Tribler Documentation

Release 7.5.0

Tribler devs

Jan 29, 2021

CONTENTS

Contents:

CHAPTER

ONE

TRIBLER

Towards making Bittorrent anonymous and impossible to shut down.

We use our own dedicated Tor-like network for anonymous torrent downloading. We implemented and enhanced the *Tor protocol specifications* plus merged them with Bittorrent streaming. More info: https://github.com/Tribler/tribler/wiki Tribler includes our own Tor-like onion routing network with hidden services based seeding and end-to-end encryption, detailed specs: https://github.com/Tribler/tribler/wiki/Anonymous-Downloading-and-Streaming-specifications

The aim of Tribler is giving anonymous access to online (streaming) videos. We are trying to make privacy, strong cryptography and authentication the Internet norm.

Tribler currently offers a Youtube-style service. For instance, Bittorrent-compatible streaming, fast search, thumbnail previews and comments. For the past 11 years we have been building a very robust Peer-to-Peer system. Today Tribler is robust: “the only way to take Tribler down is to take The Internet down” (but a single software bug could end everything).

We make use of submodules, so remember using the –recursive argument when cloning this repo.

1.1 Obtaining the latest release

Just click here and download the latest package for your OS.

1.2 Obtaining support

If you found a bug or have a feature request, please make sure you read our contributing page and then open an issue. We will have a look at it ASAP.

1.3 Contributing

Contributions are very welcome! If you are interested in contributing code or otherwise, please have a look at our contributing page. Have a look at the issue tracker if you are looking for inspiration :).

1.3.1 Running Tribler from the repository

First clone the repository:

```
git clone --recursive https://github.com/Tribler/tribler.git
```

Second, install the dependencies.

Setting up your development environment

We support development on Linux, macOS and Windows. We have written documentation that guides you through installing the required packages when setting up a Tribler development environment.

- Linux
- Windows
- macOS

Running

Now you can run tribler by executing the `tribler.sh` script on the root of the repository:

```
./src/tribler.sh
```

On Windows, you can use the following command to run Tribler:

```
python run_tribler.py
```

1.4 Packaging Tribler

We have written guides on how to package Tribler for distribution on various systems. Please take a look here.

1.5 Submodule notes

- As updated submodules are in detached head state, remember to check out a branch before committing changes on them.
- If you forgot to check out a branch before doing a commit, you should get a warning telling you about it. To get the commit to a branch just check out the branch and do a git cherry-pick of the commit.
- Take care of not accidentally committing a submodule revision change with `git commit -a`.
- Do not commit a submodule update without running all the tests first and making sure the new code is not breaking Tribler.

CHAPTER

TWO

HOW TO CONTRIBUTE TO THE TRIBLER PROJECT?

2.1 Checking out the Stabilization Branch

The stabilization branch `release-X.Y.Z` contains the most up to date bugfixes. If your issue cannot be reproduced there, it is most likely already fixed.

To backup your Tribler installation and checkout the latest version of the stabilization branch, please perform the following steps. * Copy the `.Tribler` folder to a safe location on your system (for instance the desktop) Make sure to leave the original folder on its original location. This folder is located at `~/.Tribler/` (Linux/OS X) or `%APPDATA\.Tribler` (Windows). * Remove the `tribler` installation folder. * Go to the latest tested version of Tribler and under 'Build Artifacts', download the package appropriate to your operating system. * Install/unzip this package.

To revert back to your original version of Tribler, download the installer again and install it. Afterwards you can restore your backed up Tribler data folder.

2.2 Reporting bugs

- Make sure the issue/feature you want to report doesn't already exist.
- If you want to report more than one bug or feature, create individual issues for each of them.
- Use a clear descriptive title.
- **Provide at least the following information:**
  - The version of Tribler that you are using, if you are running from a branch, branch name and commit ID.
  - The OS and version you are running.
  - Step by step instructions to reproduce the issue in case it's a bug.
- **Attach Tribler's log file. On Windows, these are found in `%APPDATA%`. On Linux distributions, the log file is located in `~/`**
  - Does it still happen if you move `%APPDATA%\.Tribler` away temporarily? (Do **not** delete it!)
  - Do you have any other software installed that might interfere with Tribler?

2.3 Pull requests

When creating a new Pull request, please take note of the following:

- New features always go to `devel`.
- If there is an unreleased `release-X.Y.Z` branch, fixes go there.
- Otherwise, fixes go to `devel`.
- Before starting to work on a feature or fix, check that nobody else is working on it by assigning yourself the corresponding issue. Create one if it doesn't exist. This is also useful to get feedback about if a given feature would be accepted. If you are not a member of the project, just drop a comment saying that you are working on that.
- Create one PR per feature/bugfix.
- Provide tests for any new features/fixes you implement and make sure they cover all methods and at least the important branches in the new/updated code. We use the pytest framework, and use fixtures to specify the components enabled in each test by using fixtures. Please do not enable components that are not strictly necessary for the test you are writing (e.g., libtorrent) as they will slow down the duration of test execution.
- If implementing a reasonably big or experimental feature, make it toggleable if possible (For instance for a new community, new GUI stuff, etc.).
- When you make a change to the user interface, please attach a screenshot of your changes to the pull request. This helps reviewers since it avoids the need for them to manually checkout your code to see what has been changed.
- **Keep a clean and nice git history:**
  - Rebase instead of merging back from the base branch.
  - Squash fixup commits together.
  - Have nice and descriptive commit messages.
- Do not commit extraneous/auto-generated files.
- Use Unix style newlines for any new file created.
- No print statements if it's not really justified (command line tools and such).
- Do an `autopep8` pass before submitting the pull request.
- Do a `pylint` pass with the `.pylintrc` on the root of the repository and make sure you are not raising the base branch violation count, it's bad enough as it is :).
- For more PR etiquette have a look here.

CHAPTER

THREE

BRANCHING MODEL AND DEVELOPMENT METHODOLOGY

In this post we'll explain the branching model and development methodology we use at TUDelft on the Tribler project.

This is mostly targeted at new students joining the team. However, it may give you some useful ideas if you are working on a similar project type.

3.1 Branching model

Tribler is developed mainly by university students (mostly MSC and PHDs) that will work on Tribler for a relatively short period of time. So pull requests usually require several review cycles and some of them take a long time to be completed and merged (development of new features are usually part of Master thesis subjects or papers and suchlike). This makes it rather hard to implement anything like traditional unsupervised continuous integration.

Our branching model follows the GitFlow model described at length in Vincent Driessen's post.

Our main repository contains 2 permanent branches:

- **devel**: The main development branch; all new features and fixes for them belong here. Every time a new release cycle is started, a new **release-X.Y.Z(-abc)** branch is forked from it.
- **master**: Contains the code of the latest stable release. It gets updated from **release-** after every release.

3.2 Release lifecycle

We follow SemVer notation for release naming:

- Stable release name: <MAJOR>.<MINOR>.<PATCH>
- Release candidate name: <MAJOR>.<MINOR>.<PATCH>-RC<NUMBER>

We have to create a new branch For every MAJOR or MINOR release.

Therefore the following structure of the branch name is used: "release/<MAJOR>.<MINOR>".

Example: "release/7.8"

The release lifecycle:

1. A release branch forks from *devel*.
2. The release branch is checked by running the application-tester
3. In case no errors, a release candidate (RC) is published as pre-release in Github and published to the forum as release post.

4. After a while (1-2 weeks), if there are no critical bugs, the release candidate is considered stable and ready for stable release.
5. A stable release tag is created and the release is published in Github. A forum post is created to inform the users in the forum.
6. The release branch is merged to *devel*

In case of an error that needs to be fixed asap in the published stable release:

1. A fix is committed to the corresponding release branch
2. The release branch is checked by running the application-tester
3. The fixed release is delivered to users
4. The release branch is merged to *devel*

A release is started by forking from the top of **devel**. From that moment, all the bugfixes relevant to the current release must be merged into the corresponding release branch. No new features could be added to it.

When the release is ready for publishing, it is merged back to **devel** to integrate the bugfixes.

3.3 Tags

Every revision that will result in a (pre)release gets tagged with a version number.

