Remofile Documentation

Release 1.0.0.dev1

Jonathan De Wachter

Jan 15, 2019

Contents

CHAPTER 1

The User Guide

Warning: The getting started document is still incomplete. Expect some serious update in the upcoming weeks.

This document aims to get you started with using Remofile. It covers the installation, the key ideas you need to know, and shows how to use the main features such as running a server, interacting with it from a shell and from the code.

- Easy installation
- Run a testing server
- Shell interactions
- Remofile from Python codes
- Synchronizing directories
- Securing the connection
- Standalone server as a service
- Additional server options

For more exhaustive documentation, refer to the following documents; the commands list, the *API reference*, the *protocol specifications*.

1.1 Easy installation

Installing Remofile can't be easier. It doesn't have many dependencies and it can be done in one command-line.

```
pip install remofile
```

I suggest you create a virtual environment before typing this command. This can easily be done like this.

```
python3 -m virtualenv python-env
source python-env/bin/activate
```

Installing with pypi in a virtual environment provides flexibility as **Remofile** isn't installed system-wide. But you can install with the underlying operating system package manager as well. Packages for various Linux distributions do exist; check out the following document.

1.2 Run a testing server

Before deploying online, better do some **local** tests to familiarize yourself with the tool. We'll start with running a local Remofile server and interact with it.

Remofile doesn't know about users and passwords, instead you use a token to authenticate.

```
remofile generate-token
```

This generates a token which is a 16 letters long string. We'll be using **qRkVWJcFRqi7rsNMbagaDd** for demonstration purpose. Copy paste it and store it somewhere as it acts as a unique password.

Just like FTP, you want to jail and expose a directory to multiple clients. It isn't too hard, the *run* command allows you to do just that.

```
mkdir my-directory/
remofile run my-directory/ 6768 qRkVWJcFRqi7rsNMbagaDd
```

It takes the directory (that must be available across the network), the port (on which to listen) and the previously generated token as parameters.

This will run a Remofile server attached to the current console. Don't expect this command to return... and don't interrupt it! Now, open another console to continue.

Note: The *start* and *stop* commands also starts a Remofile server, but instead it daemonizes the process and you work with a PID file instead. The command you use to start the server also impacts the way you run it as a service later.

Also, if no token is given to the *run* command, one is automatically generated and printed out in the console before it starts.

Before we continue, we must understand an important aspect of Remofile; its dumb nature. To have an uncomplicated tool to work with, Remofile makes the assumption that all the files in the folder being served are readable and writable by the user who started the process. It also makes the assumption that the directory isn't modified (in any way) by external mean while it's running. **But that should be common sense, shouldn't it ?**

By not attempting to be smart, the overall interface and implementation becomes simpler and easier to use. If these two assumptions aren't respected, Remofile will still graciously fail with proper error messages if something bad occurs during file operations.

1.3 Shell interactions

Our next step in experimenting Remofile will be with its command-line interface. Even though it initially was designed for integration in software (from code), it also has a powerful set of command-lines allowing people to script interactions with the server.

Let's have a look at how to list files in the remote directory.

```
remofile list /
```

In Remofile, there is no notion of "current working directory", and therefore, the **/** refers to the root of the directory being served. This is also called the **root directory**. Earlier we started the server to expose *my-directory/*, so in this case, typing this command will list that directory.

Important: Because there is no notion of **current working directory**, expect the entire programming and command-line interface to complain if a remote path isn't absolute.

The list command also comes with options. Type *remofile list –help* to know more about them. For instance, it has options similar to the POSIX *ls* command found in Unix-like OSes such as *-l* and *-r*. In one command, you can list the entire content of the remote directory.

```
remofile list / -l -r
```

There is also another important thing to understand. This is a **sessionless** command. Instead of connecting to the server and prompting you to a different interactive shell (supposedly with the connection maintained), it does connect to the server, perform the file operation, then disconnect from the server. It isn't efficient, but it provides a flexible mean for testing and scripting.

1.4 Upload and download files

Things get interesting when we actually get some files transferred to and from the remote directory. The two main commands involved are *upload* and *download*. They both support a dozen of options to customize the behavior.

Both commands can transfer an **individual file**, an entire **directory** (if the recursive flag is passed) or a **set of files** specified by a shell glob patterns.

Have a look at the following.

```
remofile upload single-file.txt a-directory/ **/some-text-files*.txt /
remofile download single-file.txt a-directory/ .
```

We notice that the first arguments refer to the source files, and the last argument refers to the destination. The destination must imperatively be an existing directory on either the remote directory for the upload command, or the local file-system for download command.

Warning: Shell glob patterns would only work for the upload command as it's expanded by the underlying shell.

The destination directory actually is optional and is defaulted to the root directory for the upload command, and the current working directory for the download command. For instance, you can upload a file, then download it back without thinking too much

> remofile upload ubuntu-16.04.3-desktop-amd64.iso remofile download ubuntu-16.04.3-desktop-amd64.iso

But these two commands merely are a front-end to the :py:func‘upload_files‘ and `download_files()` methods of the programming API. In fact, all command-lines we saw so far have their actual corresponding available available from Python.

Now we'll have a look at the programming interface because it's richer than the command-line interface in terms of functionalities.

1.5 Remofile from Python code

Earlier, we saw how to start a Remofile server from the shell using the *run* command. In fact, it can also be done in Python.

```
from remofile import Server

server = Server('my-directory', 'qRkVWJcFRqi7rsNMbagaDd')
server.run(6768)
```

If the current working directory is the same as before, it will start the Remofile server exactly like we did previously. The *run()* method is blocking. To have it returning and thus, terminating the server, one must call the *terminate()* from external thread.

Note: By default, it listens to all IPs and this snipped of code is suitable for production code. Always refer to the API reference for exhaustive documentation.

But everything we did on the client side also trivially matches to a Python programming interface.

```
from remofile import Client

client = Client('localhost', 6768, 'qRkVWJcFRqi7rsNMbagaDd')
client.list_files('/')
client.upload_file('foo', 'bar')
client.download_file('bar', 'foo')
```

You quickly get it, it's exactly the same interface. In practice, you will always use a timeout value and ensure that every file operations complete successfully.

To be written here: words about the possible exceptions.

1.6 Synchronizing directories

Everything is already there to make decent use of Remofile. But synchronizing directories would require additional logic on top of the primitive file operations that the Client implements.

