# Robinhood v2

# Temporary Filesystem Manager

# Configuration tutorial

Thomas LEIBOVICI
CEA/DAM

<thomas.leibovici@cea.fr>

V2.1.2

February 9, 2010

# Table of contents

The purpose of this document is to guide you for the first steps of Robinhood installation and configuration. We will first run a very simple instance of Robinhood for monitoring purpose only. Then we will configure it for purging files when disk space is missing. Finally we will deal with more advanced usage, by defining filesets and associating different purge policies to them.

# 1. Installation

## *1.1. Robinhood*

### Requirements

Before building Robinhood, make sure the following packages are installed on your system:

- mysql-devel
- lustre API library (if Robinhood is to be run on a Lustre filesystem): '/usr/include/liblustreapi.h' and '/usr/lib/liblustreapi.a' are installed by lustre rpm.

### Compilation

Retrieve Robinhood tarball from sourceforge: http://sourceforge.net/projects/robinhood

Unzip and untar the sources:

```
tar zxvf  robinhood-2.1.2.tar.gz
cd robinhood-2.1.2
```

Then, use the "configure" script to generate Makefiles:

- use the `--with-purpose=TMP_FS_MGR` option for using it as a temporary filesystem manager;
- set the prefix of installation path (default is /usr/local) with '`--prefix=<path>`'

```
./configure --with-purpose=TMP_FS_MGR --prefix=/usr
```

Other '`./configure`' options:

- If you want to disable Lustre specific features (getting stripe info, purge by OST…), use the '`--disable-lustre`' option.
- On Lustre 2.0-alpha5/6: a fork() in liblustreapi results in 'defunc' robinhood process when reading MDT changelogs. Use the '`--enable-llapi-fork-support`' option to avoid this.

Finally, build the RPM:

```
make rpm
```

A ready-to-install RPM is generated in the 'rpms/RPMS/<arch>' directory. The RPM is tagged with the lustre version it was built for.

The RPM includes:

- 'robinhood' and 'rh-report' binaries
- 'rh-config' configuration helper
- Configuration templates
- /etc/init.d/robinhood script

## *1.2. MySQL database*

Robinhood needs a MySQL database for storing its data. This database can run on a different machine than the Robinhood program.

### Requirements

Install *mysql* and *mysql-server* packages on the node where you want to run the database engine.

Start the database engine:

```
service mysqld start
```

### Creating Robinhood database

**With the helper script:**

To easily create robinhood database, you can use the 'rh-config' script. Run this script on the database host to check your system configuration and perform database creation steps:

```
# check database requirements:
rh-config precheck_db

# create the database:
rh-config create_db
```

Write the database password to a file with restricted access (root/600),
e.g. `/etc/robinhood.d/.dbpassword`

**or manually:**

Alternatively, if you want a better control on the database configuration and access rights, you can perform the following steps of your own:

- Create the database (one per filesystem) using the `mysqladmin` command:
  `mysqladmin create <robinhood_db_name>`

- Connect to the database:
  `mysql <robinhood_db_name>`

  Then execute the following commands in the MySQL session:

- GRANT USAGE ON *robinhood_db_name.* TO 'robinhood'@'%'*
  identified by 'password';
- GRANT ALL PRIVILEGES ON *robinhood_db_name.* TO 'robinhood'@'%'*
  identified by 'password';

- Refresh server access settings:
  `FLUSH PRIVILEGES ;`

- You can check user privileges using:
  `SHOW GRANTS FOR robinhood ;`

- For testing access to database, execute the following command on the machine where robinhood will be running :
  `mysql --user=`*`robinhood`* `--password=`*`password`* `--host=`*`db_host`*
  *`robinhood_db_name`*

  If the command is successful, a SQL shell is started. Else, you will get a 'permission denied' error.

At this time, the database schema is empty. Robinhood will automatically create it the first time it is launched.

# 2. Run your first Robinhood instance

Let's begin with a simple case: we want to monitor the content of `/tmp` ext3 filesystem, by scanning it periodically.

