rTorrent Handbook

Release 0.1.0

pyroscope

Jun 25, 2017

Contents

This is a comprehensive manual and user guide for the rTorrent bittorrent client, written by and for the community. See also the homepage of the community project and the community wiki.

rTorrent is written in C++ and uses the `ncurses` library to provide a textual user interface. It can be used in a (SSH) terminal session together with a terminal multiplexer like `tmux`, providing a very lean bittorrent solution. Using its XMLRPC remote control API, alternative user interfaces can be provided by web clients like ruTorrent, or command line clients like pyrocore and its `rtcontrol` command.

The *Overview* chapter offers you a guided tour through this manual, or browse through the table of contents below to find what you're looking for.

If you like what is here but are missing something, the best way to fill that hole is to pour what you know into it. Every contribution counts, and instead of lamenting the situation, please go fix it by taking small steps in the right direction. If everyone chimes in, we all profit in the end.

Contributing Guidelines tells you more about how to add your changes to the project.

CHAPTER 1

Contents of This Manual

Overview

rTorrent Feature Summary

- No-frills *ncurses* interface.
- Runs as a daemon, using a terminal multiplexer like `tmux` or `screen` (and 0.9.7+ has a 'real' daemon mode).
- Resource-friendly, ideal to run on a *Raspberry Pi* or a small seedbox VPS.
- Scriptable and extensible via built-in commands and XMLRPC clients.
- Very large choice of web frontends.
- Support for DHT and PEX.
- Magnet links.
- Supported on nearly all trackers.
- Implemented in C++, runs on all major POSIX platforms.

Guided Tour

The *Installation Guide* has some pointers to common ways of installing rTorrent on your machine. It does not provide yet another way to do that, because there already are plentiful and redundant sources out there.

To help you with basic configuration tasks, the *Configuration Quick Start* contains a quick start into the 'scripting language' rTorrent uses for its configuration files.

Common Configuration Use-Cases then goes on showing how to handle a number of typical configuration needs, adding more features to the basic configuration.

Building on that, the *Scripting Guide* explains more complex commands and constructs of said language. It also helps with controlling rTorrent from the outside, via the XMLRPC protocol.

The *Commands Reference* chapter lists all relevant XMLRPC and 'private' commands of *rTorrent* with a short explanation.

Getting Help on IRC

The unofficial help & support channel for rTorrent (webchat) can be found on the freenode.net IRC network.

Web Resources Related to rTorrent

Here is a list of web links to related information:

- rtorrent GitHub project
- libtorrent GitHub project
- rTorrent Community GitHub organization
- Arch Wiki rTorrent page
- rTorrent Quick Reference Card (PDF)

Installation Guide

This chapter has some pointers to common ways of installing *rTorrent* on your machine. It does not provide yet another way to do that, because there already are plentiful and redundant sources out there.

Installation Using OS Packages

While installing using pre-compiled packages is the easiest way to get a working *rTorrent* executable onto your system, it has the unfortunate side-effect that quite often these packages contain a rather outdated version of it.

You might want to look in the "testing" or "experimental" repositories of your distribution, or alternatively install from source (see below).

- The rTorrent wiki lists package names and installation commands for a lot of *Linux* distributions and other operating systems.
- DEB packages of *rTorrent-PS* for Debian and Ubuntu are on Bintray.
- *"Arch User Repository"* (AUR) PKGBUILDs maintained by @xsmile for libtorrent-ps and rtorrent-ps. See also the *Arch Linux* wiki page.

Automated Installation

This is just a selection of the myriad of projects out there that perform automated installs. If you miss something, please make sure a potential new entry is actually still maintained, and mention what target platforms it is designed and tested for.

Projects that work on *Debian* very likely also work on *Ubuntu*. Just make sure the release dates match reasonably, i.e. *Jessie* is equivalent to either *Xenial* or *Trusty*. If you want to run *ruTorrent*, the default version of *PHP* is very important (either 5 or 7).

pimp-my-box *Ansible · Ubuntu Xenial + Trusty · Debian Jessie + Wheezy · Raspian*

This will install *rTorrent-PS*, *pyrocore*, and related software onto any remote dedicated server or VPS with root access, running *Debian* or a Debian-like OS. It does so via *Ansible*, which is in many ways superior to the usual *"call a bash script to set up things once and never be able to update them again"*, since you can run this setup repeatedly to either fix problems, or to install upgrades and new features added to the project's repository.

QuickBox *bash + Javascript*

QuickBox provides easy seedbox and services management from a web dashboard. With the click of a button users can install additional application packages.

AtoMiC-ToolKit *bash · Ubuntu/Mint · full HTPC setup*

AtoMiC Toolkit simplifies the setup of a HTPC or home server and its management, on *Ubuntu* and *Debian* variants including *Raspbian*. It currently supports:

- CouchPotato
- Emby
- Headphones
- HTPC Manager
- Lazy Librarian
- Mylar
- Nzbget
- NZBHydra
- Plex
- PlexPy
- PyLoad
- qBittorrent
- Radarr
- Sabnzbdplus
- Sickgear
- Sickrage
- Sonarr
- TransmissionBT
- Webmin

rtinst *bash · Trusty · Wheezy / Jessie*

Seedbox installation script for *Ubuntu* and *Debian* systems.

Kerwood *bash · Debian Jessie + Wheezy · Raspian*

Auto install script for *rTorrent*, with *ruTorrent* as the web client.

Installing from Source

If you compile your own executable, you are free to chose whatever version you want, including the current bleeding edge of development *(git HEAD)*, or any "release tarball".

Installing (rTorrent wiki) Installation information and some trouble-shooting hints in the *rTorrent* wiki.

Installing rTorrent-PS from Scratch Installation instructions for a working *rTorrent* instance in combination with *PyroScope* from scratch, on *Debian* and most Debian-derived distros.

Installing the "Ultimate Torrent Setup" Guide to install *rtorrent*, *ruTorrent*, *Sonarr*, and *CouchPotato* on *Ubuntu*, proxied by *Apache httpd*.

Installation Guide (JES.SC) A single-page, comprehensive guide to take you step-by-step through installation and configuration of *rTorrent* and *ruTorrent*.

Installation How-To (LinOxide) How to install / setup *rTorrent* and *ruTorrent* on *CentOS* or *Ubuntu*.

Using rtorrent on Linux like a pro An oldie (originally from 2010), but still good.

rTorrent Distributions

rTorrent-PS A *rTorrent* distribution (not a fork of it), in form of a set of patches that improve the user experience and stability of official *rTorrent* releases. The notable additions are the more condensed ncurses UI with colorization and a network bandwidth graph, and a default configuration providing many new features, based in part on an extended command set.

rTorrent-PS-CH This puts more patches and a different default configuration on top of *rTorrent-PS*. It also tries to work with the current git HEAD of *rTorrent*, which *rTorrent-PS* does not.

Configuration Quick Start

To help you with fundamental configuration tasks, this chapter contains a quick start into the 'scripting language' rTorrent uses for its configuration files. *Config Template Deconstructed* uses a basic configuration file to explain what the contained commands are doing, also showing common syntax constructs by example.

The next chapter then dives into some *Common Configuration Use-Cases*, adding more features to that basic configuration.

The ArchLinux wiki page is also a good source on *rTorrent* in general and its configuration in particular.

Note: *rTorrent* started to rename a lot of configuration commands with the release of version `0.8.9`. This handbook uses the new commands throughout, and does not mention the old ones.

See the RPC Migration 0.9 wiki page for more details. That pages also links to a sed script that can transform old snippets you found on the web and might want to use to using the new command names.

The rTorrent Command Line section shows you how you can prevent *rTorrent* from adding most of the old names as aliases for the new ones, by using the `-D -I` command line options.

---

rTorrent Basics

We're assuming you used one of the ways in the *Installation Guide* to add the *rTorrent* binary to your host, ready to be configured and started. Try calling `rtorrent -h` to make sure that worked.

To be really useful, *rTorrent* must be given a basic configuration file, with some essential settings that ensure you get more than the bare-bones defaults. Follow the configuration steps in this chapter on a fresh installation, then try to start *rTorrent* and initiate your first downloads. Or check if you see something you want to add to your existing setup.

After some time, when you're familiar with the basic operation of *rTorrent*, try to work through the *Scripting Guide* if you want to dive deeper into customizing *rTorrent*.

Modernized Configuration Template

Any configuration should start with using the modernized rTorrent wiki config template. The configuration is loaded from the file `~/.rtorrent.rc` by default (that is the hidden file `.rtorrent.rc` in your user home directory). This command fetches the template from *GitHub* and writes it into that file:

```
curl -Ls "https://raw.githubusercontent.com/wiki/rakshasa/rtorrent/CONFIG-Template.md
↪" \
    | grep -A9999 '^######' | grep -B9999 '^### END' \
    | sed -re "s:/home/USERNAME:$HOME:" >~/.rtorrent.rc
mkdir ~/rtorrent  # create user's instance directory
```

All files *rTorrent* uses or creates are located in the `~/rtorrent` directory, except the main configuration file.

Here is a copy of the template in full, see *Config Template Deconstructed* below for a detailed explanation of its parts.

```
##############################################################################
# A minimal rTorrent configuration that provides the basic features
# you want to have in addition to the built-in defaults.
#
# See https://github.com/rakshasa/rtorrent/wiki/CONFIG-Template
# for an up-to-date version.
##############################################################################

# Instance layout (base paths)
method.insert = cfg.basedir, private|const|string, (cat,"/home/USERNAME/rtorrent/")
method.insert = cfg.watch,   private|const|string, (cat,(cfg.basedir),"watch/")
method.insert = cfg.logs,    private|const|string, (cat,(cfg.basedir),"log/")
method.insert = cfg.logfile, private|const|string, (cat,(cfg.logs),"rtorrent-",
↪(system.time),".log")

# Create instance directories
execute.throw = bash, -c, (cat,\
    "builtin cd \"", (cfg.basedir), "\" ",\
    "&& mkdir -p .session download log watch/{load,start}")

# Listening port for incoming peer traffic (fixed; you can also randomize it)
network.port_range.set = 50000-50000
network.port_random.set = no

# Tracker-less torrent and UDP tracker support
# (conservative settings for 'private' trackers, change for 'public')
dht.mode.set = disable
protocol.pex.set = no
trackers.use_udp.set = no

# Peer settings
throttle.max_uploads.set = 100
throttle.max_uploads.global.set = 250

throttle.min_peers.normal.set = 20
throttle.max_peers.normal.set = 60
throttle.min_peers.seed.set = 30
throttle.max_peers.seed.set = 80
trackers.numwant.set = 80

protocol.encryption.set = allow_incoming,try_outgoing,enable_retry

# Limits for file handle resources, this is optimized for
# an `ulimit` of 1024 (a common default). You MUST leave
# a ceiling of handles reserved for rTorrent's internal needs!
network.http.max_open.set = 50
network.max_open_files.set = 600
network.max_open_sockets.set = 300
```

```
# Memory resource usage (increase if you have a large number of items loaded,
# and/or the available resources to spend)
pieces.memory.max.set = 1800M
network.xmlrpc.size_limit.set = 4M

# Basic operational settings (no need to change these)
session.path.set = (cat, (cfg.basedir), ".session")
directory.default.set = (cat, (cfg.basedir), "download/")
log.execute = (cat, (cfg.logs), "execute.log")
##log.xmlrpc = (cat, (cfg.logs), "xmlrpc.log")
execute.nothrow = bash, -c, (cat, "echo >",\
    (session.path), "rtorrent.pid", " ", (system.pid))

# Other operational settings (check & adapt)
encoding.add = utf8
system.umask.set = 0027
system.cwd.set = (directory.default)
network.http.dns_cache_timeout.set = 25
##network.http.capath.set = "/etc/ssl/certs"
##network.http.ssl_verify_peer.set = 0
##network.http.ssl_verify_host.set = 0
##pieces.hash.on_completion.set = no
##keys.layout.set = qwerty

##view.sort_current = seeding, greater=d.ratio=
schedule2 = monitor_diskspace, 15, 60, ((close_low_diskspace, 1000M))

# Some additional values and commands
method.insert = system.startup_time, value|const, (system.time)
method.insert = d.data_path, simple,\
    "if=(d.is_multi_file),\
        (cat, (d.directory), /),\
        (cat, (d.directory), /, (d.name))"
method.insert = d.session_file, simple, "cat=(session.path), (d.hash), .torrent"

# Watch directories (add more as you like, but use unique schedule names)
schedule2 = watch_start, 10, 10, ((load.start, (cat, (cfg.watch), "start/*.torrent")))
schedule2 = watch_load, 11, 10, ((load.normal, (cat, (cfg.watch), "load/*.torrent")))

# Logging:
#   Levels = critical error warn notice info debug
#   Groups = connection_* dht_* peer_* rpc_* storage_* thread_* tracker_* torrent_*
print = (cat, "Logging to ", (cfg.logfile))
log.open_file = "log", (cfg.logfile)
log.add_output = "info", "log"
##log.add_output = "tracker_debug", "log"

### END of rtorrent.rc ###
```

The rTorrent Command Line

Calling `rtorrent -h` shows this usage message regarding command line options (with the last three missing):

```
Usage: rtorrent [OPTIONS]... [FILE]... [URL]...
  -h                Display this very helpful text
```

```
  -n                 Don't try to load ~/.rtorrent.rc on startup
  -b <a.b.c.d>       Bind the listening socket to this IP
  -i <a.b.c.d>       Change the IP that is sent to the tracker
  -p <int>-<int>     Set port range for incoming connections
  -d <directory>     Save torrents to this directory by default
  -s <directory>     Set the session directory
  -o key=opt,...     Set options, see 'rtorrent.rc' file

  -D                 Disable deprecated commands
  -I                 Disable intermediate commands
  -K                 Allow intermediate commands without XMLRPC (just in config files)
```

The really useful ones are `-n` and `-o import=‹file›`, to load configuration from a non-standard location. Everything else is better set in a configuration file.

