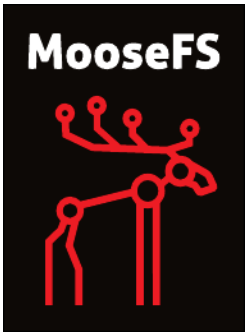


MooseFS 2.0 User's Manual

Core Technology Development & Support Team

April 7, 2016

 v. 1.2.9

Proofread by Agata Kruszona-Zawadzka
Coordination & layout by Piotr Robert Konopelko.

Please send corrections to Piotr Robert Konopelko – peter@mfs.io.

Contents

Chapter 1

About MooseFS

MooseFS is a fault-tolerant distributed file system. It spreads data over several physical locations (servers), which are visible to user as one resource. For standard file operations MooseFS acts as any other Unix-alike filesystem:

- Hierarchical structure (directory tree)
- Stores POSIX file attributes (permissions, last access and modification times)
- Supports special files (block and character devices, pipes and sockets)
- Symbolic links (file names pointing to target files, not necessarily on MooseFS) and hard links (different names of files that refer to the same data on MooseFS)
- Access to the file system can be limited based on IP address and/or password

Distinctive features of MooseFS are:

- High reliability (several copies of the data can be stored on separate physical machines)
- Capacity is dynamically expandable by adding new computers/disks
- Deleted files are retained for a configurable period of time (a file system level "trash bin")
- Coherent snapshots of files, even while the file is being written/accessed

1.1 Architecture

MooseFS consists of four components:

1. Managing servers (`master server`s) – In Community Edition one machine, in Pro any number of machines managing the whole filesystem, storing metadata for every file (information on size, attributes and file location(s), including all information about non-regular files, i.e. directories, sockets, pipes and devices).
2. Data servers (`chunk servers`) – any number of commodity servers storing files' data and synchronizing it among themselves (if a certain file is supposed to exist in more than one copy).

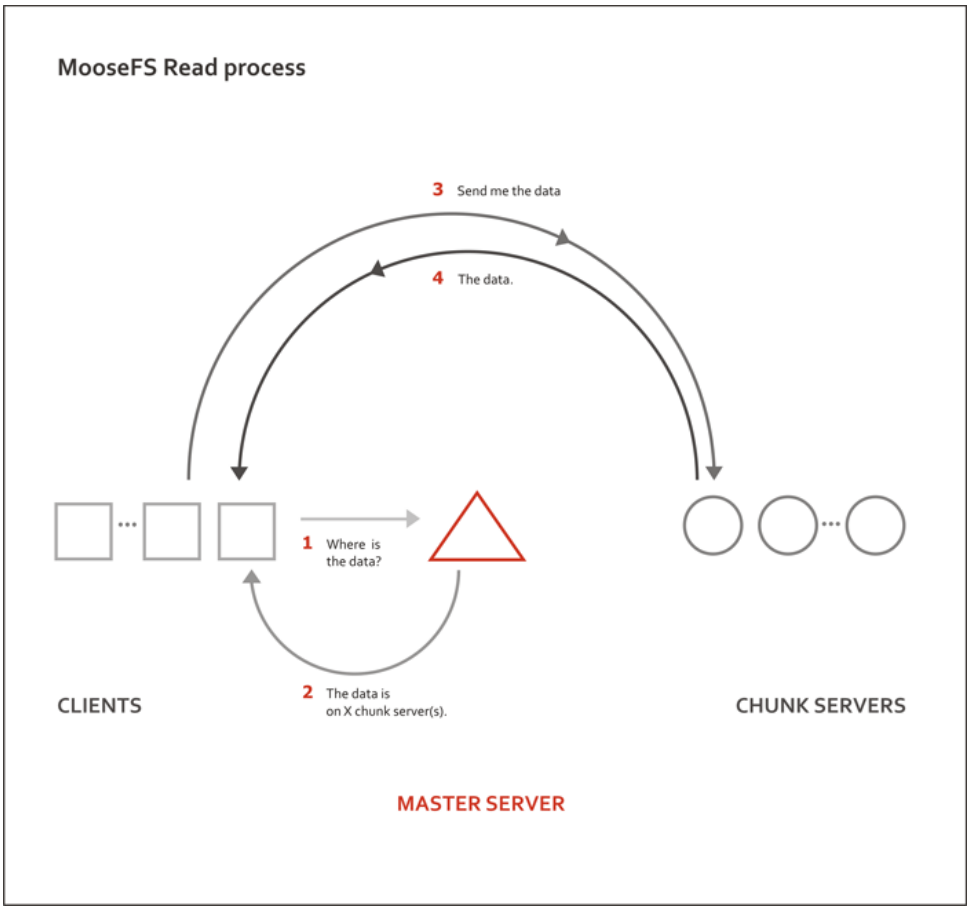
3. Metadata backup server(s) (`metalogger server`) – any number of servers, all of which store metadata changelogs and periodically download main metadata file.

   In CE version machine with metalogger can be easily set up as a master in case of main master failure.

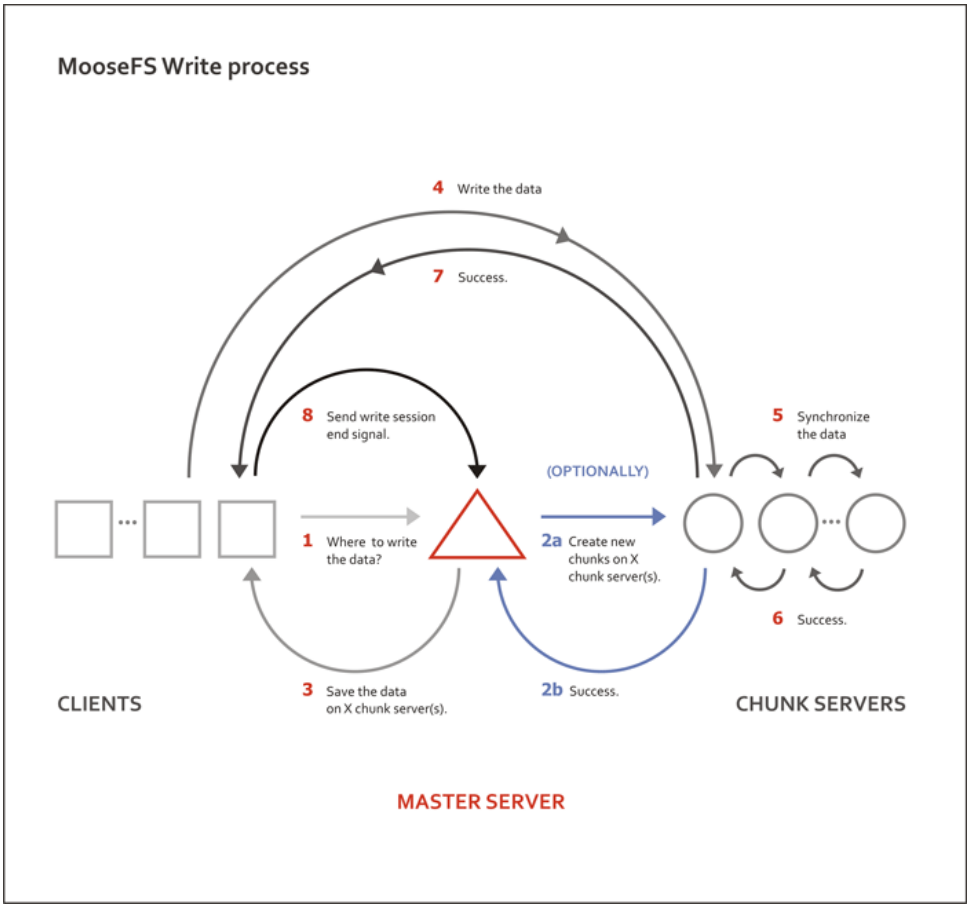
   In Pro version metalogger can be set up to provide an additional level of security.

4. Client computers that access (`mount`) the files in MooseFS – any number of machines using `mfsmount` process to communicate with the managing server (to receive and modify file metadata) and with chunkservers (to exchange actual file data).

`mfsmount` is based on the FUSE[1] mechanism (Filesystem in USErspace), so MooseFS is available on every Operating System with a working FUSE implementation (Linux, FreeBSD, MacOS X, etc.)



[1]You can read more about FUSE at `http://fuse.sourceforge.net`



Metadata is stored in the memory of the managing server and simultaneously saved to disk (as a periodically updated binary file and immediately updated incremental logs). The main binary file as well as the logs are synchronized to the metaloggers (if present) and to spare master servers in Pro version.

File data is divided into fragments (chunks) with a maximum size of 64MiB each. Each chunk is itself a file on selected disks on data servers (chunkservers).

High reliability is achieved by configuring as many different data servers as appropriate to assure the "`goal`" value (number of copies to keep) set for the given file.

1.2 How does the system work

All file operations on a client computer that has mounted MooseFS are exactly the same as they would be with other file systems. The operating system's kernel transfers all file operations to the FUSE module, which communicates with the `mfsmount` process. The `mfsmount` process communicates through the network subsequently with the managing server and data servers (chunk servers). This entire process is fully transparent to the user.

`mfsmount` communicates with the managing server every time an operation on file metadata is

required:

- creating files
- deleting files
- reading directories
- reading and changing attributes
- changing file sizes
- at the start of reading or writing data
- on any access to special files on `MFSMETA`

`mfsmount` uses a direct connection to the data server (chunk server) that stores the relevant chunk of a file. When writing a file, after finishing the write process the managing server receives information from `mfsmount` to update a file's length and the last modification time.

Furthermore, data servers (chunk servers) communicate with each other to replicate data in order to achieve the appropriate number of copies of a file on different machines.

1.3 Fault tolerance

Administrative commands allow the system administrator to specify the "`goal`", or number of copies that should be maintained, on a per-directory or per-file level. Setting the goal to more than one and having more than one data server will provide fault tolerance. When the file data is stored in many copies (on more than one data server), the system is resistant to failures or temporary network outages of a single data server.

This of course does not refer to files with the "`goal`" set to 1, in which case the file will only exist on a single data server irrespective of how many data servers are deployed in the system.

Exceptionally important files may have their `goal` set to a number higher than two, which will allow these files to be resistant to a breakdown of more than one server at the same time.

In general the setting for the number of copies available should be one more than the anticipated number of inaccessible or out-of-order servers.

In the case where a single data server experiences a failure or disconnection from the network, the files stored within it that had at least two copies, will remain accessible from another data server. The data that is now 'under its goal' will be replicated on another accessible data server to again provide the required number of copies.

It should be noted that if the number of available servers is lower than the "`goal`" set for a given file, the required number of copies cannot be preserved. Similarly if there are the same number of servers as the currently set goal and if a data server has reached 100% of its capacity, it will be unable to hold a copy of a file that is now below its goal due to another data server going offline. In these cases a new data server should be connected to the system as soon as

possible in order to maintain the desired number of copies of the file.

A new data server can be connected to the system at any time. The new capacity will immediately become available for use to store new files or to hold replicated copies of files from other data servers.

Administrative utilities exist to query the status of the files within the file system to determine if any of the files are currently below their goal (set number of copies). This utility can also be used to alter the goal setting as required.

The data fragments stored in the chunks are versioned, so re-connecting a data server with older copy of data (i.e. if it had been offline for a period of time), will not cause the files to become incoherent. The data server will synchronize itself to hold the current versions of the chunks, where the obsolete chunks will be removed and the free space will be reallocated to hold the new chunks.

Failures of a client machine (that runs the `mfsmount` process) will have no influence on the coherence of the file system or on the other client's operations. In the worst case scenario the data that has not yet been sent from the failed client computer may be lost.

1.4 Platforms

MooseFS is available on every Operating System with a working FUSE implementation:

- Linux (Linux 2.6.14 and up have FUSE support included in the official kernel)
- FreeBSD
- OpenSolaris
- MacOS X

The master server, metalogger server and chunkservers can also be run on Solaris or Windows with Cygwin. Unfortunately without FUSE it won't be possible to mount the filesystem within these operating systems.

Chapter 2

Moose File System Requirements

2.1 Requirements for Master Servers

As the managing server (master) is a crucial element of MooseFS, it should be installed on a machine which guarantees high stability and access requirements which are adequate for the whole system. It is advisable to use a server with a redundant power supply, ECC memory, and disk array RAID1/RAID5/RAID10. The managing server OS has to be POSIX compliant (systems verified so far: Linux, FreeBSD, MacOS X and OpenSolaris).