What if uploading a directory is interrupted and it's left partially uploaded. An option would to delete the directory and re-do the upload from scratch. However, this is costly, and using a synchronization approach would save tons of bandwidth and time. Remofile spares you the hard work because that logic is already there.

```
from remofile import Client, synchronize_upload, synchronize_download

client = Client('localhost', 6768)
synchronize_upload('foo', 'bar')
synchronize_download('bar', 'foo')
```

Foobar.

It also has a command version of .

```
remofile sync local foo bar
remofile sync remote bar foo
```

Foobar.

1.7 Securing your server

Securing your server with private and public keys.

To be written.

1.8 Server as a deamon or service

The last section of this guide will show you good practice when it comes to installing a Remofile server on production server.

1.9 Additional server options

Talk about chunk size.

CHAPTER 2

API Reference

This document is the API reference of Remofile. All classes, functions and exceptions are accessible from a single import of the **remofile** package.

```
from remofile import *
```

The programming interface is essentially made of two classes which are *Client* and *Server*. They implement both side of the *Remofile protocol*.

2.1 Interface overview

The client class implements all primitive file operations and therefore any more complex file operations can, in theory, be implemented on top of it.

List of primitive file operations.

- List files
- Create a file
- Create a directory
- Upload a file
- Upload a directory
- Download a file
- Download a directory
- Delete a file (or directory)

However, the algorithm module already implements a couple of useful more *advanced file operations* for you. For instance, it can upload and download trees of files, understand glob patterns and handle conflicts. It also has functions to synchronize directories from client to server and server to client.

Dealing with errors is an important aspect in a code that involves file operations. A *set of exceptions* are defined and they all have the `RemofileException` exception as base class. Therefore, you can catch most of Remofile-related exceptions in one statement.

```
try:
    do_file_operation()
except RemofileException:
    deal_with_exception()
```

Additionally, some *helper functions* like `generate_token()`, `generate_keys()` and `is_file_name_valid()` are also exposed.

2.2 Two main classes

class remofile.**Client**(*hostname*, *port*, *token*)

Remofile client.

This class implements the client side of Remofile that behaves according to the protocol.

It's a synchronous (non-threaded) and "connectionless" interface to interact with the remote directory. Indeed, a connection is etablished behind the scene but it isn't exposed. As a result, it simplifies the interface and you don't have do deal directly with connections (and eventual re-connections that may happen). Instead, you use a timeout and catch the `TimeoutError` exception that is raised if the time is out. All production code should use a timeout.

All native file operations are implemented such as listing files, creating files and directories, upload/download files and deleting files. Exceptions are raised whenever errors occur or if the file operation couldn't be successfully completed. Also note that directories and symbolic links are also reffered as 'file'.

For more complex file operations such as synchronizing directories or upload/download trees of files, while handling file conflicts and glob pattern, see the algorithm module. Otherwise implement your file operations on top of the client instance.

__init__(*hostname*, *port*, *token*)

Construct a *Client* instance.

The client instance is constructed from the hostname (which can either be "localhost" or any valid IP), the port and the token.

The token should be generated with the *generate_token()* to ensure validity of its value, or a `ValueError` exception is raised.

Parameters

- **hostname** (*str*) – The server IP address (can be "localhost").
- **port** (*int*) – The server port.
- **token** (*str*) – The token to use for authentication.

Raises **ValueError** – If the token isn't valid.

list_files(*directory*, *timeout=None*)

List files in the remote directory.

It lists files of a given directory in a remote directory and returns a dictionnary associating file names with their metadata. The metadata is a tuple that includes a boolean indicating whether the file is a directory or not, the size of the file (the value is 0 in case of directory) and the last modification time of the file.

Return value example.

```
{
    'foo.bin' : (False, 423, 4687421324),
    'bar'     : (True, 0, 1654646515)
}
```

The directory parameter must be a path-like object that refers to an **existing** directory in the remote directory for which it must list files for. If the directory doesn't exist, the `FileNotFoundErrror` exception is raised, and if it's not an actual directory, the `NotADirectoryError` exception is raised. It must be an absolute path or a `ValueError` exception is raised.

If the operation takes longer than the given timeout, a `TimeoutError` exception is raised.

Parameters

- **directory** (*path*) – The given remote directory to list files for.
- **timeout** (*int*) – How many milliseconds to wait before giving up

Raises

- **ValueError** – If the directory is not an absolute path.
- **FileNotFoundError** – If the directory doesn't exist.
- **NotADirectoryError** – If the directory is not a directory.
- **TimeoutError** – If it takes more than the timeout value to receive a response.

Returns A dictionnary associating file name with their metadata.

Return type dict

create_file(*name*, *directory*, *timeout=None*)

Create a file in the remote directory.

It creates an empty file with a given name in a given directory located in the remote directory.

The name parameter must be a string of a *valid file name* and must not conflict with an existing file (or directory) in the given remote directory. If the name isn't valid, a *FileNameError* is raised and if the file is conflicting, a `FileExistsError` exception is raised.

The directory parameter must be a path-like object that refers to an **existing** directory in the remote directory where the file must be created. If the directory doesn't exist, the `FileNotFoundError` exception is raised, and if it's not an actual directory, the `NotADirectoryError` exception is raised. It must be an absolute path or a `ValueError` exception is raised.

If the operation takes longer than the given timeout, a `TimeoutError` exception is raised.

Parameters

- **name** (*str*) – The name of the file to create.
- **directory** (*path*) – The given remote directory where to create the file.
- **timeout** (*int*) – How many milliseconds to wait before giving up

Raises

- **ValueError** – If the directory is not an absolute path.
- **FileNotFound** – If the directory doesn't exist.

:raises NotADirectoryError If the directory is not a directory. :raises FileNameError: If the name of the file isn't valid. :raises FileExistsError: If the name conflicts with the name of an existing file or directory. :raises TimeoutError: If it takes more than the timeout value to receive a response.

make_directory(*name*, *directory*, *timeout=None*)
Create a directory in the remote directory.

It creates an empty directory with a given name in a given directory located in the remote directory.

The name parameter must be a string of a *valid file name* and must not conflict with an existing file (or directory) in the given remote directory. If the name isn't valid, a *FileNameError* is raised and if the file is conflicting, a FileExistsError exception is raised.

The directory parameter must be a path-like object that refers to an **existing** directory in the remote directory where the directory must be created. If the directory doesn't exist, the NotADirectoryError exception is raised, and if it's not an actual directory, the NotADirectoryError exception is raised. It must be an absolute path or a ValueError exception is raised.

If the operation takes longer than the given timeout, a TimeoutError exception is raised.