It is recommended to add `-D` and `-I` to your start script, so that all the old command names are gone. However, some external software (web UIs and so on) might not be able to work with such a reduced command set. Also be aware that those undocumented switches changed their semantics with the release of `0.9.6` – the above shows the current situation.

And here is a simple start script that you should use before you tackle auto-starting *rTorrent* at boot time. First make it work for you, then add the bells and whistles. Copy the script to `~/rtorrent/start`, and make it executable using `chmod a+x ~/rtorrent/start`.

```
#! /bin/bash
#
# rTorrent startup script
#
umask 0027
cd $(dirname "$0")

# Check for running process
export RT_SOCKET=$PWD/.scgi_local
test -S $RT_SOCKET && lsof $RT_SOCKET >/dev/null \
    && { echo "rTorrent already running"; exit 1; }
test ! -e $RT_SOCKET || rm $RT_SOCKET

# Clean up after rTorrent ends
_at_exit() {
    stty sane
    test ! -e $RT_SOCKET || rm $RT_SOCKET
}
trap _at_exit INT TERM EXIT

# Start rTorrent (optionally with configuration loaded
# from the directory this script is stored in)
rtorrent -D -I  # -n -o import=$PWD/rtorrent.rc
```

You can call it in a simple shell prompt first, but for normal operation it must be launched in a `tmux` session, like so:

```
tmux -2u new -n rTorrent -s rtorrent "~/rtorrent/start; exec bash"
```

The `exec bash` keeps your `tmux` window open if *rTorrent* exits, which allows you to actually read any error messages in case it exited unexpectedly.

You can of course add more elaborate start scripts, like a cron watchdog, init.d scripts or systemd units, see the *rTorrent* wiki for examples.

Basic Syntax Elements

The configuration 'scripts' have some usual syntax elements, and some not so usual ones. If you're versed in *any* computer language, you surely spotted some of them in the *Modernized Configuration Template*. Comments start with a `#`, and you can break long lines apart by escaping the line ends with `\`.

The basic structure of lines is `‹ command › = ‹ arg1 ›[, ‹ arg2 ›, ...]`. In configuration files, the `command` either sets some value, or has some side effect: defining a method or schedule, executing a OS command, and so on. This is the 'old' syntax, and still relevant on the top level of configuration files.

Other elements are escaped text in quotes (these are *not* strings in the classical sense), lists in braces `{...}`, and commands in single or double parentheses `(...)`. At some places, a semicolon `;` separates multiple commands executed in sequence.

Quoted text keeps words separated by spaces together, passing all of it as a single argument to a command – quite similar to a string. However, simple words do not need to be escaped; simply put, everything that's not a command is a string.

Quoting can be nested, but the inner quotes have to be escaped using `\`, and on the third level the backslashes have to be escaped too, leading to abominations like `"...\"...\\\"...\\\"...\"..."`. Just avoid that, keep it to two levels at most, e.g. quoted text within a quoted sequence of commands. If you need more complex structures, work with helper methods where you can 'start fresh' when it comes to escaping levels.

Be pragmatic, and have no fear of mixing 'old' and 'new' syntax to your advantage. Prefer the new one with parentheses, but that your syntax works and does the thing you want is most important, readability is next, and any theoretical purity of syntax ideas come in last.

Config Template Deconstructed

With the most basic syntax elements explained, let's look at the configuration template again.

First, some manifest constants used in later commands are defined, with the most important one being the instance's root directory, named `cfg.basedir`. The `cfg.` part is nothing special, just a way to group command names and establish namespaces to avoid naming collisions.

```
method.insert = cfg.basedir, private|const|string, (cat,"/home/USERNAME/rtorrent/")
method.insert = cfg.watch,   private|const|string, (cat,(cfg.basedir),"watch/")
method.insert = cfg.logs,    private|const|string, (cat,(cfg.basedir),"log/")
method.insert = cfg.logfile, private|const|string, (cat,(cfg.logs),"rtorrent-",
↪(system.time),".log")
```

The *method.insert* defines new commands, in this case `private` ones that are only visible within *rTorrent*, but not exposed via the XMLRPC API. They're `const` and thus only evaluated once – if you look at `cfg.logfile` that becomes important, because *system.time* is called only once, during definition. Their type is `string`, other types are `value` and `simple`.

The *cat* command concatenates its arguments to a single string, in this case the 3rd argument to *method.insert*, which is the value that is assigned to the method's name. Text in parentheses are command calls, most notably `(cfg.basedir)` is used to refer to the definition of the root directory everything else is based upon.

The root directory and sub-folders contained in it, that are referenced by various commands further below, are created by calling `mkdir`. It is wrapped in a call to `bash`, because we `cd` into the instance root first and use `&&` to execute `mkdir` after it. Also, the `{brace expansion}` syntax helps to concisely list all the sub-folder names.

```
execute.throw = bash, -c, (cat,\
    "builtin cd \"", (cfg.basedir), "\" ",\
    "&& mkdir -p .session download log watch/{load,start}")
```

Next, the listening port for incoming peer traffic is set using the associated commands *network.port_range.set* and *network.port_random.set*. As shown, the single port number `50000` is used.

```
network.port_range.set = 50000-50000
network.port_random.set = no
```

The settings for tracker-less torrents *dht.mode.set*, peer exchanges *protocol.pex.set*, and UDP tracker support *trackers.use_udp.set* are conservative ones for 'private' trackers. Change them accordingly for using 'public' trackers.

```
dht.mode.set = disable
protocol.pex.set = no
trackers.use_udp.set = no
```

The *throttle.* commands* set minimal demands and upper limits on the amount of peers for incomplete and seeding items.

```
throttle.max_uploads.set = 100
throttle.max_uploads.global.set = 250

throttle.min_peers.normal.set = 20
throttle.max_peers.normal.set = 60
throttle.min_peers.seed.set = 30
throttle.max_peers.seed.set = 80
trackers.numwant.set = 80

protocol.encryption.set = allow_incoming,try_outgoing,enable_retry
```

Next file handle resource limits are defined using some *network.* commands*. The values used are optimized for an *ulimit* of 1024, which is a common default in many Linux systems. You **MUST** leave a ceiling of handles reserved for internal use, that is why they only add up to 950.

```
network.http.max_open.set = 50
network.max_open_files.set = 600
network.max_open_sockets.set = 300
```

The command *pieces.memory.max.set* determines the size of the memory region used by *rTorrent* to map chunks of files for receiving from and sending to peers.

XMLRPC payloads cannot be larger than what *network.xmlrpc.size_limit.set* specifies, the size you need depends on how many items you have loaded, and also what software is using the XMLRPC port.

```
pieces.memory.max.set = 1800M
network.xmlrpc.size_limit.set = 4M
```

The *session.path.set* command sets the location of the directory where *rTorrent* saves its status between starts – a command you should *always* have in your configuration. The default download location for data is set by *directory.default.set*.

```
session.path.set = (cat, (cfg.basedir), ".session")
directory.default.set = (cat, (cfg.basedir), "download/")
log.execute = (cat, (cfg.logs), "execute.log")
##log.xmlrpc = (cat, (cfg.logs), "xmlrpc.log")
execute.nothrow = bash, -c, (cat, "echo >",\
    (session.path), "rtorrent.pid", " ", (system.pid))
```

The *log.execute* and *log.xmlrpc* commands open related log files, which can be very helpful when debugging problems of added extensions. The *execute.nothrow* writes a PID file to the session directory.

There are some other operational settings that don't apply equally to every setup, so check if the values fit for you, and uncomment those settings you want to activate.

```
encoding.add = utf8
system.umask.set = 0027
system.cwd.set = (directory.default)
network.http.dns_cache_timeout.set = 25
##network.http.capath.set = "/etc/ssl/certs"
##network.http.ssl_verify_peer.set = 0
##network.http.ssl_verify_host.set = 0
##pieces.hash.on_completion.set = no
##keys.layout.set = qwerty

##view.sort_current = seeding, greater=d.ratio=
schedule2 = monitor_diskspace, 15, 60, ((close_low_diskspace, 1000M))
```

The next section defines some additional values and commands. `system.startup_time` memorizes the time *rTorrent* was last started, *d.data_path* returns the path to an item's data, and *d.session_file* the path to its session file.

```
method.insert = system.startup_time, value|const, (system.time)
method.insert = d.data_path, simple,\
    "if=(d.is_multi_file),\
        (cat, (d.directory), /),\
        (cat, (d.directory), /, (d.name))"
method.insert = d.session_file, simple, "cat=(session.path), (d.hash), .torrent"
```

Watch directories are an important concept to automatically load metafiles you drop into those directories. They use the *schedule2* command to *watch* these locations, by calling one of the *load.* commands* on a regular basis, taking a directory path and a pattern of files to watch out for. Each schedule must be given a *unique* name, in the simplest case just give them numbers like `watch_01`, `watch_02`, and so on.

```
schedule2 = watch_start, 10, 10, ((load.start, (cat, (cfg.watch), "start/*.torrent")))
schedule2 = watch_load, 11, 10, ((load.normal, (cat, (cfg.watch), "load/*.torrent")))
```

Finally, the logging facility of *rTorrent* is configured, opening a log file using *log.open_file*, giving it a name and a location. The path to that file is also shown on the console at startup, with the *print* command. You can have several of these files, and if you enable the `debug` level for a logging group (see below), it is recommended to put that in a separate file.

Log messages are classified into groups (`connection`, `dht`, `peer`, `rpc`, `storage`, `thread`, `tracker`, and `torrent`), and have a level of `critical`, `error`, `warn`, `notice`, `info`, or `debug`.

With *log.add_output* you can add a logging scope to a named log file. Scopes can either be a whole level, or else a group on a specific level by using ‹ `group` ›_‹ `level` › as the scope's name.

```
print = (cat, "Logging to ", (cfg.logfile))
log.open_file = "log", (cfg.logfile)
log.add_output = "info", "log"
##log.add_output = "tracker_debug", "log"
```

And that's it, more details on using commands are in the *Scripting Guide*, and more examples can be found in the following chapter.

Common Configuration Use-Cases

After you went through *Configuration Quick Start* and got familiar with the handling of *rTorrent*, it's time to look at settings that you should consider for your configuration, but which weren't necessary to start using it.

The Common Tasks in rTorrent wiki page contains more of these typical configuration use-cases.