3.4 Setting up the local repo

1. Fork Tribler's upstream repository.
2. Make a local clone of it:

```
git clone -o MrStudent --recursive --recurse-submodules --single-branch \
git@github.com:MrStudent/tribler
```

3. Add the upstream remote:

```
git remote add upstream https://github.com/Tribler/tribler
```

Note that an /HTTPS/ URL is used here instead of an /SSH/ one (git@github.com/yadayada). This is done in order to prevent accidental pushes to the main repository. However, it will only work if you don't set up /HTTPS/ auth for github. Any attempt to push something there and git will ask you for credentials.

4. Profit!

3.5 Working on new features or fixes

1. Make sure there's an issue for it and get it assigned to you. If there isn't, create it yourself. Otherwise you risk your changes not getting accepted upstream or wasting time on changes that are already being worked on by other developers.
2. Create your feature or bugfix branch. New feature branches can be created like this:

```
git fetch --all && git checkout upstream/devel -b fix_2344_my_new_feature
```

For bug fixes:

```
git fetch --all && git checkout upstream/release-X.Y.Z -b fix_2344_my_new_bugfix
```

2344 would be the issue number this branch is dealing with. This makes it trivial to identify the purpose of a branch if one hasn't had been able to work on it for a while and can't remember right away.

3. Create a Pull Request.

It is usually a good idea to create a pull request for a branch even if it's a work in progress. Doing so will make our Jenkins instance run all the checks, tests and experiments every time you push a change so you can have continuous feedback on the state of your branch.

When creating a PR, always prepend the PR title with **WIP** until it's ready for the final round of reviews. More about this on the next section.

Notes:

- Always fork directly from upstream's remote branches as opposed to your own (remote or local) **devel** or **release-** branches. Those are useless as they will quickly get out of date, so kill them with fire:

```
git branch -d release-X.Y.Z
git branch -d devel
```

- Once one of your branches has been merged upstream try to always delete them from your remote to avoid cluttering other people's remote listings (I've got around 15 remotes on my local Tribler repos and it can become annoying to look for a particular branch among dozens and dozens of other people's stale branches). This can be done either from github's PR web interface by clicking on the "delete branch" button after the merge has been done or with:

```
git push MrStudent :fix_2344_my_new_bugfix
```

3.6 Getting your changes merged upstream

When you think your PR is complete you need to get at least one peer to review your proposed changes as many times as necessary until it's ready. If you can't agree on something add another peer to the discussion to break the tie or talk to the lead developer.

All updates during the review/fix iteration cycles should be made with fixup commits to make it easier for the reviewer(s) to spot the new changes that need review on each iteration. (read the `--fixup` argument on the git-commit manpage if you don't know what a fixup commit is).

Once the reviewer gives the OK and the tests and checks are passing, the fixup commits can then be squashed and the **WIP** prefix can be switched to **READY**. The lead developer will then do the final review round.

As mentioned before, any requested modifications should come in the form of fixup commits to ease reviewing.

Once the final OK is given, all fixup commits should be squashed and the branch will get merged.

3.7 Misc guidelines

- **Keep an eye on the PRs you've reviewed** You will probably learn something from other reviewers and find out what you missed out during yours.
- **Don't send PR from your remote's ~devel~ branch** Use proper names for your branches. It will be more informative and they become part of the merge commit message.
- **Keep it small** The smaller the PRs are, the less review cycles will be needed and the quicker they will get merged.
- **Try to write as many tests as you can before writing any code** It will help you think about the problem you are trying to solve and it usually helps to write code that's easier to test.
- **Have the right amount of commits on your PRs** Don't have a feature implementation spread across a gazillion commits. For instance if a given feature requires some refactoring, your history could look like this:
  - "Refactor foo class to allow for bar" (At this point, the code should still work)
  - "Tests for feature $X"
  - "Implement feature $X"
- **Write clean and self contained commits** Each commit should make sense and be reviewable by itself. It doesn't make sense to break something on one commit and fix it on another later on in the same PR. It also makes reviews much harder.
- **Avoid unrelated and/or unnecessary modifications** If you are fixing a bug or implementing a feature, avoid unnecessary refactoring, white space changes, cosmetic code reordering, etc. It will introduce gratuitous merge conflicts to your and others' branches and make it harder to track changes (for instance with git blame).
- **Don't rename a file and modify it on the same commit** If you need to rename and modify a file on the same PR, do so in two commits. This way git will always know what's going on and it will be easier to track changes across file renames.
- **Don't send pull requests with merge commits on them** Always rebase or cherry pick. If a commit on **devel** introduces merge conflicts in your branch, fix your commits by rebasing not by back merging and creating a conflict resolution commit.
- **If one of your commits fixes an issue, mention it** Add a "Closes #1234" line to the comment's body section (from line 3 onwards). This way a reference to this particular commit will be created on the issue itself and once the commit hits the target branch the issue will be closed automatically. If a whole PR is needed to close a particular issue, add the "Closes" comment on the PR body.
- **Capitalize the commit's subject** We are civilized people after all :D
- **Write concise commit messages** If a particular commit deserves a longer explanation, write a short commit message, leave a blank line after it and then go all Shakespeare from the third line (message body) onwards.
- **Read this** Really, do it.

CHAPTER

FOUR

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

This page contains instructions on how to setup your development environment to run Tribler from source.

4.1 Windows

This section contains information about setting up a Tribler development environment on Windows. Unlike Linux based systems where installing third-party libraries is often a single `apt-get` command, installing and configuring the necessary libraries requires more attention on Windows. Moreover, the Windows environment has different file structures. For instance, where Linux is working extensively with .so (shared object) files, Windows uses DLL files.

4.1.1 Introduction

In this guide, all required dependencies of Tribler will be explained. It presents how to install these dependencies. Some dependencies have to be built from source whereas other dependencies can be installed using a .msi or .exe installer. The guide targets Windows 7 or higher, 64-bit systems, however, it is probably not very hard to install 32-bit packages.

First, Python 3 should be installed. If you already have a Python version installed, please check whether this version is 64 bit before proceeding.

```
python -c "import struct;print( 8 * struct.calcsize('P'))"
```

This outputs whether your current installation is 32 or 64 bit.

Python can be downloaded from the official Python website. You should download the Windows x86-64 MSI Installer which is an executable. **During the setup, remember to add Python to the PATH variable to access Python from the command line. The option to add Python to the PATH variable is unchecked by default!** You can verify whether Python is installed correctly by typing `python` in the command line. If they are not working, verify whether the PATH variables are correctly set. Instructions on how to set path variable can be found here.

In order to compile some of the dependencies of Tribler, you will need the Visual Studio installed which can be downloaded from here or here. You should select the community edition. Visual Studio ships with a command line interface and all required tools that are used for building some of the Python packages. After the installation of Visual Studio, you should install the Visual C++ tools. This can be done from within Visual Studio by creating a new Visual C++ project. Visual Studio then gives an option to install the Visual C++ developer tools.

In case importing one of the modules fail due to a DLL error, you can inspect if there are files missing by opening it with Dependency Walker. It should show missing dependencies.

4.1.2 libtorrent

First, install Boost which can be downloaded from SourceForge. Make sure to select the latest version and choose the version is compatible with your version of Visual C++ tools (probably msvc-14).

After installation, you should set an environment variable to let libtorrent know where Boost can be found. You can do this by going to Control Panel > System > Advanced > Environment Variables (more information about setting environment variables can be found here). Now add a variable named BOOST_ROOT and with the value of your Boost location. The default installation location for the Boost libraries is `C:\\local\\boost_<BOOST VERSION>` where `<BOOST VERSION>` indicates the installed Boost version.

Next, you should build Boost.build. You can do this by opening the Visual Studio command prompt and navigating to your Boost libraries. Navigate to `tools\\build` and execute `bootstrap.bat`. This will create the `b2.exe` file. In order to invoke `b2` from anywhere in your command line, you should add the Boost directory to your user PATH environment variable. After modifying your PATH, you should reopen your command prompt.