Parameters

- **name** (*str*) – The name of the file to create.
- **directory** (*path*) – The given remote directory where to create the file.
- **timeout** (*int*) – How many milliseconds to wait before giving up

Raises

- **ValueError** – If the directory is not an absolute path.
- **FileNotFound** – If the directory doesn't exist.

:raises NotADirectoryError If the directory is not a directory. :raises FileNameError: If the name isn't valid. :raises FileExistsError: If the name conflicts with the name of an existing file or directory. :raises TimeoutError: If it takes more than the timeout value to receive a response.

upload_file(*source*, *destination*, *name=None*, *chunk_size=512*, *process_chunk=None*, *timeout=None*)
Upload a file to the remote directory.

This method uploads a single file to a given directory in the remote directory.

The **source** parameter refers to the local file to be transfered to the remote directory and must to be a path-like object. If it's a relative path, it's treated like relative to the current working directory. If the source file can't be found or is not a file, the *SourceNotFound* exception is raised.

The **destination** parameter refers to the remote directory in which the file must be transfered to. It must be a path-like object of an **existing** directory and it must be an absolute path or the ValueError exception is raised. If the destination directory can't be found or is not a directory, the *DestinationNotFound* exception is raised.

The name parameter can be used to rename the source file while uploading it (the content is guaranteed to be the same). It must be a string of a *valid file name* and must not conflict with an existing file (or directory) in the destination directory. By default, it reads the name from the source to leave it unchanged. If the name isn't valid, a *FileNameError* is raised and if the file is conflicting, a FileExistsError exception is raised.

Additionally, you can adjust the chunk size value which defines how fragmented the file has to be sent to the server and/or pass a callback that process each fragment **before** it's sent to the server. Usually, the chunk value is between 512 and 8192.

The callback is called with various parameters and in a specific order; the chunk data, the remaining bytes, the file size and the file name. The chunk data is a bytes string of the actual data about to be sent to the server. The remaining bytes is an integer indicating the number of bytes left to

be sent (and this includes the current chunk of data). The file size is a fixed integer telling how large the file is, and the file name is the file name currently being processed.

For instance, it can be used to display a progress indicator. Here is an example.

```
def display_progress(chunk_data, remaining_bytes, file_size, file_
↪name):
    chunk_size = 512
    progress = (file_size - (remaining_bytes - len(chunk_data))) /␣
↪file_size * 100

    sys.stdout.write("
```

{0:0.2f}% | {1}".format(progress, file_name)) sys.stdout.flush()

if remaining_bytes <= chunk_size: sys.stdout.write('

')

return True

If the operation takes longer than the given timeout, a `TimeoutError` exception is raised.

param source The (local) source file to upload.

param destination The (remote) destination directory where to upload the file.

param name The name of the file after it's uploaded.

param chunk_size How fragmented (in bytes) the file is during the upload process.

param process_chunk Function processing chunks before they are sent out.

param timeout How many milliseconds to wait before giving up.

raises ValueError If the destination directory isn't an absolute path.

raises SourceNotFound If the source file doesn't exist or isn't a file.

raises DestinationNotFound If the destination directory doesn't exist or isn't a directory.

raises FileExistsError If the source file conflicts with an existing file or directory.

raises FileNameError If the source file doesn't have a valid name.

raises ValueError If the chunk size or file size is invalid.

raises TimeoutError If it takes more than the timeout value to receive a response.

upload_directory(*source*, *destination*, *name=None*, *chunk_size=512*, *process_chunk=None*, *timeout=None*)

Upload a directory to the remote directory.

This method uploads an entire directory to a given directory in the remote directory.

The **source** parameter refers to the local directory to be transfered to the remote directory and must to be a path-like object. If it's a relative path, it's treated like relative to the current working directory. If the source directory can't be found or is not a directory, the *SourceNotFound* exception is raised.

The **destination** parameter refers to the remote directory in which the directory must be transfered to. It must be a path-like object of an **existing** directory and it must be an absolute path or the `ValueError` exception is raised. If the destination directory can't be found or is not a directory, the *DestinationNotFound* exception is raised.

The name parameter can be used to rename the source directory while uploading it (the content is guaranteed to be the same). It must be a string of a *valid file name* and must not conflict with an existing directory (or file) in the destination directory. By default, it reads the name from the source to leave it unchanged. If the name isn't valid, a `FileNameError` is raised and if the file is conflicting, a `FileExistsError` exception is raised.

Additionally, you can adjust the chunk size value which defines how fragmented files have to be sent to the server and/or pass a callback that process each fragment **before** it's sent to the server. Usually, the chunk value is between 512 and 8192.

The callback is called with various parameters and in a specific order; the chunk data, the remaining bytes, the file size and the file name. The chunk data is a bytes string of the actual data about to be sent to the server. The remaining bytes is an integer indicating the number of bytes left to be sent (and this includes the current chunk of data). The file size is a fixed integer telling how large the file is, and the file name is the file name currently being processed.

For instance, it can be used to display a progress indicator. Here is an example.

```
def display_progress(chunk_data, remaining_bytes, file_size, file_
↪name):
    chunk_size = 512
    progress = (file_size - (remaining_bytes - len(chunk_data))) /␣
↪file_size * 100

    sys.stdout.write("
```

{0:0.2f}% | {1}".format(progress, file_name)) sys.stdout.flush()

if remaining_bytes <= chunk_size: sys.stdout.write('

')

return True

If the operation takes longer than the given timeout, a `TimeoutError` exception is raised.

param source The (local) source directory to upload.

param destination The (remote) destination directory where to upload the directory.

param name The name of the directory after it's uploaded.

param chunk_size How fragmented (in bytes) files are during the upload process.

param process_chunk Function processing chunks before they are sent out.

param timeout How many milliseconds to wait before giving up.

raises ValueError If the destination directory isn't an absolute path.

raises SourceNotFound If the source directory doesn't exist or isn't a directory.

raises DestinationNotFound If the destination directory doesn't exist or isn't a directory.

raises FileExistsError If the source directory conflicts with an existing file or directory.

raises FileNameError If the source directory doesn't have a valid name.

raises ValueError If the chunk size or file size is invalid.

raises TimeoutError If it takes more than the timeout value to receive a response.

download_file(*source*, *destination*, *name=None*, *chunk_size=512*, *process_chunk=None*, *timeout=None*)

Download a file from the remote directory.

This method downloads a single file from a given directory in the remote directory.

The **source** parameter refers to the remote file to be transfered from the remote directory and must to be a path-like object. It must be an absolute path or it will raise the ValueError exception. If the source file can't be found or is not a file, the SourceNotFound exception is raised.