## *2.1. Configuration file*

We first need to write a very basic configuration file: after installing robinhood rpm, get the template file `/etc/robinhood.d/tmpfs/templates/tmp_fs_mgr_tuto.conf`.
It contains the minimal set of configuration variables:

```
General
{
    fs_path = "/tmp";
    fs_type = ext3;
}

Log
{
    log_file       = "/var/log/robinhood/tmp_fs.log";
    report_file    = "/var/log/robinhood/reports.log";
    alert_file     = "/var/log/robinhood/alerts.log";
}

ListManager
{
    MySQL
    {
        server = db_host;
        db = robinhood_test;
        user = robinhood;
```

```
        password_file = /etc/robinhood.d/.dbpassword;
    }
}
```

**General section:**
'fs_path' is the mount point of the filesystem we want to monitor.
'fs_type' is the type of filesystem (as returned by mount). This parameter is used for sanity checks.

**Log section:**
Make sure the log directory exists.
Note: you can also use special values 'stderr' or 'stdout' for log files, so you can directly read log messages in your terminal when testing your configuration.

**ListManager::MySQL section:**
This section is for configuring database access.
Set the host name of the database server (*server* parameter), the database name (*db* parameter), the database user (*user* parameter) and specify a file where you wrote the password for connecting to the database (*password_file* parameter).
/!\ Make sure the password file cannot be read by any user, by setting it a '600' mode for example.
If you don't care about security, you can directly specify the password in the configuration file, by setting the *password* parameter.
E.g.: `password = 'passw0rd' ;`

## 2.2. Running robinhood

For this first example, we just want to scan the filesystem once and exit, so we can get stats about current content of /tmp.
Thus, we are going to run robinhood command with the '--scan' option, and the '--once' option so it will exit when the scan is finished.
You can specify the configuration file using the '–f' option. By default, it uses the first file it finds in the '`/etc/robinhood.d/tmpfs`' directory.
If you want to override configuration values for log file, use the '–L' option. For example, let's use '-L stdout'

```
robinhood –f /etc/robinhood.d/tmpfs/test.conf –L stdout --scan –once
```
or just:
```
robinhood –L stdout --scan –once
```
(if your config file is the only one in /etc/robinhood.d/tmpfs)

That's it!
You should get something like this:

```
2009/07/17 13:49:06: FS Scan | Starting scan of /tmp
2009/07/17 13:49:06: FS Scan | Full scan of /tmp completed, 7130 entries
found. Duration = 0.07s
2009/07/17 13:49:06: FS Scan | File list of /tmp has been updated
2009/07/17 13:49:06: Main | All tasks done! Exiting.
```

## 2.3. Getting stats on filesystem content

Now we performed a scan, we can get stats about users, files, directories, etc. using the `rh-report` command.

- Getting stats about user ‘foo’:

```
rh-report –u foo
```

```
User:                foo

    Type:               directory
    Count:                     90
    Space used:         720.00 KB    (1440 blks)
    Dircount min:               0
    Dircount max:              54
    Dircount avg:               8

    Type:                    file
    Count:                    609
    Space used:          20.26 MB    (41496 blks)
    Size min:                   8    (8 bytes)
    Size max:            1.05 MB    (1096625 bytes)
    Size avg:           27.21 KB    (27865 bytes)

    Type:                 symlink
    Count:                      9
    Space used:          36.00 KB    (72 blks)
    Size min:                  13    (13 bytes)
    Size max:                  22    (22 bytes)
    Size avg:                  17    (17 bytes)
```

- Getting largest files:

```
rh-report –-topsize
```

```
Rank:              1
Path:              /tmp/robinhood-2.1.1/rpms/BUILD/robinhood-2.1.1/src/Robinhood/rh-
report
Size:              1.05 MB   (1096625 bytes)
Last access:       2009/07/15 14:25:45
Last modification: 2009/07/15 14:25:44
Owner/Group:       foo/grp1

Rank:              2
Path:              /tmp/robinhood-2.1.1/rpms/BUILD/robinhood-
2.1.1/src/Robinhood/robinhood
Size:              1.03 MB   (1080539 bytes)
Last access:       2009/07/15 14:25:44
Last modification: 2009/07/15 14:25:43
Owner/Group:       foo/grp2

...
```

- Getting top space consummers:

```
rh-report –f –-topusers
```

```
Rank:                       1
User:                     foo
Space used:          21.00 MB    (43008 blks)
Nb entries:               708
Size min:                   8    (8 bytes)
Size max:            1.05 MB    (1096625 bytes)
Size avg:           23.92 KB    (24489 bytes)

Rank:                       2
User:                    root
Space used:          32.00 KB    (64 blks)
```

```
Nb entries:                 4
Size min:                 112    (112 bytes)
Size max:            4.00 KB    (4096 bytes)
Size avg:            2.05 KB    (2104 bytes)

...
```

- …and more:
  you can also generate reports, or dump files per directory, per OST, etc…
  To get more details about available reports, run '`rh-report --help`'.

