with a slash. You must specify certain configuration options to enable successful caching of **S3**-type objects. Minimally, you need the following set of directives in addition to any other applicable directives.

```
all.export *?

pss.namelib -lfncache libXrdN2No2p.so [parms]

parms:    [-maxfnlen len] [-slash {c | xx}] [prefix]
```

The **all.export** directive specifies that object ID's are allowed and that any **CGI** information (i.e. characters after the first question mark) are to be removed from the object ID and passed as separate information to the proxy plugin. The plugin requires that any **CGI** information be passed separately so must specify '***?**' to get consistent results.

A special Name2Name plugin must be used to convert object ID's to file paths so that the data they refer to can be cached as local disk files. You don't need this plugin if the storage system plugin you specified using the **pfc.osslib** directive is an object store (e.g. **Ceph**). The default cache storage system is a file system store via the **oss** plugin.

The plugin handles object ID conversion as follows:

1) All occurrences of a slash (i.e. '/') are changed to be a reverse slash (i.e. '\' or escape). This is necessary to prevent object id's containing slashes from colliding. If your object ID's use reverse slashes then you must choose another character that you won't use in the ID's and specify that character using the '**-slash**' option in the *parms* passed to the plugin. You can specify it as a single character or as a two character hexadecimal code (e.g. 32 for a space character).
2) The maximum allowed filename length is determined by what is allowed to be created in '/' directory. If this is incorrect, use the **-maxfnlen** *parms* option to specify the correct maximum length.

3) If the object ID is longer than what is allowed for a filename length, the object ID is transformed into a path by splitting the ID into filename max units until the ID is exhausted. Otherwise, two 1-character hash codes are developed from the object ID itself and each is used as a directory prefix to the object ID. The directory names are actually two character hexadecimal digits representing each hash character.
4) If a *prefix* is specified in the *parms* passed to the plugin, it is used to prefix the result from the previous step. This allows you to place all object ID's in a separate directory in the cache.

Using the above steps, an object ID of say "foobar" would be placed in

/osslocalroot/*prefix*/68/7F/foobar

Where *osslocalroot* comes from the **oss.localroot** directive and *prefix* comes from *parms* passed to the plugin. Either or both may be null.

5.1.2 Cache Integrity Options

The following matrix shows how combinations of integrity options interact.

	cache	**nocache**
net	Verifies checksums or checksum equivalents when sent with data over the network and records them with the cached file. The checksum is verified when the data is read from the cached file.	Verifies checksums or checksum equivalents when sent with data over the network. No checksum integrity checks are performed on the cached file.
nonet	Does not perform a transmission integrity check but does generate unverified checksums on the received data and records them with the cached file. The checksum is verified when the data is read from the cached file.	Does not perform any integrity checking on received or read data. This combination is equivalent to specifying **off**.

Network data integrity is very much dependent on the data provider supporting checksums on transmitted data. All **XRootD** data servers, release 5 and above, support transmission integrity using **crc32c** checksums. If a server does not support checksum integrity but does support **TLS**, the data is transmitted using **TLS** to verify integrity and **crc32c** checksums are then locally generated unless **notls** has been specified. In either case, these are called verified checksums.

When a data server supports neither checksum integrity nor **TLS**, the data cannot be verified. However, if the cache option is selected checksums are still generated locally and recorded with the cached file so that data at rest integrity can be performed. While these checksums are unverified they still detect media data alteration when the data is read back; even if the data was transmitted with data errors. The same is true if the combination **cache nonet** is specified.

Since a file can come from multiple sources, some of which providing unverified data; a cached file can be composed of verified and unverified pages. The **uvkeep** option may be used to limit exposure to corrupt data. This option specifies the longest time a cached file containing unverified pages may be used to provide data; after which the cached file is deleted and the data is fetched again, if needed. This option is useful when dealing with a large number of unverifiable data sources.

When only a handful of data sources provide unverified data, the **uvkeep 0** option can be used to ensure that a cached file never contains unverified data. Any data received without a checksum or checksum equivalent (i.e. **TLS**) is never written to media and is only served from memory. While this does increase overhead as data may need to be fetched multiple times, the overhead should be small if most of the data sources support data transmission integrity.