The **destination** parameter refers to **an existing** local directory in which the file must be transfered to. It must be a path-like object and if it's a relative path, it's treated like relative to the current working directory. If the destination directory can't be found or is not a directory, the DestinationNotFound exception is raised.

The name parameter can be used to rename the source file while downloading it (the content is guaranteed to be the same). It must be a string of a *valid file name* and must not conflict with an existing file (or directory) in the destination directory. By default, it reads the name from the source to leave it unchanged. If the name isn't valid, a `FileNameError` is raised and if the file is conflicting, a `FileExistsError` exception is raised.

Additionally, you can adjust the chunk size value which defines how fragmented the file has to be received from the server and/or pass a callback that process each fragment **before** it's written to the local file. Usually, the chunk value is between 512 and 8192.

The callback is called with various parameters and in a specific order; the chunk data, the remaining bytes, the file size and the file name. The chunk data is a bytes string of the actual data just received from the server. The remaining bytes is an integer indicating the number of bytes left to be received (and this includes the current chunk of data). The file size is a fixed integer telling how large the file is, and the file name is the file name currently being processed.

For instance, it can be used to display a progress indicator. Here is an example.

```
def display_progress(chunk_data, remaining_bytes, file_size, file_
↪name):
    chunk_size = 512
    progress = (file_size - (remaining_bytes - len(chunk_data))) /␣
↪file_size * 100

    sys.stdout.write("
```

{0:0.2f}% | {1}".format(progress, file_name)) sys.stdout.flush()

if remaining_bytes <= chunk_size: sys.stdout.write('

')

return True

If the operation takes longer than the given timeout, a `TimeoutError` exception is raised.

param source The (remote) source file to download.

param destination The (local) destination directory where to download the file.

param name The name of the file after it's downloaded.

param chunk_size Foobar.

param process_chunk Foobar.

param timeout How many milliseconds to wait before giving up.

raises ValueError If the source directory isn't an absolute path.

raises SourceNotFound If the source file doesn't exist or isn't a file.

raises DestinationNotFound If the destination directory doesn't exist or isn't a directory.

raises FileExistsError If the source file conflicts with an existing file or directory.

raises FileNameError If the source file doesn't have a valid name.

raises TimeoutError If it takes more than the timeout value to receive a response.

download_directory(*source*, *destination*, *name=None*, *chunk_size=512*, *process_chunk=None*, *timeout=None*)

Download a directory from the remote directory.

This method downloads an entire directory from a given directory in the remote directory.

The **source** parameter refers to the remote directory to be transfered from the remote directory and must to be a path-like object. It must be an absolute path or it will raise the ValueError exception. If the source directory can't be found or is not a directory, the SourceNotFound exception is raised.

The **destination** parameter refers to **an existing** local directory in which the directory must be transfered to. It must be a path-like object and if it's a relative path, it's treated like relative to the current working directory. If the destination directory can't be found or is not a directory, the DestinationNotFound exception is raised.

The name parameter can be used to rename the source directory while downloading it (the content is guaranteed to be the same). It must be a string of a *valid file name* and must not conflict with an existing directory (or file) in the destination directory. By default, it reads the name from the source to leave it unchanged. If the name isn't valid, a *FileNameError* is raised and if the file is conflicting, a FileExistsError exception is raised.

Additionally, you can adjust the chunk size value which defines how fragmented files have to be received from the server and/or pass a callback that process each fragment **before** it's written to the local file. Usually, the chunk value is between 512 and 8192.

The callback is called with various parameters and in a specific order; the chunk data, the remaining bytes, the file size and the file name. The chunk data is a bytes string of the actual data just received from the server. The remaining bytes is an integer indicating the number of bytes left to be received (and this includes the current chunk of data). The file size is a fixed integer telling how large the file is, and the file name is the file name currently being processed.

For instance, it can be used to display a progress indicator. Here is an example.

```
def display_progress(chunk_data, remaining_bytes, file_size, file_
↪name):
    chunk_size = 512
    progress = (file_size - (remaining_bytes - len(chunk_data))) /␣
↪file_size * 100

    sys.stdout.write("
```

{0:0.2f}% | {1}".format(progress, file_name)) sys.stdout.flush()

if remaining_bytes <= chunk_size: sys.stdout.write('

')

return True

If the operation takes longer than the given timeout, a `TimeoutError` exception is raised.

param source The (remote) source directory to download.

param destination The (local) destination directory where to download the directory.

param name The name of the directory after it's downloaded.

param chunk_size Foobar.

param process_chunk Foobar.

param timeout How many milliseconds to wait before giving up.

raises ValueError If the source directory isn't an absolute path.

raises SourceNotFound If the source file doesn't exist or isn't a file.

raises DestinationNotFound If the destination directory doesn't exist or isn't a directory.

raises FileExistsError If the source file conflicts with an existing file or directory.

raises FileNameError If the source file doesn't have a valid name.

raises TimeoutError If it takes more than the timeout value to receive a response.

delete_file(*timeout=None*)

Delete a file in the remote directory.

Long description.

Parameters **timeout** (*int*) – How many milliseconds to wait before giving up

class remofile.**Server**(*root_directory*, *token*, *private_key=None*, ***kwargs*)

Remofile server.

This class implements the server side of Remofile that behaves according to the protocol.

It's a single-threaded server that will jail a given directory called **root directory**, and exposes it on Internet using a given range of IP addresses and a port. It doesn't start listening to connections until the blocking *run()* method is called. To interrupt the loop, call the *terminate()* method from a another thread. The server can also be configured with various options.

The *run()* may be found within a try-except statement to catch the `KeyboardInterrupt()` exception during testing.

```
try:
    server.run(port)
except KeyboardInterrupt:
    server.terminate()
```

The configuration options includes the file size limit and the chunk size range. The file size limit prevents clients from transfering files exceeding a given size (expressed in bytes), and the chunk size range prevents clients from

__init__(*root_directory*, *token*, *private_key=None*, ***kwargs*)

Construct a *Server* instance.

The server instance is constructed from the root directory, the token, an optional private key and the server options values.

The root directory parameter must be a path-like object of an **existing** directory or a NotADirectoryError exception is raised. The token and private key should respectively be generated with the *generate_token()* and *generate_keys()* to ensure validity of their values, or it will raise a ValueError.

For the other parameters, check out the corresponding server options properties *file_size_limit* and *chunk_size_range*.

