

Robinhood PolicyEngine

Temporary Filesystem Manager

Configuration tutorial

Thomas LEIBOVICI
CEA/DAM

<thomas.leibovici@cea.fr>

V2.3.0

May 6th, 2011

Table of contents

1. Installation	3
1.1. Robinhood	3
1.1.1. Install from RPM.....	3
1.1.2. Build and install from the source tarball	3
Requirements.....	3
Compilation.....	3
1.2. MySQL database	4
Requirements.....	4
Creating Robinhood database.....	4
2. Run your first Robinhood instance.....	5
2.1. Configuration file	5
2.2. Running robinhood.....	6
2.3. Getting stats on filesystem content.....	7
3. Scan options and alerts	8
4. Resource monitoring and purges	10
4.1. Defining purge triggers	10
4.2. Minimal purge policy	11
4.3. Space monitoring and purges	12
4.4. Purge parameters	12
4.5. Using file classes	13
5. Directory removal	14
6. Robinhood on Lustre	15
6.1. Purge triggers on OST usage.....	15
6.2. [new 2.3] File classes and conditions on OST indexes	16
6.3. File classes/conditions on OST pool names	16
6.4. Use Lustre changelogs instead of scanning the filesystem with Lustre 2!.....	16
7. Optimizations and compatibility.....	17
7.1. [new 2.2] Periodic fileclass matching	17
7.2. [new 2.3] Optimize reporting speed	18
7.3. rpmbuild compatibility issue	18
7.4. SLES init dependencies.....	18
Get more information	18

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 statistics and 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*

1.1.1. Install from RPM

Pre-generated RPMs can be downloaded on sourceforge, for the following configuration:

- x86_64 architecture , RedHat 5/6 Linux family
- MySQL database 5.x
- Posix filesystems, Lustre 1.8 filesystems, Lustre 2.0 filesystems

The RPM includes:

- ‘robinhood’ and ‘rbh-report’ binaries
- ‘rbh-config’ configuration helper
- Configuration templates
- /etc/init.d/robinhood script

1.1.2. Build and install from the source tarball

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-2.3.0.tar.gz
cd robinhood-2.3.0
```

Then, use the “configure” script to generate Makefiles:

- use the --with-purpose=TMP_FS_MGR option for using it as a temporary filesystem manager;

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

Other './configure' options:

- You can change the default prefix of installation path (default is /usr) using:
'--prefix=<path>'
- If you want to disable Lustre specific features (getting stripe info, purge by OST...), use the '--disable-lustre' option.

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.

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 'rbh-config' script. Run this script on the database host to check your system configuration and perform database creation steps:

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

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

Note: if no option is given to *rbh-config*, it prompts for configuration parameters interactively. Else, if you specify parameters on command line, it runs in batch mode.

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';`
- The 'super' privilege is required for creating DB triggers (needed for accounting optimizations):
`GRANT SUPER ON *.* 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.

Note1: you can also specify special values ‘stderr’, ‘stdout’ or ‘syslog’ for log parameters.

Note2: robinhood is compliant with log rotation (if its log file is renamed, it will automatically switch to a new log file).

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)

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 `rbh-report` command.

- Getting stats for a user: `-u` option

```
rbh-report -u foo
```

```
User:                foo

Type:                directory
Count:               90
Space used:          720.00 KB    (1440 blks)

Type:                file
Count:               609
Space used:          20.26 MB    (41496 blks)
Size avg:            27.21 KB    (27865 bytes)

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

- Splitting user's usage per group: `-S` option
(use of `--csv` option for row formatting)

```
rbh-report -u foo -S --csv
```

user	,	group,	type,	count,	spc_used,	avg_size
foo	,	group1,	dir,	208,	106496,	0
foo	,	group1,	file,	7908,	278658000,	35237
foo	,	group2,	dir,	125,	64000,	0
foo	,	group2,	file,	34525,	4262732700,	123468

- Getting largest files: `--top-size` option

```
rbh-report --top-size
```

```
Rank:                1
Path:                /tmp/robinhood-2.1.1/rpms/BUILD/robinhood/src/Robinhood/rbh-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/grpl
*Purge class:        tmp_file_class

Rank:                2
Path:                /tmp/robinhood-2.1.1/rpms/BUILD/robinhood/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
Purge class:      [default]
```

...

*new in robinhood 2.2

- Getting top space consumers: --top-users option

```
rbh-report --top-users
```

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

```
Rank:                2
User:                root
Space used:          32.00 KB    (64 blks)
Nb entries:           4
Size avg:            2.05 KB    (2104 bytes)
```

...