Load 'Drop-In' Config Fragments

The examples here and in the wiki are mostly short snippets written to serve a specific purpose. To easily add those by just dropping them into a new file, add this to your *main* configuration file (which then can be the last change you apply to it).

```
method.insert = cfg.drop_in, private|const|string, (cat, (cfg.basedir), "config.d")
execute.nothrow = bash, -c, (cat,\
    "find ", (cfg.drop_in), " -name '*.rc' ",\
    "| sort | sed -re 's/^/import=/' >", (cfg.drop_in), "/.import")
try_import = (cat, (cfg.drop_in), "/.import")
```

To test the change, excute these commands:

```
mkdir -p ~/rtorrent/config.d
echo 'print="Hello from config.d!"' >$_/hello.rc
```

Then restart *rTorrent*, and you should see `Hello from config.d!` amongst the initial console messages.

Note: If a drop-in file just contains commands that can be repeated several times, they can be re-imported making them way easier to test after changes. For example, schedules can be redefined, but method definitions can not (under the same name).

Log Rotation, Archival, and Pruning

The following longer snippet adds logs that don't endlessly grow, get archived after some days, and are finally deleted after a while. See rtorrent.d/15-logging.rc for the full snippet.

> **Warning:** If you include this, take care to comment out any conflicting logging commands that you already have in your main configuration.

The time spans for archival and pruning are set using `pyro.log_archival.days` (default: 2) and `pyro.log_retention.days` (default: 7). You can change these in your main configuration, *after* including the snippet via *import*.

```
# Note that the "main" log is only rotated when using rTorrent-PS 1.1+ (after 2017-03-
↪26),
# since 'log.open_file' needed to learn how to re-open first. Otherwise, you'll get a␣
↪daily
# console warning.

# Settings for archival delay, and retention [days]
method.insert.value = pyro.log_retention.days, 7
method.insert.value = pyro.log_archival.days,  2
```

Log files are time stamped (see `pyro.date_iso.log_stamp` and `pyro.log_stamp.current`). Full log file paths for different types are created using `pyro.logfile_path`, which takes the type as an argument.

```
# Create a "YYYY-mm-dd-HHMMSS" time stamp
method.insert = pyro.date_iso.log_stamp, simple|private,\
    "execute.capture_nothrow = bash, -c, \"echo -n $(date +%Y-%m-%d-%H%M%S)\""

# String value for the currently used time stamp, changed on rotation
method.insert = pyro.log_stamp.current, string

# Create a full logfile path using the current stamp
method.insert = pyro.logfile_path, simple|private,\
    "cat = (cfg.logs), (argument.0), \"-\", (pyro.log_stamp.current), .log"
```

The `pyro.log_rotate` multi-method takes care of calculating a new time stamp, and rotating all the log files by re-opening them with their new name. A daily schedule calls this method and thus triggers the rotation.

```
# (Re-)open all logs with a current time stamp; the main log file
# is just opened, you need to add some logging scopes yourself!
method.insert = pyro.log_rotate, multi|rlookup|static
method.set_key = pyro.log_rotate, !stamp,\
    "pyro.log_stamp.current.set = (cat, (pyro.date_iso.log_stamp))"
method.set_key = pyro.log_rotate, execute,\
    "log.execute = (pyro.logfile_path, execute)"
method.set_key = pyro.log_rotate, messages,\
    "branch = (pyro.extended), ((log.messages, (pyro.logfile_path, messages) ))"
method.set_key = pyro.log_rotate, xmlrpc,\
    "log.xmlrpc = (pyro.logfile_path, xmlrpc)"
method.set_key = pyro.log_rotate, ~main,\
    "log.open_file = log, (pyro.logfile_path, rtorrent)"

# Logrotate schedule (rotating shortly after 1AM, so DST shenanigans
# are taken care of, and rotation is always near the next day's begin)
schedule2 = pyro_daily_log_rotate, 01:05:00, 24:00:00, ((pyro.log_rotate))
```

Finally, two schedules take care of daily archival (1:10 AM) and pruning (1:20 AM), passing the command built by `pyro._logfile_find_cmd` to `bash` for execution. The `pyro.log_rotate` method is used near the end to open log files at startup.

```
# Log file archival and pruning
method.insert = pyro._logfile_find_cmd, simple|private,\
     "cat = \"find \", (cfg.logs),\
            \" -type f -name '*.\", (argument.0),\"'\",\
            \" -mtime +\", (argument.1),\
            \" -exec nice \", (argument.2), \" '{}' ';'\""

schedule2 = pyro_logfile_archival, 01:10:00, 24:00:00,\
    "execute.nothrow = bash, -c, (pyro._logfile_find_cmd, log, (pyro.log_archival.
↪days), gzip)"

schedule2 = pyro_logfile_pruning, 01:20:00, 24:00:00,\
    "execute.nothrow = bash, -c, (pyro._logfile_find_cmd, log.gz, (pyro.log_retention.
↪days), rm)"

# Open logs initially on startup
pyro.log_rotate=
```

Set a Download to "Seed Only"

The `d.seed_only` command helps you to stop all download activity on an item. Select any unfinished item, press `Ctrl-X`, and enter `d.seed_only=` followed by . Then all files in that item are set to `off`, and any peers still sending you data are cut off. The data you have is still seeded, as long as the item is not stopped.

```
method.insert = d.seed_only, private|simple,\
    "f.multicall = *, f.priority.set=0 ;\
     d.update_priorities= ;\
     d.disconnect.seeders="
```

f.multicall calls *f.priority.set* on every file, *d.update_priorities* makes these changes known, and finally *d.disconnect.seeders* kicks any active seeders.

Scheduled Bandwidth Shaping

This example shows how to use *schedule2* with absolute start times, to set the download rate depending on the wall clock time, at 10AM and 4PM. The result is a very simple form of bandwidth shaping, with full speed transfers enabled while you're at work (about 16 MiB/s in the example), and only very moderate bandwidth usage when you're at home.

```
schedule2 = throttle_full, 10:00:00, 24:00:00, ((throttle.global_down.max_rate.set_kb,
↪ 16000))
schedule2 = throttle_slow, 16:00:00, 24:00:00, ((throttle.global_down.max_rate.set_kb,
↪  1000))
```

Use *throttle.global_up.max_rate.set_kb* for setting the upload rate.

If you call these commands via XMLRPC from an outside script, you can implement more complex rules, e.g. throttling when other computers are visible on the network.

External scripts should also be used when saving money is the goal, in cases where you have to live with disadvantageous ISP plans with bandwidth caps. Run such a script very regularly (via `cron`), to enforce the bandwidth rules continuously.

Scripting Guide

Building on the *Configuration Quick Start*, this chapter explains more complex commands and constructs of the scripting language. It also helps with controlling *rTorrent* from the outside, via the XMLRPC protocol.

It is to become the comprehensive reference to rTorrent's command language that was always missing, and will only be a success when enough people join forces and thus spread the workload to many shoulders. If you're a developer or power user, this will be an invaluable tool, so please take the time and *contribute what you know*.

See the *Commands Reference* chapter for a list of all relevant XMLRPC and 'private' commands of *rTorrent* with a short explanation. The generated index also lists all the commmand names.

Another helpful tool is the quite powerful *GitHub* search. Use it to find information on commands, e.g. their old vs. new syntax variants, what they actually do (i.e. "read the source"), and internal uses in predefined methods, handlers, and schedules. Consider the view.add example.

Introduction

rTorrent scripting uses a strictly line-oriented syntax, with no control structures that span several logical lines. Read *Basic Syntax Elements* (again) regarding the most fundamental syntax rules. If you skipped *Config Template Decon-*

structed, now is the time to go through it, since it exposes you to common idioms while explaining the core config commands.

Here's also a short command syntax summary:

- Comments start with a `#`.
- Line continuations work by escaping the line end with `\`.
- Commands take the form `cmd = arg, ...` ('old' syntax) or `(cmd, arg, ...)` ('new' syntax).
- Arguments are a comma-separated list: `arg1, arg2, ...`.
- `$cmd=...` evaluates `cmd` and inserts its result value in place of the call. A command in single parentheses is also immediately evaluated.
- Use double quotes for preserving whitespace, or to pass statements as arguments to other commands (like *method.insert* and *schedule2*). Use `\"` to escape quotes within quoted strings.
- Use double parentheses to pass a command unevaluated to another command.
- Braces `{...,  ...}` pass a list as an argument, used for setting list values, or with boolean operators.
- All commands are defined in the *C++* source files `rtorrent/src/command_*.cc` of the client's source code, which is the ultimate reference when it comes to intricate details.

Commands

A command is ... It's called by ...

A deep-dive into defining your own commands can be found in the *reference of related commands*.

Use the *Commands Reference* for details on specific commands, and the generated index to find them by name.

Escaping

The most basic form of escaping is when you have to supply a command with multiple arguments to another command as part of an argument list. You have to tell rTorrent which comma belongs to the inner argument list, and which to the outer one, by quoting the inner command using double quotation marks:

```
outer = arg1, "inner=arg21,arg22", arg3
```

It's also good style to avoid deep nesting by defining your own custom commands (see *method.insert*, and also *Config Template Deconstructed* and *Common Configuration Use-Cases* for many examples). You can then use these building blocks in another command, instead of a literal nested group. The additional benefit is you can name things for documentation purposes, and also avoid overly long lines.

In practice, anything but a single nested quote should be avoided, because the next level already gives you the `\\\"` awkwardness.

Make *plenty* use of line continuations, i.e. escaping of line ends to break up long physical lines into several short ones. Put the breaks into places where you can use any amount of whitespace, and then indent the parts according to the structure of the logical line.

```
method.insert = indent_sequence_of_cmds_and_their_args, private|simple,\
    "load.verbose =\
        (cat, (cfg.watch), (argument.0), /*.torrent),\
        (cat, d.category.set=, (argument.0)) ;\
     category.view.update = (argument.0)"
```

```
schedule2 = polling, 10, 120,\
    ((d.multicall2, main,\
        "branch=\"or={d.up.rate=,d.down.rate=,}\",\
            poll=$interval.active=,\
            poll=$interval.idle="))
```

Also note how using combinations of 'new' and 'old' syntax keeps the needed amount of escaping at bay (double parentheses are also a form of escaping).

Object Types

This is a summary about the possible object types in command_dynamic.cc (applies to `0.9.6`).

- multi (with subtypes: static, private, const, rlookup)
  - **TODO:** what is it
- simple (with subtypes: static, private, const)
  - **TODO:** why is it "simple"
- value, bool, string, list (with subtypes: static, private, const)
  - Standard types, `value` is an integer.

Formatting & Type Conversions

TODO

Custom Attributes

TODO

Advanced Concepts

'.multicall' Demystified

TODO

Scripting Best Practices

TODO

Using XMLRPC for Remote Control

TODO

- TCP vs. Unix domain sockets
- raw SCGI vs. HTTP gateways
- XMLRPC buffer size
- client libs

- daemon mode

Frequently Asked Questions

How Can I Stop All Torrents From a Shell?

This is most useful when *rTorrent* consistently crashes shortly after starting up. That often means you have an item that refers to a data file with an I/O error or a similar fault. To solve this, you need stop all torrents from the outside, since you cannot use the crashing client for it.

Before you do this, **make a backup of your session folder!** Then call this command:

```
for i in ~/rtorrent/.session/*.torrent.rtorrent; do \
    sed -i -re 's/5:stateii1e/5:stateii0e/' $i; done
```

Now you can start the client, and there's a good chance it won't crash this time. To start items one by one, use this:

```
while true; do rtcontrol --from-view stopped is_complete=y -/1 \
                         --start --flush -qo name || break; sleep 2; done
```

When the bad item is started, the crash might be triggered immediately, or with some delay. Increase the sleep time if removing the last item shown (and possibly the one following it) does not solve the problem.

What is the Difference Between 'paused' and 'stopped'?

TODO

Commands Reference

The reference chapter lists all relevant XMLRPC and 'private' commands provided by *rTorrent* with a short explanation. See the *Scripting Guide* on how to combine them into meaningful command sequences.

Use the search box in the sidebar to find specific commands, or the search. The generated index also lists all the commmand names.