Parameters

- **root_directory** – The directory which will be exposed to clients.
- **token** – The token that clients must use to be granted access.
- **private_key** – The private key to use to encrypt communication with clients.
- **file_size_limit** – The file size limit (in bytes) files can't exceed during upload/download.
- **chunk_size_range** – The minimum and maximum chunk size (in bytes) allowed during upload/download.

Raises

- **NotADirectoryError** – If the root directory isn't a valid (isn't a directory or doesn't exist).
- **ValueError** – If the token or private_key aren't valid.

root_directory

The directory to be exposed.

The root directory must be a path-like object that refers to an existing directory. If a relative path is passed, it's combined with the current working directory to get an absolute path. If the root directory doesn't exist, the NotADirectoryError exception is raised.

Note that the root directory can't be changed while the server is running, or it will raise a RuntimeError exception.

Getter Returns the root directory.

Setter Changes the root directory.

Type path-like object

Raises

- **NotADirectoryError** – If the root directory doesn't exist or isnt' a directory.
- **RuntimeError** – If the value is changed while the server is running.

file_size_limit

The file size limit.

The file size limit must be an integer specificying in bytes the maximum size a file can have to be accepted and transferred. The value can't zero or negative or it will raise a ValueError exception.

Note that the file size limit can't be changed while the server is running, or it will raise a RuntimeError exception.

Getter Returns the file size limit in bytes.

Setter Changes the file size limit in bytes.

Type integer

Raises

- **ValueError** – If the value is less or equal to 0.
- **RuntimeError** – If the value is changed while the server is running.

chunk_size_range
The chunk size range.

The chunk size range must be a tuple of two integers specifying the minimum chunk size and maximum chunk size (in bytes) allowed during negotiating upload and download transfers. The minimum chunk size value can't zero or negative or it will raise a ValueError exception.

Note that the chunk size range can't be changed while the server is running, or it will raise a RuntimeError exception.

Getter Returns the chunk size range.

Setter Changes the chunk size range.

Type tuple of two integer

Raises

- **ValueError** – If the value is less or equal to 0.
- **RuntimeError** – If the value is changed while the server is running.

run(*port*, *address=None*)
Start the main loop.

This method starts the main loop after it initializes the sockets which listen on a given port and optionally a specific IP address. By default, it listens on all available IP addresses. If it's unable to listen on a specific port and/or IP address, it will raise a `RuntimeError` exception.

It's a blocking method that won't return until the *terminate()* method is called. Usually, the *run()* method is called from an external thread and the main thread calls the *terminate()* method.

Parameters

- **port** (*int*) – The port to listen.
- **address** (*str*) – The IP address to use (all available IP addresses by default).

Raises

- **RuntimeError** – If the server is unable to listen on the IP address(es) and/or the port.
- **RuntimeError** – If the server is already running.

terminate()
Terminate the main loop.

This method interrupt the main loop causing the server to terminate. It can safely be called from a different thread where the initial call to *run()* was made. If the server wsa transferring files, operations are all interupted and the client is disconnected.

Raises **RuntimeError** – If the server is not running.

2.3 Advanced algorithms

A bunch of advanced algorithms are implemented on top of the primitive file operations because in practice, we need more than that.

You will particularly want to have a look at *upload_files()* and *download_files()* as they provide a more sophisticated interface to fine-grain control what is to be uploaded or downloaded. And you

may also need synchronization features in both directions which is done with *synchronize_upload()* and *synchronize_download()*.

remofile.**upload_files**(*client*, *source*, *destination*, *relative_directory*)
Upload files to the remote directory.

This method uploads one file, an entire directory or a set of files (specified by glob patterns) to a given directory in the remote directory.

The source parameter refers to the local file(s) to be transfered to the remote directory and must to be a path-like object of a relative directory that can contain glob patterns. Source files are related to the directory specified by the relative_directory parameter which itself must be a path-like object of an **existing** absolute directory. By default, the relative directory is the current working directory. If the source is a directory or refers to sub-directories, the recursive parameter must be set.

The destination parameter refers to the remote directory in which the file(s) must be transfered to. It must be a path-like object aof an **existing** absolute directory.

The exclude_files parameter is a sequence of path-like objects of relative path (that may or may not contain glob patterns) that refers to files to be excluded from the transfer.

The timeout parameter corresponds to time allowed to send each chunk.

Parameters

- **source** – Foobar.
- **destination** – Foobar.
- **chunk_size** (*int*) – Foobar.
- **process_chunk** – Foobar.
- **timeout** (*int*) – How many milliseconds to wait before giving up

remofile.**download_files**(*client*, *source*, *destination*, *relative_directory*)
Download files from the remote directory.

This method downloads one file, an entire directory or a set of files (specified by shell glob pattern) to a given directory in in the local filesystem. Additional parameters are there to refine the process.

The source parameter refers to the remote file(s) to be transfered to the local filesystem and is expected to be a path-like object, unless it's a shell glob pattern in which case a string is expected. The path must be relative and is by default relative to the root directory. It can be altered with the relative_directory parameter which itself must be path-like object refering to an absolute directory.

The destination parameter refers to the remote directory in which the file(s) must be transfered to. It must be a path-like object and must be absolute.

Long description.

Parameters

- **source** – Foobar.
- **destination** – Foobar.
- **chunk_size** (*int*) – Foobar.
- **process_chunk** – Foobar.
- **timeout** (*int*) – How many milliseconds to wait before giving up

remofile.**synchronize_upload**(*client*, *source*, *destination*)
Synchronize remote files with a local directory.

Long description.

remofile.**synchronize_download**(*client*, *source*, *destination*)
Synchronize local files with a remote directory.

Long description.

2.4 Handling exceptions

This section isn't written yet.

exception remofile.**RemofileException**
Base exception for Remofile-related exceptions.

Remofile-related exceptions excludes filesystem-related exceptions, except for *SourceNotFound* and *DestinationNotFound* which exist to simplify catching exceptions in some functions.

exception remofile.**SourceNotFound**
Source is not found.

This exception is raised in upload/download/synchronize related functions to catch the numberous different exceptions that a bad source could raise.

It's triggered if a source file doesn't exist, or if it isn't a file or a directory (according to the context).

exception remofile.**DestinationNotFound**
Destination is not found.

This exception is raised in upload/download/synchronize related functions to catch the numberous different exceptions that a bad destination could raise.

It's triggered if a destination directory doesn't exist or if it's not a directory.

exception remofile.**UnexpectedError**(*message*)
Unexpected error occured.