2.1.1 RAM size

The most important factor in sizing requirements for the master machine is RAM, as the full file system structure is cached in RAM for speed. The master server should have approximately 350 MiB of RAM allocated to handle 1 million files.

2.1.2 HDD free space

The necessary size of HDD depends both on the number of files and chunks used (main metadata file) and on the number of operations made on the files (metadata changelog); for example the space of 20GiB is enough for storing information for 25 million files and for changelogs to be kept for up to 50 hours.

You can calculate the minimum amount of space we recommend using the following formula:

- `RAM` – amount of RAM
- `BACK_LOGS` – number of metadata change log files, default is 50 (from `/etc/mfs/mfsmaster.cfg`)
- `BACK_META_KEEP_PREVIOUS` – number of previous metadata files to be kept (default is 1) (also from `/etc/mfs/mfsmaster.cfg`)

The formula:

```
SPACE = RAM * (BACK_META_KEEP_PREVIOUS + 2) + 1 * (BACK_LOGS + 1) [GiB]
```

(If default values from `/etc/mfs/mfsmaster.cfg` are used, it is `RAM * 3 + 51 [GiB]`)

The value `1` (before multiplying by `BACK_LOGS + 1`) is an estimation of size used by one `changelog.[number].mfs` file. On highly loaded MooseFS instance it uses a bit less than 1 GB.

Example:
If you have 128 GiB of RAM, using the formula above, you should reserve for `/var/lib/mfs`:

128*3 + 51 = 384 + 51 = **435 GiB minimum**.

2.2 Requirements for Metalogger(s)

MooseFS metalogger simply gathers metadata backups from the MooseFS master server – so the hardware requirements are not higher than for the master server itself; it needs about the same disk space. Similarly to the master server – the OS has to be POSIX compliant (Linux, FreeBSD, Mac OS X, OpenSolaris, etc.).

If you would like to use the metalogger as a master server in case of the main master's failure, the metalogger machine should have at least the same amount of RAM and HDD as the main master server.

2.3 Requirements for Chunkservers

Chunkserver machines should have appropriate disk space (dedicated exclusively for MooseFS) and POSIX compliant OS (verified so far: Linux, FreeBSD, Mac OS X and OpenSolaris).

Minimal configuration should start from several gigabytes of storage space (only disks with more than 256 MB and chunkservers reporting more than 1 GB of total free space are accessible for new data).

2.4 Requirements for Clients (`mfsmount`)

`mfsmount` requires FUSE to work; FUSE is available on several operating systems: Linux, FreeBSD, OpenSolaris and MacOS X, with the following notes:

- In case of Linux a kernel module with API 7.8 or later is required (it can be checked with dmesg command – after loading kernel module there should be a line fuse init (API version 7.8)). It is available in fuse package 2.6.0 (or later) or in Linux kernel 2.6.20 (or later). Due to some minor bugs, the newer module is recommended (fuse 2.7.2 or Linux 2.6.24, although fuse 2.7.x standalone doesn't contain getattr/write race condition fix).
- In case of FreeBSD we recommed using fuse-freebsd[1], which is a successor to fuse4bsd.
- For MacOSX we recommend using OSXFUSE[2], which is a successor to MacFUSE and has been tested on MacOSX 10.6, 10.7, 10.8, 10.9 and 10.11.

[1] `https://github.com/glk/fuse-freebsd`
[2] `http://osxfuse.github.com`

Chapter 3

Installing MooseFS 2.0

This is a Very Quick Start Guide describing basic MooseFS 2.0 installation in configuration of two Master Servers and three Chunkservers.

Please note that complete installation process is described in "MooseFS 2.0 Step by Step Tutorial".

For the sake of this document, it's assumed that your machines have following IP addresses:

- Master servers: 192.168.1.1, 192.168.1.2
- Chunkservers: 192.168.1.101, 192.168.1.102 and 192.168.1.103
- Users' computers (clients): 192.168.2.x

In this tutorial it is assumed that you have MooseFS 2.0 Pro version. If you use Community Edition, please remove '`-pro`' from packages names.

In this tutorial it is also assumed that you have Ubuntu/Debian installed on your machines. If you have another distribution, please use appropriate package manager instead of `apt`.

3.1 Configuring DNS Server

Before you start installing MooseFS, you need to have working DNS. It's needed for MooseFS to work properly with several master servers, because DNS can resolve one host name as more than one IP address.
All IPs of machines which will be master servers must be included in DNS configuration file and resolved as "`mfsmaster`" (or any other selected name), e.g.:

Listing 3.1: DNS entries

```
mfsmaster   IN  A   192.168.1.1     ; address of first master server
mfsmaster   IN  A   192.168.1.2     ; address of second master server
```

More information about configuring DNS server is included in supplement to "MooseFS Step by Step Tutorial".

3.2 Adding repositories

Before installing MooseFS you need to add MooseFS Official Supported Repositories to your system.

3.2.1 Ubuntu / Debian

First, add the key:

Listing 3.2: Adding the repo key

```
# wget -O - http://ppa.moosefs.com/moosefs.key | apt-key add -
```

Then add the appropriate entry in `/etc/apt/sources.list`:

- For Ubuntu 14.04 Trusty:
  `deb http://ppa.moosefs.com/stable/apt/ubuntu/trusty trusty main`
- For Ubuntu 12.04 Precise:
  `deb http://ppa.moosefs.com/stable/apt/ubuntu/precise precise main`
- For Ubuntu 10.10 Maverick:
  `deb http://ppa.moosefs.com/stable/apt/ubuntu/maverick maverick main`
- For Debian 7.0 Wheezy:
  `deb http://ppa.moosefs.com/stable/apt/debian/wheezy wheezy main`
- For Debian 6.0 Squeeze:
  `deb http://ppa.moosefs.com/stable/apt/debian/squeeze squeeze main`
- For Debian 5.0 Lenny:
  `deb http://ppa.moosefs.com/stable/apt/debian/lenny lenny main`

After that do:
`# apt-get update`

3.2.2 RedHat / CentOS (EL6)

Red Hat 6 family OS use `SysV init` runlevel system to start processes. To use service command to start MooseFS processes use this steps to add SysV repository.
Add the appropriate key to package manager:

Listing 3.3: Adding the repo key

```
# curl "http://ppa.moosefs.com/yum/RPM-GPG-KEY-MooseFS" > /etc/pki/rpm-gpg/RPM-GPG-
    KEY-MooseFS
```

Next you need to add the repository entry to yum repo:

Listing 3.4: Adding the MooseFS repo

```
# curl "http://ppa.moosefs.com/stable/yum/rhsysv/MooseFS.repo" > /etc/yum.repos.d/
    MooseFS.repo
# yum update
```

3.2.3 RedHat / CentOS (EL7)

Red Hat 7 familiy OS use `systemd` Linux system and service manager to start processes. To use systemctl command to start MooseFS processes use this steps to add `systemd` repository.

Add the appropriate key to package manager:

Listing 3.5: Adding the repo key

```
# curl "http://ppa.moosefs.com/yum/RPM-GPG-KEY-MooseFS" > /etc/pki/rpm-gpg/RPM-GPG-
    KEY-MooseFS
```

Next you need to add the repository entry to yum repo:

Listing 3.6: Adding MooseFS repo

```
# curl "http://ppa.moosefs.com/stable/yum/rhsystemd/MooseFS.repo" > /etc/yum.repos.d
    /MooseFS.repo
# yum update
```

3.2.4 Apple MacOS X

It's possible to run all components of the system on Mac OS X systems, but most common scenario would be to run the client (`mfsmount`) that enables Mac OS X users to access resources available in MooseFS infrastructure.

In case of MacOS X – since there's no default package manager – we release `.pkg` files containing only binaries without any startup scripts, that normally are available in Linux packages.

To install MooseFS on Mac please follow these steps:

- download and install FUSE for Mac OS X package from `http://osxfuse.github.io`
- download and install MooseFS packages from `http://ppa.moosefs.com/stable/osx/`

You should be able to mount MooseFS filesystem in `/mnt/mfs` issuing the following command:

```
$ sudo mfsmount /mnt/mfs
```

If you've exported filesystem with additional options like password protection, you should include those options in `mfsmount` invocation as in documentation.

3.3 Differences in package names between Pro and CE version

The packages in MooseFS 2.0 Pro version are named according to following pattern:

- `moosefs-pro-master`
- `moosefs-pro-cli`
- `moosefs-pro-chunkserver`
- `moosefs-pro-metalogger`

- `moosefs-pro-client`

etc.

In MooseFS 2.0 Community Edition (CE) the packages are named according to the following pattern:

- `moosefs-master`
- `moosefs-cli`
- `moosefs-chunkserver`
- `moosefs-metalogger`
- `moosefs-client`

etc.

3.4 MooseFS Master Server(s) installation

Install package `moosefs-pro-master` by running the following command:

For Debian OS family:

```
# apt-get install moosefs-pro-master
```

For RedHat OS family:

```
# yum install moosefs-pro-master
```

Now, set MooseFS Master Server basic configuration:

```
# cd /etc/mfs
# cp mfsmaster.cfg.dist mfsmaster.cfg
# cp mfsexports.cfg.dist mfsexports.cfg
```

File `mfsexports.cfg` specifies which users' computers can mount the file system and with what privileges. For example, to specify that only machines addressed as `192.168.2.x` can use the whole structure of MooseFS resources (`/`) in read/write mode, in the first line which is not commented out change the asterisk (`*`) to `192.168.2.0/24`, so that you'll have:

```
192.168.2.0/24      /       rw,alldirs,maproot=0
```

Now, if you use MooseFS Pro, place proper `mfslicence.bin` file into `/etc/mfs` directory. This file **must** be available on **all** Master Servers.

At this point it is possible to run the MooseFS Master Server:

```
# mfsmaster start
```

If you use `SysV` init script manager, which is by default available in Debian, Ubuntu and RedHat 6 family operating systems, you can also start Master by issuing the following command:

```
# service moosefs-pro-master start
```

To start MooseFS Master Server with latest `systemd` Linux system and service manager, which is available in RedHat 7 family operating systems, use this command:

```
# systemctl start moosefs-pro-master.service
```

You need to repeat these steps on each machine intended for running MooseFS Master Server (in this example – on `192.168.1.1` and `192.168.1.2`).

You can also find more detailed description how to add Master Followers in **MooseFS Upgrade Guide - Chapter 6: Adding master follower(s) server(s) procedure** (Pro only).

3.5 MooseFS CGI Monitor, CGI Server and Command Line Interface installation

MooseFS CGI Monitor and MooseFS CGISERV can be installed on any machine, but good practice tells that it should be installed on every Master Server.

MooseFS Command Line Interface (CLI) tool allows you to see various information about MooseFS status. The `mfscli` with `-SIN` option displays basic info similar to the "Info" tab in CGI. To install CGI, CGISERV and CLI, use the following commands.

For Debian OS family:

```
# apt-get install moosefs-pro-cgi
# apt-get install moosefs-pro-cgiserv
# apt-get install moosefs-pro-cli
```

Set `MFSCGISERV_ENABLE` variable to `true` in file `/etc/default/mfs-cgiserv` to configure `mfscgiserv` autostart.

For RedHat OS family:

```
# yum install moosefs-pro-cgi
# yum install moosefs-pro-cgiserv
# yum install moosefs-pro-cli
```

Run MooseFS CGI Monitor with `SysV`:

```
# service moosefs-pro-cgiserv start
```

Run MooseFS CGI Monitor with `systemd`:

```
# systemctl start moosefs-pro-cgiserv.service
```

MooseFS CGI Monitor website should now be available at `http://192.168.1.1:9425` address(for the moment there would be no data about chunk servers).

3.6 Chunk servers installation

For Debian OS family:

```
# apt-get install moosefs-pro-chunkserver
```

For RedHat OS family:

```
# yum install moosefs-pro-chunkserver
```

Now you need to prepare basic configuration files for the `mfschunkserver`:

```
# cd /etc/mfs
# cp mfschunkserver.cfg.dist mfschunkserver.cfg
# cp mfshdd.cfg.dist mfshdd.cfg
```

In the `mfshdd.cfg` file you'll give locations in which you have mounted hard drives/partitions designed for the chunks of the system. It is recommended that they are used exclusively for the MooseFS – this is necessary to manage the free space properly. For example if you'll use `/mnt/mfschunks1` and `/mnt/mfschunks2` locations, add these two lines to `mfshdd.cfg` file:

```
/mnt/mfschunks1
/mnt/mfschunks2
```

Before you start chunkserver, make sure that the user `mfs` has rights to write in the mounted partitions (which is necessary to create a .lock file):

```
# chown -R mfs:mfs /mnt/mfschunks1
# chown -R mfs:mfs /mnt/mfschunks2
```

At this moment you are ready to start the chunk server:

For `SysV` init script system

```
# service moosefs-pro-chunkserver start
```

For `systemd` Linux system and service manager

```
# systemctl start moosefs-pro-chunkserver.service
```

You need to repeat these steps on each machine intended for running MooseFS Chunkserver (in this example – on `192.168.1.101`, `192.168.1.102` and `192.168.1.103`.