Now, download the libtorrent source code from GitHub and extract it. It is advised to compile version 1.0.8. Note that you if you have a 32-bit system, you can download the `.msi` installer so you do not have to compile libtorrent yourself. Open the Developer Command Prompt shipped with Visual Studio (not the regular command prompt) and navigate to the location where you extracted the libtorrent source. In the directory where the libtorrent source code is located, navigate to `bindings\\python` and build libtorrent by executing the following command (this takes a while so make sure to grab a coffee while waiting):

```
b2 boost=source libtorrent-link=static address-model=64
```

This command will build a static libtorrent 64-bit debug binary. You can also build a release binary by appending `release` to the command given above. After the build has been completed, the resulting `libtorrent.pyd` can be found in `LIBTORRENT_SOURCE\\bindings\\python\\bin\\msvc-14\\debug\\address-model-64\\boost-source\\link-static\\` where `LIBTORRENT_SOURCE` indicates the directory with the libtorrent source files. Copy `libtorrent.pyd` to your site-packages location (the default location is `C:\\Python37\\Lib\\site-packages`)

After successfully copying the `libtorrent.pyd` file either compiled or from the repository, you can check if the installation was successful:

```
python -c "import libtorrent" # this should work without any error
```

4.1.3 libsodium

Libsodium is required for the `libnacl` library, used for cryptographic operations. Libsodium can be download as precompiled binary from their website. Download the latest version, built with msvc. Extract the archive to any location on your machine. Next, you should add the location of the dynamic library to your `PATH` variables (either as system variable or as user variable). These library files can be found in `LIBSODIUM_ROOT\\x64\\Release\\v142\\dynamic\\` where `LIBSODIUM_ROOT` is the location of your extracted libsodium files. After modifying your PATH, you should reopen your command prompt. You test whether Python is able to load `libsodium.dll` by executing:

```
python -c "import ctypes; ctypes.cdll.LoadLibrary('libsodium')"
```

Note that this might fail on Python 3.8, since directories have to be explicitly whitelisted to load DLLs from them. You can either copy the `libsodium.dll` to your `System32` directory or by whitelisting that directory using `os.add_dll_directory` when running Tribler.

4.1.4 Additional Packages

There are some additional packages which should be installed. They can easily be installed using pip:

```
pip install aiohttp aiohttp_apispec cffi chardet configobj cryptography decorator␣
↪gmpy2 idna libnacl lz4 \
netifaces networkx numpy pillow psutil pyasn1 PyQt5 pyqtgraph pywin32 pyyaml
```

To enable Bitcoin wallet management (optional), you should install the bitcoinlib library (support for this wallet is highly experimental):

```
pip install bitcoinlib==0.4.10
```

4.1.5 Running Tribler

You should now be able to run Tribler from command line. Grab a copy of the Tribler source code and navigate in a command line interface to the source code directory. Start Tribler by executing the Batch script in the `tribler/src` directory:

```
tribler.bat
```

If there are any problems with the guide above, please feel free to fix any errors or create an issue so we can look into it.

4.2 MacOS

Tribler development environment setup on MacOS (10.10 to latest).

4.2.1 HomeBrew

This guide will outline how to setup a Tribler development environment on Mac.

PyQt5

If you wish to run the Tribler Graphical User Interface, PyQt5 should be available on the system. To install PyQt5, we first need to install Qt5, a C++ library which can be installed with Brew:

```
brew install python3 qt5 sip pyqt5
brew cask install qt-creator # if you want the visual designer
brew link qt5 --force # to allow access qmake from the terminal

qmake --version # test whether qt is installed correctly
```

Libtorrent

You can install libtorrent with Brew using the following command:

```
brew install libtorrent-rasterbar
```

To verify a correct installation, you can execute:

```
python3
>>> import libtorrent
```

Symbol not found: _kSCCompAnyRegex error

If you see *Symbol not found: _kSCCompAnyRegex* error, then follow https://github.com/Homebrew/homebrew-core/pull/43858 for the explanation.

You can build libtorrent by yourself: http://libtorrent.org/python_binding.html or by using this workaround from PR:

1. Edit brew formula:

```
brew edit libtorrent-rasterbar
```

2. Add on the top of the *install* function *ENV.append* string as described below:

```
def install
    ENV.append "LDFLAGS", "-framework SystemConfiguration -framework CoreFoundation"
```

3. Build *libtorrent-rasterbar* from source:

```
brew install libtorrent-rasterbar --build-from-source
```

Other Packages

There are a bunch of other packages that can easily be installed using pip and brew:

```
brew install gmp mpfr libmpc libsodium
cd src && python3 -m pip install -r requirements.txt
```

To enable Bitcoin wallet management (optional), you should install the bitcoinlib library (support for this wallet is experimental):

```
python3 -m pip install bitcoinlib==0.4.10
```

4.2.2 Tribler

The security system on MacOS can prevent `libsodium.dylib` from being dynamically linked into Tribler when running Python. If this library cannot be loaded, it gives an error that libsodium could not be found. This is because the `DYLD_LIBRARY_PATH` cannot be set when Python starts. More information about this can be read here.

The best solution to this problem is to link or copy `libsodium.dylib` into the Tribler root directory.

```
git clone --recursive  https://github.com/Tribler/tribler.git
cd tribler
cp /usr/local/lib/libsodium.dylib ./ || cp /opt/local/lib/libsodium.dylib ./
```

You can now run Tribler by executing the following bash script in the `src` directory:

```
./tribler.sh
```

Proceed proceed to Build instructions

Help

If there are any problems with the guide above, please feel free to fix any errors or create an issue so we can look into it.

4.3 Linux

This section contains information about setting up a Tribler development environment on Linux systems.

4.3.1 Debian/Ubuntu/Mint

First, install the required dependencies by executing the following command in your terminal:

```
sudo apt install git libssl-dev libx11-6 libgmp-dev python3 python3-minimal python3-
↪pip python3-libtorrent python3-pyqt5 python3-pyqt5.qtsvg python3-scipy
```

Secondly, install python packages

```
pip3 install aiohttp aiohttp_apispec chardet configobj decorator libnacl matplotlib␣
↪netifaces networkx pony psutil pyasn1 requests lz4 pyqtgraph pyyaml
```

Then, install py-ipv8 python dependencies

```
cd src/pyipv8
pip3 install --upgrade -r requirements.txt
```

You can now clone the Tribler source code, and run Tribler by executing the following commands:

```
git clone https://github.com/tribler/tribler --recursive
cd tribler/src
./tribler.sh
```

Alternatively, you can run the latest stable version of Tribler by downloading and installing the .deb file from here. This option is only recommended for running Tribler and is not suitable for development.

If there are any problems with the guide above, please feel free to fix any errors or create an issue so we can look into it.

CHAPTER

FIVE

BUILDING TRIBLER

This page contains instructions on how to build and package Tribler.

5.1 Windows

This section contains information about building Tribler on Windows. In the end you should be left with a `.exe` file which, when opened, enables users to install Tribler on their system. This guide installs a 64-bit version of Tribler and has been tested on Windows 10 and Windows 2008 Server R2, 64-bit. It is recommended to create this builder on a system that is already able to run Tribler from a git checkout (it means that all the required packages required by Tribler are installed already). In case you want to build a 32 bit version, just install all the dependencies mentioned in 32 bit version. Information about setting up a developer environment on Windows can be found here.

When you have installed zope, an empty `__init__.py` file must be present in the zope folder. If this file is missing, a `No module named zope` error will be thrown. Create this file in the `site-packages/zope` folder if it does not exist.

5.1.1 Required packages

To build a Tribler installer, you'll need some additional scripts and packages. The versions used as of writing this guide are mentioned next to the package or script. * The git command tools (version 2.7.0) are required to fetch the latest release information. These can be downloaded from here. * PyInstaller, a tool to create an executable from python files. Install the latest version from pip: `pip install pyinstaller`. * The builder needs to find all packages that are required by Tribler so make sure you can run Tribler on your machine and that there are no missing dependencies. * Nullsoft Scriptable Install System (NSIS) (version 2.5.0) is a script-driven Installer authoring tool for Microsoft Windows with minimal overhead. It can be downloaded here. We selected version 2.5 as the uninstall functions were not called properly in 3.03b. * Three plugins are required.The UAC plugin is the first. This can be downloaded from here (version 0.2.4c). How to install a plugin can be found here. * The second plugin that is needed is AccessControl plug-in (version 1.0.8.1). It can be downloaded here. * The third plugin required is NSIS Simple Firewall Plugin (version 1.2.0). You can download it here. * The fourth plugin needed is NSProcess (Version 1.6.7), which can be downloaded here. * A version of Microsoft Visual Studio should be installed (we use 2012), but make sure you do not have the build-tools only. The full (community) edition can be downloaded here.