# 3. Scan options and alerts

In the first example, we have used Robinhood as a 'one-shot' command. It can also be used as a daemon that will periodically scan the filesystem for updating its database. You can also configure alerts when entries in the filesystem match a condition you specified.

Let's configure Robinhood to scan the filesystem once a day.
Add a FS_Scan section to the config file:

```
FS_Scan
{
    min_scan_interval      =   1d ;
    max_scan_interval      =   1d ;

    nb_threads_scan        =     2 ;

    Ignore
    {
        # ignore ".snapshot" and ".snapdir" directories (don't scan them)
        type == directory
        and
        ( name == ".snapdir" or name == ".snapshot" )
    }

    Ignore
    {
        # ignore the content of /tmp/dir1
        tree == "/tmp/dir1"
    }
}
```

It is possible to have a dynamic scan interval, depending on filesystem usage. For this example, we want a constant scan interval so we set `min_scan_interval` and `max_scan_interval` to 1 day (`1d`).

Robinhood uses a parallel algorithm for scanning large filesystems efficiently. You can set the number of threads used for scanning using the `nb_threads_scan` parameter.

You may need to ignore some parts of the filesystem (like .snapshot dirs etc…). For this, you can use a "ignore" sub-block, like in the example above, and specify complex Boolean expressions on file properties (refer to Robinhood admin guide for more details about available attributes).

To create alerts about filesystem entries, add an EntryProcessor section to configuration file:

```
EntryProcessor
{
    Alert
    {
        type == directory
        and
        dircount > 10000
    }

    # Raise alerts for large files
    Alert
    {
        type == file
        and
        size > 100GB
    }
}
```

In this example, we want to raise alerts for directories with more that 10.000 entries and file larger than 100GB.
Those alerts are written to the `alert_file` specified in Log section. They can also be sent by mail by specifying a mail address in the `alert_mail` parameter of Log section.

Now, let's run Robinhood as a daemon. Use the '--detach' option to start it in background and detach it from your terminal:

```
robinhood --scan --detach
```

# 4. Resource monitoring and purges

One of the main purpose of robinhood is to monitor disk space usage and trigger purges when disk space runs low.
Purge triggering is based on high/low watermarks and files removal is based on a LRU list: when the used space reaches a high watermark, Robinhood will build a list of least recently accessed files (from its database) and purge them until the disk space is back to the low watermark.

## *4.1. Defining purge triggers*

First of all, let's specify watermarks levels for triggering purge.
This can be done on several criteria:

- Global filesystem usage
- OST usage (for Lustre filesystems)
- User usage (quota-like purge)

Write the following block to configuration file for triggering purge on global filesystem usage:

```
Purge_Trigger
{
    trigger_on         = global_usage ;
```

```
    high_watermark_pct = 90% ;
    low_watermark_pct  = 85% ;
    check_interval     = 5min ;
}
```

*trigger_on* specifies the type of trigger.
*high_watermark_pct* indicates the disk usage that must be reached for starting purge.
*low_watermark_pct* indicates the disk usage that must be reached for stopping purge.
*check_interval* is the interval for checking disk usage.

We can also do the same for users, be specifying a kind of “quota”:

```
Purge_Trigger
{
    trigger_on         = user_usage ;
    high_watermark_vol = 10GB ;
    low_watermark_vol  = 9GB ;
    check_interval     = 12h ;
}
```

Every 12h, the daemon will check the space used by users. If a user uses more than 10GB, its files will be purged from the least recently accessed until the space he uses decrease to 9GB.

## 4.2. Minimal purge policy

Robinhood makes it possible to define different purge policies for several file classes.
In this example, we will only define a single purge policy for all files.
This can be done in a ‘Purge_Policies’ section of config file:

```
Purge_policies
{
      Policy default
      {
            Condition
            {
                  last_access > 1h
            }
      }
}
```

‘default’ policy is a special policy that applies to files that are not in a file class.
In a policy, you must specify a condition for allowing entries to be purged. In this example, we don’t want recently accessed entries (read or written within the last hour) to be purged.