Now at `http://192.168.1.1:9425` full information about the system is available, including the master server and chunk servers.

3.7 MooseFS Clients installation

MooseFS client uses `FUSE` library. During installation process, your operating system also downloads and installs `FUSE` library if it is not installed.

Debian OS family:

```
# apt-get install moosefs-pro-client
```

RedHat OS family:

```
# yum install moosefs-pro-client
```

Let's assume that you want to mount the MooseFS share in a `/mnt/mfs` folder on a client's machine. Issue the following commands:

```
# mkdir -p /mnt/mfs
# mfsmount /mnt/mfs -H mfsmaster
```

Now after running the `df -h | grep mfs` command you should get information similar to this:

```
/storage/mfschunks/mfschunks1
    2.0G    69M     1.9G    4%  /mnt/mfschunks1
/storage/mfschunks/mfschunks2
    2.0G    69M     1.9G    4%  /mnt/mfschunks2
mfs#mfsmaster:9421
    3.2G    0       3.2G    0%  /mnt/mfs
```

You need to repeat these steps on each machine intended to be MooseFS 2.0 Client (in this example – on `192.168.2.x`.

3.8 Enabling MooseFS services during OS boot

Each operating system has it's own method to manage services start during boot. Below you can find a few examples of enabling MooseFS autostart in supported operating systems.

3.8.1 RedHat / Centos (EL6)

MooseFS Chunkserver:

To enable MooseFS Chunkserver autostart during OS boot, use `chkconfig` command like in example below:

```
chkconfig moosefs-chunkserver on
```

MooseFS Master Server:

To enable MooseFS Master Server autostart during OS boot, use `chkconfig` command like in example below:

```
chkconfig moosefs-master on
```

MooseFS Client:

To enable MooseFS Client automount during boot, first of all check if the `fuse` package is installed:

```
# rpm -qa | grep fuse
fuse-2.8.3-4.el6.x86_64
fuse-libs-2.8.3-4.el6.x86_64
```

If fuse and fuse-libs packages are installed, add similar entry to the following one in `/etc/fstab`:

```
mfsmount    /mnt/mfs    fuse    defaults,mfsmaster=mfsmaster.host,mfsport=9421    0
       0
```

If MooseFS Client has to be mounted on the same machine that MooseFS Master Server runs, please put the following `fstab` entry instead of the one listed above:

```
mfsmount    /mnt/mfs    fuse    defaults,mfsdelayedinit,mfsmaster=mfsmaster.host,
    mfsport=9421    0    0
```

3.8.2 RedHat / Centos (EL7)

In operating systems with `systemd`, use `systemctl` command to manage init processes at boot:

MooseFS Chunkserver:

To enable MooseFS Chunkserver autostart during OS boot:

```
systemctl enable moosefs-chunkserver.service
```

MooseFS Master Server:

To enable MooseFS Master Server autostart during OS boot:

```
systemctl enable moosefs-master.service
```

MooseFS Client:

To enable MooseFS Client automount during boot, first of all check if the `fuse` package is installed:

```
# rpm -qa | grep fuse
fuse-2.9.2-6.el7.x86_64
fuse-libs-2.9.2-6.el7.x86_64
```

If fuse and fuse-libs packages are installed, add similar entry to the following one in `/etc/fstab`:

```
mfsmount    /mnt/mfs    fuse    mfsmaster=mfsmaster.host,mfsport=9421    0    0
```

If MooseFS Client has to be mounted on the same machine that MooseFS Master Server runs, please put the following `fstab` entry instead of the one listed above:

```
mfsmount    /mnt/mfs    fuse    defaults,mfsdelayedinit,mfsmaster=mfsmaster.host,
    mfsport=9421    0    0
```

3.8.3 Debian / Ubuntu

This method works in Debian 6, Debian 7, Ubuntu 12, Ubuntu 14.

MooseFS Chunkserver:

To enable MooseFS Chunkserver autostart during OS boot, find `/etc/default/moosefs-chunkserver` file and change `MFSCHUNKSERVER_ENBLE` variable to `true`:

```
MFSCHUNKSERVER_ENABLE=true
```

MooseFS Master:

To enable MooseFS Master Server autostart during OS boot, edit `/etc/default/moosefs-master` file and change `MFSMASTER_ENBLE` variable to `true`:

```
MFSMASTER_ENABLE=true
```

MooseFS Client:

To enable MooseFS Client automount during boot, add similar entry to the following one in `/etc/fstab`:

```
mfsmount    /mnt/mfs    fuse    mfsmaster=mfsmaster.host,mfsport=9421    0    0
```

3.8.4 FreeBSD

MooseFS Chunkserver:

To enable MooseFS Chunkserver autostart during OS boot, add an entry to `/etc/rc.conf`:

```
mfschunkserver_enable="YES"
```

MooseFS Master:

To enable MooseFS Chunkserver autostart during OS boot, add entry to `/etc/rc.conf`:

```
mfsmaster_enable="YES"
```

MooseFS Client:

To enable MooseFS Client automount during boot, add entry in `/etc/fstab`:

```
mfsmount_magic /mnt/mfs moosefs rw,mfsmaster=mfsmaster,mountprog=/usr/local/bin/
    mfsmount,late 0 0
```

3.9 Basic MooseFS use

Create `folder1` in `/mnt/mfs`, in which you store files in one copy (setting `goal=1`):

```
mkdir -p /mnt/mfs/folder1
```

and `folder2`, in which you store files in two copies (setting `goal=2`):

```
mkdir -p /mnt/mfs/folder2
```

The number of copies for the folder is set with the `mfssetgoal -r` command:

```
# mfssetgoal -r 1 /mnt/mfs/folder1
/mnt/mfs/folder1:
inodes with goal changed:                0
inodes with goal not changed:            1
inodes with permission denied:           0

# mfssetgoal -r 2 /mnt/mfs/folder2
/mnt/mfs/folder2:
inodes with goal changed:                0
inodes with goal not changed:            1
inodes with permission denied:           0
```

3.10 Stopping MooseFS

In order to safely stop the MooseFS cluster you have to perform the following steps:

- Unmount the file system on all machines using umount command (in our example it would be: `umount /mnt/mfs`)
- Stop the Chunk Servers processes:
  For `SysV`: `service moosefs-pro-chunkserver stop`
  For `systemd`: `systemctl stop moosefs-pro-chunkserver.service`
- Stop the Master Server processes (starting from the FOLLOWER, you shuould stop the LEADER Master Server as the last one):
  For `SysV`: `service moosefs-pro-master stop`
  For `systemd`: `systemctl stop moosefs-pro-master.service`
- Stop the Metalogger process:
  For `SysV`: `service moosefs-pro-metalogger stop`
  For `systemd`: `systemctl stop moosefs-pro-metalogger.service`

Chapter 4

Troubleshooting

4.1 Metadata save

Sometimes MFS master server freezes during the metadata save. To overcome this problem you should change one setting in your system. On your master machines, you should enable overcommit memory setting by issuing the following command as root:

```
# echo 1 > /proc/sys/vm/overcommit_memory
```

To do it permanently, you can add the following line to your `/etc/sysctl.conf` file (it works only on Linux):

```
vm.overcommit_memory=1
```

More detail about the reasons for this behavior:
Master server performs a fork operation, effectively spawning another process to save metadata to disk. Theoretically, when you fork a process, the process memory is copied. In real life it is done the lazy way – the memory is marked, so that if any changes are to occur, a block with changes is copied as needed, but only then. Now, if you fork a process that has 180GB of memory in use, the system can "just do it", or check if it has 180GB of free memory and reserve it for the forked "child", and only then do it and, when it doesn't have enough memory, the fork operation fails – this is the case in Linux, so actually saving metadata is done in the main process, because fork operation failed.

This behavior differs between systems and even between distributions of one system.

It is safe to enable overcommit memory (the "just do it" way) with mfsmaster, because the forked process is short lived. It terminates as soon as it manages to save metadata, and during the time that it works, there are usually not that many changes to the main process' memory, so the amount of additional RAM needed is relatively small.

Alternatively, you can add huge (at least equal to the amount of physical RAM or even more) amounts of swap space on your master servers – then the fork should succeed, because it should always find the needed memory space in your swap.

4.2 Master metadata restore from metaloggers

MooseFS Community Edition have only one master, but can have several metaloggers deployed for backup. If for some reason you loose all metadata files and changelogs from master server you can use data form metalogger to restore your data. To start dealing with recovery first you need to transfer all data stored on metalogger in `/var/lib/mfs` to master metadata folder. Files on metalogger will have `_ml_` prefix prepended to the filenames. After all files are copied, you need to create `metadata.mfs` file from changelogs and `metadata.back` files. To do this we need to use the command `mfsmaster -a`. `Mfsmaster` starts to build new metadata file and starts `mfsmaster` process.

4.3 Maintenance mode

Maintenance mode in general is helpful when there is need for maintenance on Chunkserver(s), like Chunkserver package upgrade to a newer version, adding new HDD / replacing broken ones or system upgrade (and e.g. reboot).

Maintenance mode has been introduced, because in MooseFS 1.6, when there was need for maintenance on Chunkserver(s) and necessity to turn server(s) off, a lot of replications were being performed, because MooseFS had started to replicate all undergoal chunks from another available copy to fulfill the goal (it's one of MooseFS principals). Then, when it was back again – a lot of deletions were running, because of presence of overgoal chunks, created during replications. So a lot of unnecessary I/O operations.

By enabling maintenance mode before stopping Chunkserver(s) process(es) / turning machine(s) off or *post factum*, you can prevent MooseFS from replicating chunks from such turned off Chunkserver(s). **Note: Server(s) in maintenance mode must match currently off (disconnected) servers. If they don't match, all chunks are replicated.**

Additionally, MooseFS treats Chunkservers in maintenance mode as overloaded (no chunk creations, replications etc.). It means, that new chunks are not created on Chunkservers in maintenance mode. The reason of such behavior is because when you want to turn Chunkserver off / stop the Chunkserver process, at the moment of stopping, some I/O operations may go to this Chunkserver and when you just stop it, some write operations must be re-tried (because they haven't been finished on this stopped Chunkserver). When you turn maintenance mode on for specific Chunkserver a few seconds before stop, MooseFS will finish write operations and won't start a new ones on this Chunkserver.

Maintenance mode is designed to be a temporary state and it is not recommended to put Chunkservers in this mode for a long time.

You can enable or disable maintenance mode in CGI monitor by clicking "switch on / switch off" in "maintenance" column, or sending a command using:

- `mfscli -CM1/ip/port` – to switch maintenance mode on
- `mfscli -CM0/ip/port` – to switch maintenance mode off

4.4 Chunk replication priorities

In MooseFS 2.0 a few chunk replication classes and priorities have been introduced:

- Replication limit class 0 and class 1 – replication for data safety
- Replication limit class 2 and class 3 – equalization of used disk space

These classes and priorities are described below:

- Replication limit class 0 (for endangered chunks):
  - priority 0: 0 (chunk) copies on regular disks and 1 copy on disk marked for removal
  - priority 1: 1 copy on regular disks and 0 copies on disks marked for removal
- Replication limit class 1 (for undergoal chunks):
  - priority 2: 1 copy on regular disk and some copies on disks marked for removal
  - priority 3: >1 copy on regular disks and at least 1 copy on disks marked for removal
  - priority 4: just undergoal chunks ("goal" > "valid copies", no copies on disks marked for removal)
- Replication limit class 2: Rebalancing between chunkservers with disk space usage around arithmetic mean
- Replication limit class 3: Rebalancing between chunkserver with disk space usage strongly above or strongly below arithmetic mean (very low or very high disk space usage, e.g. when new chunkserver is added)

Chapter 5

MooseFS Tools

5.1 For MooseFS Master Server(s)

5.1.1 `mfsmaster`

`mfsmaster` – start, restart or stop Moose File System master process

SYNOPSIS

- `mfsmaster [-c CFGFILE] [-u] [-f] [-t LOCKTIMEOUT] [ACTION]`
- `mfsmaster -v`
- `mfsmaster -h`

DESCRIPTION

`mfsmaster` is the master program of Moose File System.