This exception is raised whenever the server couldn't fulfill the request because an unexpected error occured on the server side.

It's well explained in the protocol specifications *document*.

Variables **message** (*str*) – Underlying exception message that occured on the server.

exception remofile.**BadRequestError**
Bad request error occured.

This exception is raised when the client sends a badly formatted request. For instance, it can occur when the server isn't ready to handle the request because it's not in a valid state.

It's well explained in the protocol specifications *document*.

exception remofile.**CorruptedResponse**(*message*, *error*)
Corrupted response was received.

This exception is raised when the client is unable to process the response returned by the server. The client stricly implements the protocol and if a response isn't expected or doesn't have the correct format, the response is said corrupted.

A `message` describing how the response could not be processed is available in attribute.

Examples of message.

- Unable to extract response type from response

- Invalid reason type in refuse response
- Unable to extract message from error response

The `error` attribute is the underlying exception message that was raised while processing the corrupted response.

Variables

- **message** (*str*) – Explicit message explaining how response is corrupted.
- **error** (*str*) – Underlying exception message.

exception remofile.**FileNameError**

File name is invalid.

This exception is raised when the file name is incorrect as stated by the Remofile protcol.

2.5 Helper functions

These helper functions will come in handy to generate a random valid token and generate a pair of private and public key to configure the server with. There is also a function to check validity of a filename as defined by the Remofile protocol.

remofile.**generate_token**()

Brief description.

Long description.

remofile.**generate_keys**()

Brief description.

Long description.

remofile.**is_file_name_valid**(*name*)

Brief description.

Long description.

Parameters **name** – Description.

CHAPTER 3

Commands List

This document is the reference of the command-line interface (CLI) of **Remofile**, which allows one to start a server and perform various file operations in a sessionless fashion, from a terminal.

Commands are divided into *client-related commands* and *server-related commands*.

3.1 Client-related commands

The client-related commands consists of 6 commands, **list**, **file**, **folder**, **upload**, **download** and **remove**, and they all rely on a previously configured environment to locate the remote server and pass the authentication process. Therefore, a set of 4 environment variables must be set or the command-lines will stop prematurely.

REMOFILE_HOSTNAME

The address of the Remote server. Set it to 'localhost' if the server runs locally.

REMOFILE_PORT (optional)

The port to use when connecting to the server. By default, it uses **6768**.

REMOFILE_TOKEN

The token with which the server was configured (a token was generated by the server if it wasn't explicitly set).

REMOFILE_PUBLIC_KEY (optional)

The public key that goes in pair with the private key used on the server side. By default, a **Remofile** connection is not encrypted.

Example of configuring an environment on Unix-like OSes.

```
export REMOFILE_HOSTNAME=localhost
export REMOFILE_PORT=6768
export REMOFILE_TOKEN=something
export REMOFILE_PUBLIC_KEY=something
```

3.1.1 rmf list

List files in the remote directory.

This is a client-related command that lists files of a given directory located in the remote directory. This command is akin to the POSIX **ls** command found in Unix-like OSes.

It takes only one **optional** parameter which is the remote directory to list files for, and must be an absolute path of an **existing** directory. By default, it lists the root directory.

By default, it only displays file names and doesn't list the directory recursively. If the **-l** flag is set, it also lists the file metadata (file or directory indicator, file size and last modification time), and if the **-r** flag is set, the sub-directories are listed as well.

Additionally, the **–timeout** flag allows you to adjust the number of milliseconds to wait before giving up on the server response.

```
rmf list [OPTIONS] [DIRECTORY]
```

Options

-a, --all
Display additional file information.

-r, --recursive
List directories and their contents recursively.

-t, --timeout <timeout>
Adjust the timeout value in milliseconds.

Arguments

DIRECTORY
Optional argument

3.1.2 rmf file

Create a file in the remote directory.

This is a client-related command that creates an empty file in the a given directory located in the remote directory. This command is akin to the POSIX **touch** command found in Unix-like OSes.

It takes the name of the file and an optional remote directory (in which to create the file) in parameters. The directory parameter must be an absolute path of an **existing** directory. By default, it creates the file in the root directory.

If the file already exists in the given directory, the command fails unless the **–update** flag is set. Note that unlike the *touch* command, it doesn't update the file timestamp.

Additionally, the **–timeout** flag allows you to adjust the number of milliseconds to wait before giving up on the server response.

```
rmf file [OPTIONS] NAME [DIRECTORY]
```

Options

-u, --update
Ignore (and don't fail) if files already exist.

-t, --timeout <timeout>
Adjust the timeout value in milliseconds.

Arguments

NAME
Required argument

DIRECTORY
Optional argument

3.1.3 rmf folder

Create a folder in the remote directory.

This is a client-related command that creates an empty folder in the a given directory located in the remote directory. This command is akin to the POSIX **mkdir** command found in Unix-like OSes.

It takes the name of the folder and an optional remote directory (in which to create the folder) in parameters. The directory parameter must be an absolute path of an **existing** directory. By default, it creates the folder in the root directory.

If the folder already exists in the given directory, the command fails unless the **–update** flag is set. Note that it leaves the existing directory unchanged.

Additionally, the **–timeout** flag allows you to adjust the number of milliseconds to wait before giving up on the server response.

```
rmf folder [OPTIONS] NAME [DIRECTORY]
```

Options

-u, --update
Ignore (and don't fail) if directories already exist.

-t, --timeout <timeout>
Adjust the timeout value in milliseconds.

Arguments

NAME
Required argument

DIRECTORY
Optional argument

3.1.4 rmf upload

Upload files to the remote directory.

This is a client-related command that uploads files to the remote directory. The source must be files or directories located on the local filesystem and the destination must be an **existing** directory located in the remote directory. Unlike the source, the destination must be an absolute path.

If source refers to one or more directories, the recursive flag must be set otherwise they'll be skipped. The progress flag allows to display the progression of the transfer which is useful for large files.

Examples.

```
rmf upload -r -p src/my-file.txt src/my-directory/ /dst
```

Additionally, the **–timeout** flag allows you to adjust the number of milliseconds to wait before giving up on the server response.

```
rmf upload [OPTIONS] [SOURCE]... DESTINATION
```

Options

-r, --recursive
Upload directories and their content recursively.

-p, --progress
Display a progress indicator.

-t, --timeout <timeout>
Adjust the timeout value in milliseconds.

Arguments

SOURCE
Optional argument(s)

DESTINATION
Required argument

3.1.5 rmf download

Download files from the remote directory.