5.1.2 Building & Packaging Tribler

Start by cloning Tribler if you haven't done already (using the `git clone --recursive` command). Next, create a `build` folder directly on your `C:\` drive. Inside the `build` folder, put the following items:

1. A folder `certs` containing a `.pfx` key. In our case it's named `swarmplayerprivatekey.pfx`. Make sure to rename paths in `makedist_win.bat` to match your file name.
2. `vc_redist_110.exe` (Visual C++ Redistributable for Visual Studio 2012), which is available here. In case you build 32 bit, get the x86 version. Once more, don't forget to rename the file.
3. `libsodium.dll` which can be downloaded from libsodium.org (as of writing version 1.0.8).
4. The openssl dll files `libeay32.dll`, `libssl32.dll` and `ssleay32.dll` (place them in a directory named `openssl`).

Then, set a `PASSWORD` environment variable with its value set to the password matching the one set in your `.pfx` file.

Finally, open a command prompt and enter the following commands (Change 11.0 depending on your version of Microsoft Visual Studio): Note that for building 32 bit you need to pass anything but 64, i.e. 32 or 86 to the `update_version_from_git.py` script.

```
cd tribler
python build/update_version_from_git.py 64
build\win\makedist_win.bat 64
```

This builds an `.exe` installer which installs Tribler.

5.2 MacOS

This guide explains how to build Tribler on MacOS (10.10 to 10.13). The final result is a `.dmg` file which, when opened, allows `Tribler.app` to be copied to the Applications directory and or launched. Make sure the required packages required by Tribler are installed from the Development instructions.

5.2.1 Required packages

- eulagise: In order to attach the EULA to the `.dmg` file, we make use of the `eulagise` script. This script is written in PERL and is based on a more fully-featured script. The script can be downloaded from GitHub. The builder expects the script to be executable and added to the `PATH` environment variable. This can be done with the following commands:

```
cp eulagise.pl /usr/local/bin/eulagise
chmod +x /usr/local/bin/eulagise
eulagise # to test it - it should show that you should add some flags
```

5.2.2 Building Tribler on macOS

Start by checking out the directory you want to clone (using `git clone --recursive`). Open a terminal and `cd` to this new cloned directory (referenced to as `tribler_source` in this guide).

Next, we should inject version information into the files about the latest release. This is done by the `update_version_from_git.py` script found in `Tribler/Main/Build`. Invoke it from the `tribler_source` directory by executing:

```
build/update_version_from_git.py
```

Now execute the builder with the following command:

```
build/mac/makedist_macos.sh
```

This will create the `.dmg` file in the `tribler_source/dist` directory.

5.3 Debian and derivatives

Run the following commands in your terminal:

```
sudo apt-get install devscripts python-setuptools
cd tribler
build/update_version_from_git.py
python3 -m PyInstaller tribler.spec
cp -r dist/tribler build/debian/tribler/usr/share/tribler
dpkg-deb -b build/debian/tribler tribler.deb
```

This will build a `tribler.deb` file, including all dependencies and required libraries.

5.4 Other Unixes

We don't have a generic setup.py yet.

So for the time being, the easiest way to package Tribler is to put `Tribler/` in `/usr/share/tribler/` and `debian/bin/tribler` in `/usr/bin/`. A good reference for the dependency list is `debian/control`.

CHAPTER

SIX

TRIBLER REST API

6.1 Overview

The Tribler REST API allows you to create your own applications with the channels, torrents and other data that can be found in Tribler. Moreover, you can control Tribler and add data to Tribler using various endpoints. This documentation explains the format and structure of the endpoints that can be found in this API. **Note that this API is currently under development and more endpoints will be added over time**.

6.2 Making requests

The API has been built using the aiohttp library. Requests go over HTTP where GET requests should be used when data is fetched from the Tribler core and POST requests should be used if data in the core is manipulated (such as adding a torrent or removing a download). Responses of the requests are in JSON format. Tribler should be running either headless or with the GUI before you can use this API. To make successful requests, you should pass the *X-Api-Key* header, which can be found in your Tribler configuration file (*triblerd.conf*).

Some requests require one or more parameters. These parameters are passed using the JSON format. An example of performing a request with parameters using the curl command line tool can be found below:

```
curl -X PUT -H "X-Api-Key: <YOUR API KEY>" http://localhost:8085/mychannel/rssfeeds/
↪http%3A%2F%2Frssfeed.com%2Frss.xml
```

Alternatively, requests can be made using Swagger UI by starting Tribler and opening *http://localhost:8085/docs* in a browser.

6.3 Error handling

If an unhandled exception occurs the response will have code HTTP 500 and look like this:

```
{
    "error": {
        "handled": False,
        "code": "SomeException",
        "message": "Human readable error message"
    }
}
```

If a valid request of a client caused a recoverable error the response will have code HTTP 500 and look like this:

```
{
    "error": {
        "handled": True,
        "code": "DuplicateChannelIdError",
        "message": "Channel name already exists: foo"
    }
}
```

6.4 Download states

There are various download states possible which are returned when fetching downloads. These states are explained in the table below.

DLSTATUS_ALLOCATING_DISKSPACE	Libtorrent is allocating disk space for the download
DLSTATUS_WAITING4HASHCHECK	The download is waiting for the hash check to be performed
DLSTATUS_HASHCHECKING	Libtorrent is checking the hashes of the download
DLSTATUS_DOWNLOADING	The torrent is being downloaded
DLSTATUS_SEEDING	The torrent has been downloaded and is now being seeded to other peers
DLSTATUS_STOPPED	The torrent has stopped downloading, either because the downloading has completed or the user has stopped the download
DLSTATUS_STOPPED_ON_ERROR	The torrent has stopped because an error occurred
DLSTATUS_METADATA	The torrent information is being fetched from the DHT
DLSTATUS_CIRCUITS	The (anonymous) download is building circuits

6.5 Endpoints

6.5.1 Create torrent

`POST /createtorrent`

Create a torrent from local files and return it in base64 encoding.

Query Parameters

- **`download`** (*boolean*) – Flag indicating whether or not to start downloading

Request JSON Object

- **`description`** (*string*) –
- **`export_dir`** (*string*) –
- **`files[]`** (*string*) –
- **`name`** (*string*) –
- **`trackers[]`** (*string*) –

Status Codes

- 200 OK –
- 400 Bad Request –

6.5.2 Debug

GET /debug/circuits/slots

Return information about the slots in the tunnel overlay.

Status Codes

- 200 OK –

Response JSON Object

- **slots[].competing** (*integer*) –
- **slots[].random** (*integer*) –

GET /debug/cpu/history

Return information about CPU usage history.

Status Codes

- 200 OK –

Response JSON Object

- **cpu_history[].cpu** (*number*) –
- **cpu_history[].time** (*integer*) –

GET /debug/log

Return content of core or gui log file & max_lines requested.

Query Parameters

- **process** (*string*) – Specifies which log to return
- **max_lines** (*integer*) – Maximum number of lines to return from the log file

Status Codes

- 200 OK –

Response JSON Object

- **content** (*string*) –
- **max_lines** (*integer*) –

GET /debug/memory/dump

Return a Meliae-compatible dump of the memory contents.

Status Codes

- 200 OK – The content of the memory dump file

GET /debug/memory/history

Return information about memory usage history.

Status Codes

- 200 OK –

Response JSON Object

- **memory_history[].mem** (*integer*) –

- **`memory_history[].time`** (*integer*) –

`GET /debug/open_files`
Return information about files opened by Tribler.

Status Codes

- 200 OK –

Response JSON Object

- **`open_files[].fd`** (*integer*) –
- **`open_files[].path`** (*string*) –

`GET /debug/open_sockets`
Return information about open sockets.

Status Codes

- 200 OK –

Response JSON Object

- **`open_sockets[].family`** (*integer*) –
- **`open_sockets[].laddr`** (*string*) –
- **`open_sockets[].raddr`** (*string*) –
- **`open_sockets[].status`** (*string*) –
- **`open_sockets[].type`** (*integer*) –

`DELETE /debug/profiler`
Stop the profiler.