OPTIONS

- `-v` print version information and exit
- `-h` print usage information and exit
- `-c CFGFILE` specify alternative path of configuration file (default is mfs master.cfg in system configuration directory)
- `-u` log undefined configuration values (when default is assumed)
- `-f` run in foreground, don't daemonize
- `-t LOCKTIMEOUT` how long to wait for lockfile (in seconds; default is 60 sec onds)
- `ACTION` is the one of `start`, `stop`, `restart`, `reload`, `test` or `kill`. Default action is restart. The test action will yeld one of two responses: "`mfsmaster pid: PID`" or "`mfsmaster is not running`". The kill action will send a SIGKILL to the currently running master process. SIGHUP or reload action forces `mfsmaster` to reload all configuration files.

FILES

- `mfsmaster.cfg` configuration file for MooseFS master process; refer to mfsmaster.cfg(5) manual for defails
- `mfsexports.cfg` MooseFS access control file; refer to mfsexports.cfg(5) manual for defails
- `mfstopology.cfg` Network topology definitions; refer to mfstopology.cfg(5) manual for defails
- `.mfsmaster.lock` lock file of running MooseFS master process (created in data directory)
- `metadata.mfs`, `metadata.mfs.back` MooseFS filesystem metadata image (created in data directory)
- `changelog.*.mfs` MooseFS filesystem metadata change logs (created in data directory; merged into metadata.mfs once per hour)
- `data.stats` MooseFS master charts state (created in data directory)

5.1.2 `mfsmetarestore`

`mfsmetarestore` is a tool that replays MooseFS metadata change logs or dump MooseFS metadata image.

SYNOPSIS

- `mfsmetarestore -m OLDMETADATAFILE -o NEWMETADATAFILE [CHANGELOGFILE...]`
- `mfsmetarestore -m METADATAFILE`
- `mfsmetarestore -a [-d DIRECTORY]`
- `mfsmetarestore -v`
- `mfsmetarestore -?`

DESCRIPTION

When `mfsmetarestore` is called with both -m and -o options, it replays given `CHANGELOGFILE`s on `OLDMETADATAFILE` and writes result to `NEWMETADATAFILE`. Multiple change log files can be given. `mfsmetarestore` with just `-m METADATAFILE` option dumps MooseFS metadata image file in human readable form. `mfsmetarestore` called with `-a` option automatically performs all operations needed to merge change log files. Master data directory can be specified using `-d DIRECTORY` option.

- `-v` – print version information and exit
- `-?` – print version information and exit
- `-a` – autorestore mode (see above)
- `-d DATAPATH` – master data directory (for autorestore mode)
- `-m METADATAFILE` – specify input metadata image file
- `-o NEWMETADATAFILE` – specify output metadata image file

FILES

- `metadata.mfs` – Moose File System metadata image as read by `mfsmaster` process
- `metadata.mfs.back` – Moose File System metadata image as left by killed or crashed `mfsmaster` process
- `changelog.*.mfs` – Moose File System metadata change logs

5.1.3 `mfsmetadump`

`mfsmetadump` is a tool which dumps metadata to file.

USAGE

```
mfsmetadump metadata_file
```

5.2 For MooseFS Supervisor

5.2.1 `mfssupervisor`

`mfssupervisor` – choose or switch leader master

SYNOPSIS

- `mfssupervisor [-xdfi] [-l new leader ip] [-H master host name] [-P master supervising port]`
- `mfssupervisor -v`
- `mfssupervisor -h`

DESCRIPTION

`mfssupervisor` is the supervisor program of Moose File System. It is needed to start a completely new system or a system after a big crash. It can be also used to force select a new leader master.

OPTIONS

- `-v` – print version information and exit
- `-h` – print usage information and exit
- `-x` – produce more verbose output
- `-d` – dry run (print info, but do not change anything)
- `-f` – force electing not synchronized follower; use this option to initialize a new system
- `-i` – print info only about masters state

- `-l` – try to switch current leader to given ip
- `-H` – use given host to find your master servers (default: `mfsmaster`)
- `-P` – use given port to connect to your master servers (default: `9419`)

5.3 For MooseFS Command Line Interface

5.3.1 `mfs`

`mfscli` – monitoring MooseFS from command line

SYNOPSIS

- `/usr/bin/mfscli [-hpn28] [-H master_host] [-P master_port] [-f 0..3] -S(IN|LI|IG|MU|IC|IL|CS|ML|HD|EX|MS|MO|QU) [-o order_id [-r]] [-m mode_id]`
- `/usr/bin/mfscli [-hpn28] [-H master_host] [-P master_port] [-f 0..3] -C(RC/ip/port|BW/ip/port)`

OPTIONS:

- `-h` – print help
- `-p` – force plain text format on tty devices
- `-s separator` – field separator to use in plain text format on tty devices (forces -p)
- `-2` – force 256-color terminal color codes
- `-8` – force 8-color terminal color codes
- `-H` master_host – master address (default: mfsmaster)
- `-P` master_port – master client port (default: 9421)
- `-n` – do not resolve ip adresses (default when output device is not tty)
- `-f<0;3>` – set frame charset to be displayed as table frames in ttymode

MONITORING:

- `-S data set` – defines data set to be displayed
- `-SIN` – show full master info
- `-SIM` – show only masters states
- `-SIG` – show only general master (leader) info
- `-SLI` – show only licence info
- `-SIC` – show only chunks info (goal/copies matrices)
- `-SIL` – show only loop info (with messages)
- `-SCS` – show connected chunk servers

- `-SMB` – show connected metadata backup servers
- `-SHD` – show hdd data
- `-SEX` – show exports
- `-SMS` – show active mounts
- `-SMO` – show operation counters
- `-SQU` – show quota info
- `-o order_id` – sort data by column specified by 'order id' (depends on data set)
- `-r` – reverse order
- `-m mode_id` – show data specified by 'mode id' (depends on data set)

COMMANDS:

- `-C command` – perform particular command
- `-CRC/ip/port` – remove given chunkserver from list of active chunkservers
- `-CBW/ip/port` – send given chunkserver back to work (from grace state)
- `-CRS/sessionid` – remove given session

5.4 For MooseFS CGI Server

5.4.1 `mfscgiserv`

`mfscgiserv` – start HTTP/CGI server for Moose File System monitoring

SYNOPSIS

- `mfscgiserv [-H BIND_HOST] [-P BIND_PORT] [-R ROOT_PATH] [-t LOCKTIMEOUT] [-f [-v]] [ACTION]`
- `mfscgiserv -h`

DESCRIPTION

`mfscgiserv` is a very simple HTTP server capable of running CGI scripts for Moose File System monitoring.

OPTIONS

- `-h` – print usage information and exit
- `-H BIND_HOST` – local address to listen on (default: any)
- `-P BIND_PORT` – port to listen on (default: `9425`)
- `-R ROOT_PATH` – local path to use as HTTP document root (default is `CGIDIR` set up at configure time)

- `-f` –run in foreground, don't daemonize
- `-v` – log requests on stderr
- `-t LOCKTIMEOUT` – how long to wait for lockfile (in seconds; default is 60 seconds)

`ACTION` is one of `start`, `stop`, `restart` or `test`. Default action is `restart`. The `test` action will yeld one of two responses: "`mfscgiserv pid: PID`" or "`mfscgiserv is not running`".

5.5 For MooseFS Metalogger(s)

5.5.1 `mfsmetalogger`

`mfsmetalogger` – start, restart or stop Moose File System metalogger process

SYNOPSIS

- `mfsmetalogger [-f] [-c CFGFILE] [-u] [-d] [-t LOCKTIMEOUT] [ACTION]`
- `mfsmetalogger -s [-c CFGFILE]`
- `mfsmetalogger -v`
- `mfsmetalogger -h`

DESCRIPTION

`mfsmetalogger` is the metadata replication server of Moose File System. Depending on parameters it can start, restart or stop MooseFS metalogger process. Without any options it starts MooseFS metalogger, killing previously run process if lock file exists.

`SIGHUP` (or '`reload`' `ACTION`) forces `mfsmetalogger` to reload all configuration files.

`mfsmetalogger` exists since 1.6.5 version of MooseFS; before this version `mfschunkserver` was responsible of logging metadata changes.

- `-v` – print version information and exit
- `-h` – print usage information and exit
- `-f` – (**deprecated**, use start action instead) forcily run MooseFS metalogger process, without trying to kill previous instance (this option allows to run MooseFS metalogger if stale PID file exists)
- `-s` – (**deprecated**, use stop action instead) stop MooseFS metalogger process
- `-c CFGFILE` – specify alternative path of configuration file (default is `mfsmetalogger.cfg` in system configuration directory)
- `-u` – log undefined configuration values (when default is assumed)
- `-d` – run in foreground, don't daemonize
- `-t LOCKTIMEOUT` – how long to wait for lockfile (default is 60 seconds)

ACTION is the one of start, stop, restart, reload, test or kill.

Default action is restart unless -s (stop) or -f (start) option is given. Note that -s and -f options are **deprecated**, likely to disappear and ACTION parameter to become obligatory in MooseFS 2.0.

FILES

- mfsmetalogger.cfg – configuration file for MooseFS metalogger process
- mfsmetalogger.lock – PID file of running MooseFS metalogger process (created in RUN_PATH by MooseFS $< 1.6.9$)
- .mfsmetalogger.lock – lock file of running MooseFS metalogger process (created in data directory since MooseFS 1.6.9)
- changelog_ml.*.mfs – MooseFS filesystem metadata change logs (backup of master change log files)
- metadata.ml.mfs.back – Latest copy of complete metadata.mfs.back file from MooseFS master.
- sessions.ml.mfs – Latest copy of sessions.mfs file from MooseFS master.

5.6 For MooseFS Chunkserver(s)

5.6.1 mfschunkserver

mfschunkserver – start, restart or stop Moose File System chunkserver process

SYNOPSIS

- mfschunkserver [-c CFGFILE] [-u] [-f] [-t LOCKTIMEOUT] [ACTION]
- mfschunkserver -v
- mfschunkserver -h

DESCRIPTION

mfschunkserver is the data server of Moose File System.

OPTIONS

- -v – print version information and exit
- -h – print usage information and exit
- -c CFGFILE – specify alternative path of configuration file (default is mfschunkserver.cfg in system configuration directory)
- -u – log undefined configuration values (when default is assumed)
- -f – run in foreground, don't daemonize

- `-t LOCKTIMEOUT` – how long to wait for lockfile (in seconds; default is 60 seconds)

`ACTION` is the one of `start`, `stop`, `restart`, `reload`, `test` or `kill`. Default action is restart. The test action will yeld one of two responses: "`mfschunkserver pid: PID`" or "`mfschunkserver is not running`". The kill action will send a `SIGKILL` to the currently running chunkserver process. `SIGHUP` or reload action forces `mfschunkserver` to reload all configuration files.