## 4.3. Running resource monitoring

Robinhood is now able to monitor disk usage and purge entries when needed.
Start the daemon with the ‘--purge’ option. If your are not sure of your configuration, you can specify the ‘--dry-run’ option, so it won’t really purge files.

```
robinhood --purge --detach –-dry-run
```

You will get something like this in the log file:

```
Main | Resource Monitor successfully initialized
ResMonitor | Filesystem usage: 92.45% (786901 blocks) / high watermark:
90.00% (764426 blocks)
ResMonitor | 34465 blocks (x512) must be purged on Filesystem (used=786901,
target=734426, block size=4096)
Purge | Building a purge list from last full FS Scan: 2009/07/17 13:49:06
Purge | Starting purge on global filesystem
ResMonitor | Global filesystem purge summary: 34465 blocks purged/34465
blocks needed in /tmp
```

The list of purged files is written in the report file.

Note 1: you can use the same daemon for performing scans periodically and monitoring resource, by combining options on command line.
E.g.: `robinhood --purge --scan –-detach`
By default, if you start Robinhood with no action, it will perform both.
Note 2: if you do not want to have a daemon on your system, you can perform resource monitoring with a 'one-shot' command that you can launch in cron, for example:
`robinhood --purge --once`
It will check the disk usage: it will exit immediately if disk usage does not exceed the high watermark; else, it will purge entries.

## 4.4. Purge parameters

You can add a "Purge_parameters" block to the configuration file, to have a better control on purges:

```
Purge_Parameters
{
    nb_threads_purge      = 4;
    post_purge_df_latency = 1min;
    db_result_size_max    = 10000;
}
```

Purge actions are performed in parallel. You can specify the number of purge threads by setting the *nb_threads_purge* parameter.
On filesystems where releasing data is asynchronous, the 'df' command may take a few minutes before returning an up-to-date value after purging a lot of files. Thus, Robinhood must wait before checking disk usage again after a purge. This is driven by the *post_purge_df_latency* parameter.

## 4.5. Using file classes

Robinhood makes it possible to apply different purge policies to files, depending on their properties (path, posix attributes, …). This can be done by defining file classes that will be addressed in policies.

In this section of the tutorial, we will define 3 classes and apply different policies to them:

- We don't want "*.log" files that owns to 'root' to be purged;

- We want to keep files from directory /tmp/A to be kept longer on disk than files from other directories.

First, we need to define those file classes, in a ‘filesets’ section of the configuration file. We associate a custom name to each FileClass, and specify the definition of the class:

```
Filesets
{
      # log files owned by root
      FileClass root_log_files
      {
            definition
            {
                  owner == root
                  and
                  name == “*.log”
            }
      }

      # files in filesystem tree /fs/A
      FileClass A_files
      {
            definition
            {
                  tree == “/fs/A”
            }
      }

}
```

Then, those classes can be used in policies:

- entries can be white-listed using a ‘ignore_fileclass’ statement;
- they can be targeted in a policy, using a ‘target_fileclass’ directive.

```
Purge_Policies
{
      # whitelist log files of ‘root’
      ignore_fileclass = root_log_file;

      # keep files in /fs/A at least 12h after their last access
      Policy purge_A_files
      {
            target_fileclass = A_files;
            condition
            {
                  last_access > 12h
            }
      }

      # The default policy applies to all other files
      # (files not in /fs/A and that don’t own to root)
      Policy default
      {
            condition
            {
                  last_access > 1h
            }
      }
}
```

Notes:

- a given FileClass cannot be targeted simultaneously in several purge policies;
- policies are matched in the order they appear in configuration file. In particular, if 2 policy targets overlap, the first matching policy will be used;
- For ignoring entries, you can directly specify a condition in the ‘purge_policies’ section, using a ‘ignore’ block:

```
Purge_Policies
{
   Ignore
   {
      owner == root
      and
      name == “*.log”
   }

   …
```