Status Codes

- 200 OK –

Response JSON Object

- **`success`** (*boolean*) –

`GET /debug/profiler`
Return information about the state of the profiler.

Status Codes

- 200 OK –

Response JSON Object

- **`state`** (*string*) – State of the profiler (STARTED or STOPPED)

`PUT /debug/profiler`
Start the profiler.

Status Codes

- 200 OK –

Response JSON Object

- **`success`** (*boolean*) –

`GET /debug/threads`
Return information about running threads.

Status Codes

- 200 OK –

Response JSON Object

- **threads[].frames[]** (*string*) –
- **threads[].thread_id** (*integer*) –
- **threads[].thread_name** (*string*) –

6.5.3 Downloads

GET /downloads
Return all downloads, both active and inactive

This endpoint returns all downloads in Tribler, both active and inactive. The progress is a number ranging from 0 to 1, indicating the progress of the specific state (downloading, checking etc). The download speeds have the unit bytes/sec. The size of the torrent is given in bytes. The estimated time assumed is given in seconds.

Detailed information about peers and pieces is only requested when the get_peers and/or get_pieces flag is set. Note that setting this flag has a negative impact on performance and should only be used in situations where this data is required.

Query Parameters

- **get_peers** (*boolean*) – Flag indicating whether or not to include peers
- **get_pieces** (*boolean*) – Flag indicating whether or not to include pieces
- **get_files** (*boolean*) – Flag indicating whether or not to include files

Status Codes

- 200 OK –

Response JSON Object

- **downloads.anon_download** (*boolean*) –
- **downloads.availability** (*number*) –
- **downloads.destination** (*string*) –
- **downloads.error** (*string*) –
- **downloads.eta** (*integer*) –
- **downloads.files** (*string*) –
- **downloads.hops** (*integer*) –
- **downloads.infohash** (*string*) –
- **downloads.max_download_speed** (*integer*) –
- **downloads.max_upload_speed** (*integer*) –
- **downloads.name** (*string*) –
- **downloads.num_peers** (*integer*) –
- **downloads.num_seeds** (*integer*) –
- **downloads.peers** (*string*) –
- **downloads.progress** (*number*) –

- **downloads.ratio** (*number*) –
- **downloads.safe_seeding** (*boolean*) –
- **downloads.size** (*integer*) –
- **downloads.speed_down** (*number*) –
- **downloads.speed_up** (*number*) –
- **downloads.status** (*string*) –
- **downloads.time_added** (*integer*) –
- **downloads.total_down** (*integer*) –
- **downloads.total_pieces** (*integer*) –
- **downloads.total_up** (*integer*) –
- **downloads.trackers** (*string*) –
- **downloads.vod_mode** (*boolean*) –
- **downloads.vod_prebuffering_progress** (*number*) –
- **downloads.vod_prebuffering_progress_consec** (*number*) –

PUT /downloads

Start a download from a provided URI.

Query Parameters

- **get_peers** (*boolean*) – Flag indicating whether or not to include peers
- **get_pieces** (*boolean*) – Flag indicating whether or not to include pieces
- **get_files** (*boolean*) – Flag indicating whether or not to include files

Request JSON Object

- **anon_hops** (*integer*) – Number of hops for the anonymous download. No hops is equivalent to a plain download
- **destination** (*string*) – the download destination path of the torrent
- **safe_seeding** (*boolean*) – Whether the seeding of the download should be anonymous or not
- **uri** (*string*) – The URI of the torrent file that should be downloaded. This URI can either represent a file location, a magnet link or a HTTP(S) url. (required)

Status Codes

- 200 OK –

Response JSON Object

- **infohash** (*string*) –
- **started** (*boolean*) –

DELETE /downloads/{infohash}

Remove a specific download.

Parameters

- **infohash** (*string*) – Infohash of the download to remove

Request JSON Object

- **`remove_data`** (*`boolean`*) – Whether or not to remove the associated data

Status Codes

- 200 OK –

Response JSON Object

- **`infohash`** (*`string`*) –
- **`removed`** (*`boolean`*) –

`PATCH /downloads/{infohash}`

Update a specific download.

Parameters

- **`infohash`** (*`string`*) – Infohash of the download to update

Request JSON Object

- **`anon_hops`** (*`integer`*) – The anonymity of a download can be changed at runtime by passing the anon_hops parameter, however, this must be the only parameter in this request.
- **`selected_files[]`** (*`integer`*) –
- **`state`** (*`string`*) – State parameter to be passed to modify the state of the download (resume/stop/recheck)

Status Codes

- 200 OK –

Response JSON Object

- **`infohash`** (*`string`*) –
- **`modified`** (*`boolean`*) –

`GET /downloads/{infohash}/files`

Return file information of a specific download.

Parameters

- **`infohash`** (*`string`*) – Infohash of the download to from which to get file information

Status Codes

- 200 OK –

Response JSON Object

- **`files[].included`** (*`boolean`*) –
- **`files[].index`** (*`integer`*) –
- **`files[].name`** (*`string`*) –
- **`files[].progress`** (*`number`*) –
- **`files[].size`** (*`integer`*) –

`GET /downloads/{infohash}/stream/{fileindex}`

Stream the contents of a file that is being downloaded.

Parameters

- **`infohash`** (*`string`*) – Infohash of the download to stream
- **`fileindex`** (*`string`*) – The fileindex to stream

Status Codes

- 206 Partial Content – Contents of the stream

GET /downloads/{infohash}/torrent
Return the .torrent file associated with the specified download.

Parameters

- **infohash** (*string*) – Infohash of the download from which to get the .torrent file

Status Codes

- 200 OK – The torrent

6.5.4 Events

GET /events
Open an EventStream for receiving Tribler events.

Status Codes

- 200 OK –

Response JSON Object

- **event** (*object*) –
- **type** (*string*) –

6.5.5 Libtorrent

GET /libtorrent/session
Return Libtorrent session information.

Query Parameters

- **hop** (*string*) – The hop count of the session for which to return information

Status Codes

- 200 OK – Return a dictonary with key-value pairs from the Libtorrent session information

Response JSON Object

- **hop** (*integer*) –
- **settings** (*object*) –

GET /libtorrent/settings
Return Libtorrent session settings.

Query Parameters

- **hop** (*string*) – The hop count of the session for which to return settings

Status Codes

- 200 OK – Return a dictonary with key-value pairs from the Libtorrent session settings

Response JSON Object

- **hop** (*integer*) –
- **settings** (*object*) –

6.5.6 Metadata

`DELETE /metadata`

Delete channel entries.

Status Codes

- 200 OK – Returns a list of deleted entries
- 400 Bad Request –

`PATCH /metadata`

Update channel entries.

Status Codes

- 200 OK – Returns a list of updated entries
- 400 Bad Request –
- 404 Not Found –

`GET /metadata/torrents/{infohash}/health`

Fetch the swarm health of a specific torrent.

Parameters

- **`infohash`** (*`string`*) – Infohash of the download to remove

Query Parameters

- **`timeout`** (*`integer`*) – Timeout to be used in the connections to the trackers
- **`refresh`** (*`integer`*) – Whether or not to force a health recheck. Settings this to 0 means that the health of a torrent will not be checked again if it was recently checked.
- **`nowait`** (*`integer`*) – Whether or not to return immediately. If enabled, results will be passed through to the events endpoint.

Status Codes

- 200 OK –

Response JSON Object

- **`tracker.error`** (*`string`*) –
- **`tracker.infohash`** (*`string`*) –
- **`tracker.leechers`** (*`integer`*) –
- **`tracker.seeders`** (*`integer`*) –

`GET /metadata/{public_key}/{id}`

Get channel entries.

Parameters

- **`public_key`** (*`string`*) –
- **`id`** (*`string`*) –

Status Codes

- 200 OK – Returns a list of entries
- 404 Not Found –

`PATCH /metadata/{public_key}/{id}`

Update a single channel entry.

Parameters

- **`public_key`** (*string*) –
- **`id`** (*string*) –

Status Codes

- 200 OK – The updated entry
- 400 Bad Request –
- 404 Not Found –

6.5.7 Search

`GET /search`

Perform a search for a given query.