List of Command Groups

The following are similar, but incomplete resources:

- PyroScope's reference
- wikia.com Reference

Download Items and Attributes

*d.** commands

All `d.*` commands take an info hash as the first argument when called over the XMLRPC API, to uniquely identify the *target* object. 'Target' is the term used for that 1st parameter in error messages and so on.

```
d.name = ‹hash›  string ‹name›
```

When called within configuration methods or in a `Ctrl-X` prompt, the target is implicit.

d.multicall2

d.multicall.filtered

download_list

```
# 'd.multicall.filtered' is rTorrent-PS 1.1+ only
d.multicall2 = ‹view›, [‹cmd1›=[‹args›][, ‹cmd2›=...]]  list of lists of results␣
↪‹rows of results›
d.multicall.filtered = ‹view›, ‹predicate›, [‹cmd1›=[‹args›][, ‹cmd2›=...]]  same␣
↪as 'multicall2'
download_list = ‹view›  list of strings ‹info hashes›
```

These commands iterate over the content of a given view, or `default` when the view is omitted or empty. `download_list` always just returns a list of the contained infohashes.

`d.multicall2` iterates over all items in `view` and calls the given commands on each, assembling the results of those calls in a row per item. Typically, the given commands either just have a side effect (e.g. *d.stop*), or return some item attribute (e.g. *d.name*).

`d.multicall.filtered` is only available in *rTorrent-PS*, and evaluates the `predicate` condition as a filter for each item, only calling the commands for items that match it. See *elapsed.greater* for an example.

If you request a lot of attribute values on *all* items, make sure you set a big enough value for *network.xmlrpc.size_limit* to hold all the returned data serialized to XML. It is also valid to pass no commands at all to `d.multicall2`, but all you get from that is a list of empty lists.

Example:

```
$ rtxmlrpc --repr d.multicall2 '' tagged d.hash= d.name= d.custom=category
[['91C588B9A9B5A71F0462343BC74E2A88C1E0947D',
  'sparkylinux-4.0-x86_64-lxde.iso',
  'Software'],
 ['17C14214B60B92FFDEBFB550380ED3866BF49691',
  'sparkylinux-4.0-x86_64-xfce.iso',
  'Software']]

$ rtxmlrpc --repr download_list '' tagged
['91C588B9A9B5A71F0462343BC74E2A88C1E0947D',
 '17C14214B60B92FFDEBFB550380ED3866BF49691']
```

d.name

d.base_filename

d.base_path

d.directory

d.directory.set

d.directory_base

d.directory_base.set

```
d.name = ‹hash›  string ‹name›
d.base_filename = ‹hash›  string ‹basename›
d.base_path = ‹hash›  string ‹path›
d.directory = ‹hash›  string ‹path›
d.directory_base = ‹hash›  string ‹path›
d.directory.set = ‹hash›, ‹path›  0
d.directory_base.set = ‹hash›, ‹path›  0
```

These commands return various forms of an item's data path and name, and the last two can change the path, and sometimes the name in the file system. Note that *rTorrent-PS* can also change the displayed name, by setting the `displayname` custom attribute using *d.custom.set*.

Basics:

- `d.base_filename` is always the basename of `d.base_path`.
- `d.directory_base` and `d.directory` are always the same.
- `d.base_filename` and `d.base_path` are empty on closed items, after a restart, i.e. not too useful (since 0.9.1 or so).

Behaviour when `d.directory.set` + `d.directory_base.set` are used (tested with 0.9.4):

- `d.base_path` always remains unchanged, and item gets closed.
- `d.start` sets `d.base_path` if resume data is ok.
- 'single' file items (no containing folder, see *d.is_multi_file*):
  - `d.directory[_base].set` → `d.name` is **never** appended (only in `d.base_path`).
  - after start, `d.base_path` := `d.directory/d.name`.
- 'multi' items (and yes, they can contain just one file):
  - `d.directory.set` → `d.name` is appended.
  - `d.directory_base.set` → `d.name` is **not** appended (i.e. item renamed to last path part).
  - after start, `d.base_path` := `d.directory`.

Making sense of it (trying to at least):

- `d.directory` is *always* a directory (thus, single items auto-append `d.name` in `d.base_path` and cannot be renamed).
- `d.directory_base.set` means set path **plus** basename together for a multi item (thus allowing a rename).
- only `d.directory.set` behaves consistently for single+multi, regarding the end result in `d.base_path`.

The definition below is useful, since it *always* contains a valid path to an item's data, and can be used in place of the unreliable `d.base_path`.

```
# Return path to item data (never empty, unlike `d.base_path`);
# multi-file items return a path ending with a '/'.
method.insert = d.data_path, simple,\
    "if=(d.is_multi_file),\
        (cat, (d.directory), /),\
        (cat, (d.directory), /, (d.name))"
```

d.is_active

d.is_open

d.open

d.pause

d.resume

d.close

d.close.directly

d.start

d.state

d.state_changed

d.state_counter

d.stop

d.try_close

d.try_start

d.try_stop **TODO**

d.loaded_file

d.tied_to_file

d.tied_to_file.set `d.loaded_file` is the metafile from which this item was created. After loading from a watch directory, this points to that watch directory, but after a client restart it is the session file (since the item is then loaded from there).

`d.tied_to_file` also starts out as the file the item is initially created from, but can be set to arbitrary values, and an item can be *untied* using *d.delete_tied*, leading to an empty value and the deletion of the tied file.

One of the *stop_untied*, *close_untied*, or *remove_untied* commands can then be used in a schedule to stop, close, or remove an item that lost its tied file, including when you delete or move it from the outside in a shell or cron job.

d.accepting_seeders

d.accepting_seeders.disable

d.accepting_seeders.enable **TODO**

d.bitfield

d.bytes_done **TODO**

d.check_hash Checks the piece hashes of an item against its data. Started items are paused during the rehashing.

d.chunk_size

```
d.chunk_size = ‹hash›  value ‹size›
```

Returns the item's chunk size in bytes (also known as the "piece size").

d.chunks_hashed

d.chunks_seen **TODO**

d.complete

d.completed_bytes

d.completed_chunks TODO

d.connection_current

d.connection_current.set

d.connection_leech

d.connection_seed TODO

d.create_link

d.delete_link TODO

d.delete_tied Delete the *d.tied_to_file*, which obviously also unties the item. This command is bound to the `U` key by default, and also called whenever an item is erased.

Example:

```
# Delete metafile from a watch dir directly after loading it
# (note that a copy still remains in the session directory)
schedule2 = watch_cleaned, 29, 10, \
    ((load.normal, (cat,(cfg.watch),"cleaned/*.torrent"), "d.delete_tied="))
```

d.creation_date TODO

d.custom

d.custom.set

d.custom_throw

d.custom1

d.custom1.set

d.custom2...5

d.custom2...5.set

```
d.custom[_trow] = string ‹key›  string ‹value›
d.custom.set = string ‹key›, string ‹value›  0
d.custom1 =  string ‹value›
d.custom1.set = string ‹value›  0
```

Set and return custom values using either arbitrary keys, or a limited set of 5 numbered slots. Note that `d.custom1` is *not* the same as `d.custom=1` or `d.custom=custom1`, and can only be accessed by its assigned commands.

If `d.custom` is called for a key that doesn't exist, it will return an empty string, unlike `d.custom_throw` which throws a `No such custom value` error.

Try to avoid the numbered versions, they're obviously limited, and collisions with other uses are quite likely. *ruTorrent* for example uses #1 for its label, and the other slots for various other purposes.

d.disconnect.seeders TODO

d.down.choke_heuristics

d.down.choke_heuristics.leech

d.down.choke_heuristics.seed

d.down.choke_heuristics.set TODO

d.down.rate

d.down.total

```
d.down.rate = ‹hash›  value ‹rate›
d.down.total = ‹hash›  value ‹total›
```

The total amount and current rate of download traffic for this item. It's possible for the total download to be greater than *d.size_bytes*, due to error correction or discarded data.

d.downloads_max

d.downloads_max.set

d.downloads_min

d.downloads_min.set

```
d.downloads_max = ‹hash›  value ‹max›
d.downloads_max.set = ‹hash›, value ‹max›  0
d.downloads_min = ‹hash›  value ‹max›
d.downloads_min.set = ‹hash›, value ‹max›  0
```

Control the maximum and minimum download slots that should be used per item. *rTorrent* will attempt to balance the number of active connections so that the number of unchoked connections is between the minimum and maximum, which means that these are not hard limits, but are instead goals that *rTorrent* will try to reach.

`0` means unlimited, and while `d.downloads_max` can be set to less than `d.downloads_min`, *rTorrent* will then use `d.downloads_min` as the maximum instead.

d.erase

d.free_diskspace **TODO**

d.group

d.group.name

d.group.set **TODO**

d.hash

```
d.hash = ‹hash›  string ‹hash›
```

Returns the hash of the torrent in hexadecimal form, with uppercase letters. The most common use is in the command list of a *d.multicall2*, to return the hash in a list of results. It can also be used to check if a hash already exists in the client – while most other getters can serve the same purpose, this is the obvious one to use for that.

If you are looking to cause a hash check, see *d.check_hash*.

d.hashing

d.hashing_failed

d.hashing_failed.set **TODO**

d.ignore_commands

d.ignore_commands.set **TODO**

d.incomplete

d.is_hash_checked

d.is_hash_checking

d.is_meta TODO

d.is_multi_file

```
d.is_multi_file = ‹hash›  bool (0 or 1)
```

Returns `1` if the torrents is marked as having multiple files, `0` if it's a single file. Note that multifile-marked torrents are able to only have 1 actual file in them. See *d.size_files* for returning the number of files in an item.

d.is_not_partially_done

d.is_partially_done TODO

d.is_pex_active

```
d.is_pex_active = ‹hash›  bool (0 or 1)
```

Return whether PEX is active for this item. See *protocol.pex* to determine if PEX is active globally.

d.is_private

```
d.is_private = ‹hash›  bool (0 or 1)
```

Indicates if the private flag is set. If it is, the client will not attempt to find new peers in addition to what a tracker returned (i.e. PEX and DHT are inactive).

d.left_bytes

d.load_date

d.local_id

d.local_id_html TODO

d.max_file_size

d.max_file_size.set Controls the maximum size of any file in the item. If a file exceeds this amount, the torrent cannot be opened and an error will be shown. Defaults to the value of *system.file.max_size* at the time the torrent is added.

d.max_size_pex TODO

d.message

d.message.set

```
d.message = ‹hash›  string ‹message›
d.message.set = ‹hash›, string ‹message›  0
```

Used to store messages relating to the item, such as errors in communicating with the tracker or a hash check failure.

d.mode TODO

d.peer_exchange

d.peer_exchange.set TODO

d.peers_accounted

d.peers_complete

d.peers_connected TODO

d.peers_max

d.peers_max.set

d.peers_min

d.peers_min.set

d.peers_not_connected TODO

d.priority

d.priority.set

d.priority_str TODO

d.ratio Returns the current upload/download ratio of the torrent. This is the amount of uploaded data divided by the completed bytes multiplied by 1000. If no bytes have been downloaded, the ratio is considered to be `0`.

d.save_full_session Flushes the item's state to files in the session directory (if enabled). This writes *all* files that contribute to an item's state, i.e. the 'full' state.

See also *session.save* and *d.save_resume* below.

d.save_resume Similar to *d.save_full_session*, but skips writing the original metafile, only flushing the data in the `*.libtorrent_resume` and `*.rtorrent` files.

The new data is written to `*.new` files and afterwards renamed, if writing those files succeeded.

d.size_bytes

d.size_chunks

d.size_files

d.size_pex Returns the various size attributes of an item.

- **bytes**: The total number of bytes in the item's files.
- **chunks**: The number of chunks, including the trailing chunk.
- **files**: The number of files (does not include directories).
- **pex**: The number of peers that were reported via the PEX extension. If *d.is_pex_active* is false, this will be always be 0.

d.skip.rate

d.skip.total

```
d.skip.rate = ‹hash›  value ‹rate›
d.skip.total = ‹hash›  value ‹total›
```

Skipped pieces are ones that were received from peers, but weren't needed and thus ignored. These values are part of the main download statistics, i.e. *d.down.rate* and *d.down.total*.

d.throttle_name

d.throttle_name.set

d.timestamp.finished

d.timestamp.started

d.tracker.insert

d.tracker.send_scrape

d.tracker_announce

d.tracker_focus

d.tracker_numwant

d.tracker_numwant.set

d.tracker_size

d.up.choke_heuristics

d.up.choke_heuristics.leech

d.up.choke_heuristics.seed

d.up.choke_heuristics.set

d.up.rate

d.up.total **TODO**

d.update_priorities After a scripted change to priorities using *f.priority.set*, this command **must** be called. It updates the internal state of a download item based on the new priority settings.

d.uploads_max

d.uploads_max.set

d.uploads_min

d.uploads_min.set

```
d.uploads_max = ‹hash›  value ‹max›
d.uploads_max.set = ‹hash›, value ‹max›  0
d.uploads_min = ‹hash›  value ‹min›
d.uploads_min.set = ‹hash›, value ‹min›  0
```

Control the maximum and minimum upload slots that should be used. *rTorrent* will attempt to balance the number of active connections so that the number of unchoked connections is between the given minimum and maximum.