This is a client-related command that downloads files from the remote directory. The source must be files or directories located on the remote directory and the destination must be an **existing** directory located on the local filesystem. Unlike the destination, the source must be absolute paths.

If source refers to one or more directories, the recursive flag must be set otherwise they'll be skipped. The progress flag allows to display the progression of the transfer which is useful for large files.

Examples.

```
rmf download -r -p /src/my-file.txt /src/my-directory/ dst/
```

Additionally, the **–timeout** flag allows you to adjust the number of milliseconds to wait before giving up on the server response.

```
rmf download [OPTIONS] [SOURCE]... DESTINATION
```

Options

-r, --recursive
Download directories and their content recursively.

-p, --progress
Display a progress indicator.

-t, --timeout <timeout>
Adjust the timeout value in milliseconds.

Arguments

SOURCE
Optional argument(s)

DESTINATION
Required argument

3.1.6 rmf remove

Remove files in the remote directory.

This is a client-related command that removes a file or a folder located in the remote directory. This command is akin to the POSIX **rm** command found in Unix-like OSes.

Rest of the description here.

```
rmf remove [OPTIONS] NAME [DIRECTORY]
```

Options

-t, --timeout <timeout>
Adjust the timeout value in milliseconds.

Arguments

NAME
Required argument

DIRECTORY
Optional argument

3.2 Server-related commands

Unlike the client-related commands, server-related commands don't require to configure the environment. There are 3 main commands which are **run**, **start** and **stop**, and they allow you to start and stop a Remofile with various options. There are also two utility commands which are **generate-token** and **generate-keys**.

3.2.1 rmf run

Start a (non-daemonized) server.

This is a server-related command that start a non-daemonized server (not detached from the shell). The directory parameter is the root directory which will be served and therefore must be an existing directory. The server listens on port 6768 by default but it can be changed with the port parameter. If the token is not specified, it's generated and printed out to the console before the server starts running.

Additionally, the file size limit and the chunk size range can be altered. The file size limit and minimum chunk size must be both be greater than 0, and maximum chunk size must be greater or equal to minimum chunk size.

```
rmf run [OPTIONS] DIRECTORY [PORT] [TOKEN]
```

Options

--file-size-limit <file_size_limit>
Foobar

--min-chunk-size <min_chunk_size>
Foobar

--max-chunk-size <max_chunk_size>
Foobar

Arguments

DIRECTORY
Required argument

PORT
Optional argument

TOKEN
Optional argument

3.2.2 rmf start

Start a daemonized server.

This is a server-related command that start a daemonized server (detached from the shell). Unlike the run command, it accepts the –pidfile flag which tells the pidfile location. By default, the pidfile is created in the current working directory and named 'daemon.pid'.

Refer to the run command for more information.

```
rmf start [OPTIONS] DIRECTORY [PORT] [TOKEN]
```

Options

--pidfile <pidfile>
Foobar

--file-size-limit <file_size_limit>
Foobar

--min-chunk-size <min_chunk_size>
Foobar

--max-chunk-size <max_chunk_size>
Foobar

Arguments

DIRECTORY
Required argument

PORT
Optional argument

TOKEN
Optional argument

3.2.3 rmf stop

Stop a daemonized server.

This is a server-related command that stop a daemonized server from its pidfile. By default, it expects the pidfile in the current working directory with the name 'daemon.pid' but it can be altered with the –pidfile flag.

```
rmf stop [OPTIONS]
```

Options

--pidfile <pidfile>
Foobar

3.2.4 rmf generate-token

Generate a token.

This is an utility command that generates a valid token needed to configure both the client and the server.

Note that by default, the server will generate a token if none was explicitly set.

```
rmf generate-token [OPTIONS]
```

3.2.5 rmf generate-keys

Generate a pair of keys.

This is an utility command that generates a valid pair of keys to encrypt communication with clients.

The first key is a public key that must be shared across clients connecting to the Remofile server and the second key is the private key that must be kept secret. Both `Client` and `Server` instances must be configured with their respective keys.

```
rmf generate-keys [OPTIONS]
```

CHAPTER 4

Protocol Specifications

This document defines the protocol used in **Remofile** that clients use to transfer files back and forth from/to the remote server.

- Request and response pattern
- The four transferring states
- Error and refused responses
- File name validity
- Absolute and relative directory
- List files request
- Create file request
- Make directory request
- Upload request-response cycle
- Download request-response cycle
- Delete file resquest

See client.py and server.py source file for an example of the implementation of this protocol.

valid file name This is the description of valid file name term.

```
Todo:
- talk about timeout (stateless connection), immediate consequence of the underlying␣
↪ZeroMQ protocol
- talk about authentication and encryption
- talk about zeromq socket identity
- talk about timeout
- talk about authentication and encryption
```

4.1 Request and response pattern

The protocol is entirely based on **ZeroMQ** using a single pair of socket in the **REQ-REP** mode. Documenting and understanding becomes easy because the client and the server are locked in a two steps communication pattern; the client sends one **request** and the server sends back one **response**. Additionally, all responses may come along with a **reason**.

The list of possible requests.

- LIST_FILES
- CREATE_FILE
- MAKE_DIRECTORY
- UPLOAD_FILE
- SEND_CHUNK
- DOWNLOAD_FILE
- RECEIVE_CHUNK
- CANCEL_TRANSFER
- REMOVE_FILE

The list of possible responses.

- ACCEPTED
- REFUSED
- ERROR

The list of possible reasons.

- FILE_LISTED
- FILE_CREATED
- DIRECTORY_CREATED
- INVALID_FILE_NAME
- FILE_NOT_FOUND
- FILE_ALREADY_EXISTS
- NOT_A_FILE
- NOT_A_DIRECTORY
- INCORRECT_FILE_SIZE
- INCORRECT_CHUNK_SIZE
- TRANSFER_ACCEPTED
- CHUNK_ACCEPTED
- CHUNK_SENT
- TRANSFER_COMPLETED
- TRANSFER_CANCELLED
- BAD_REQUEST

- UNKNOWN_ERROR

Requests and reasons are **Python tuple** whose first element is a **request type** for requests, or **response type** for responses. Information related to the request or response compose the rest of the tuple. The tuple and the elements in it are serialized and sent over using the *send_pyobj()* and *recv_pyobj()* method. Refer to the **pyZMQ** documentation to understand the serialization process.

4.2 The four transferring states

There are four transferring states in which the server can be and they condition the possible responses to a request. These states are **exclusive**; the server is at one of those states at a time.

The four transferring states.