- Filesystem content summary: -i option

```
rbh-report -i
```

```
Type:                directory
Count:               90
```

```
Type:                file
Count:               609
Size avg:            23.92 KB    (24489 bytes)
```

- **[new 2.2]** fileclasses summary: --class-info option

```
rbh-report --class-info
```

```
Class:               important_files
Nb entries:          879
Space used:          9.00 GB    (18874368 blks)
Size min:            0          (0 bytes)
Size max:            1.00 GB    (19327352832 bytes)
Size avg:            10.48 MB    (10993664 bytes)
```

```
Class:               project_B
Nb entries:          878
Space used:          2.23 GB    (4666481 blks)
Size min:            0          (0 bytes)
Size max:            95.37 MB    (100000000 bytes)
Size avg:            2.60 MB    (2727157 bytes)
```

- ...and more:
you can also generate reports, or dump files per directory, per OST, etc...
To get more details about available reports, run 'rbh-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 variable scan interval, depending on the 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 Large_Directory
    {
        type == directory
        and
        dircount > 10000
        and
        last_mod < 1d
    }

    # Raise alerts for large files
    Alert Large_file
    {
        type == file
        and
        size > 100GB
        and
        last_mod < 1d
    }
}
```

```
}  
}
```

In this example, we want to raise alerts for directories with more than 10.000 entries and file larger than 100GB.

Note that these rules are matched at each scan. If you don't want to be alerted about a given file at every scan, it is advised to specify a condition on `last_mod`, so you will only be alerted for recently modified entries.

[new 2.2] You can now specify a name for alerts, so you can easily identify the issue in the alert title (in e-mail title).

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.

[new 2.2] You can now batch alert e-mails as a single summary (instead of sending 1 mail per file). This is driven by the `batch_alert_max` parameter of Log section (0=no batch limit, 1=no batching).

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 purposes of robinhood is to monitor disk space usage and trigger purges when disk space runs low.

Purge triggering is based on high/low thresholds and files removal is based on a LRU list: when the used space reaches a high threshold, 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 threshold.

4.1. *Defining purge triggers*

First of all, let's specify thresholds for triggering purge.

This can be done on several criteria:

- Global filesystem usage
- OST usage (for Lustre filesystems)
- User usage (quota-like purge)
- Periodic purge (whatever the filesystem usage is) **[new 2.2.3]**

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

```
Purge_Trigger  
{  
    trigger_on          = global_usage ;  
    high_threshold_pct  = 90% ;  
    low_threshold_pct   = 85% ;  
    check_interval      = 5min ;  
}
```

trigger_on specifies the type of trigger.

high_threshold_pct indicates the disk usage that must be reached for starting purge.

low_threshold_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, by specifying a kind of “quota”:

```
Purge_Trigger
{
    trigger_on          = user_usage ;
    high_threshold_vol  = 10GB ;
    low_threshold_vol   = 9GB ;
    check_interval      = 12h ;
}
```

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

A list of users (separated by coma) can also be specified for quotas (wildcards are allowed), e.g:

```
Purge_Trigger {
    trigger_on          = user_usage(foo,charlie,project*) ;
    ...
}
```

[new 2.2.3] Periodic trigger periodically purge all entries that match purge policies (no threshold is expected in this case):

```
Purge_Trigger {
    trigger_on = periodic;
    check_interval = 12h;
}
```

[new 2.2.3] To receive a mail notification each time a high threshold (or quota) is reached, add this parameter to a trigger:

```
    alert_high = true ;
```

By default, robinhood raises an alert if it can't purge enough data to reach the low threshold.

[new 2.2.3] You can disable those alerts by adding this in a trigger definition:

```
    alert_low = false ;
```

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. *Space monitoring and purges*

Robinhood is now able to monitor disk usage and purge entries when needed. Start the daemon with the ‘--purge’ option. If you 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 threshold:
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 parameter, it will run all configured actions.

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 threshold; else, it will purge entries.

Only check thresholds without purging data:

if robinhood is started with the ‘--check-thresholds’ option instead of ‘--purge’, it will only check for trigger conditions without purging data.

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 belong 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;
- If you modify a fileclass definition or target fileclasses of policies, you need to reset fileclass information in database. To do so, run this command:
rbh-config reset_classes
- 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"
    }
    ...
}

```

[new 2.2] FileClasses can be defined as the union or the intersection of other FileClasses. To do so, use the special keywords “union”, “inter” and “not” in the fileclass definition:

```

FileClass root_log_A
{
    Definition {
        (root_log_files inter A_files)
        union (not B_files)
    }
}

```

5. 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.