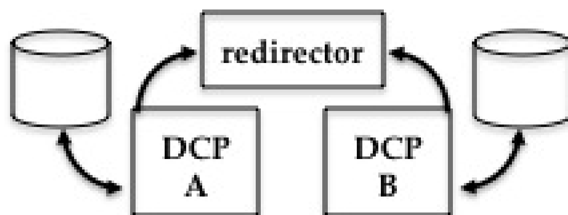
If the major problem is media corruption and virtually all network transmissions are error-free, the combination of **cache nonet** may make sense.

Finally, the cache integrity relies on the data source to ensure that correct data is actually being sent over the network. That means the data source should also have verified that the data has not been corrupted not only upon creation but also due to media failures when it is read back. The data integrity option provided by the caching proxy can only detect, at best, transmission and cache media errors.

5.2 Disk Caching Proxy Clusters

You may cluster disk-caching proxies (**DCP**'s) to provide for additional bandwidth and disk space using a redirector. However, clustering **DCP**'s is not the same as clustering standard proxy servers because **DCP**'s maintain a semi-permanent disk cache and cannot see the contents of each other's cache. Furthermore, clients opening a file must be vectored to the **DCP** that has already partially or fully cached the requested file and if the file is not present in any cache the client must be redirected to a **DCP** with sufficient disk space to cache the file.

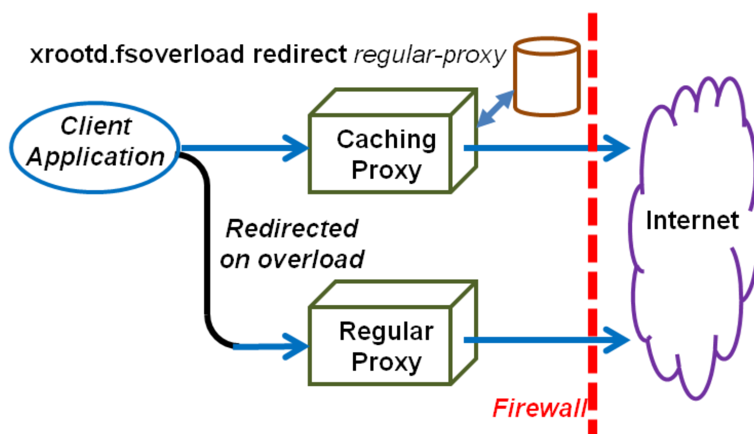
In some sense a **DCP** is a combination of a proxy and a disk server. Hence, they need to be clustered as disk servers not as standard proxy servers. In the figure to the left, two disk caching proxy servers, **DCP-A** and **DCP-B** are clustered using a redirector. Each **DCP** has its own disk cache that may not be shared. In fact, any attempt to share the same space using, say, a distributed shared file system will result in data corruption or loss. The minimal configuration file for such a cluster is shown on the next page. The configuration file uses host names of **dcp-a**, **dcp-b**, and redirector for each server component in the cluster.



```
# Tell everyone who the manager is
#
all.manager redirector:1213

# The redirector and all servers export /data with the cache attribute. Paths tagged with
# cache request that if the file isn't found in the cluster the redirector should send
# the client to a PFC server with enough space to cache the file.
#
all.export /data cache

# Everyone needs to know where the files will be placed (i.e. the root of the namespace).
# The namespace logically starts at /data but is physically located at /pfc-cache
#
oss.localroot /pfc-cache

# Configuration is different for the redirector, the server cmsd, and for the server xrootd.
# We break those out in the if-else-fi clauses. Specifically, the redirector must be told
# that it is a manager. The server cmsd is tols it's a server and should use the standard oss
# plug-in to locate files. The server xrootd, however, needs to use the proxy file cache as
# it's a virtual data server using specialized plug-ins.
#
if redirector

all.role manager

else if exec cmsd

all.role server

else

all.role server

# For xrootd, load the proxy plugin and the disk caching plugin.
#
ofs.osslib   libXrdPss.so
pss.cachelib libXrdPfc.so

# Tell the proxy where the data is coming from (arbitrary).
#
pss.origin someserver.domain.org:1094

# Tell the PFC's available RAM
#
pfc.ram 100g

fi
```