Query Parameters

- **`exclude_deleted`** (*boolean*) –
- **`remote_query`** (*boolean*) –
- **`hide_xxx`** (*boolean*) – Toggles xxx filter
- **`txt_filter`** (*string*) – FTS search on the chosen word* terms
- **`category`** (*string*) –
- **`metadata_type`** (*array*) –
- **`last`** (*integer*) – Limit the range of the query
- **`sort_by`** (*string*) – Sorts results in forward or backward, based on column name (e.g. “id” vs “-id”)
- **`first`** (*integer*) – Limit the range of the query
- **`sort_desc`** (*boolean*) –

Status Codes

- 200 OK –

Response JSON Object

- **`chant_dirty`** (*boolean*) –
- **`torrents[].category`** (*string*) –
- **`torrents[].commit_status`** (*integer*) –
- **`torrents[].date`** (*integer*) –
- **`torrents[].id`** (*integer*) –
- **`torrents[].infohash`** (*string*) –
- **`torrents[].last_tracker_check`** (*integer*) –
- **`torrents[].name`** (*string*) –
- **`torrents[].num_leechers`** (*integer*) –

- **`torrents[].num_seeders`** (*`integer`*) –
- **`torrents[].public_key`** (*`string`*) –
- **`torrents[].relevance_score`** (*`integer`*) –
- **`torrents[].size`** (*`integer`*) –
- **`torrents[].type`** (*`string`*) –

`GET /search/completions`

Return auto-completion suggestions for a given query.

Query Parameters

- **`q`** (*`string`*) – Search query

Status Codes

- 200 OK –

Response JSON Object

- **`completions[]`** (*`string`*) –

6.5.8 Settings

`GET /settings`

Return all the session settings that can be found in Tribler.

This endpoint returns all the session settings that can be found in Tribler.

It also returns the runtime-determined ports, i.e. the ports for the SOCKS5 servers. Please note that a port with a value of -1 in the settings means that the port is randomly assigned at startup.

Status Codes

- 200 OK –

`POST /settings`

Update Tribler settings.

Status Codes

- 200 OK –

Response JSON Object

- **`modified`** (*`boolean`*) –

6.5.9 Shutdown

`PUT /shutdown`

Shutdown Tribler.

Status Codes

- 200 OK –

Response JSON Object

- **`shutdown`** (*`boolean`*) –

6.5.10 State info

`GET /state`

Return the current state of the Tribler core.

Status Codes

- 200 OK –

Response JSON Object

- **`last_exception`** (*string*) –
- **`readable_state`** (*string*) –
- **`state`** (*string*) – One of three stats: STARTING, UPGRADING, STARTED, EXCEPTION

6.5.11 Statistics

`GET /statistics/ipv8`

Return general statistics of IPv8.

Status Codes

- 200 OK –

Response JSON Object

- **`statistics.total_down`** (*integer*) –
- **`statistics.total_up`** (*integer*) –

`GET /statistics/tribler`

Return general statistics of Tribler.

Status Codes

- 200 OK –

Response JSON Object

- **`statistics.database_size`** (*integer*) –
- **`statistics.num_channels`** (*integer*) –
- **`statistics.torrent_queue_stats[].failed`** (*integer*) –
- **`statistics.torrent_queue_stats[].pending`** (*integer*) –
- **`statistics.torrent_queue_stats[].success`** (*integer*) –
- **`statistics.torrent_queue_stats[].total`** (*integer*) –
- **`statistics.torrent_queue_stats[].type`** (*string*) –

6.5.12 Torrent info

GET /torrentinfo

Return metainfo from a torrent found at a provided URI.

Query Parameters

- **torrent** (*string*) – URI for which to return torrent information. This URI can either represent a file location, a magnet link or a HTTP(S) url.

Status Codes

- 200 OK – Return a hex-encoded json-encoded string with torrent metainfo

Response JSON Object

- **metainfo** (*string*) –

6.5.13 TrustChain

6.5.14 Trust View

GET /trustview

Return the trust graph.

Status Codes

- 200 OK –

Response JSON Object

- **bootstrap.download** (*integer*) –
- **bootstrap.progress** (*number*) –
- **bootstrap.upload** (*integer*) –
- **graph.edge[][]** (*integer*) –
- **graph.node.id** (*integer*) –
- **graph.node.key** (*string*) –
- **graph.node.pos[]** (*number*) –
- **graph.node.sequence_number** (*integer*) –
- **graph.node.total_down** (*integer*) –
- **graph.node.total_up** (*integer*) –
- **num_tx** (*integer*) –
- **root_public_key** (*string*) –

6.5.15 Upgrader

`POST /upgrader`

Skip the DB upgrade process, if it is running.

Request JSON Object

- **`skip_db_upgrade`** (*boolean*) – Whether to skip the DB upgrade process or not

Status Codes

- 200 OK –
- 400 Bad Request –
- 404 Not Found –

Response JSON Object

- **`skip_db_upgrade`** (*boolean*) –

CHAPTER

SEVEN

TRUSTCHAIN

TrustChain is a tamper-resistant data structure that is used in Tribler to record community contributions. This blockchain-based distributed ledger can then be used to build a reputation mechanism that is able to identify free-riders. A basic implementation of TrustChain is available in our code base and available for other developers.

TrustChain is specifically designed to be transaction-agnostic which means that any transaction can be stored in TrustChain. In Tribler, this consists of the amount of uploaded and downloaded data.

7.1 Using TrustChain

Using TrustChain to store transaction is straightforward. Creating a new block is done by invoking the `sign_block` method of the community. The required arguments are the destination candidate (the counterparty of the transaction), your public key and the transaction itself, in the Python dictionary format. Note that this dictionary can only contain Python primitive types and no custom objects due to the serialization of the transaction when sending it to the other party.

Assuming that the transaction counterparty is online and the block is valid, the counterparty signs the block and sends it back where the `received_half_block` method is invoked, processing the received block.

7.2 Using TrustChain in your project

To use TrustChain in your own projects, one can create a subclass of `TrustChainCommunity` or use the `TrustChainCommunity` directly. This should be enough for basic usage. For more information about communities, we reference the reader to a Dispersy tutorial.

In order to implement custom transaction validation rules, a subclass of `TrustChainBlock` should be made and the `BLOCK_CLASS` variable in the `TrustChainCommunity` should be updated accordingly. By overriding the `validate_transaction` method, you can add your own custom validation rules.

This document briefly describes the main ideas and design decisions behind the Channels 2.0 (GigaChannels) system.

CHAPTER

EIGHT

CHANNELS ARCHITECTURE

In Tribler, users can share their torrents collection by creating a "Channel" (*a la* Youtube) and adding some torrents to it. The channel and its contents will then be shown to other Tribler users in the common list of "Discovered" channels. Users can keyword search channels for content. Channels popularity rating helps users identify quality content.

Basically, Channels 2.0 works by spreading gossip about available channels, downloading channels "subscribed" by the user and putting them into the local channels database (*Metadata Store*). When subscribed, channels are transferred through Libtorrent. The infohashes linking to these "channel torrents" are sent around by specialized IPv8 communities (*GigaChannel Community* and *RemoteQuery Community*). Each metadata entry is individually signed by its author's private key, preventing other users from modifying other's channels. The user can create multiple channels with a tree of internal "folders". Channels and metadata entries can be updated or deleted by their author.

The Channels system consists of a number of disparate, complementary subsystems:

- *Metadata Store*
- *GigaChannel Manager*
- *GigaChannel Community*
- *RemoteQuery Community*
- *VSIDS heuristics* for channels popularity

8.1 What is a channel?

A *channel* is a collection of *signed* metadata entries. These entries form a forest of channel/folder(collection) trees. Every metadata entry contains an id, a timestamp and a pointer to the id of its parent channel/folder. Every metadata entry is individually signed by the creator host's private key. The signature is included in the serialized form of the entry, along with the public key. This allows the entries to be trustfully spread through a network of untrusting strangers, if all of them trust the entry creator's public key.

A channel exists in two forms:

- a set of DB entries in a PonyORM-backed SQLite DB (*The DB form*).
- a stream of serialized entries broken down into *lz4-packed chunks* dumped into files in a torrent (*the Torrent form*).

A channel entry is added to the local DB only if it passes the "logical firewall" with strict criteria for correctness (e.g. signature check, etc.). This is true for no matter the way an entry enters the system, be it from the torrent form, or from any kind of network packet.