FILES

- `mfschunkserver.cfg` – configuration file for MooseFS chunkserver process
- `mfshdd.cfg` – list of directories (mountpoints) used for MooseFS storage
- `.mfschunkserver.lock` – lock file of running MooseFS chunkserver process (created in data directory)
- `data.csstats` – chunkserver charts state (created in data directory)

5.7 For MooseFS Client

5.7.1 `mfsmount`

`mfsmount` – mount Moose File System

SYNOPSIS

- `mfsmount mountpoint [-d] [-f] [-s] [-m] [-n] [-p] [-H HOST] [-P PORT] [-S PATH] [-o opt[,opt]...]`
- `mfsmount -h|--help`
- `mfsmount -V|--version`

DESCRIPTION

Mount Moose File System.
General options:

- `-h`, `--help` – display help and exit
- `-V` – display version information and exit

FUSE options:

- `-d`, `-o debug` – enable debug mode (implies -f)
- `-f` – foreground operation
- `-s` – disable multi-threaded operation

MooseFS options:

- `-c CFGFILE, -o mfscfgfile=CFGFILE` – loads file with additional mount options
- `-m`, `--meta`, `-o mfsmeta` – mount MFSMETA companion filesystem instead of primary MooseFS
- `-n` – omit default mount options (-o allow_other,default_permissions)
- `-p` – prompt for password (interactive version of -o mfspassword=PASS)
- `-H HOST`, `-o mfsmaster=HOST` – connect with MooseFS master on HOST (default is mfsmaster)
- `-P PORT`, `-o mfsport=PORT` – connect with MooseFS master on PORT (default is 9421)
- `-B HOST`, `-o mfsbind=HOST` – local address to use for connecting with master instead of default one
- `-S PATH`, `-o mfssubfolder=PATH` – mount specified MooseFS directory (default is /, i.e. whole filesystem)
- `-o mfspassword=PASSWORD` – authenticate to MooseFS master with PASSWORD
- `-o mfsmd5pass=MD5` – authenticate to MooseFS master using directly given MD5 (only if mfspassword option is not specified)
- `-o mfsdebug` – print some MooseFS-specific debugging information
- `-o mfscachemode=CACHEMODE` – set cache mode (see DATA CACHE MODES; default is AUTO)
- `-o mfscachefiles` – (deprecated) preserve file data in cache (equivalent to '-o mfscachemode=YES')
- `-o mfsattrcacheto=SEC` – set attributes cache timeout in seconds (default: 1.0)
- `-o mfsentrycacheto=SEC` – set file entry cache timeout in seconds (default: 0.0, i.e. no cache)
- `-o mfsdirentrycacheto=SEC` – set directory entry cache timeout in seconds (default: 1.0)
- `-o mfswritecachesize=N` – specify write cache size in MiB (in range: 16..2048 - default: 250)
- `-o mfsrlimitnofile=N` – try to change limit of simultaneously opened file descriptors on startup (default: 100000)
- `-o mfsnice=LEVEL` – try to change nice level to specified value on startup (default: -19)
- `-o mfsioretries=N` – specify number of retiries before I/O error is returned (default: 30)

General mount options:

- `-o rw | -o ro` – Mount file-system in read-write (default) or read-only mode respectively.
- `-o suid | -o nosuid` – Enable or disable suid/sgid attributes to work.
- `-o dev | -o nodev` – Enable or disable character or block special device files interpretation.
- `-o exec | -o noexec` – Allow or disallow execution of binaries.

DATA CACHE MODES

There are three cache modes: `NO`, `YES` and `AUTO`. Default option is `AUTO` and you shuldn't change it unless you really know what you are doing. In AUTO mode data cache is managed automatically by `mfsmaster`.

- `NO`, `NONE` or `NEVER` never allow files data to be kept in cache (safest but can reduce efficiency)
- `YES` or `ALWAYS` always allow files data to be kept in cache (dangerous)
- `AUTO` file cache is managed by `mfsmaster` automatically (should be very safe and efficient)

5.7.2 `mfstools`

`mfstools` – perform MooseFS-specific operations

SYNOPSIS

- `mfsgetgoal [-r] [-n|-h|-H|-k|-m|-g] OBJECT...`
- `mfsrgetgoal [-n|-h|-H|-k|-m|-g] OBJECT...`
- `mfssetgoal [-r] [-n|-h|-H|-k|-m|-g] [+|-]N OBJECT...`
- `mfsrsetgoal [-n|-h|-H|-k|-m|-g] [+|-]N OBJECT...`
- `mfsgettrashtime [-r] [-n|-h|-H|-k|-m|-g] OBJECT...`
- `mfsrgettrashtime [-n|-h|-H|-k|-m|-g] OBJECT...`
- `mfssettrashtime [-r] [-n|-h|-H|-k|-m|-g] [+|-]SECONDS OBJECT...`
- `mfsrsettrashtime [-n|-h|-H|-k|-m|-g] [+|-]SECONDS OBJECT...`
- `mfsgeteattr [-r] [-n|-h|-H|-k|-m|-g] OBJECT...`
- `mfsseteattr [-r] [-n|-h|-H|-k|-m|-g] -f ATTRNAME [-f ATTRNAME ...] OBJECT...`
- `mfsdeleattr [-r] [-n|-h|-H|-k|-m|-g] -f ATTRNAME [-f ATTRNAME ...] OBJECT...`
- `mfscheckfile FILE...`
- `mfsfileinfo FILE...`
- `mfsdirinfo [-n|-h|-H|-k|-m|-g] OBJECT...`
- `mfsfilerepair [-n|-h|-H|-k|-m|-g] FILE...`
- `mfsappendchunks SNAPSHOT_FILE OBJECT...`
- `mfsmakesnapshot [-o] SOURCE... DESTINATION`
- `mfsgetquota [-n|-h|-H|-k|-m|-g] dirname [dirname ...]`
- `mfssetquota [-n|-h|-H|-k|-m|-g] [-i|-I inodes] [-l|-L length] [-s|-S size] [-r|-R realsize] dirname [dirname ...]`
- `mfsdelquota [-a|-A|-i|-I|-l|-L|-s|-S|-r|-R] [-n|-h|-H|-k|-m|-g] -f dirname [dirname ...]`

DESCRIPTION

- `mfsgetgoal` and `mfssetgoal` operate on object's goal value, i.e. the number of copies in which all file data are stored. It means that file should survive failure of one less chunkservers than its goal value. Goal must be set between `1` and `9` (note that `1` is strongly unadvised). `mfsgetgoal` prints current goal value of given object(s). `-r` option enables recursive mode, which works as usual for every given file, but for every given directory additionally prints current goal value of all contained objects (files and directories). `mfssetgoal` changes current goal value of given object(s). If new value is specified in `+N` form, goal value is increased to `N` for objects with lower goal value and unchanged for the rest. Similarly, if new value is specified as `-N`, goal value is decreased to `N` for objects with higher goal value and unchanged for the rest. `-r` option enables recursive mode. These tools can be used on any file, directory or deleted (trash) file.
- `mfsrgetgoal` and `mfsrsetgoal` are deprecated aliases for `mfsgetgoal -r` and `mfssetgoal -r` respectively.
- `mfsgettrashtime` and `mfssettrashtime` operate on object's trashtime value, i.e. the number of seconds the file is preserved in special trash directory before it's finally removed from filesystem. Trashtime must be non-negative integer value. mfsgettrashtime prints current trashtime value of given object(s). `-r` option enables recursive mode, which works as usual for every given file, but for every given directory additionally prints current trashtime value of all contained objects (files and directories). `mfssettrashtime` changes current trashtime value of given object(s). If new value is specified in `+N` form, trashtime value is increased to `N` for objects with lower trashtime value and unchanged for the rest. Similarly, if new value is specified as `-N`, trashtime value is decreased to `N` for objects with higher trashtime value and unchanged for the rest. `-r` option enables recursive mode. These tools can be used on any file, directory or deleted (trash) file.
- `mfsrgettrashtime` and `mfsrsettrashtime` are deprecated aliases for `mfsgettrashtime -r` and `mfssettrashtime -r` respectively.
- `mfsgeteattr`, `mfsseteattr` and `mfsdeleattr` tools are used to get, set or delete some extra attributes. Attributes are described below.
- `mfscheckfile` checks and prints number of chunks and number of chunk copies belonging to specified file(s). It can be used on any file, included deleted (trash).
- `mfsfileinfo` prints location (chunkserver host and port) of each chunk copy belonging to specified file(s). It can be used on any file, included deleted (trash).
- `mfsdirinfo` is extended, MooseFS-specific equivalent of `du -s` command. It prints summary for each specified object (single file or directory tree).
- `mfsfilerepair` deals with broken files (those which cause I/O errors on read operations) to make them partially readable. In case of missing chunk it fills missing parts of file with zeros; in case of chunk version mismatch it sets chunk version known to `mfsmaster` to highest one found on chunkservers. Note: because in the second case content mismatch can occur in chunks with the same version, it's advised to make a copy (not a snapshot!) and delete original file after "repairing".
- `mfsappendchunks` (equivalent of `mfssnapshot` from MooseFS 1.5) appends a lazy copy

of specified file(s) to specified snapshot file ("lazy" means that creation of new chunks is delayed to the moment one copy is modified). If multiple files are given, they are merged into one target file in the way that each file begins at chunk (64MB) boundary; padding space is left empty.

- `mfsmakesnapshot` makes a "real" snapshot (lazy copy, like in case of `mfsappendchunks`) of some object(s) or subtree (similarly to `cp -r command`). It's atomic with respect to each `SOURCE` argument separately. If `DESTINATION` points to already existing file, error will be reported unless `-o` (overwrite) option is given. Note: if `SOURCE` is a directory, it's copied as a whole; but if it's followed by trailing slash, only directory content is copied.
- `mfsgetquota`, `mfssetquota` and `mfsdelquota` tools are used to check, define and delete quotas. Quota is set on a directory. It can be set in one of 4 ways: for number of inodes inside the directory (total sum of the subtree's inodes) with `-i`, `-I` options, for sum of (logical) file lengths with `-l`, `-L` options, for sum of chunk sizes (not considering goals) with `-s`, `-S` options and for physical hdd space (more or less chunk sizes multiplied by goal of each chunk) with `-r`, `-R` options. Small letters set soft quota, capital letters set hard quota. `-a` and `-A` options in `mfsdelquota` mean all kinds of quota. Quota behavior is described below.

GENERAL OPTIONS

Most of mfstools use `-n`, `-h`, `-H`, `-k`, `-m` and `-g` options to select format of printed numbers. `-n` causes to print exact numbers, `-h` uses binary prefixes (`Ki`, `Mi`, `Gi` as 2^{10}, 2^{20} etc.) while `-H` uses `SI` prefixes (`k`, `M`, `G` as 10^3, 10^6 etc.). `-k`, `-m` and `-g` show plain numbers respectivaly in kibis (binary kilo - 1024), mebis (binary mega - 1024^2) and gibis (binary giga - 1024^3). The same can be achieved by setting `MFSHRFORMAT` environment variable to: 0 (exact numbers), `1` or `h` (binary prefixes), `2` or `H` (`SI` prefixes), `3` or `h+` (exact numbers and binary prefixes), `4` or `H+` (exact numbers and SI prefixes). The default is to print just exact numbers.

EXTRA ATTRIBUTES

`noowner` – This flag means, that particular object belongs to current user (`uid` and `gid` are equal to `uid` and `gid` values of accessing process). Only `root` (`uid=0`) sees the real `uid` and `gid`.

`noattrcache` – This flag means, that standard file attributes such as `uid`, `gid`, `mode`, `length` and so on won't be stored in kernel cache. In MooseFS 1.5 this was the only behavior, and `mfsmount` always prevented attributes from being stored in kernel cache, but in MooseFS 1.6 attributes can be cached, so in very rare ocassions it could be useful to turn it off.

`noentrycache` – This flag is similar to above. It prevents directory entries from being cached in kernel.

QUOTAS

Quota is always set on a directory. Hard quota cannot be exceeded any time. Soft quota can be exceeded for a period of time (7 days). Once a quota is exceeded in a directory, user must go below the quota during the next 7 days. If not, the soft quota for this particular

directory starts to behave like a hard quota. The 7 days period is global and cannot currently be modified.

INHERITANCE

When new object is created in MooseFS, attributes such as `goal`, `trashtime` and extra attributes are inherited from parent directory. So if you set i.e. "`noowner`" attribute and `goal` to `3` in a directory then every new object created in this directory will have `goal` set to `3` and "`noowner`" flag set. A newly created object inherits always the current set of its parent's attributes. Changing a directory attribute does not affect its already created children. To change an attribute for a directory and all of its children use "`-r`" option.

Chapter 6

MooseFS Configuration Files

6.1 For MooseFS Master Server(s)

Warning: Configuration files on all Master Servers must be consistent!

6.1.1 `mfsmaster.cfg`

`mfsmaster.cfg` – main configuration file for `mfsmaster`

DESCRIPTION

The file `mfsmaster.cfg` contains configuration of MooseFS master process.

SYNTAX

Syntax is:

```
OPTION = VALUE
```

Lines starting with `#` character are ignored as comments.