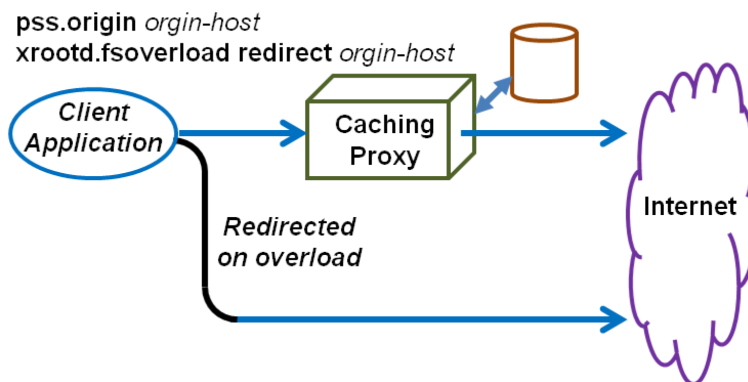
5.3 Handling Cache Overloads

The disk caching service used by the proxy server is able to detect overload conditions and notify the server that it is not able to service an incoming request. The default action taken by the proxy server is to delay the client until the cache can handle additional requests. However, it is also possible to redirect the client to another server. This can be another proxy server because of a firewall or to redirect he client to the actual origin of the data when there is no firewall. Redirection choices are specifying using the **xrootd.fsoverload** directive; refer to the "**Xrd/XRootD Configuration Referenc**e" for full details. This section explains overload redirection concepts to assist you in your configuration choices.

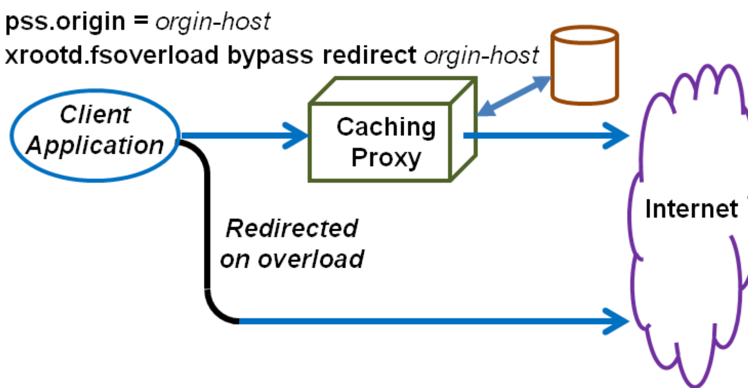


The graphic on the left shows a typical proxy that sits behind a firewall. All access to the internet requires that a proxy server be used. Normally, clients connect to the caching proxy server. This can be a direct mode, forwarding mode, or combination mode proxy. It doesn't really matter. When the caching layer indicates an overload, the client is redirected to another regular (i.e. non-caching) proxy to effect access to the data through the firewall. Both proxies specify the same origin configuration (see the **pss.origin** directive). However, the caching proxy server also specifies the **xrootd.fsorverload** directive using the **redirect** option to specify the alternate proxy server to use should the caching proxy become overloaded.

If your site does not have a firewall but you still want to cache data closer to your site; you would also setup a caching proxy server. The **pss.origin** would specify the host that normally clients would connect on the internet. If the caching proxy



becomes overloaded the **xrootd.fsoverload** directive specifies, via the redirect option, to send the client directly to the origin.



Finally, you can also use a forwarding mode or combination mode caching proxy without using a firewall. However, the **XRootD** layer must be told that client can be redirected to their original destination should a forwarding location be specified. This is done by adding the **bypass** option to the **xrootd.fsoverload** directive. The reason that you must specify bypass is to allow the proxy service to apply outgoing restrictions via the **pss.permit** directive before any redirection occurs. If you do not specify bypass, then the client is redirected to the host specified via the **redirect** option which usually is another proxy server that knows how to handle site forwarding.

Cache overload handling can be specified on an individual proxy server or all proxy servers within a cluster.