`0` means unlimited, and when `d.uploads_max` is less than `d.uploads_min`, *rTorrent* will use `d.uploads_min` as the maximum instead.

d.views

d.views.has

d.views.push_back

d.views.push_back_unique

d.views.remove **TODO**

d.wanted_chunks **TODO**

Note: The following are only available in *rTorrent-PS*!

---

d.tracker_domain Returns the (shortened) tracker domain of the given download item. The chosen tracker is the first HTTP one with active peers (seeders or leechers), or else the first one.

```
# Trackers view (all items, sorted by tracker domain and then name).
# This will ONLY work if you use rTorrent-PS!
view.add          = trackers
view.sort_new     = trackers, "compare=,d.tracker_domain=,d.name="
view.sort_current = trackers, "compare=,d.tracker_domain=,d.name="
```

Note: The following commands are part of the default `pyrocore` configuration!

d.data_path Return path to an item's data – this is never empty, unlike *d.base_path*. Multi-file items return a path ending with a `/`.

Definition:

```
method.insert = d.data_path, simple,\
    "if=(d.is_multi_file),\
        (cat, (d.directory), /),\
        (cat, (d.directory), /, (d.name))"
```

d.session_file Return path to session file.

Definition:

```
method.insert = d.session_file, simple, "cat=(session.path), (d.hash), .torrent"
```

d.tracker.bump_scrape Send a scrape request for an item, set its `tm_last_scrape` custom attribute to now, and save the session data. Part of auto-scape.rc, and bound to the `&` key in *rTorrent-PS*, to manually request a scrape update.

d.timestamp.downloaded

d.last_active **TODO**

*f.** commands

These commands can be used as arguments in a *f.multicall*. They can also be called directly, but you need to pass *‹infohash›:f‹index›* as the first argument. Index counting starts at `0`, the array size is *d.size_files*.

f.multicall

```
f.multicall = ‹infohash›, ‹pattern›, [‹cmd1›=[‹args›][, ‹cmd2›=...]]  list of␣
↪lists of results ‹rows of results›
```

Iterates over the files in an item, calling the given `f.*` commands. The second argument, if non-empty, is a glob-like pattern (e.g. `*.mkv`) and filters the result for matching filenames. That pattern matching is very simplistic, be cautious and test that you get the results you expect.

See also *d.multicall2* on basics regarding multi-calls.

f.completed_chunks

f.frozen_path

f.is_create_queued

f.is_created

f.is_open

f.is_resize_queued

f.last_touched

f.match_depth_next

f.match_depth_prev

f.offset

f.path

f.path_components

f.path_depth

f.prioritize_first

f.prioritize_first.disable

f.prioritize_first.enable

f.prioritize_last

f.prioritize_last.disable

f.prioritize_last.enable TODO

f.priority

f.priority.set TODO

See also *d.update_priorities*.

f.range_first

f.range_second

f.set_create_queued

f.set_resize_queued

f.size_bytes

f.size_chunks

f.unset_create_queued

f.unset_resize_queued TODO

*p.** commands

These commands can be used as arguments in a *p.multicall*. They can also be called directly, but you need to pass *‹infohash›:p‹id›* as the first argument. The ‹ id › is the peer ID as returned by *p.id*, encoded as a hexadecimal string.

p.multicall

```
p.multicall = ‹infohash›, "", [‹cmd1›=[‹args›][, ‹cmd2›=...]]  list of lists of␣
↪results ‹rows of results›
```

Iterates over the peers in an item, calling the given `p.*` commands.

The second argument is ignored, pass an empty string. See also *d.multicall2* on basics regarding multi-calls.

p.address

p.banned

p.banned.set

p.call_target

p.client_version

p.completed_percent

p.disconnect

p.disconnect_delayed

p.down_rate

p.down_total

p.id

p.id_html

p.is_encrypted

p.is_incoming

p.is_obfuscated

p.is_preferred

p.is_snubbed

p.is_unwanted

p.options_str

p.peer_rate

p.peer_total

p.port

p.snubbed

p.snubbed.set

p.up_rate

p.up_total TODO

*t.** commands

These commands can be used as arguments in a *t.multicall*. They can also be called directly, but you need to pass *‹infohash›:t‹index›* as the first argument. Index counting starts at `0`, the array size is *d.tracker_size*.

t.multicall

```
t.multicall = ‹infohash›, "", [‹cmd1›=[‹args›][, ‹cmd2›=...]]  list of lists of␣
↪results ‹rows of results›
```

Iterates over the trackers in an item, calling the given `t.*` commands.

The second argument is ignored, pass an empty string. See also *d.multicall2* on basics regarding multi-calls.

t.activity_time_last

t.activity_time_next

t.can_scrape

t.disable

t.enable

t.failed_counter

t.failed_time_last

t.failed_time_next

t.group

t.id

t.is_busy

t.is_enabled

t.is_enabled.set

t.is_extra_tracker

t.is_open

t.is_usable

t.latest_event

t.latest_new_peers

t.latest_sum_peers

t.min_interval

t.normal_interval

t.scrape_complete

t.scrape_counter

t.scrape_downloaded

t.scrape_incomplete

t.scrape_time_last

t.success_counter

t.success_time_last

t.success_time_next

t.type

t.url **TODO**

*load.** commands

The client may be configured to check a directory for new metafiles and load them. Items loaded in this manner will be tied to the metafile's path (see *d.tied_to_file*).

This means when the metafile is deleted, the item may be stopped (see *stop_untied*), and when the item is removed the metafile is also. Note that you can untie an item by using the `U` key (which will also delete the tied file), and using `Ctrl-K` also implictly unties an item.

load.normal

load.verbose

load.start

load.start_verbose **TODO** Synopsis

> Load a metafile or watch a pattern for new files to be loaded (in watch directory schedules).
>
> `normal` loads them stopped, and `verbose` reports problems to the console (like when a new file's infohash collides with an already loaded item).

TODO Post-load commands

load.raw

load.raw_start

load.raw_start_verbose

load.raw_verbose **TODO**

*session.** commands

session.name

session.name.set

session **TODO**

session.on_completion

session.on_completion.set **TODO**

session.path

session.path.set

```
session.path  string ‹path›
session.path.set = ‹path›
```

`session.path.set` specifies the location of the directory where *rTorrent* saves its status between starts – a command you should *always* have in your configuration.

It enables session management, which means the metafiles and status information for all open downloads will be stored in this directory. When restarting *rTorrent*, all items previously loaded will be restored. Only one instance of *rTorrent* should be used with each session directory, though at the moment no locking is done.

An empty string will disable session handling. Note that you cannot change to another directory while a session directory is already active.

session.save Flushes the full session state for all torrents to the related files in the session folder. Note that this can cause heavy IO with many torrents. The default interval this command runs at can be audjusted, however if *rTorrent* restarts or goes down, there may be a loss of statistics and resume data for any new torrents added after the last snapshot.

See also *d.save_full_session*, which saves the state of a single item.

session.use_lock

session.use_lock.set **TODO**

Scripting

*method.** commands

method.insert

```
method.insert = ‹name›, ‹type›[|‹sub-type›...][, ‹definition›]  0
```

The general way to define *any* kind of command. See *Object Types* for the possible values in the 2nd argument.

TODO more details

method.insert.simple

```
method.insert.simple = ‹name›, ‹definition›  0
```

This is a shortcut to define commands that are `simple` non-private functions.

method.insert.c_simple

```
method.insert.c_simple = ‹name›, ‹definition›  0
```

Defines a `const` simple function. **TODO** Meaning what?

method.insert.s_c_simple

```
method.insert.s_c_simple = ‹name›, ‹definition›  0
```

Defines a `static const` simple function. **TODO** Meaning what?

method.insert.value

```
method.insert.value = ‹name›, ‹default›  0
```

Defines a value that you can query and set, just like with any built-in value.

The example shows how to do optional logging for some new command you define, and also how to split a complicated command into steps using the `multi` method type.

```
# Enable verbose mode by setting this to 1
method.insert.value = sample.verbose, 0

# Do something with optional logging
method.insert = sample.action, multi|rlookup|static
method.set_key = sample.action, 10, ((print, "action"))
method.set_key = sample.action, 20, ((print, "action2"))
method.set_key = sample.action, 99,\
    ((branch, sample.verbose=,\
        "print=\"Some log message\""\
    ))
method.const.enable = sample.action
```

method.const

method.const.enable

```
method.const = ‹name›  bool (0 or 1)
method.const.enable = ‹name›  0
```

Set a method to immutable (or final). `method.const` queries whether a given command is. If you try to change a `const` method, you'll get an `Object is wrong type or const.` error.

See *method.insert.value* for an example.

method.erase Doesn't work, don't bother.

method.get

```
method.get = ‹name›  various (see text)
```

Returns the definition of a method, i.e. its current integer or string value, the definition for `simple` methods, or a dict of command definitions for `multi` methods. Querying any built-in method (a/k/a non-*dynamic* commands) results in a `Key not found.` fault.

The type of the definition can be either string or list, depending on whether `"..."` or `((...))` was used during insertion.

An example shows best what you get here, if you query the commands defined in the *method.insert.value* example, you'll get this:

```
$ rtxmlrpc --repr method.get '' sample.verbose
1

$ rtxmlrpc --repr method.get '' sample.verbose.set
ERROR    While calling method.get('', 'sample.verbose.set'): <Fault -503: 'Key␣
↪not found.'>

$ rtxmlrpc --repr method.get '' sample.action
{'10': ['print', 'action'],
 '20': ['print', 'action2'],
 '99': ['branch', 'sample.verbose=', 'print="Some log message"']}
```

`method.get` is also great to see what system handlers are registered. They often begin with a `!` or ~ to ensure they sort before / after any user-defined handlers.

```
$ rtxmlrpc --repr method.get '' event.download.closed
{'!view.indemand': 'view.filter_download=indemand',
 'log': 'print="CLOSED ",$d.name=," [",$convert.date=$system.time=,"]"'}
```

The `!view.‹viewname›` handler is added dynamically when you register it for an event using *view.filter_on*.

method.set TODO

method.set_key

method.has_key

method.list_keys

```
method.set_key = ‹name›, ‹key›[, ‹definition›]  0
method.has_key = ‹name›, ‹key›  bool (0 or 1)
method.list_keys = ‹name›  list of strings
```

Set entries in a `multi` method, query a single key, or list them all. If you omit the definition in a `method.set_key` call, the key is erased – it is safe to do that with a non-existing key.

`method.set_key` is commonly used to add handler commands to event types like *event.download.finished*. It can also be used to split complicated command definitions, see *method.insert.value* for an example.

method.rlookup

method.rlookup.clear TODO

method.redirect

```
method.redirect = ‹alias›, ‹target›  0
```

Defines an alias for an existing command, the arguments are command names. Aliases cannot be changed, using the same alias name twice causes an error.

*event.** commands

event.download.closed

event.download.erased

event.download.finished

event.download.hash_done

event.download.hash_failed

event.download.hash_final_failed

event.download.hash_queued

event.download.hash_removed

event.download.inserted

event.download.inserted_new

event.download.inserted_session

event.download.opened

event.download.paused

event.download.resumed TODO

event.view.hide

event.view.show

```
# rTorrent-PS 1.1+ only
event.view.hide = ‹new-view-name›  0
event.view.show = ‹old-view-name›  0
```

These events get called shortly before and after the download list canvas changes to a new view. Each gets passed the view name that is *not* available via *ui.current_view* at the time of the trigger, i.e. either the new or the old view name.