OPTIONS

Configuration options:

- `WORKING_USER` user to run daemon as
- `WORKING_GROUP` group to run daemon as; optional value - if empty then default user group will be used
- `SYSLOG_IDENT` name of process to place in syslog messages; default is `mfsmaster`
- `LOCK_MEMORY` whether to perform mlockall() to avoid swapping out `mfsmaster` process; default is 0, i.e. no

- NICE_LEVEL nice level to run daemon with; default is -19; note: process must be started as root to increase priority, if setting of priority fails, process retains the nice level it started with
- FILE_UMASK set default umask for group and others (user has always 0); default is 027 – block write for group and block all for others
- DATA_PATH where to store metadata files and lock file
- EXPORTS_FILENAME alternate location/name of mfsmaster.cfg file
- TOPOLOGY_FILENAME alternate location/name of mfstopology.cfg file
- BACK_LOGS number of metadata change log files (default is 50)
- BACK_META_KEEP_PREVIOUS number of previous metadata files to be kept (default is 1)
- CHANGELOG_PRESERVE_SECONDS how many seconds of change logs have to be preserved in memory (default is 1800; this sets the minimum, actual number may be a bit bigger due to logs being kept in 5k blocks; zero disables extra logs storage)
- MATOML_LISTEN_HOST IP address to listen on for metalogger, masters and supervisors connections (* means any)
- MATOML_LISTEN_PORT port to listen on for metalogger, masters and supervisors connections
- MASTER_RECONNECTION_DELAY delay in seconds before next try to reconnect to master-leader if not connected (default is 5)
- MASTER_TIMEOUT timeout in seconds for master-leader connections (default is 10)
- BIND_HOST local address to use for connecting with master-leader (default is *, i.e. default local address)
- MATOCS_LISTEN_HOST IP address to listen on for chunkserver connections (* means any)
- MATOCS_LISTEN_PORT port to listen on for chunkserver connections
- REPLICATIONS_DELAY_INIT initial delay in seconds before starting replications (default is 300)
- REPLICATIONS_DELAY_DISCONNECT replication delay in seconds after chunkserver disconnection (default is 3600)
- CHUNKS_LOOP_MAX_CPS Chunks loop shouldn't check more chunks per seconds than given number (default is 100000)
- CHUNKS_LOOP_MIN_TIME Chunks loop shouldn't be done in less seconds than given number (default is 300)
- CHUNKS_SOFT_DEL_LIMIT Soft maximum number of chunks to delete on one chunkserver (default is 10)
- CHUNKS_HARD_DEL_LIMIT Hard maximum number of chunks to delete on one chunkserver (default is 25)
- CHUNKS_WRITE_REP_LIMIT Maximum number of chunks to replicate to one chunkserver (default is 2)

- CHUNKS_READ_REP_LIMIT Maximum number of chunks to replicate from one chunkserver (default is 10)
- CS_HEAVY_LOAD_THRESHOLD Threshold for chunkserver load. Whenever load reaches this threshold, chunkserver is switched into 'grace' mode (less operations)
- CS_HEAVY_LOAD_GRACE_PERIOD Defines how long chunkservers will remain in 'grace' mode (in seconds)
- ACCEPTABLE_DIFFERENCE Maximum difference between space usage of chunkservers (default is 0.01 = 1%)
- MATOCL_LISTEN_HOST IP address to listen on for client (mount) connections (* means any)
- MATOCL_LISTEN_PORT port to listen on for client (mount) connections
- SESSION_SUSTAIN_TIME How long to sustain a disconnected client session (in seconds; default is 86400 = 1 day)
- QUOTA_TIME_LIMIT Time limit in seconds for soft quota (default is 604800 = 7 days)

NOTES Chunks in master are tested in a loop. Speed (or frequency) is regulated by two options CHUNKS_LOOP_MIN_TIME and CHUNKS_LOOP_MAX_CPS. First defines minimal time between iterations of the loop and second defines maximal number of chunk tests per second. Typically at the beginning, when number of chunks is small, time is constant, regulated by CHUNK_LOOP_MIN_TIME, but when number of chunks becomes bigger then time of loop can increase according to CHUNKS_LOOP_MAX_CPS.

Example: CHUNKS_LOOP_MIN_TIME is set to 300, CHUNKS_LOOP_MAX_CPS is set to 100000 and there is 1000000 (one million) chunks in the system. 1000000/100000 = 10, which is less than 300, so one loop iteration will take 300 seconds. With 1000000000 (one billion) chunks the system needs 10000 seconds for one iteration of the loop.

Deletion limits are defined as 'soft' and 'hard' limit. When number of chunks to delete increases from loop to loop, current limit can be temporary increased above soft limit, but never above hard limit.

6.1.2 mfsexports.cfg

mfsexports.cfg – MooseFS access control for mfsmounts

DESCRIPTION

The file mfsexports.cfg contains MooseFS access list for mfsmount clients.

SYNTAX

Syntax is: ADDRESS DIRECTORY [OPTIONS]

Lines starting with # character are ignored as comments.

ADDRESS can be specified in several forms:

- `*` – all addresses
- `n.n.n.n` – single IP address
- `n.n.n.n/b` – IP class specified by network address and number of significant bits
- `n.n.n.n/m.m.m.m` – IP class specified by network address and mask
- `f.f.f.f-t.t.t.t` – IP range specified by from-to addresses (inclusive)

`DIRECTORY` can be `/` or path relative to MooseFS root; special value `.` means MFSMETA companion filesystem.

OPTIONS list:

- `ro`, `readonly` – export tree in read-only mode; this is default
- `rw`, `readwrite` – export tree in read-write mode
- `alldirs` – allows to mount any subdirectory of specified directory (similarly to NFS)
- `dynamicip` – allows reconnecting of already authenticated client from any IP address (the default is to check IP address on reconnect)
- `ignoregid` – disable testing of group access at `mfsmaster` level (it's still done at `mfsmount` level) - in this case "group" and "other" permissions are logically added; needed for supplementary groups to work (`mfsmaster` receives only user primary group information)
- `admin` – administrative privileges – currently: allow changing of quota values
- `maproot=USER[:GROUP]` – maps root (`uid=0`) accesses to given user and group (similarly to `maproot` option in NFS mounts); USER and GROUP can be given either as name or number; if no group is specified, USER's primary group is used. Names are resolved on `mfsmaster` side (see note below).
- `mapall=USER[:GROUP]` – like above but maps all non privileged users (`uid!=0`) accesses to given user and group (see notes below).
- `password=PASS`, `md5pass=MD5` – requires password authentication in order to access specified resource
- `minversion=VER` – rejects access from clients older than specified
- `mingoal=N`, `maxgoal=N` – specify range in which goal can be set by users
- `mintrashtime=TDUR`, `maxtrashtime=TDUR` – specify range in which trashtime can be set by users

Default options are:
`ro`, `maproot=999:999`, `mingoal=1`, `maxgoal=9`, `mintrashtime=0`, `maxtrashtime=4294967295`.

NOTES

`USER` and `GROUP` names (if not specified by explicit `uid`/`gid` number) are resolved on `mfsmaster` host.

TDUR can be specified as number without time unit (number of seconds) or combination of numbers with time units. Time units are: W,D,H,M,S. Order is important – less significant time units can't be defined before more significant time units. Time units are case insensitive.

Option mapall works in MooseFS in different way than in NFS, because MooseFS is using FUSE's "default_permissions" option. When mapall option is used, users see all objects with uid equal to mapped uid as their own and all other as root's objects. Similarly objects with gid equal to mapped gid are seen as objects with current user's primary group and all other objects as objects with group 0 (usually wheel). With mapall option set attribute cache in kernel is always turned off.

EXAMPLES

```
*                    /          ro
192.168.1.0/24       /          rw
192.168.1.0/24       /          rw,alldirs,maproot=0,password=passcode
10.0.0.0-10.0.0.5    /test      rw,maproot=nobody,password=test
10.1.0.0/255.255.0.0 /public    rw,mapall=1000:1000
10.2.0.0/16          /          rw,alldirs,maproot=0,mintrashtime=2h30m,maxtrashtime
    =2w
```

6.1.3 mfstopology.cfg

mfstopology.cfg – MooseFS network topology definitions

DESCRIPTION

The file mfstopology.cfg contains assignments of IP addresses into network locations (usually switch numbers). This file is optional. If your network has one switch or decreasing traffic between switches is not necessary then leave this file empty.

SYNTAX

Syntax is:
ADDRESS SWITCH-NUMBER

Lines starting with # character are ignored as comments.

ADDRESS can be specified in several forms:

- * – all addresses
- n.n.n.n – single IP address
- n.n.n.n/b – IP class specified by network address and bits number
- n.n.n.n/m.m.m.m – IP class specified by network address and mask
- f.f.f.f-t.t.t.t – IP range specified by from-to addresses (inclusive)

SWITCH-NUMBER can be specified as any positive 32-bit numer.

NOTES

If one IP belongs to more than one definition then last definition is used.

As for now distance between switches is constant. So distance between machines is calculated as: 0 when IP numbers are the same, 1 when IP numbers are different, but switch numbers are the same and 2 when switch numbers are different

Distances are used only to sort chunkservers during read and write operations. New chunks are still created randomly. Also rebalance routines do not take distances into account.

6.2 For MooseFS Metalogger(s)

6.2.1 `mfsmetalogger.cfg`

codemfsmetalogger.cfg – configuration file for mfsmetalogger

DESCRIPTION

The file mfsmetalogger.cfg contains configuration of MooseFS metalogger process.

SYNTAX

Syntax is:

`OPTION = VALUE`

Lines starting with `#` character are ignored as comments.

OPTIONS

Configuration options:

- `DATA_PATH` – where to store metadata files
- `LOCK_FILE` – (deprecated) daemon lock/pid file
- `WORKING_USER` – user to run daemon as
- `WORKING_GROUP` – group to run daemon as (optional – if empty then default user group will be used)
- `SYSLOG_IDENT` – name of process to place in syslog messages (default is mfsmetalogger)
- `LOCK_MEMORY` – whether to perform mlockall() to avoid swapping out mfsmetalogger process (default is 0, i.e. no)
- `NICE_LEVEL` – nice level to run daemon with (default is -19 if possible; note: process must be started as root to increase priority)
- `BACK_LOGS` – number of metadata change log files (default is 50)
- `BACK_META_KEEP_PREVIOUS` – number of previous metadata files to be kept (default is 3)
- `META_DOWNLOAD_FREQ` – metadata download frequency in hours (default is 24, at most `BACK_LOGS`/2)
- `MASTER_HOST` – address of MooseFS master host to connect with (default is mfs master)

- MASTER_PORT – number of MooseFS master port to connect with (default is 9420)
- MASTER_RECONNECTION_DELAY – delay in seconds before trying to reconnect to master after dis connection (default is 30)
- MASTER_TIMEOUT – timeout (in seconds) for master connections (default is 60)

6.3 For MooseFS Chunkservers

6.3.1 mfschunkserver.cfg

mfschunkserver.cfg – main configuration file for mfschunkserver

DESCRIPTION

The file mfschunkserver.cfg contains configuration of MooseFS chunkserver process.

SYNTAX

Syntax is:

OPTION = VALUE

Lines starting with # character are ignored as comments.

OPTIONS

Configuration options:

- WORKING_USER – user to run daemon as
- WORKING_GROUP – group to run daemon as; optional value – if empty then default user group will be used
- SYSLOG_IDENT – name of process to place in syslog messages; default is mfschunkserver
- LOCK_MEMORY – whether to perform mlockall() to avoid swapping out mfschunkserver process; default is 0, i.e. no
- NICE_LEVEL – nice level to run daemon with; default is -19; note: process must be started as root to increase priority, if setting of priority fails, process retains the nice level it started with
- FILE_UMASK – set default umask for group and others (user has always 0); default is 027 - block write for group and block all for others
- DATA_PATH – where to store daemon lock file
- HDD_CONF_FILENAME – alternate location/name of mfshdd.cfg file
- HDD_TEST_FREQ – chunk test period in seconds; default is 10

- HDD_LEAVE_SPACE_DEFAULT – how much space should be left unused on each hard drive; number format: [0-9]*(.[0-9]*)?([kMGTPE]|[KMGTPE]i)?B?; default is 256MiB; examples: 0.5GB, .5G, 2.56GiB, 1256M etc.
- BIND_HOST – local address to use for master connections; default is *, i.e. default local address
- MASTER_HOST – MooseFS master host, IP is allowed only in single-master installations; default is mfsmaster
- MASTER_PORT – MooseFS master command port; default is 9420
- MASTER_CONTROL_PORT – MooseFS master control port; default is 9419
- MASTER_TIMEOUT – timeout in seconds for master connections; default is 60
- MASTER_RECONNECTION_DELAY – delay in seconds before trying to reconnect to master after disconnection (default is 5)
- BIND_HOST – local address to use for connecting with master (default is *, i.e. default local address)
- CSSERV_LISTEN_HOST – IP address to listen on for client (mount) connections (* means any)
- CSSERV_LISTEN_PORT – port to listen on for client (mount) connections (default is 9422)
- CSSERV_TIMEOUT – timeout (in seconds) for client (mount) connections (default is 5)

6.3.2 mfshdd.cfg

mfshdd.cfg – list of MooseFS storage directories for mfschunkserver

DESCRIPTION

The file mfshdd.cfg contains list of directories (mountpoints) used for MooseFS storage.