6 Memory Caching Proxy

The memory caching proxy is almost identical to the standard proxy except that memory is used to hold recently read data. The memory cache is used to satisfy future reads if the requested data is present in the cache. This avoids re-fetching data from the originating server. A memory cache proxy may make **LAN** transfers more efficient, depending on the data access pattern.

You configure a memory caching proxy the same way you configure a standard proxy server with the addition of a **pss.cache** directive, described in the next section. Be aware that disk caching proxies and memory caching proxies are mutually exclusive (i.e. you cannot configure both in the same server).

6.1 Configuration

Proxy memory caching specific directives and options are described below.

```
pss.cache [cacheopts]

cacheopts: [debug dbg] [logstats] [max2cache mc] [minpages mp]
           [pagesize ps][preread [prpgs [minrd]]
           [perf prf [rcalc]]] [r/w]
           [sfiles {off | on | .sfx}] [size sz] [cachepots]
```

Where:

cache

Configures memory caching of data. By default, memory caching is turned off. Specifying the **cache** directive with no *cachepots* is equivalent to specifying:

debug 0 minpages 256 pagesize 32k preread 0 sfiles off size 100m

If the **cachelib** directive is specified, the cache options have no effect. Otherwise, the following options may be specified:

debug sets the debugging level: 0 is off, 1 is low, 2 is medium, and 3 is high. The default is 0.

logstats prints statistics about cache usage for every file that is fully closed.

max2cache the largest read to cache in memory. The value is set to the **pagesize** if it is smaller than **pagesize** or not specified (i.e. default). The limit does not apply to pre-read operations. The *mc* value may be suffixed by **k**, **m**, or **g** to scale *mc* by 2^{10}, 2^{20}, or 2^{30}, respectively.

minpages specifies the minimum number of pages that the cache should hold. If it is unspecified or is less than one it is set to 256.

pagesize the size of each page in the cache. It is adjusted to be a multiple of 4k. This also establishes the minimum read size from a data server. The *ps* value may be suffixed by **k**, **m**, or **g** to scale *ps* by 2^{10}, 2^{20}, or 2^{30}, respectively. The default is 32K.

preread enables pre-read operations. By default no data is pre-read. Specify the number of additional pages to be read past the last read page. You also follow *prpgs* by the read size that triggers a pre-read. Reads less than *minrd* cause a pre-read to occur. The *minrd* value may be suffixed by **k**, **m**, or **g** to scale *minrd* by 2^{10}, 2^{20}, or 2^{30}, respectively. The default is the **pagesize** value plus one. Specifying **preread** with no options is equivalent to specifying:

preread 1 perf 90

preread perf specifies the minimum required performance from automatic pre-reads and is part of the **preread** options. When this performance cannot be obtained for a file, pages are no longer automatically pre-read for the file. Performance is measured as the number pre-read pages used divided by the number actually pre-read times 100. Specify a value between 0 and 100 for *prf*. The default is 90 (i.e., 90% of all pre-reads must be useful). The *prf* may be followed by *rcalc*, the number of pre-read bytes that trigger a new performance calculation. The *rcalc* value may be suffixed by **k**, **m**, or **g** to scale *num* by 2^{10}, 2^{20}, or 2^{30}, respectively. The default is 52428800 (i.e. 50m).

r/w caches files opened in read/write mode. By default, only files opened read/only are cached.

sfiles enables or disables optimization for structured file (e.g., root files). Cache usage is optimized for the typical access patterns associated with structured files. Specify **off** to turn off this feature (the default), **on** to always use this feature, or a dot followed by a suffix (i.e. **.***sfx*). Only files whose names end with this suffix, including the dot, will have this optimization applied.

size specifies the size of the cache in bytes. See the notes on how this value is adjusted irrespective of the specification The *sz* value may be suffixed by **k**, **m**, or **g** to scale *sz* by 2^{10}, 2^{20}, or 2^{30}, respectively. The default is 100m.

Notes

1) The **size** value may be adjusted upwards or downwards depending on the **pagesize** and **minpages** values. The following adjustments are made:
   a. The **size** value is adjusted downwards to be a multiple of **pagesize**.
   b. The final **size** value is either the *adjusted* **size** *or* **minpages** multiplied by the **pagsize**,,whichever is greater.