- A user can have an arbitrary number of channels in the "domain" of his or her public key.
- A channel can have an arbitrary number of nested folders/metadata entries.

Detailed description of the serialization ("Commit") process and serialization formats used by Channels system can be found in the following documents:

This document describes the format for storing, saving and loading metadata entries to/from file-based format that is sent as torrents by the Channels system.

8.1.1 Metadata lifecycle in the Channels system

It is easiest to show the inner workings of the Channels system by following the full path of a metadata entry from its creation to the moment it arrives into another host's database. In brief, the channel is going through the following stages during its lifetime:

- *Channel creation* or *Torrent metadata entry creation* - as a tree of metadata entries in the PonyORM DB
- *Committing the channel contents to disk* in the form of a serialized, append-only stream of updates, broken down into mdblob files
- Creating a torrent from the serialized stream/mdblob files
- *Delivering the channel entry to another host* via e.g. gossip through RemoteQuery community
- Downloading the channel by infohash (using Libtorrent)
- *Processing the channel from disk* on the remote host and updating the remote hosts' DB accordingly

The discrete clock

An important part of the Channels design is its usage of discrete timestamps to note the order of creation/modification of metadata entries. The timestamps' meaning is local to the public key domain. This allows us to use a simple increasing discrete counter that is initiated at Tribler startup from the local time. The counter is increased by +1 on each `tick` (e.g. generation of a timestamp).

Channel creation

The user creates a personal channel by creating a `ChannelTorrent` entry in the database:

Property	Value	Comment
metadata_type	400 (CHANNEL_TORRENT)	used as the "discriminator" by PonyORM
flags	0	
public_key	host's public key	identifies the "user domain" of the entry
id	random integer	uniquely identifies the entry in the domain
origin	0	Points to the parent or the channel/folder.
timestamp	current discrete_clock value	
infohash	assigned randomly	
size	0	
torrent_date	current datetime	
title	given by the user	indexed by FTS
tags		
tracker_info		
num_entries	0	
start_timestamp	same as timestamp	
signature	generated with the host's private key from all the serialized fields (above)	
status	NEW	
local_version	same as timestamp	

Note that `discrete_clock` is increased by +1 after each usage.

Torrent metadata entry creation

The user adds a torrent to Tribler by creating a `TorrentMetadata` entry in the database:

Property	Value
metadata_type	300 (REGULAR_TORRENT)
flags	0
public_key	host's public key
id	random integer
origin	the id of the parent channel
timestamp	current discrete_clock value
infohash	assigned from the provided torrent
size	assigned from the provided torrent
torrent_date	assigned from the provided torrent
title	assigned from the provided torrent
tags	Text-based "category" of the torrent, assigned by analyzing the contents from the provided torrent
tracker_info	assigned from the torrent provided by user
signature	generated with the host's private key from all the serialized fields (above)
status	NEW

Committing the channel contents to disk

The basic idea of "commit" is to represent the *changes* to the channel tree as a stream of data, serialize that stream, compress it, break down into files, add the files to the existing channel directory, update the channel torrent from it, then update and re-sign the toplevel channel entry with the infohash of the updated channel torrent.

- After the GUI (or the user directly) initializes the commit channel action, the Metadata Store scans the domain of the host's public key for entries in the `NEW`, `UPDATED` or `DELETED` status.
- For every new/updated entry, the Core builds the path from the entry to its root channel (the channel with `origin_id==0`).
- New/updated entries are sorted by timestamp.
- Folder entries on the path to the channel root are updated with new counts of torrents in them (`num_entries`), recursively. These receive a new ``timestamp``s from ``discrete_clock`` and signature. Their `status` is changed to `UPDATED`.
- All the `UPDATED` and `NEW` entries in the channel are serialized and concatenated into a single stream, that is incrementally compressed with lz4 algorithm and split into 1 MB-sized chunks called mdblobs. The compression is performed incrementally, until the compressed mdblob size fits in 1 MB. Every blob is individually lz4-unpackable.
- The mdblobs are written into the channel's directory that is located at `.Tribler/<version>/channels/<first 32 characters of hexlified public key><hexademical representation of channel's id padded to 16 characters>` (`hexlify(public_key)[:32] + "{:0>16x}".format(id)`)
- Afterwards, the mdblob is dumped into a new file in the channel's directory. The filename follows the pattern `<last_metadata_entry's_timestamp>.mdblob.lz4`.
- The `DELETE` entries require special treatment. For each metadata entry marked `DELETE`, a `DeletedMetadataPayload` is created with its `delete_signature` field containing the signature of the metadata entry to be deleted. Due to a design mistake, these payloads got no `timestamp` field. This resulted in a nasty error in those cases when a `DELETE` entry would be placed the last in an mdblob file. As a workaround, the `DELETE` entries are now serialized *last* in the stream, but whenever an mdblob file ends with such an entry, the it gets the filename from a `discrete_clock` tick.
- When all the changes are dumped to the disk, Metadata Store calls Libtorrent to create a torrent from the channel directory to obtain the new infohash.
- The toplevel channel entry is updated with the new `infohash`, `num_entries`, `size`, and a `timestamp` from the `discrete_clock` tick. The entry's `signature` is updated accordingly.
- If the whole commit procedure finishes without errors, all the newly serialized entries change their status to `COMMITTED`.

Delivering the channel entry to another host

A channel entry can end up at another host in multiple ways, most of which involve GigaChannel or RemoteQuery community: * A search query through GigaChannel community can net a channel entry * A channel entry can be queried by RemoteQuery community walking an querying for subscribed channels * A channel entry can be gossiped by GigaChannel community (along with some preview contents).

Initially, when a host receives an unknown Channel entry, it will set its `local_version` to `0`. Whenever a host receives a channel entry with the same public key and id as it already know, but with a higher timestamp it will update the entry with the data from the new one. If a channel is subscribed by user and its `local_version` is lower that its `timestamp`, GigaChannelManager will initiate the process of downloading/processing the channel. Note that `local_version` and other local properties are *not* overwritten by updating the channel entry. Only `non-local` (i.e. payload-serialized) attributes are updated.

Processing the channel from disk

After the channel torrent was downloaded by GigaChannel Manager, the processing procedure is initiated.

- Processing is performed by sequentially reading the next mdblob file that has a filename/number that is higher than the current `local_timestamp` of the processed channel.
- `start_timestamp` puts the lower bound on the mdblob's names/timestamps that should be processed. Its main purpose is enabling the possibility of `defragmentation` or `complete reset` of a channel.
- All the metadata entries in each processed mdblob are unpacked, deserialized, checked for correct signature and added to the database *the same way as if they were received from network* (e.g. trough a query in by RemoteQuery community)
- As soon as the mdblob file processing is finished, the channel's `local_version` is set to the filename number. This guarantees that if processing is interrupted by e.g. a power fail, processing will restart from the same mdblob.
- The last mdblob's filename is equal to the channel's `timestamp`. Therefore, as soon as the last mdblob is processed, the channel's `local_version` becomes equal to its `timestamp`, thus putting the channel to "Complete" state.

Free-for-all metadata entries

Free-for-all (FFA) metadata entry is a special kind of *unsigned* metadata entry. It is impossible to attribute such an entry to any particular host, because the entry contain no traces of the host that created them. In other words, given one torrent file, two different hosts will independently create the same FFA entry.

FFA entries are created whenever the user downloads a torrent that is unknown to MDS. Whenever MDS receives a FFA entries are not attributed to any channel. Whenever MDS receives a signed metadata entry that has the same infohash as an FFA entry, it will remove the FFA entry. FFA entries exist for the reason of helping local and remote keyword search.

This document contains information about serialization format used by Metadata Store to serialize metadata entries before sending them over the network and saving them on disk

8.1.2 Payload types and serialization formats

All payloads follow the same pattern:

serialized parent metadata type + type specific data + signature.

Payload field types are expressed in Python Struct notation.

Metadata types are identified by integers as follows:

ORM type	Payload type	Python constant	Numeric metadata type
		TYPELESS	100
ChannelNode	ChannelNodePayload	CHANNEL_NODE	200
MetadataNode	MetadataNodePayload	METADATA_NODE	210
CollectionNode	CollectionNodePayload	COLLECTION_NODE	220
TorrentMetadata	TorrentMetadataPayload	REGULAR_TORRENT	300
ChannelMetadata	ChannelMetadataPayload	CHANNEL_TORRENT	400
	DeletedMetadataPayload	DELETED	500

SignedPayload

This is the base class/payload type. It is never used on the wire "as is" and should only be used as a parent class for other payloads.