It is also possible to recursively remove entire parts of the filesystem (“rm -rf”), by specifying a condition on the top-level directory.

Directory removal is driven by a ‘rmdir_policy’ section in the configuration file:

- The condition for removing empty directories is given by the ‘age_rm_empty_dirs’ parameter, that indicates the duration after which an empty directory is removed.

- You can specify one or several ‘ignore’ condition for directories you never want to be removed.
- Use ‘recursive_rmdir’ sub-blocks to indicate what matching directories are to be removed recursively. *#!/*The whole directories content is removed without checking policies on their content (whitelist rules...).

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

    # recursively remove ".trash" directories after 15 days
    recursive_rmdir
    {
        name == ".trash" and last_mod > 15d
    }

    # whitelist directories matching the following condition
    ignore
    {
        depth < 2
        or
        owner == 'foo'
        or
        tree == /fs/subdir/A
    }
}
```

You can specify advanced parameters by writing an ‘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_threshold_pct = 85% ;
    low_threshold_pct = 80% ;
    check_interval = 5min ;
}
```

6.2. *[new 2.3] File classes and conditions on OST indexes*

On Lustre filesystem, you can specify policy conditions and fileclasses based on ost indexes.

E.g.:

```
Filesets
{
    FileClass retired_ost
    {
        ost_index == 1 or ost_index == 4 or ost_index == 7
        and ost_index != 9
    }
}
```

“ost_index == *N*” is true if at least one part of the file is stored in OST index #*N*.

“ost_index != *N*” is true if no part of the file is stored in OST index #*N*.

(Note: ‘<’ and ‘>’ comparators are not allowed for this condition).

6.3. *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 dalx_storage
    {
        ost_pool == "dal*"
    }
}
```

6.4. *Use Lustre changelogs instead of scanning the 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 avoids over-loading the filesystem with namespace scans!

To activate this feature in Lustre, use the ‘rbh-config’ script on MDS:

```
rbh-config enable_chglogs
```


Then, set related information into robinhood's configuration:

```
ChangeLog
{
    MDT
    {
        mdt_name   = "MDT0000";
        reader_id  = "c11";
    }
}
```

By default, on Lustre v2, robinhood performs event handling when it is started with no option (instead of scanning).

You can start a robinhood instance for reading events only, using the `'-read-log'` option.

Note: in Lustre 2.0, there is a couple of bugs in ChangeLog mechanism ("defunc" processes in daemon mode, thread-safety issues, ...). Those bugs are now fixed in git master branch of Lustre, and fixes will be included in the next Lustre release (probably 2.1). Thus, it is recommended to wait this new version of Lustre for using this feature.

7. Optimizations and compatibility

7.1. *[new 2.2] Periodic fileclass matching*

In previous versions, robinhood matched fileclasses every time it applied policies. This resulted in doing a lot of filesystem calls at this time. Now, it can match fileclasses at regular interval, and only the policy condition for the fileclass is checked at each policy application. Additionally, the fileclass of each file is now stored in the database, so you can see it using the `rbh-report` command.

The fileclass matching interval is set using the `'fileclass_update'` parameter in the new `'db_update_policy'` section:

```
db_update_policy
{
    fileclass_update = periodic( 1h );
}
```

Possible values are:

- `never`: match file class once, and never again
- `always`: always re-match file class when applying policies (this is the old behavior)
- `periodic(<period>)`: match fileclasses if this has not been done for a while.

If you modify fileclass definitions in config file, you need to clear robinhood cache, to force matching them again. This can be done using `'rbh-config'`:

```
> rbh-config reset_classes
```

7.2. [new 2.3] Optimize reporting speed

By default, robinhood is optimized for speeding up common accounting reports (by user, by group, ...), but this can slow database operations during filesystem scans. If you only need specific reports, you can disable some parameters to make scan faster.

For instance, if you only need usage reports by user, you had better disable *group_acct* parameter; this will improve scan performance. In this case, reports on groups will still be available, but their generation will be slower: if you request a *group-info* report and if *group_acct* is *off*, the program will iterate through all the entries (complexity: $O(n)$ with n =number of entries). If *group_acct* is *on*, robinhood will directly access the data in its accounting table, which is quite instantaneous (complexity: $O(1)$).

E.g: if *group_acct* parameter is activated, *group-info* report will be generated in 0.01sec for 1M entries. If *group_acct* is disabled, the same report will take about 10sec.

7.3. 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
```

7.4. SLES init dependencies

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 'rbh-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