7 POSIX Server-less Caching

The subsystems that support caching proxy servers may also be used to setup a server-less proxy cache. This is essentially a client-side application cache. It caches data either in memory or on disk (depending on the configuration) by intercepting file system calls. Any data that is read is locally cached. Currently, this feature is available only via the **POSIX** preload library (i.e. **libXrdPosixPreload.so**) or by directly linking with **libXrdPosix.so** and using the **XRootD POSIX** interface API's defined in one of the following public include files:

- **XrdPosix.hh**,
- **XrdPosixExtern.hh**, or
- **XrdPosixXrootd.hh**

Using the **XRootD POSIX** API's is straightforward as they do not differ from the standard **POSIX** API's. The API's defined in **XrdPosix.hh** and **XrdPosixExtern.hh** accept either simple paths or **XRootD URL**'s. A simple file path vectors the request to the local file system while an **XRootD URL** (i.e. one that starts with **xroot://** or **root://**) vectors the request to the specified server. You can define virtual mount points that convert specified simple paths to actual **URL**'s. The interface defined in **XrdPosixXrootd.hh** only accepts **XRootD URL**'s.

The preload library mechanism offers a very simple way to run **POSIX**-compliant programs without needing to change them or even recompile them. You can invoke your program with a simple shell script shown below.

```
LD_PRELOAD=/usr/lib64/libXrdPosixPreload.so
export LD_PRELOAD
$*
```

Note that you may need to change the **LD_PRELOAD** value if the preload library is not installed in a standard place. So, assume the script is called **runit** then executing

runit ls /tmp

produces a directory listing of the local **/tmp** directory while

runit ls xroot://someserver//tmp

produces a directory listing of **someserver**'s **/tmp** directory, if allowed.

Regardless of whether you use the direct **API** or the preload library, you need to tell the **XRootD POSIX** subsystem that you wish to cache data (i.e. configure caching).

This is done by setting the **XRDPOSIX_CONFIG** environmental variable to the path to your configuration file. For instance, we can augment the previous **runit** script to also specify the location of the configuration file that can be used to enable caching.

```
LD_PRELOAD= /usr/lib64/libXrdPosixPreload.so
XRDPOSIX_CONFIG=/home/abh/psx.cf
export LD_PRELOAD XRDPOSIX_CONFIG
$*
```

So, **/home/abh/psx.cf** contains the needed directives to setup local caching.

7.1 Server-less Disk Caching

To enable disk caching, you need to specify at least the following directives in your configuration file.

```
posix.cachelib /usr/lib64/libXrdPfc.so
oss.localroot cachepath
pfc.diskusage fracLow fracHigh
pfc.ram bytes[m|g]
```

The **cachelib** directive tells the system to use local disk caching using the **libXrdPfc.so** plug-in. You may need to change the path is this plug-in is installed in a non-standard directory.

The **localroot** directive specifies the disk location of the cache. The *cachepath* should be some existing directory path where cached file blocks can be permanently placed.

The **diskusage** directive specifies how much disk to use and at what point unused cached blocks should be deleted. Specify for *fracLow* the space utilization percentage that triggers purging. Specify for *fracHigh* the percentage of space utilization that must be reached for purging to stop. The default is “0.9 0.95” (i.e. 90% 95%) but a more reasonable combination might be “0.45 0.50” to prevent total use of your local disk. The **diskusage** boundaries, can be specified also in **g** or **t** bytes units for **g**iga- and **t**era-bytes, respectively. Since controlling the amount of disk space that the file cache can use is tricky, you may wish to use a separate disk partition for the cache.

Finally, the **ram** directive tells the disk cache the maximum amount of memory that it may use. The larger the amount the better will be the performance.