- The signature is applied to the end of the serialized payload.
- `flags` field is reserved for future use.
- Public key and signature use LibNaCl Curve25519 ECC.
- If both the signature and public key are all-zeroes, the payload is a "Free-for-All" (FFA) metadata entry.

Metadata type	Flags (reserved)	Public key	Signature
H	H	64s	64s
Signed			

ChannelNodePayload

This is another "virtual" class/payload type. It represents an abstract node in the channel internal hierarchy graph.

- `id` is a random integer which, together with the public key, uniquely identifies the ChannelNode in the Channels system.
- `origin` is the `id` of the parent ChannelNode. `0` in this field means a top-level, parentless entry.
- `timestamp` stands for the "creation time" of this ChannelNode. Typically, it contains the value from the monotonically increasing discrete clock-like counter *DiscreteClock* in the MetadataStore.

Metadata type	Flags (reserved)	Public key	Id	Origin	Timestamp	Signature
H	H	64s	Q	Q	Q	64s

MetadataNodePayload

This is a generic "virtual" metadata-containing node in the channels graph.

- `title` field contains the entry's title, e.g. a name for a movie or a title for a torrent. It is indexed by FTS in MetadataStore.
- `tags` field is reserved for human-readable tags usage. It is **not** indexed by FTS. Currently, the first word in this field is interpreted as the entry's category. For ChannelTorrent entries, it contains channel description (for reasons of the legacy Channels system compatibility).

Metadata type	Flags (reserved)	Public key	Id	Origin	Timestamp	Title	Tags	Signature
H	H	64s	Q	Q	Q	varlenI	varlenI	64s

TorrentMetadataPayload

This is the primary type of payloads used in the Channels system - a payload for BitTorrent metadata.

- `infohash` stands for BitTorrent infohash, in binary form.
- `size` stands for the torrent size in bytes.
- *torrent_date* stands for the torrent creation date, in seconds since Unix Epoch.
- *tracker_info* is the tracker URL
- Note that this payload inherits directly from ChannelNodePayload, and **not** from MetadataNodePayload. The title and tags field are moved to the end of the payload (for a dumb reason of micro-optimizing deserialization).

Metadata type	Flags (reserved)	Public key	Id	Origin	Timestamp	Infohash	Size	Torrent date	Title	Tags	Tracker info	Signature
H	H	64s	Q	Q	Q	20S	Q	I	varlenI	varlenI	varlenI	64s

CollectionNodePayload

This payload serializes CollectionNode entries, which are like ChannelTorrent entries, but w/o infohash field. CollectionNode entries are used to represent intra-channel "folders" of torrents and/or other folders.

- `num_entries` field represents the number of entries that is contained in this Collection and its sub-collections.

Metadata type	Flags (reserved)	Public key	Id	Origin	Timestamp	Title	Tags	Num entries	Signature
H	H	64s	Q	Q	Q	varlenI	varlenI	Q	64s

ChannelMetadataPayload

This payload serializes ChannelTorrent entries, combining properties of CollectionNodePayload, and TorrentMetadataPayload.

- `start_timestamp` represents the first timestamp in this channel. It is used to limit the channel's span back in time after e.g. defragmenting a channel or restarting it anew.
- `torrent_date` is used by the GUI to show when the channel was committed the last time (the "Updated" column).

Metadata type	Flags (reserved)	Public key	Id	Origin	Timestamp	Infohash	Size	Torrent date	Title	Tags	Tracker info	Num entries	Start timestamp	Signature
H	H	64s	Q	Q	Q	20s	Q	I	varlenI	varlenI	varlenI	Q	Q	64s

DeletedMetadataPayload

This payload is a “command” for the Channels system to delete an entry. The entry to delete is pointed by its signature.

- Currently, this metadata can only exist in serialized form. It is never saved to DB.
- `delete_signature` the signature of the metadata entry to be deleted.

Metadata type	Flags (reserved)	Public key	Delete signature	Signature
H	H	64s	64s	64s

8.2 Metadata Store

The Metadata Store (MDS) consists of:

- PonyORM bindings (class definitions)
- methods for converting channels between DB and Torrent forms.

The process of dumping a channel to a torrent is called a “commit”, because it creates a serialized snapshot of the channel. For details, see the documentation on *channel torrent storage*.

8.3 GigaChannel Manager

The GigaChannel Manager (GM) is responsible for downloading channel torrents and processing them into MDS. GM regularly queries MDS and Tribler Download Manager for state changes and starts required actions:

- for subscribed channels that must be downloaded or updated GM starts downloading the corresponding torrents;
- for already downloaded channel torrents GM starts processing them into MDS;
- GM deletes old channel torrents that does not belong to any channel.

Internally, GM uses a queue of actions. GM queue tries to be smart and only do the necessary operations For instance, if the user subscribes to a channel, then immediately unsubscribes from it, and then subscribes to it again, the channel will only be downloaded once.

The looping task + actions queue design is necessary to prevent the callback / race condition nightmare of synchronizing Libtorrent’s and `asyncio`’s reactors.

8.4 GigaChannel Community

The GigaChannel Community (GC) is the original channels gossip/search overlay deployed with the first implementation of Channels 2.0. It plays the following roles:

- spreading push-based gossip of subscribed channels, along with some preview contents;
- sending and serving remote keyword search queries (initiated by searching for stuff in the Tribler GUI).

8.4.1 Channel gossip

At regular intervals, GC queries MDS for a random subscribed channel and assembles a "preview blob" containing the channel metadata entry and some of its contents. GC then sends this preview blobs to random peers in push-based gossip.

8.4.2 Keyword search

When the user initiates a keyword search, GS sends a request to a number of the host's random peers. The remote hosts respond to these requests with serialized metadata entries. Upon receiving a response, calls MDS to process them. Then:

- if the received entries were already known, nothing happens;
- if the received entries are new entries or updated version of already known entries, these will be added to MDS and shown to user through the GUI;
- if local MDS happens to have newer versions of some of the received entries, GC will send the newer entries back as a gratuitous update. ("Hey buddy, that's old news! I got a newer version of the stuff you gave me. Here, take it for free.)

GigaChannel Community is supposed to be superseded by the more robust RemoteQuery Community.

8.5 RemoteQuery Community

The RemoteQuery Community (RQC) essentially provides a way to query remote hosts' MDS with the same multi-purpose `get_entries` method that is used by the local `metadata` REST endpoint. While looking dangerous on the surface (hello, SQL injection!), it only allows for very limited types of `SELECT`-like queries. The philosophy of RQC is that it never allows to get more information from the host that is not already exposed by the network through other means (e.g. GigaChannel Community). RQC plays the following roles:

- Pulling subscribed channels from other host during endless walk. This allows for fast bootstrapping Channel contents on new clients.
- Providing a generic mechanism for exchanging metadata info between hosts.
- Enabling researches to gather useful statistics from metadata-gathering walks.

8.6 VSIDS heuristics

VSIDS stands for Variable State Independent Decaying Sum heuristic. Its basic idea is that the "weight" of some element additively increases on each access to that element, but multiplicatively decreases (decays) over time. VSIDS is close to timed PageRank in its results in the sense it selects variables with high temporal degree centrality and temporal eigenvector centrality (see. "Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers" by Jia Hui Liang et. al.).

We use VSIDS to calculate the channel popularity. Whenever a channel is received by RQC or GC, its rating is bumped additively. Over the time, channels ratings decay in such a way, that a channel's rating is reduced 2.78 times over 3 days. As a classic optimization, instead of decaying all the channel entries all the time, we increase the bump amount proportionally. The implementation also includes standard VSIDS features such as normalization and float overflow protection/renormalization. An important detail of our implementation is that the a single host can never bump the channel more than once. Instead, it can only "renew" its vote. This is achieved by storing the last vote for each host/channel pair along with its bump amount, and deducting it from the channel vote rating before applying the vote again, with the current bump amount.

VSIDS is implemented as a part of MDS.

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search

HTTP ROUTING TABLE