# 5. Empty directory removal

Purge after purge, there will be more and more empty directories in the filesystem namespace. Robinhood provides a mechanism for removing directories that have been empty for a long time.
This policy is driven by a ‘rmdir_policy’ section in the configuration file. This section only consists in a single parameter ‘age_rm_emty_dirs’, that indicates the duration after which a empty directory is removed.
You can also specify a ‘ignore’ condition for directories you never want to be removed.

```
rmdir_policy
{
      # removing empty directories after 15 days
      age_rm_empty_dirs = 15d;

      ignore
      {
            depth < 2
            or
            owner == ‘foo’
            or
            tree == /fs/subdir/A
      }
}
```

You can also specify advanced parameters by writing a ‘rmdir_parameters’ block:

```
rmdir_parameters
{
    # Interval for performing empty directory removal
    runtime_interval = 12h ;

    # Number of threads for performing rmdir operations
    nb_threads_rmdir = 4 ;
}
```

In this section, you can also specify the period for checking empty directories (*runtime_interval* parameter).

For running empty directory removal, start robinhood with the '`--rmdir`' option:

```
robinhood –f /etc/robinhood.d/test.conf --rmdir
```

# 6. Robinhood on Lustre

Robinhood provides special features for Lustre filesystems:

## *6.1. Purge triggers on OST usage*

Robinhood can monitor OST usage independently, and trigger purges only for the files of a given OST when it exceeds a threshold.

E.g.:

```
Purge_Trigger
{
      trigger_on = OST_usage ;
      high_watermark_pct = 85% ;
      low_water_mark_pct = 80% ;
      check_interval = 5min ;
}
```

## *6.2. File classes/conditions on OST pool names*

In Lustre 1.8 release and later, you can use OST pool names for specifying file classes.

E.g.:

```
Filesets
{
      FileClass da1x_storage
      {
            ost_pool == “da1*”
      }
}
```

## *6.3. Use Lustre changelogs instead of scanning filesystem with Lustre 2!*

With Lustre 2.x, file system scans are no more required to update robinhood's database: it can collect events from Lustre using Lustre's ChangeLog mechanism. This avoid over-loading the filesystem with namespace scans!
To activate this feature in Lustre, use the 'rh-config' script on MDS:

```
rh-config enable_chglogs
```

Then, set related information into robinhood's configuration:

```
ChangeLog
{
    MDT
    {
        mdt_name  = "MDT0000";
        reader_id = "cl1";
    }

    force_polling = ON;
}
```

`force_polling`: on Lustre 2.0-alpha5/6, changelog readers need to perform active polling to get new events from MDT. Thus, you need to activate polling with this option on these lustre versions.
On Lustre v2, robinhood does event handling by default, instead of scanning. You can start a robinhood instance for reading events only, using the '`--handle-events`' option.

# 7. Optimizations and compatibility

## *7.1. Optimize report generation*

By default, robinhood database is optimized for insertion and update, to get the best performances when scanning large filesystems and processing high streams of events.
On the other side, this makes it longer to generate reports: it may take a couple of minutes generating a summary for filesystems with tens of millions of entries.
If you don't care about the time it takes for scanning, and if its event processing pipeline does not appear to be busy, you may want to optimize Robinhood for generating reports faster. There is not a single universal optimization for this: you have to define specific indexes on the database, depending on the reports you need.
Don't hesitate asking on robinhood-support mailing list to determine the good index for your need : robinhood-support@lists.sourceforge.net.

## *7.2. rpmbuild compatibility issue*

Robinhood spec file (used for generating the RPM) is written for recent Linux distributions (RH5 and later). If you have troubles generating robinhood RPM (e.g. undefined rpm macros), you can switch to the older spec file (provided in the distribution tarball):

```
> mv robinhood.old_spec.in robinhood.spec.in
> ./configure ….
> make rpm
```

## *7.3. SLES init depedencies*

On SLES systems, the default dependency for boot scheduling is on "mysql" service.
However, in many cases, it should be too early for starting robinhood daemon, especially if the filesystem it manages is not yet mounted. In such case, you have to modify the following

lines in `scripts/robinhood.init.sles.in` before you run `./configure`:

```
# Required-Start:    <required service>
```

---

## Get more information

**In-line help**
'--help' options of 'robinhood' and 'rh-report' commands provide you with detailed descriptions of command-line parameters.

**Admin guide**
For more details about robinhood configuration, you can refer to the admin guide.
It is usually located in the `doc/admin_guides` directory of the distribution tarball.

**Mailing list**
If you have other specific questions, you can get support by posting to Robinhood mailing list: robinhood-support@lists.sourceforge.net