All of the “**pfc.**” prefixed directives are available for your use (see the Disk Caching Proxy Configuration section). A limited number of ”**pss.**” directives are also available but these directives must be prefixed with “**posix**.” in order to be recognized (see the section on esoteric directives). Since the disk caching subsystem uses the standard storage system plug-in to handle the disk cache, you can also use most of the “**oss.**” prefixed directives documented in the “Open File System & Open Storage System Configuration Reference”. However, it’s best to ignore the “**pss.**” and “**oss.**” directives unless you have very special needs.

7.2 Server-less Memory Caching

Using memory caching is far simpler than using disk caching because there is less to configure. Your configuration file needs to only have one directive, as shown below.

```
posix.cache [cachepots]
```

The *cacheopts* are identical to the ones described in the Memory Caching Proxy Configuration section using the **pss.cache** directive.

7.3 Esoteric Directives

The following table lists available directives that enable highly specialized features. For details on these options see the Proxy Configuration section.

POSIX Directive	Documented	Notes
posix.ccmlib	pss.ccmlib	Allows running a cache context manager.
posix.ciosync	pss.ciosync	Controls cache synchronization on close.
posix.inetmode	pss.inetmode	Controls TCP/IP stack usage.
posix.namelib	pss.namelib	Allows name mapping and cache squashing.
posix.setop	pss.setopt	Controls.underlying XRootD client.
posix.trace	pss.trace	Enables tracing.

7.4 Defining Virtual Mount Points

When using the **POSIX** preload library or the **XrdPosixExtern.hh** interface, you can define virtual mount points. A virtual mount point equates a path to an **XRootD URL**. Thus, any use of that path is internally converted to a specified **URL** using a template before being handed to the **XRootD POSIX** subsystem.

You specify virtual mounts by setting the **XROOTD_VMP** environmental variable to one or more templates that specify how to convert a path to a **URL**. The basic format is shown below.

server[**:***port*]**:***path*[**=**[*newpath*]] [. . .]

Where:

server is the DNS name or IP address of the target server that would have been specified in a URL.

port is the optional port number. If not specified, 1094 is assumed

path is the path prefix that is associated with *server*. It must start with a slash. Any specified path whose prefix matches *path causes the specified path* to be prefixed with "**xroot://***server*[**:***port*]**/**" (i.e. an **XRootD** URL) and then vectored to the server. This subject to what has been specified for the remainder of the template.

= when a simple equals sign follows path, The path prefix (i.e. path) is removed from the specified path before prefixing it with an **XRootD** URL.

newpath

when specified after the equals sign, *newpath* replaces *path* before prefixing it with an **XRootD** URL.

You may specify any number of virtual mount point templates in the environmental variable. Each one must be separated by a single space.

7.4.1 Examples

XROOTD_VMP=xnode:/xrootd/

/xrootd/home/abh is converted to **xroot://xnode:1094//xrootd/home/abh**

XROOTD_VMP=xnode:2094:/xrootd/

/xrootd/home/abh is converted to **xroot://xnode:2094//xrootd/home/abh**

XROOTD_VMP=xnode:/xrootd/=/

/xrootd/home/abh is converted to **xroot://xnode:1094//home/abh**

XROOTD_VMP=xnode:2094:/xrootd/=/atlas/

/xrootd/home/abh is converted to **xroot://xnode:1094//atlas/home/abh**

8 The netchk Utility

The **netchk** utility allows you to verify that an external client has a logical path to all of the required nodes in proxy configuration. The syntax is:

```
netchk { ssh | tcp } dest

dest:   [user@]host:port [dest]
```

Parameters

ssh only checks whether it is possible to ssh login through the *dest* host list.

tcp verifies that there is end-to-end message transitivity from the starting point through the *dest* host list.

user the username to use when doing an **ssh** login to each host in *dest*. The default is to use the current username.

host is a hostname or IP address that must be reachable. The first *host* in the list must be reachable from the starting node where the **netchk** command is issued. Each subsequent *host* in the *dest* list must be reachable from the previous *host* in the *dest* list.

port the port number where messages must be sent to. This is typically the port number for connects. This is only meaningful with **tcp** and the **ssh** parameter ignores the *port* specification.