Be aware that during startup these view names can be *empty* strings!

Event handler example:

```
method.set_key = event.view.hide, ~log,\
    ((print, "× ", ((ui.current_view)), " → ", ((argument.0))))'
method.set_key = event.view.show, ~log,\
    ((print, " ", ((argument.0)), " → ", ((ui.current_view))))'
```

Scheduling Commands

The scheduling commands define tasks that call another command or list of commands repeatedly, just like a cron job, but with a resolution of seconds.

schedule2

```
schedule2 = ‹name›, ‹start›, ‹interval›, ((‹command›[, ‹args›...]))  0
schedule2 = ‹name›, ‹start›, ‹interval›, "‹command›=[‹args›...][ ; ‹command›=...]
↪"  0
```

Call the given command(s) every `interval` seconds, starting from `start`. An interval of zero calls the task once, while a start of zero calls it immediately. Currently command is forwarded to the option handler (*ed note*: whatever that means).

The `name` serves both as a handle for *schedule_remove2*, and as an easy way to document what this task actually does. Existing tasks can be changed at any time, just use the same name.

`start` and `interval` may optionally use a time format like `[dd:]hh:mm:ss`. An interval of `07:00:00:00` would mean weekly execution.

Examples:

```
# Watch directories
schedule2 = watch_start, 11, 10, ((load.start, (cat, (cfg.watch), "start/*.torrent
↪")))
schedule2 = watch_load,  12, 10, ((load.normal, (cat, (cfg.watch), "load/*.torrent
↪")))

# Add day break to console log
# → ( 0:00:00) New day: 20/03/2017
schedule2 = log_new_day, 00:00:00, 24:00:00,\
    "print=\"New day: \", (convert.date, (system.time))"

# ... or the equivalent using "new" syntax:
schedule2 = log_new_day, 00:00:05, 24:00:00,\
    ((print, "New day: ", ((convert.date, ((system.time_seconds)) )) ))
```

schedule_remove2

```
schedule_remove2 = ‹name›  0
```

Delete an existing task referenced by `name` from the scheduler. Deleting a non-existing task is not an error.

start_tied

stop_untied

close_untied

remove_untied TODO

close_low_diskspace TODO

Importing Script Files

import

try_import TODO

Conditions (if/then/else)

branch

if TODO

Conditional Operators

false TODO

and

or

not TODO

less

equal

greater

```
less = ‹cmd1›[, ‹cmd2›]  bool (0 or 1)
equal = ‹cmd1›[, ‹cmd2›]  bool (0 or 1)
greater = ‹cmd1›[, ‹cmd2›]  bool (0 or 1)
```

The comparison operators can work with strings or values (integers), returned from the given command(s). The most common form is with one provided command, that is then called for a target (e.g. with *view.filter*) or a target pair (e.g. *view.sort_new* or *view.sort_current*).

Consider this example, where items are sorted by comparing the names of target pairs, and the `less` command is called by a typical sorting algorithm:

```
view.sort_new     = name,((less,((d.name))))
view.sort_current = name,((less,((d.name))))
```

An example for a filter with two commands returning integer values is the `important` view, showing only items with a high priority:

```
view.add = important
ui.current_view.set = important
method.insert = prio_high, value|const|private, 3
view.filter = important, "equal=d.priority=,prio_high="
```

When two commands are given, their return types must match, and each command is called with the target (or the left / right sides of a target pair, respectively).

As you can see above, to compare against a constant you have to define it as a command. If you run *rTorrent-PS*, you can use *value* instead.

For strings, you can use *cat* as the command, and pass it the text literal.

```
view.filter = important, ((not, ((equal, ((d.throttle_name)), ((cat)) )) ))
view.filter = important, ((equal, ((d.throttle_name)), ((cat, NULL)) ))
```

Looks strange, like so many things in *rTorrent* scripting. The first filter shows all items that have *any* throttle set, i.e. have a non-empty throttle name. `((cat))` is the command that returns that empty string we want to compare against. The second filter selects items that have the special unlimited throttle `NULL` set.

elapsed.greater

elapsed.less

```
elapsed.greater = ‹start-time›, ‹interval›  bool (0 or 1)
elapsed.less = ‹start-time›, ‹interval›  bool (0 or 1)
```

Compare time elapsed since a given timestamp against an interval in seconds. The timestamps are UNIX ones, like created by *system.time_seconds*. The result is `false` if the timestramp is empty / zero.

Example:

```
method.insert.value = cfg.seed_seconds, 259200
schedule2 = limit_seed_time, 66, 300, "d.multicall.filtered = started,\
    \"elapsed.greater = (d.timestamp.finished), (cfg.seed_seconds)\",\
    d.try_stop="
```

What this does is stop any item finished longer than 3 days ago (selected via *d.multicall.filtered*), unless it is set to ignore commands (*d.try_stop* checks the ignore flag before stopping).

compare

```
# rTorrent-PS 0.*+ only
compare = ‹order›, ‹sort_key›=[, ...]  bool (0 or 1)
```

Compares two items like *less* or *greater*, but allows to compare by several different sort criteria, and ascending or descending order per given field.

The first parameter is a string of order indicators, either one of `aA+` for ascending or `dD-` for descending. The default, i.e. when there's more fields than indicators, is ascending.

Field types other than value or string are treated as equal (or in other words, they're ignored). If all fields are equal, then items are ordered in a random, but stable fashion.

Example (sort a view by message *and* name):

```
view.add = messages
view.filter = messages, ((d.message))
view.sort_new = messages, "compare=,d.message=,d.name="
```

string.contains

string.contains_i

```
# rTorrent-PS 1.1+ only
string.contains[_i]=«haystack»,«needle»[,...]  bool (0 or 1)
```

Checks if a given string contains any of the strings following it. The variant with `_i` is case-ignoring, but *only* works for pure ASCII needles.

Example:

```
$ rtxmlrpc d.multicall.filtered '' 'string.contains_i=(d.name),Mate' d.name=
['sparkylinux-4.0-x86_64-mate.iso']
```

String Functions

string.map

string.replace

```
# rTorrent-PS 1.1+ only
string.map=«text»,{«old»,«new»}[,...]  string
string.replace=«text»,{«old»,«new»}[,...]  string
```

`string.map` scans a list of replacement pairs for an `old` text that matches *all* of the given string, and replaces it by `new`.

`string.replace` substitutes any occurence of the old text by the new one.

Example:

```
$ rtxmlrpc string.map '' 'foo' [foo,bar [bar,baz
baz

$ rtxmlrpc string.replace '' "it's like 1" [1,2ic [2,ma3 [3,g
it's like magic
```

```
$ rtxmlrpc -i 'print = (string.map, (cat, (value,1)), {0,off}, {1,low}, {2,""},
↪{3,high})'
# prints 'low' as a console message, this is how you map integers
```

Value Conversion & Formatting

The `to_*` forms are **deprecated**.

convert.kb

convert.mb

convert.xb

to_kb

to_mb

to_xb TODO

convert.date

convert.elapsed_time

convert.gm_date

convert.gm_time

convert.time

to_date

to_elapsed_time

to_gm_date

to_gm_time

to_time TODO

convert.throttle

to_throttle TODO

convert.human_size

```
# rTorrent-PS 1.1+ only
convert.human_size = ‹bytes›[, ‹format›]  string
```

Converts a size in bytes to a compact, human readable string. See also *convert.xb* for a similar command.

Format is a number (default 2), with these values:

- `0`: use 6 chars (one decimal place)
- `1`: just print the rounded value (4 chars)
- `2`: combine the two formats into 4 chars by rounding for values >= 9.95
- `+8`: adding 8 converts zero values to whitespace of the correct length

Examples:

```
$ rtxmlrpc --repr convert.human_size '' +970 +0
'  0.9K'
$ rtxmlrpc --repr convert.human_size '' +970 +1
'  1K'
$ rtxmlrpc --repr convert.human_size '' +970 +10
'0.9K'
$ rtxmlrpc --repr convert.human_size '' +0 +2
'0.0K'
$ rtxmlrpc --repr convert.human_size '' +0 +10
'    '
```

convert.magnitude

```
# rTorrent-PS 1.1+ only
convert.magnitude = ‹number›  string
```

Converts any positive number below 10 million into a very compact string representation with only 2 characters. Above 99, only the first significant digit is retained, plus an order of magnitude indicator using roman numerals (c = 10^2, m = 10^3, X = 10^4, C = 10^5, M = 10^6). Zero and out of range values are handled special (see examples below).

Examples:

```
$ rtxmlrpc convert.magnitude '' +0
 .
$ rtxmlrpc convert.magnitude '' +1
 1
$ rtxmlrpc convert.magnitude '' +99
99
$ rtxmlrpc convert.magnitude '' +100
1c
$ rtxmlrpc convert.magnitude '' +999
9c
$ rtxmlrpc convert.magnitude '' +1000
1m
$ rtxmlrpc convert.magnitude '' +9999999
9M
$ rtxmlrpc convert.magnitude '' +10000000

$ rtxmlrpc -- convert.magnitude '' -1
```

value

```
# rTorrent-PS 1.1+ only
value = ‹number›[, ‹base›]  value
```

Converts a given number with the given base (or 10 as the default) to an integer value.

Examples:

```
$ rtxmlrpc -qi 'view.filter = rtcontrol, "equal = d.priority=, value=3"'
# the 'rtcontrol' view will now show all items with priority 'high'
$ rtxmlrpc --repr value '' 1b 16
27
$ rtxmlrpc --repr value '' 1b
ERROR    While calling value('', '1b'): <Fault -503: 'Junk at end of number: 1b'>
```

Logging, Files, and OS

*execute.** commands

Call operating system commands, possibly catching their output for use within *rTorrent*.

Note: The `.bg` variants detach the child process from the *rTorrent* parent, i.e. it runs in the background. This **must** be used if you want to call back into *rTorrent* via XMLRPC, since otherwise there *will* be a deadlock.

`throw` means to raise an error when the called command fails, while the `nothrow` variants will return the exit code.

---

execute.throw

execute.throw.bg

execute2

```
execute.throw[.bg] = {command, arg1, arg2, ...}  0
```

This will execute a system command with the provided arguments. These commands either raise an error or return `0`. `execute2` is the same as `execute.throw`, and should be avoided.

Since internally `spawn` is used to call the OS command, the shell is not involved and things like shell redirection will not work here. There is also no reason to use shell quoting in arguments, just separate them by commas. If you need shell features, call `bash -c "‹command›"` like shown in this example:

```
# Write a PID file into the session directory
execute.throw = bash, -c, (cat, "echo >", (session.path), "rtorrent.pid", " ",␣
↪(system.pid))
```

Note that the result of the `(cat, ...)` command ends up as a *single* argument passed on to `bash`.

execute.nothrow

execute.nothrow.bg

```
execute.nothrow[.bg] = {command, arg1, arg2, ...}  value ‹exit status›
```

Like *execute.throw*, but return the command's exit code (*warning:* due to a bug the return code is shifted by 8 bits, so `1` becomes `0x100`).

The `.bg` variant will just indicate whether the child could be successfully spawned and detached.

execute.capture

execute.capture_nothrow

```
execute.capture[_nothrow] = {command, arg1, arg2, ...}  string ‹stdout›
```

Like `execute.[no]throw`, but returns the command's standard output. The `nothrow` variant returns any output that was written before an error, in case one occurs. The exit code is never returned.

Note that any line-endings are included, so if you need a plain string value, wrap the command you want to call into an `echo -n` command:

```
method.insert = log_stamp, private|simple,\
    "execute.capture_nothrow = bash, -c, \"echo -n $(date +%Y-%m-%d-%H%M%S)\""
```

execute.raw

execute.raw.bg

execute.raw_nothrow

execute.raw_nothrow.bg **TODO**

*system.** commands

Commands related to the operating system and the XMLRPC API.

system.listMethods

system.methodExist

system.methodHelp

system.methodSignature

system.capabilities

system.getCapabilities **TODO**

system.multicall **TODO**

system.shutdown **TODO**

system.api_version

system.client_version

system.library_version **TODO**

system.colors.enabled

system.colors.max

system.colors.rgb **TODO**

system.cwd

system.cwd.set **TODO**

system.env

```
# 0.9.7+ / rTorrent-PS 0.*+ only
system.env = ‹varname›  string ‹env-value›
```

Query the value of an environment variable, returns an empty string if `$varname` is not defined.