SYNTAX

Syntax is: [*|<|>|~]PATH [SPACE LIMIT]

Lines starting with # character are ignored as comments.

Path can be prefixed with one or more special characters. These characters work as additional options:

- * means this directory (hard drive) is "marked for removal" and all data will be replicated to other hard drives, usually on other chunkservers
- < means that all data from this hard drive should be moved to other hard drives
- > means that all data from other hard drives should be moved to this hard drive
- ~ means that significant (more than 10% in less than minute) change of total blocks count will not mark this drive as damaged (useful for compressed filesystems)

`PATH` is path to the mounting point of storage directory, usually a single hard drive.

`SPACE LIMIT` is optional space limit, that allows to set one of two values: how much space should be left unused on this device or how much space is to be used on this device. Definition format: `[0-9]*(.[0-9]*)?([kMGTPE]|[KMGTPE]i)?B?`, positive value means how much space to use, negative value means how much space should be left unused.

Chapter 7

Frequently Asked Questions

7.1 What average write/read speeds can we expect?

Aside from common (for most filesystems) factors like: block size and type of access (sequential or random), in MooseFS the speeds depend also on hardware performance. Main factors are hard drives performance and network capacity and topology (network latency). The better the performance of the hard drives used and the better throughput of the network, the higher performance of the whole system.

7.2 Does the goal setting influence writing/reading speeds?

Generally speaking, it does not. In case of reading a file, goal higher than one may in some cases help speed up the reading operation, i. e. when two clients access a file with goal two or higher, they may perform the read operation on different copies, thus having all the available throughtput for themselves. But in average the goal setting does not alter the speed of the reading operation in any way.

Similarly, the writing speed is negligibly influenced by the goal setting. Writing with goal higher than two is done chain-like: the client send the data to one chunk server and the chunk server simultaneously reads, writes and sends the data to another chunk server (which may in turn send them to the next one, to fulfill the goal). This way the client's throughtput is not overburdened by sending more than one copy and all copies are written almost simultaneously. Our tests show that writing operation can use all available bandwidth on client's side in 1Gbps network.

7.3 Are concurrent read and write operations supported?

All read operations are parallel – there is no problem with concurrent reading of the same data by several clients at the same moment. Write operations are parallel, execpt operations on the same chunk (fragment of file), which are synchronized by Master server and therefore need to be sequential.

7.4 How much CPU/RAM resources are used?

In our environment (ca. 1 PiB total space, 36 million files, 6 million folders distributed on 38 million chunks on 100 machines) the usage of chunkserver CPU (by constant file transfer) is about 15-30% and chunkserver RAM usually consumes in between 100MiB and 1GiB (dependent on amount of chunks on each chunk server). The master server consumes about 50% of modern 3.3 GHz CPU (ca. 5000 file system operations per second, of which ca. 1500 are modifications) and 12GiB RAM. CPU load depends on amount of operations and RAM on the total number of files and folders, not the total size of the files themselves. The RAM usage is proportional to the number of entries in the file system because the master server process keeps the entire metadata in memory for performance. HHD usage on our master server is ca. 22 GB.

7.5 Is it possible to add/remove chunkservers and disks on the fly?

You can add/remove chunk servers on the fly. But keep in mind that it is not wise to disconnect a chunk server if this server contains the only copy of a chunk in the file system (the CGI monitor will mark these in orange). You can also disconnect (change) an individual hard drive. The scenario for this operation would be:

1. Mark the disk(s) for removal (see How to mark a disk for removal?)
2. Reload the chunkserver process
3. Wait for the replication (there should be no "undergoal" or "missing" chunks marked in yellow, orange or red in CGI monitor)
4. Stop the chunkserver process
5. Delete entry(ies) of the disconnected disk(s) in `mfshdd.cfg`
6. Stop the chunkserver machine
7. Remove hard drive(s)
8. Start the machine
9. Start the chunkserver process

If you have hotswap disk(s) you should follow these:

1. Mark the disk(s) for removal (see How to mark a disk for removal?)
2. Reload the chunkserver process
3. Wait for the replication (there should be no "undergoal" or "missing" chunks marked in yellow, orange or red in CGI monitor)
4. Delete entry(ies) of the disconnected disk(s) in `mfshdd.cfg`
5. Reload the chunkserver process
6. Unmount disk(s)
7. Remove hard drive(s)

If you follow the above steps, work of client computers won't be interrupted and the whole operation won't be noticed by MooseFS users.

7.6 How to mark a disk for removal?

When you want to mark a disk for removal from a chunkserver, you need to edit the chunkserver's `mfshdd.cfg` configuration file and put an asterisk '`*`' at the start of the line with the disk that is to be removed. For example, in this `mfshdd.cfg` we have marked "`/mnt/hdd`" for removal:

```
/mnt/hda
/mnt/hdb
/mnt/hdc
*/mnt/hdd
/mnt/hde
```

After changing the mfshdd.cfg you need to reload chunkserver (on Linux Debian/Ubuntu: `service moosefs-pro-chunkserver reload`).

Once the disk has been marked for removal and the chunkserver process has been restarted, the system will make an appropriate number of copies of the chunks stored on this disk, to maintain the required "goal" number of copies.

Finally, before the disk can be disconnected, you need to confirm there are no "undergoal" chunks on the other disks. This can be done using the CGI Monitor. In the "Info" tab select "Regular chunks state matrix" mode.

7.7 My experience with clustered filesystems is that metadata operations are quite slow. How did you resolve this problem?

During our research and development we also observed the problem of slow metadata operations. We decided to aleviate some of the speed issues by keeping the file system structure in RAM on the metadata server. This is why metadata server has increased memory requirements. The metadata is frequently flushed out to files on the master server.

Additionally, in Community Edition, the metadata logger server(s) also frequently receive updates to the metadata structure and write these to their file systems.

In Pro version metaloggers are optional, because master followers are keeping synchronised with leader master. They're also saving metadata to the hard disk.

7.8 When I perform `df -h` on a filesystem the results are different from what I would expect taking into account actual sizes of written files.

Every chunkserver sends its own disk usage increased by 256MB for each used partition/hdd, and the master sends a sum of these values to the client as total disk usage. If you have 3 chunkservers with 7 hdd each, your disk usage will be increased by 3*7*256MB (about 5GB).

The other reason for differences is, when you use disks exclusively for MooseFS on chunkservers `df` will show correct disk usage, but if you have other data on your MooseFS disks `df` will count your own files too.

If you want to see the actual space usage of your MooseFS files, use `mfsdirinfo` command.

7.9 Can I keep source code on MooseFS? Why do small files occupy more space than I would have expected?

The system was initially designed for keeping large amounts (like several thousands) of very big files (tens of gigabytes) and has a hard-coded chunk size of 64MiB and block size of 64KiB. Using a consistent block size helps improve the networking performance and efficiency, as all nodes in the system are able to work with a single 'bucket' size. That's why even a small file will occupy 64KiB plus additionally 4KiB of checksums and 1KiB for the header.

The issue regarding the occupied space of a small file stored inside a MooseFS chunk is really more significant, but in our opinion it is still negligible. Let's take 25 million files with a goal set to 2. Counting the storage overhead, this could create about 50 million 69 KiB chunks, that may not be completely utilized due to internal fragmentation (wherever the file size was less than the chunk size). So the overall wasted space for the 50 million chunks would be approximately 3.2TiB. By modern standards, this should not be a significant concern. A more typical, medium to large project with 100,000 small files would consume at most 13GiB of extra space due to block size of used file system.

So it is quite reasonable to store source code files on a MooseFS system, either for active use during development or for long term reliable storage or archival purposes.

Perhaps the larger factor to consider is the comfort of developing the code taking into account the performance of a network file system. When using MooseFS (or any other network based file system such as NFS, CIFS) for a project under active development, the network filesystem may not be able to perform file IO operations at the same speed as a directly attached regular hard drive would.

Some modern integrated development environments (IDE), such as Eclipse, make frequent IO requests on several small workspace metadata files. Running Eclipse with the workspace folder on a MooseFS file system (and again, with any other networked file system) will yield slightly slower user interface performance, than running Eclipse with the workspace on a local hard drive.

You may need to evaluate for yourself if using MooseFS for your working copy of active development within an IDE is right for you.

In a different example, using a typical text editor for source code editing and a version control system, such as Subversion, to check out project files into a MooseFS file system, does not typically resulting any performance degradation. The IO overhead of the network file system nature of MooseFS is offset by the larger IO latency of interacting with the remote Subversion repository. And the individual file operations (open, save) do not have any observable latencies when using simple text editors (outside of complicated IDE products).

A more likely situation would be to have the Subversion repository files hosted within a MooseFS file system, where the svnserver or Apache + mod_svn would service requests to the Subversion

repository and users would check out working sandboxes onto their local hard drives.

7.10 Do chunkservers and metadata server do their own checksumming?

Chunk servers do their own checksumming. Overhead is about 4B per a 64KiB block which is 4KiB per a 64MiB chunk. Metadata servers don't. We thought it would be CPU consuming. We recommend using ECC RAM modules.

7.11 What resources are required for the Master server?

The most important factor is RAM of `mfsmaster` machine, as the full file system structure is cached in RAM for speed. Besides RAM `mfsmaster` machine needs some space on HDD for main metadata file together with incremental logs. The size of the metadata file is dependent on the number of files (not on their sizes). The size of incremental logs depends on the number of operations per hour, but length (in hours) of this incremental log is configurable.

7.12 When I delete files or directories, the MooseFS free space size doesn't change. Why?

MooseFS does not immediately erase files on deletion, to allow you to revert the delete operation. Deleted files are kept in the trash bin for the configured amount of time (default: 24h / 86400 seconds) before they are deleted.

You can configure for how long files are kept in trash and empty the trash manually (to release the space).

You cant mount the trash e.g. in the following way: First of all, create the directory to mount `mfsmeta`

```
# mkdir /mnt/mfsmeta
```

Then, mount `mfsmeta` (like normally, but with `-m` parameter:

```
# mfsmount -H master.host.name -m /mnt/mfsmeta
```

or:

```
# mfsmount -H master.host.name -o mfsmeta /mnt/mfsmeta
```

Then, go into trash subdirectory:

```
# cd /mnt/mfsmeta/trash
```

If you use MooseFS 2, you can see the list of deleted files in this directory. If you use MooseFS 3.0, you can see 4096 sub-trashes in this directory (named `000 .. FFF`). The reason of divide the trash into sub-trashes is a huge amount of files in trash on big instances. In such case, commands like `ls` or even `find` are not able to functionate properly. So since MooseFS 3, deleted files are located inside these directories (sub-trashes). The best way to locate a file you are looking for is to use `find` command.

If you use MooseFS 3 and you want to see the old trash structure (because you e.g. don't have a lot of files in trash and you like old, simple structure), you should mount the trash with a specific `mfsflatrash` parameter, e.g.:

```
# mfsmount -H master.host.name -o mfsmeta,mfsflattrash /mnt/mfsmeta
```

You can delete files from trash on MooseFS 2, just issue `rm` command, e.g.:

```
# mkdir /mnt/mfsmeta
# mfsmount -H master.host.name -o mfsmeta /mnt/mfsmeta
# cd /mnt/mfsmeta/trash
# rm *
```

If you want to delete files from trash with old structure on MooseFS 3, just issue `rm` command like above, but firstly mount it with `mfsflattrash` parameter, e.g.:

```
# mkdir /mnt/mfsmeta
# mfsmount -H master.host.name -o mfsmeta,mfsflattrash /mnt/mfsmeta
# cd /mnt/mfsmeta/trash
# rm *
```

In case you want to delete files from trash on MooseFS 3 with new trash structure, you should combine `find` and `rm` commands together, e.g.:

```
# mkdir /mnt/mfsmeta
# mfsmount -H master.host.name -o mfsmeta /mnt/mfsmeta
# cd /mnt/mfsmeta/trash
# find . -type f -exec rm {} \;
```

The time of storing a deleted file can be verified by the `mfsgettrashtime` command and changed with `mfssettrashtime`.

7.13 When I added a third server as an extra chunkserver, it looked like the system started replicating data to the 3rd server even though the file goal was still set to 2.

Yes. Disk usage balancer uses chunks independently, so one file could be redistributed across all of your chunkservers.