Notes

1) The starting node must have **netchk** available. No other node need have **netchk** installed as the utility automatically but temporarily propagates itself across all of the hosts in *dest*.
2) All nodes, including the issuing node, must have **perl** installed along with the **IO::Socket** and **IPC::Open2** packages.
3) All hosts in *dest* must have **ssh** installed and the command issuer (or specified *user*) must have login access to each host in *dest*.
4) The **netchk** utility can be found in the **utils** directory of the **XRootD** distribution or in the add-ons section of the **xrootd.org** main page.

5) You should run **netchk** with the **ssh** parameter first to make sure there is a clear ssh login path to all hosts in *dest*.
6) Since the goal of **netchk** is to make sure there is message transitivity through all the listed hosts in *dest* using the ports that will be used by various daemons, the listed port must be available for use. This essentially means that the daemons that would normally use these ports must not be running on he hosts listed in *dest* when **netchk** is started.
7) A full transitivity test consists of trying paths between an external public node and all possible end-points.

Example

The following example tests whether there is appropriate transitivity between the **XRootD** proxy server to be run at z.domain.edu and the redirector to be run at x.domain.edu. Both servers will be using port 1024.

```
netchk tcp z.domain.edu:1094 x.domain.edu:1024
```

The test will make sure that the issuing node (which should be an external public node) can connect to z.domain.edu:1024 and that z.domain.edu:1024 can connect to x.domain.edu:1024. Two-way messages are sent across the connections to make sure they are not blocked.

9 Document Change History

16 Jun 2014

- Split description of proxy services from the ofs/oss manual. Hence, this is a new manual.
- Document the caching proxy service.

29 Jul 2014

- Document forwarding proxies.
- Document the **pss.origin** directive that configures a forwarding proxy.

5 Aug 2014

- Document the **pss.permit** directive that restricts outgoing connections of a forwarding proxy.

2 Dec 2014

- Document the disk caching proxy configuration directives (i.e. the ones that start with **pfc**).

14 Feb 2015

- Change the "**pfc.hdfs**" prefixed directives to "**pfc.file**" prefixed directives.
- Remove the **pfc.chachedir** directive.

11 Oct 2015

- Clarify the meaning of the high and low watermarks in the **pfc.diskusage** directive .

11 Jan 2017

- Document version 2 of the disk caching proxy which includes new and changed directives.

21 Mar 2017

- Correct **pfc.ram** directive (was incorrectly **pss.ram**).
- Document the **-lfncache** and **-lfn2pfn pss.namelib** directive options.

21 May 2017

- Describe automatic proxy selection via a client plug-in.
- Describe how disk caching proxy overloads may be handled.

26 June 2017

- Describe how to cache S3-type objects.

28 June 2017

- Describe the **pss.ciosync** directive.
- Document the **pss.config** directive's **stream** option.
- Document server-less caching
- Document **XRootD POSIX** interface.

18 September 2018

- Describe the enhanced **pfc.diskusage** directive.

16 May 2019

- Describe the **pfc.writequeue** directive.
- Describe the **pss.ccmlib** directive.
- Describe the **pss.dca** directive.
- Add section on Direct Cache Access.

21 January 2020

- Describe **URL** specification as the origin server using the **pss.origin** directive.

27 March 2020

- Document the **pss.persona** directive.
- Restructure the configuration section for readability.

18 May 2020

- Document that forwarding specification of the **pss.origin** directive may accept additional protocols that are to be considered acceptable.

2 June 2020

- Document the **pfc.acchistorysize**, **pfc.flush**, and the **-lfncachesrc** option of the **pss.namelib** directive.
- Expand security section and add privacy and integrity section.
- Simplify cluster configuration using the **cache export** attribute.

24 August 2020

- Correct documentation for the **pss.dca** directive.

31 August 2020

- Correct documentation for the **pfc.cschk** directive.

9 September 2020

- Correct documentation for the **pfc.cschk** directive and add a section describing data integrity options.

6 November 2020

- Describe the [**no**]**tls** option on the **pfc.cschk**.

21 June 2021

- Describe the **pss.reproxy**.
- Change the **pfc** default block size to 256K.
- Add description of the interaction between the **pfc.blocksize** and the **xrootd.async** directives.

7 June 2022

- Change the **pfc** default block size to 128K.