Example:

```
session.path.set = (cat, (system.env, RTORRENT_HOME), "/.session")
```

system.file.allocate

system.file.allocate.set **TODO**

system.file.max_size

system.file.max_size.set **TODO**

system.file.split_size

system.file.split_size.set

system.file.split_suffix

system.file.split_suffix.set TODO

system.file_status_cache.prune

system.file_status_cache.size TODO

system.files.closed_counter

system.files.failed_counter

system.files.opened_counter TODO

system.hostname TODO

system.pid TODO

system.random

```
# rTorrent-PS 1.0+ only
system.random = [[‹lower›,] ‹upper›]  value
```

Generate *uniformly* distributed random numbers in the range defined by `lower ... upper`.

The default range with no args is `0 ... RAND_MAX`. Providing just one argument sets an *exclusive* upper bound, and two args define an *inclusive* range.

An example use-case is adding jitter to time values that you later check with *elapsed.greater*, to avoid load spikes and similar effects of clustered time triggers.

system.time

system.time_seconds

system.time_usec TODO

system.umask.set TODO

*log.** commands

log.add_output

```
log.add_output = ‹scope›, ‹name›  0
```

This command adds another logging scope to a named log file, opened by one of the *log.open_file* commands.

Log messages are classified into groups (`connection`, `dht`, `peer`, `rpc`, `storage`, `thread`, `torrent`, and `tracker`), and have a level of `critical`, `error`, `warn`, `notice`, `info`, or `debug`.

Scopes can either be a whole level, or else a group on a specific level by using `‹group›_‹level›` as the scope's name.

Example:

```
log.add_output = tracker_debug, tracelog
```

log.execute

log.xmlrpc TODO

log.open_file

log.open_gz_file

log.open_file_pid

log.open_gz_file_pid

```
log.open_file = ‹name›, ‹log file path›[, ‹scope›...]  0
log.open_gz_file
log.open_file_pid
log.open_gz_file_pid
```

All these commands open a log file, giving it a name to refer to. Paths starting with ~ are expanded. You can immediately add some logging scopes, see *log.add_output* for details on those.

The `pid` variants add the PID of *rTorrent* at the end of the file name (see *Log Rotation, Archival, and Pruning* for a way better scheme for log separation). Adding `gz` opens the logfile directly as a compressed streams, note that you have to add an appropriate extension yourself.

There is an arbitary limit on the number of log streams you can open (64 in 0.9.6). The core of the logging subsystem is implemented in `torrent/utils/log` of *libtorrent*.

You can re-open existing logs in *rTorrent-PS* 1.1+ (and maybe in *rTorrent* 0.9.7+), by just calling an open command with a new path. To 'close' one, bind it to `/dev/null`.

Example:

```
log.open_file_pid = tracker, /tmp/tracker.log, tracker_debug
# ... opens '/tmp/tracker.log.NNNNN' for debugging tracker announces etc.
```

Warning: Compressed log files do not seem to work, in version 0.9.6 at least.

log.vmmap.dump

```
log.vmmap.dump = ‹dump file path›  0
```

Dumps all memory mapping regions to the given file, each line contains a region in the format `‹begin›-‹end› [‹size in KiB›k]`.

log.messages

```
# rTorrent-PS 0.*+ only
log.messages = ‹log file path›  0
```

(Re-)opens a log file that contains the messages normally only visible on the main panel and via the `l` key. Each line is prefixed with the current date and time in ISO8601 format. If an empty path is passed, the file is closed.

Example:

```
log.messages = (cat, (cfg.logs), "messages.log")
```

Network (Sockets, HTTP, XMLRPC)

*network.** commands

network.bind_address

network.bind_address.set TODO

network.http.dns_cache_timeout

network.http.dns_cache_timeout.set

```
network.http.dns_cache_timeout.set = ‹seconds›  0
network.http.dns_cache_timeout  ‹seconds›
```

Controls the DNS cache expiry (in seconds) for HTTP requests. The default is 60 seconds.

Set to zero to completely disable caching, or set to -1 to make the cached entries remain forever.

network.http.current_open

network.http.max_open

network.http.max_open.set

```
network.http.current_open  value ‹num›
network.http.max_open  value ‹max›
network.http.max_open.set = ‹max›  0
```

`network.http.current_open` returns the number of currently opened HTTTP connections, and `network.http.max_open` determines the upper limit for simultaneous HTTP connections.

Be wary of setting this too high, as even if your connection can support that many requests, the target host may not be able to respond quickly enough, leading to timeouts.

network.http.proxy_address

network.http.proxy_address.set TODO

network.http.cacert

network.http.cacert.set

network.http.capath

network.http.capath.set TODO

network.http.ssl_verify_host

network.http.ssl_verify_host.set

network.http.ssl_verify_peer

network.http.ssl_verify_peer.set TODO

network.listen.backlog

network.listen.backlog.set

network.listen.port TODO

network.local_address

network.local_address.set TODO

network.max_open_files

network.max_open_files.set TODO

network.max_open_sockets

network.max_open_sockets.set

network.open_sockets TODO

network.port_open

network.port_open.set

network.port_random

network.port_random.set

network.port_range

network.port_range.set TODO

network.proxy_address

network.proxy_address.set TODO

network.receive_buffer.size

network.receive_buffer.size.set

network.send_buffer.size

network.send_buffer.size.set

```
network.receive_buffer.size  value ‹size›
network.receive_buffer.size.set = ‹size›  0
network.send_buffer.size  value ‹size›
network.send_buffer.size.set = ‹size›  0
```

Sets or gets the maximum socket receive / send buffer in bytes.

On Linux, the default buffer size for receiving data is set by the `/proc/sys/net/core/rmem_default` file (`wmem_default` for sending). The maximum allowed value is set by the `/proc/sys/net/core/rmem_max` file (`wmem_max` for sending).

See the tuning guide for tweaking these values

network.scgi.dont_route

network.scgi.dont_route.set

```
network.scgi.dont_route  bool (0 or 1)
network.scgi.dont_route.set = ‹bool›  0
```

Enable / disable routing on SCGI connections, directly calling setsockopt to modify the `SO_DONTROUTE` flag.

network.scgi.open_local

network.scgi.open_port

```
network.scgi.open_local = ‹path›  0
network.scgi.open_port = ‹port›  0
```

Open up a TCP port or a Unix domain socket for SCGI communication (i.e. the XMLRPC socket). Only use *one* of these!

Note: Using `network.scgi.open_port` means *any* user on the machine you run *rTorrent* on can execute *arbitrary* commands with the permission of the *rTorrent* runtime user. Most people don't realize that, now you do! Also, **never** use any other address than `127.0.0.1` with it.

---

network.tos.set

```
network.tos.set = ‹flag›  0
```

Set the type of service flag to use in IP packets.

The options as pulled from *strings.ip_tos* are:

- default
- lowdelay
- throughput
- reliability
- mincost

default uses the system default setting. A raw hexadecimal value can also be passed in for custom flags.

network.xmlrpc.dialect.set

```
network.xmlrpc.dialect.set = ‹dialect [value 0...2]›  0
```

Set the XMLRPC dialect to use, as defined by xmlrpc-c. The dialect parameter can have these values:

- 0: dialect_generic
- 1: dialect_i8
- 2: dialect_apache

dialect_i8 is the default value, which means the XMLRPC API will use the xmlrpc-c i8 extension type for returning long integers.

See its documentation for more information on how xmlrpc-c handles dialects.

network.xmlrpc.size_limit

network.xmlrpc.size_limit.set

```
network.xmlrpc.size_limit =  value ‹bytes›
network.xmlrpc.size_limit.set = ‹max-size›  0
```

Set or return the maximum size of any XMLRPC requests in bytes. Human-readable forms such as 2M are also allowed (for 2 MiB, i.e. 2097152 bytes).

Note: The following are only available in *rTorrent-PS*!

---

network.history.auto_scale

network.history.auto_scale.set

network.history.depth

network.history.depth.set

network.history.refresh

network.history.sample Commands to add network traffic charts to the bottom of the collapsed download display.

Add these lines to your configuration:

```
# rTorrent-PS 0.** only!

# Show traffic of the last hour (112*32 = 3584  3600)
network.history.depth.set = 112

method.insert = network.history.auto_scale.toggle, simple|private,\
    "branch=(network.history.auto_scale),\
        ((network.history.auto_scale.set, 0)),\
        ((network.history.auto_scale.set, 1))"
method.insert = network.history.auto_scale.ui_toggle, simple|private,\
    "network.history.auto_scale.toggle= ; network.history.refresh="
```

```
schedule2 = network_history_sampling, 1, 32, "network.history.sample="
schedule2 = bind_auto_scale, 0, 0,\
    "ui.bind_key=download_list, =, network.history.auto_scale.ui_toggle="
```

This will add the graph above the footer, you get the upper and lower bounds of traffic within your configured time window, and each bar of the graph represents an interval determined by the sampling schedule. Pressing = toggles between a graph display with base line 0, and a zoomed view that scales it to the current bounds.

*ip_tables.** commands

ip_tables.add_address

ip_tables.get

ip_tables.insert_table

ip_tables.size_data TODO

*ipv4_filter.** commands

ipv4_filter.add_address

ipv4_filter.dump

ipv4_filter.get

ipv4_filter.load

ipv4_filter.size_data TODO

Bittorrent Protocol

*dht.** commands

dht.add_node TODO

dht.mode.set

dht TODO

dht.port

dht.port.set

dht_port TODO

dht.statistics Returns `{'active': 0, 'dht': 'disable', 'throttle': ''}` when DHT is off, and ...

TODO

dht.throttle.name

dht.throttle.name.set TODO

*pieces.** commands

pieces.hash.on_completion

pieces.hash.on_completion.set

pieces.hash.queue_size

pieces.memory.block_count

pieces.memory.current

pieces.memory.max

pieces.memory.max.set

pieces.memory.sync_queue

pieces.preload.min_rate

pieces.preload.min_rate.set

pieces.preload.min_size

pieces.preload.min_size.set

pieces.preload.type

pieces.preload.type.set

pieces.stats.total_size

pieces.stats_not_preloaded

pieces.stats_preloaded

pieces.sync.always_safe

pieces.sync.always_safe.set

pieces.sync.queue_size

pieces.sync.safe_free_diskspace

pieces.sync.timeout

pieces.sync.timeout.set

pieces.sync.timeout_safe

pieces.sync.timeout_safe.set **TODO**

*protocol.** commands

protocol.choke_heuristics.down.leech

protocol.choke_heuristics.down.leech.set

protocol.choke_heuristics.down.seed

protocol.choke_heuristics.down.seed.set

protocol.choke_heuristics.up.leech

protocol.choke_heuristics.up.leech.set

protocol.choke_heuristics.up.seed

protocol.choke_heuristics.up.seed.set **TODO**

protocol.connection.leech

protocol.connection.leech.set

protocol.connection.seed

protocol.connection.seed.set **TODO**

protocol.encryption.set **TODO**

See also BitTorrent protocol encryption.

protocol.pex

protocol.pex.set **TODO**

*throttle.** commands

Throttles are names for bandwidth limitation rules (for upload, download, or both). The throttle assigned to the item in focus can be changed using `Ctrl-T` – it will rotate through all defined ones.

There are two system throttles, `NULL` and the one with an empty name. `NULL` is a special throttle for *unlimited*, and the latter is the *global* throttle, which is the default for new items and what's shown in the status bar on the left as `[Throttle ‹UP›/‹DOWN› KB]`.

TODO Explain how throttles work, borrowing from the global throttle.

Other commands in this group determine the limits for upload / download slots, and the amount of peers requested in tracker announces.

Warning: Note that since named throttles *borrow* from the global throttle, the global one has to be set to a non-zero value for the named ones to work (because borrowing from ∞ means there is no limit).

throttle.down

throttle.up

```
throttle.down = ‹name›, ‹rate›  0
throttle.up = ‹name›, ‹rate›  0
```

Define a named throttle. The `rate` must be a string (important when using XMLRPC), and is always in KiB/s.