7.14 Is MooseFS 64bit compatible?

Yes!

7.15 Can I modify the chunk size?

No. File data is divided into fragments (chunks) with a maximum of 64MiB each. The value of 64 MiB is hard coded into system so you cannot modify its size. We based the chunk size on real-world data and determined it was a very good compromise between number of chunks and speed of rebalancing / updating the filesystem. Of course if a file is smaller than 64 MiB it occupies less space.

In the systems we take care of, several file sizes significantly exceed 100GB with no noticable chunk size penalty.

7.16 How do I know if a file has been successfully written to MooseFS?

Let's briefly discuss the process of writing to the file system and what programming consequences this bears.

In all contemporary filesystems, files are written through a buffer (write cache). As a result, execution of the write command itself only transfers the data to a buffer (cache), with no actual writing taking place. Hence, a confirmed execution of the write command does not mean that the data has been correctly written on a disk. It is only with the invocation and completion of the fsync (or close) command that causes all data kept within the buffers (cache) to get physically written out. If an error occurs while such buffer-kept data is being written, it could cause the fsync (or close) command to return an error response.

The problem is that a vast majority of programmers do not test the close command status (which is generally a very common mistake). Consequently, a program writing data to a disk may "assume" that the data has been written correctly from a success response from the write command, while in actuality, it could have failed during the subsequent close command.

In network filesystems (like MooseFS), due to their nature, the amount of data "left over" in the buffers (cache) on average will be higher than in regular file systems. Therefore the amount of data processed during execution of the close or fsync command is often significant and if an error occurs while the data is being written [from the close or fsync command], this will be returned as an error during the execution of this command. Hence, before executing close, it is recommended (especially when using MooseFS) to perform an fsync operation after writing to a file and then checking the status of the result of the fsync operation. Then, for good measure, also check the return status of close as well.

NOTE! When stdio is used, the fflush function only executes the "write" command, so correct execution of fflush is not sufficient to be sure that all data has been written successfully – you should also check the status of fclose.

The above problem may occur when redirecting a standard output of a program to a file in shell. Bash (and many other programs) do not check the status of the close execution. So the syntax of "`application > outcome.txt`" type may wrap up successfully in shell, while in fact there has been an error in writing out the "`outcome.txt`" file. You are strongly advised to avoid using the above shell output redirection syntax when writing to a MooseFS mount point. If necessary, you can create a simple program that reads the standard input and writes everything to a chosen file, where this simple program would correctly employ the appropriate check of the result status from the fsync command. For example, "`application | mysaver outcome.txt`", where `mysaver` is the name of your writing program instead of `application > outcome.txt`.

Please note that the problem discussed above is in no way exceptional and does not stem directly from the characteristics of MooseFS itself. It may affect any system of files – network type systems are simply more prone to such difficulties. Technically speaking, the above recommendations should be followed at all times (also in cases where classic file systems are used).

7.17 Does MooseFS have file size limit?

The maximum file size limit in MooseFS 2.0 is 128 PiB.

The maximum file system size limit is 16 EiB.

7.18 Can I set up HTTP basic authentication for the mfscgiserv?

`mfscgiserv` is a very simple HTTP server written just to run the MooseFS CGI scripts. It does not support any additional features like HTTP authentication. However, the MooseFS CGI scripts may be served from another full-featured HTTP server with CGI support, such as lighttpd or Apache. When using a full-featured HTTP server such as Apache, you may also take advantage of features offered by other modules, such as HTTPS transport. Just place the CGI and its data files (`index.html`, `mfs.cgi`, `chart.cgi`, `mfs.css`, `acidtab.js`, `logomini.png`, `err.gif`) under chosen `DocumentRoot`. If you already have an HTTP server instance on a given host, you may optionally create a virtual host to allow access to the MooseFS CGI Monitor through a different hostname or port.

7.19 Can I run a mail server application on MooseFS? Mail server is a very busy application with a large number of small files – will I not lose any files?

You can run a mail server on MooseFS. You won't lose any files under a large system load. When the file system is busy, it will block until its operations are complete, which will just cause the mail server to slow down.

7.20 Are there any suggestions for the network, MTU or bandwidth?

We recommend using jumbo-frames[1] (MTU=9000). With a greater amount of chunkservers, switches should be connected through optical fiber or use aggregated links[2]. The network should be built upon 1Gb Ethernet. It is unlikely there would be any performance benefit from using 10Gb Ethernet, as the inherit latencies in the hard drives [on the chunk servers] would become the bottleneck of the entire system.

7.21 Does MooseFS support supplementary groups?

Yes.

[1] `https://en.wikipedia.org/wiki/Jumbo_frame`
[2] `https://en.wikipedia.org/wiki/Link_aggregation`

7.22 Does MooseFS support file locking?

Yes, since MooseFS 3.0.

7.23 Is it possible to assign IP addresses to chunk servers via DHCP?

Yes, but we highly recommend setting "DHCP reservations" based on MAC addresses.

7.24 Some of my chunkservers utilize 90% of space while others only 10%. Why does the rebalancing process take so long?

Our experiences from working in a production environment have shown that aggressive replication is not desirable, as it can substantially slow down the whole system. The overall performance of the system is more important than equal utilization of hard drives over all of the chunk servers. By default replication is configured to be a non-aggressive operation. At our environment normally it takes about 1 week for a new chunkserver to get to a standard hdd utilization. Aggressive replication would make the whole system considerably slow for several days.

Replication speeds can be adjusted on master server startup by setting these two options:

- `CHUNKS_WRITE_REP_LIMIT`
  Maximum number of chunks to replicate to one chunkserver (default is `2,1,1,4`).
  One number is equal to four same numbers separated by colons.
  - First limit is for endangered chunks (chunks with only one copy)
  - Second limit is for undergoal chunks (chunks with number of copies lower than specified goal)
  - Third limit is for rebalance between servers with space usage around arithmetic mean
  - Fourth limit is for rebalance between other servers (very low or very high space usage)

  Usually first number should be grater than or equal to second, second greater than or equal to third, and fourth greater than or equal to third (1st >= 2nd >= 3rd <= 4th)
- `CHUNKS_READ_REP_LIMIT`
  Maximum number of chunks to replicate from one chunkserver (default is `10,5,2,5`).
  One number is equal to four same numbers separated by colons. Limit groups are the same as in write limit, also relations between numbers should be the same as in write limits (1st >= 2nd >= 3rd <= 4th)

Tuning these in your environment requires some experiments.

7.25 I have metalogger running – should I make additional backup of the metadata file on the master server?

Yes, it is highly advisable to make additional backup of the metadata file. This provides a worst case recovery option if, for some reason, the metalogger data is not useable for restoring the master server (for example the metalogger server is also destroyed).

The master server flushes metadata kept in RAM to the `metadata.mfs.back` binary file every hour on the hour (xx:00). So a good time to copy the metadata file is every hour on the half hour (30 minutes after the dump). This would limit the amount of data loss to about 1.5h of data. Backing up the file can be done using any conventional method of copying the metadata file – cp, scp, rsync, etc.

After restoring the system based on this backed up metadata file the most recently created files will have been lost. Additionally files, that were appended to, would have their previous size, which they had at the time of the metadata backup. Files that were deleted would exist again. And files that were renamed or moved would be back to their previous names (and locations). But still you would have all of data for the files created in the X past years before the crash occurred.

In MooseFS Pro version, master followers flush metadata from RAM to the hard disk once an hour. The leader master downloads saved metadata from followers once a day.

7.26 I think one of my disks is slower / damaged. How should I find it?

In the CGI monitor go to the "Disks" tab and choose "switch to hour" in "I/O stats" column and sort the results by "write" in "max time" column. Now look for disks which have a significantly larger write time. You can also sort by the "fsync" column and look at the results. It is a good idea to find individual disks that are operating slower, as they may be a bottleneck to the system.

It might be helpful to create a test operation, that continuously copies some data to create enough load on the system for there to be observable statisics in the CGI monitor. On the "Disks" tab specify units of "minutes" instead of hours for the "I/O stats" column.

Once a "bad" disk has been discovered to replace it follow the usual operation of marking the disk for removal, and waiting until the color changes to indicate that all of the chunks stored on this disk have been replicated to achieve the sufficient goal settings.

7.27 How can I find the master server PID?

Issue the following command:

```
# mfsmaster status
```

7.28 Web interface shows there are some copies of chunks with goal 0. What does it mean?

This is a way to mark chunks belonging to the non-existing (i.e. deleted) files. Deleting a file is done asynchronously in MooseFS. First, a file is removed from metadata and its chunks are marked as unnecessary (`goal=0`). Later, the chunks are removed during an "idle" time. This is much more efficient than erasing everything at the exact moment the file was deleted.

Unnecessary chunks may also appear after a recovery of the master server, if they were created shortly before the failure and were not available in the restored metadata file.

7.29 Is every error message reported by `mfsmount` a serious problem?

No. `mfsmount` writes every failure encountered during communication with chunkservers to the syslog. Transient communication problems with the network might cause IO errors to be displayed, but this does not mean data loss or that `mfsmount` will return an error code to the application. Each operation is retried by the client (`mfsmount`) several times and only after the number of failures (reported as `try counter`) reaches a certain limit (typically 30), the error is returned to the application that data was not read/saved.

Of course, it is important to monitor these messages. When messages appear more often from one chunkserver than from the others, it may mean there are issues with this chunkserver – maybe hard drive is broken, maybe network card has some problems – check its charts, hard disk operation times, etc. in the CGI monitor.

Note: `XXXXXXXX` in examples below means IP address of chunkserver. In `mfsmount` version < 2.0.42 chunkserver IP is written in hexadecimal format. In `mfsmount` version >= 2.0.42 IP is "human-readable".

What does
`file: NNN, index: NNN, chunk: NNN, version: NNN - writeworker: connection with (XXXXXXXX:PPPP) was timed out (unfinished writes: Y; try counter: Z)`
message mean?

This means that Zth try to write the chunk was not successful and writing of Y blocks, sent to the chunkserver, was not confirmed. After reconnecting these blocks would be sent again for saving. The limit of trials is set by default to 30.

This message is for informational purposes and doesn't mean data loss.

What does
`file: NNN, index: NNN, chunk: NNN, version: NNN, cs: XXXXXXXX:PPPP - readblock error (try counter: Z)`
message mean?

This means that Zth try to read the chunk was not successful and system will try to read the block again. If value of Z equals 1 it is a transitory problem and you should not worry about it. The limit of trials is set by default to 30.

7.30 How do I verify that the MooseFS cluster is online? What happens with `mfsmount` when the master server goes down?

When the master server goes down while `mfsmount` is already running, `mfsmount` doesn't disconnect the mounted resource, and files awaiting to be saved would stay quite long in the queue while trying to reconnect to the master server. After a specified number of tries they eventually return EIO – "input/output error". On the other hand it is not possible to start `mfsmount` when the master server is offline.

There are several ways to make sure that the master server is online, we present a few of these below. Check if you can connect to the TCP port of the master server (e.g. socket connection test). In order to assure that a MooseFS resource is mounted it is enough to check the inode number – MooseFS root will always have inode equal to 1. For example if we have MooseFS installation in `/mnt/mfs` then `stat /mnt/mfs` command (in Linux) will show:

```
$ stat /mnt/mfs
  File: '/mnt/mfs'
  Size: xxxxxx      Blocks: xxx      IO Block: 4096   directory
Device: 13h/19d     Inode: 1         Links: xx
(...)
```

Additionaly `mfsmount` creates a virtual hidden file `.stats` in the root mounted folder. For example, to get the statistics of `mfsmount` when MooseFS is mounted we can `cat` this `.stats` file, eg.:

```
$ cat /mnt/mfs/.stats
fuse_ops.statfs: 241
fuse_ops.access: 0
fuse_ops.lookup-cached: 707553
fuse_ops.lookup: 603335
fuse_ops.getattr-cached: 24927
fuse_ops.getattr: 687750
fuse_ops.setattr: 24018
fuse_ops.mknod: 0
fuse_ops.unlink: 23083
fuse_ops.mkdir: 4
fuse_ops.rmdir: 1
fuse_ops.symlink: 3
fuse_ops.readlink: 454
fuse_ops.rename: 269
(...)
```

If you want to be sure that master server properly responds you need to try to read the goal of any object, e.g. of the root folder:

```
$ mfsgetgoal /mnt/mfs
/mnt/mfs: 2
```

If you get a proper goal of the root folder, you can be sure that the master server is up and running.