You can also set a new rate for existing throttles this way (i.e. repeated definitions are no error).

throttle.down.max

throttle.up.max

```
throttle.down.max = ‹name›  value ‹limit›
throttle.up.max = ‹name›  value ‹limit›
```

Get the current limit of a named throttle in bytes/s.

Unknown throttles return `-1`, unlimited ones `0`. If the global throttle is not set, you also get `0` for any call.

throttle.down.rate

throttle.up.rate

```
throttle.down.rate = ‹name›  value ‹rate›
throttle.up.rate = ‹name›  value ‹rate›
```

Get the current rate of a named throttle in bytes/s, averaged over recent history.

Unknown throttles always return `0`. If the global throttle is not set, you also get `0` for any call.

throttle.global_down.max_rate

throttle.global_down.max_rate.set

throttle.global_down.max_rate.set_kb

throttle.global_up.max_rate

throttle.global_up.max_rate.set

throttle.global_up.max_rate.set_kb Query or change the current value for the global throttle. Always use `set_kb` to change these values (the `set` commands have bugs), and be aware that you always get bytes/s when querying them.

throttle.global_down.rate

throttle.global_up.rate

```
throttle.global_down.rate  value ‹rate›
throttle.global_up.rate  value ‹rate›
```

Current overall bandwidth usage in bytes/s, averaged over recent history.

throttle.global_down.total

throttle.global_up.total

```
throttle.global_down.total  value ‹bytes›
throttle.global_up.total  value ‹bytes›
```

Amount of data moved over all items, in bytes.

TODO . . . in this session, including deleted items?

throttle.max_downloads

throttle.max_downloads.set

throttle.max_downloads.div

throttle.max_downloads.div.set

throttle.max_downloads.div._val

throttle.max_downloads.div._val.set

throttle.max_uploads

throttle.max_uploads.set

throttle.max_uploads.div

throttle.max_uploads.div.set

throttle.max_uploads.div._val

throttle.max_uploads.div._val.set **TODO**

throttle.max_downloads.global

throttle.max_downloads.global.set

throttle.max_downloads.global._val

throttle.max_downloads.global._val.set

throttle.max_uploads.global

throttle.max_uploads.global.set

throttle.max_uploads.global._val

throttle.max_uploads.global._val.set **TODO**

throttle.min_downloads

throttle.min_downloads.set

throttle.min_uploads

throttle.min_uploads.set **TODO**

throttle.max_peers.normal

throttle.max_peers.normal.set

throttle.max_peers.seed

throttle.max_peers.seed.set

throttle.min_peers.normal

throttle.min_peers.normal.set

throttle.min_peers.seed

throttle.min_peers.seed.set **TODO**

throttle.unchoked_downloads

throttle.unchoked_uploads **TODO**

throttle.ip

```
throttle.ip = ‹throttle name›, ‹IP or domain name›  0
```

Throttle a specific peer by its IP address.

User Interface

*ui.** commands

Commands in this group control aspects of the 'curses' UI.

ui.current_view

ui.current_view.set

```
ui.current_view  string ‹viewname›
ui.current_view.set = ‹viewname›  0
```

Query or change the current view the user is seeing (querying since `0.9.7`). *view.list* gives you a list of all the added views.

Typical uses are to change and then restore the active view, or rotate through a set of views. Rotating through views requires querying the current view and the view list, to find the next one.

In *rTorrent-PS* 1.1+, view changes trigger event handlers for *event.view.hide* and *event.view.show*.

ui.torrent_list.layout

ui.torrent_list.layout.set Offers a choice between `full` and `compact` layout (since `0.9.7`).

ui.unfocus_download Used internally before erasing an item, to move the focus away from it.

Note: The following are only available in *rTorrent-PS*!

---

ui.bind_key

ui.bind_key.verbose

ui.bind_key.verbose.set

```
# rTorrent-PS 0.*+ only
ui.bind_key = display, key, "command=[...]"  0
# rTorrent-PS 1.1+ only
ui.bind_key.verbose =  bool (0 or 1)
```

Binds the given key on a specified display to execute the given command when pressed. Note that this needs to be called in a one-shot schedule, after *rTorrent* is fully initialized.

`display` must always be `download_list`, for the moment.

`key` can be either a single character for normal keys, `^` plus a character for control keys, or a 4 digit octal code for special keys.

The *ui.bind_key.verbose* flag determines whether replacing an existing binding is logged (`1`, the default) or not (`0`).

Configuration example:

```
# Bind '^' to show the "rtcontrol" result
schedule2 = bind_view_rtcontrol, 1, 0,\
    "ui.bind_key = download_list, ^, ui.current_view.set=rtcontrol"
```

ui.color.alarm

ui.color.complete

ui.color.even

ui.color.focus

ui.color.footer

ui.color.incomplete

ui.color.info

ui.color.label

ui.color.leeching

ui.color.odd

ui.color.progress0

ui.color.progress20

ui.color.progress40

ui.color.progress60

ui.color.progress80

ui.color.progress100

ui.color.progress120

ui.color.queued

ui.color.seeding

ui.color.stopped

ui.color.title

ui.color.‹type›.set

```
# rTorrent-PS 0.*+ only
ui.color.‹type›=  string ‹color-spec›
ui.color.‹type›.set=‹color-spec›  0
```

These commands allow you to set colors for selected elements of the user interface in *rTorrent-PS*, in some cases depending on their status. You can either provide colors by specifying the numerical index in the terminal's color table, or by name (for the first 16 colors). The possible color names are "black", "red", "green", "yellow", "blue", "magenta", "cyan", "gray", and "white"; you can use them for both text and background color, in the form "«fg» on «bg»", and you can add "bright" in front of a color to select a more luminous version. If you don't specify a color, the default of your terminal is used.

Also, these additional modifiers can be placed in the color definitions, but it depends on the terminal you're using whether they have an effect: "bold", "standout", "underline", "reverse", "blink", and "dim".

See the color scheme for 256 xterm colors for an example.

ui.focus.end

ui.focus.home

ui.focus.pgdn

ui.focus.pgup

ui.focus.page_size

ui.focus.page_size.set

```
# rTorrent-PS 0.*+ only
```

TODO

ui.style.progress

ui.style.progress.set

ui.style.ratio

ui.style.ratio.set

```
# rTorrent-PS 0.*+ only
```

TODO

*view.** commands

view.add

view.list

view.size

view.persistent **TODO**

view.event_added

view.event_removed TODO

view.filter

view.filter_all

view.filter_download

view.filter_on TODO

view.set

view.set_visible

view.set_not_visible

view.size_not_visible TODO

view.sort

view.sort_current

view.sort_new TODO

view.collapsed.toggle

```
# rTorrent-PS 0.*+ only
view.collapsed.toggle=‹view-name›  0
```

This command changes between the normal item display, where each item takes up three lines, to a more condensed form exclusive to *rTorrent-PS*, where each item only takes up one line.

Note that each view has its own state, and that if the view name is empty, the current view is toggled. You can set the default state in your configuration, by adding a toggle command for each view you want collapsed after startup (the default is expanded).

Miscellaneous

*strings.** commands

strings.choke_heuristics

- `upload_leech`
- `upload_leech_dummy`
- `download_leech`
- `download_leech_dummy`
- `invalid`

strings.choke_heuristics.download

- `download_leech`
- `download_leech_dummy`

strings.choke_heuristics.upload

- `upload_leech`
- `upload_leech_dummy`

strings.connection_type

- `leech`
- `seed`

- `initial_seed`
- `metadata`

strings.encryption

- `none`
- `allow_incoming`
- `try_outgoing`
- `require`
- `require_RC4`
- `require_rc4`
- `enable_retry`
- `prefer_plaintext`

strings.ip_filter

- `unwanted`
- `preferred`

strings.ip_tos

- `default`
- `lowdelay`
- `throughput`
- `reliability`
- `mincost`

Options for *network.tos.set*.

strings.tracker_mode

- `normal`
- `aggressive`

TODO (Groups)

- choke_group
- fi
- file
- group
- group2
- keys
- ratio
- scheduler

directory.default

directory.default.set

directory

encoding.add

encoding_list TODO

trackers.disable

trackers.enable

trackers.numwant

trackers.numwant.set

trackers.use_udp

trackers.use_udp.set TODO

trackers.alias.items

trackers.alias.set_key TODO

TODO (singles)

cat

print

add_peer

bind

catch

check_hash

connection_leech

connection_seed

download_rate

encoding_list

encryption

ip

key_layout

max_downloads

max_downloads_div

max_downloads_global

max_memory_usage

max_peers

max_peers_seed

max_uploads

max_uploads_div

max_uploads_global

min_downloads

min_peers

min_peers_seed

min_uploads

on_ratio

port_random

port_range

proxy_address

scgi_local

scgi_port

torrent_list_layout

upload_rate TODO

'Intermediate' Commands

The *intermediate* commands are kept around as aliases for 'new' ones – at least for the time being. Probably best avoided.

Avoiding the *deprecated* commands is a must, these will disappear at some time.

method.use_deprecated

method.use_deprecated.set

```
method.use_deprecated  bool (0 or 1)
method.use_deprecated.set = ‹0 or 1›  bool ‹current› (0 or 1)
```

The default is `true`. The undocumented `-D` command line options sets this to `false` with a ''Disabled deprecated commands'' console message.

method.use_intermediate

method.use_intermediate.set

```
method.use_intermediate  value (0 ... 2)
method.use_intermediate.set = ‹0 ... 2›  value ‹current› (0 ... 2)
```

The default is 1 (allow everywhere), values other than 1 or 2 are treated like 0. The undocumented `-I` command line options sets this to 0 with a ''Disabled intermediate commands'' console message, while `-K` sets it to 2, printing `Allowing intermediate commands without xmlrpc`.

All the command aliases can be found in these three source files: `command_local.cc`, `command_throttle.cc`, and `main.cc`. Search for `REDIRECT` using `grep`.

These are called *intermediate:*

- `execute` → `execute2` (ignore both, just use `execute.throw`)
- `schedule` → `schedule2`
- `schedule_remove` → `schedule_remove2`
- `group.‹name›.view` → `group2.‹name›.view`
- `group.‹name›.view.set` → `group2.‹name›.view.set`
- `group.‹name›.ratio.min` → `group2.‹name›.ratio.min`
- `group.‹name›.ratio.min.set` → `group2.‹name›.ratio.min.set`
- `group.‹name›.ratio.max` → `group2.‹name›.ratio.max`
- `group.‹name›.ratio.max.set` → `group2.‹name›.ratio.max.set`
- `group.‹name›.ratio.upload` → `group2.‹name›.ratio.upload`

- `group.‹name›.ratio.upload.set` → `group2.‹name›.ratio.upload.set`

Contributing Guidelines

See contribution-guide.org for the basics on contributing to an open source project.

Reporting an Error, or Requesting an Addition

Any corrections and change requests are managed using GitHub's issue tracker. If you never opened an issue on GitHub before, consult the Mastering Issues guide.

Adding Your Own Contributions

The handbook is rendered to HTML using the Sphinx tool, the text itself is written using reStructuredText markup.

The easiest way to edit text is using GitHub's built-in editor. When you click the edit button (pencil), it will fork the project for you and then open the rich-text web editor. It is very **convenient to fix small spelling or grammatical errors** on the fly, while you're reading the handbook.

If you're reading this on *Read the Docs*, take note of the *"Edit on GitHub"* button in the top-right corner of each page, which will take you to the under-lying text file on GitHub.

The GitHub editor has a *Preview* button (use it) and can save directly into a so-called 'pull request' (PR) for integration into the project. Please do *not* save changes into a PR just to "try it out", you can however save changes into your fork at your pleasure.

The **more technical but also more powerful way** is to clone the project to your machine. Here are some resources to help you with getting started, if you never wrote anything for a `Sphinx` document before:

- the Sphinx reStructuredText Primer explains the text markup language used.
- the project's README shows you how to build the handbook on your own machine.

After you wrote, spell-checked and reviewed your text, open a pull request as explained in the GitHub help. Try to keep PRs at a reasonable size, ideally only changing one file, especially when it comes to the command reference. This reduces the potential of merge conflicts and rework, and also makes reviews take a manageable amount of time.

CHAPTER 2

Indices & Tables

- genindex
- search

Index

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V