- IDLE
- UPLOAD
- DOWNLOAD
- DELETE

At connection time, the transferring state always is **IDLE** and because it's a one to one network architecture, the client is expected to be aware of the server's current state at any time; there is no getter to know the current transferring state.

Some file operations can take long to complete and the transferring states allow to model their processing time. For instance, uploading will happen in several request-response cycles when the server is marked in the **UPLOAD** transferring state. Unrelated requests to uploading a file during this process obviously are errors.

4.3 Error and refused responses

The difference between a **REFUSED** and **ERROR** response lies in the *correct usage of the protocol* and *the expected behavior*.

Regardless of the current business, the client is expected to communicate **flawlessly** with the server in a known language and the server is expected to deal with all possible errors that may happen during the fulfillment of the request... and reply with an ACCEPTED or **REFUSED** response.

If the client sends a bad request, this is an **ERROR** because it failed to follow the protocol specifications. If an unexpected error occurs on the server side, this is an **ERROR**. All other events such as a failure to complete a request because of possible unmet runtime conditions are **not** errors.

We also notice that there are only two possible error responses; bad requests and unknown errors. Unknown errors come along with a message describing the error. Theoretically, all requests may return an **ERROR** response.

4.4 File name validity

The name of the file must be valid which is any sequence that doesn't contain one the following forbidden character.

- <
- >
- :
- /

- \
- |
- ?
- *

Abc.

4.5 Absolute and relative directory

Unlike the local filesystem, there is no notion of **current working directory** when working with the remote directory exposed by Remofile.

As a direct consequence, all paths that refer to the remote directory should be absolute paths. If relative paths are given to the server, an implementation can be tolerant and constructs an absolute paths out of the relative paths by combining them to the root directory.

4.6 List files request

Listing files is the only request that can be made regardless of the current transferring state. It's a non-lasting operation that should be canceled on the client side with a timeout value if ever the server takes long to reply.

The **LIST_FILES** request is constructed with the path to the directory to list files for.

Request example:

```
request = (Request.LIST_FILES, '/foo/bar')
```

This will list the */foo/bar* directory and compute their metadata if the request is accepted. Metadata is a tuple that includes a boolean indicating whether the file is a directory or not, the size of the file (the value is 0 in case of directory) and the last modification time of the file.

```
response = (Response.ACCEPTED, Reason.FILE_LISTED,
    {'foo.bin' : (False, 423, 4687421324), 'bar' : (True, 0, 1654646515)})
```

The path to the directory to list files for must be an absolute path that refers to an **existing directory**. Possible refuse reason is **NOT_A_DIRECTORY** if this directory doesn't exist.

Another response include **BAD_REQUEST** error response if the directory to list files for isn't an absolute path.

4.7 Create file request

Creating a file can only happen when the file server is in the **IDLE** state. It's a non-lasting operation that should be canceled on the client side with a timeout value if ever the server takes long to reply.

The **CREATE_FILE** request is constructed with the **name of the file** to be created followed by the **destination directory**.

Request example.

```
request = (Request.CREATE_FILE, 'qaz.bin', '/foo/bar')
```

This will create an empty file with name *qaz.bin* in */foo/bar* directory if the request is accepted.

Response example.

```
response = (Response.ACCEPTED, Reason.FILE_CREATED)
```

The name of the file must be *valid file name* that doesn't conflict with an existing file (or directory) in the destination directory. The destination directory must be an absolute path of an **existing directory**.

Possible refuse reasons.

- **INVALID_FILE_NAME** when the file name isn't valid
- **NOT_A_DIRECTORY** when the destination directory doesn't exist
- **FILE_ALREADY_EXISTS** when it conflicts with an existing file (or directory)

Another response include **BAD_REQUEST** error response if the destination directory isn't an absolute path.

4.8 Make directory request

Creating a directory can only happen when the file server is in the **IDLE** state. It's a non-lasting operation that should be canceled on the client side with a timeout value if ever the server takes long to reply.

The **MAKE_DIRECTORY** request is constructed with the **name of the directory** to be created followed by the **destination directory**.

Request example.

```
request = (Request.MAKE_DIRECTORY, 'qaz', '/foo/bar')
```

This will create an empty directory with name *qaz* in */foo/bar* directory if the request is accepted.

Response example.

```
response = (Response.ACCEPTED, Reason.DIRECTORY_CREATED)
```

The name of the directory must be *valid file name* that doesn't conflict with an existing directory (or file) in the destination directory. The destination directory must be an absolute path of an **existing directory**.

Possible refuse reasons.

- **INVALID_FILE_NAME** when the directory name isn't valid
- **NOT_A_DIRECTORY** when the destination directory doesn't exist
- **FILE_ALREADY_EXISTS** when it conflicts with an existing directory (or file)

Another response include **BAD_REQUEST** error response if the destination directory isn't an absolute path.

4.9 Upload request-response cycle

Initiating an upload will turn the server into **UPLOAD** state and it can only be requested when the server is in **IDLE** mode. Transfers is interrupted in the middle if an error occurs on the server side, or can be explicitly interrupted on request by the client.

The three requests involved in uploading files are.

- UPLOAD_FILE

- SEND_CHUNK
- CANCEL_TRANSFER

The **UPLOAD_FILE** request initiates the uploading process and turns the server into **UPLOAD** state. The subsequent requests must either be **SEND_CHUNK** to send the file data to the server, or **CANCEL_TRANSFER** to interrupt the transfer. When the file data is entirely sent over (when all data chunks are sent) or if the transfer explicitely interrupted, the server goes back to **IDLE** state.

4.9.1 The upload file request

The **UPLOAD_FILE** request is constructed with the **file name**, the **destination directory**, the **file size** and the **chunk size**.

Request example.

```
request = (Request.UPLOAD_FILE, 'qaz.bin', '/foo/bar', 23735613, 4096)
```

This request initiates the upload of the file *qaz.bin* (supposedly located on the client file system) to the */foo/bar* directory (on the server side). The file is 23735613 bytes long and has to be transfered by chunk of 4096 bytes. If the response is accepted, the client and server have now agreed upon a given cycle of upload request-response.

Response example.

```
response = (Response.ACCEPTED, Reason.TRANSFER_ACCEPTED)
```

The file name must be a *valid file name* that doesn't conflict with an existing file (or directory) in the destination directory. The destination directory must be an absolute path to an **existing directory**. The file size can't be 0 or greater than the maximum set by the server, and the chunk size must be within the range set on the server side (by default between 512 and 8192).

Possible refuse reasons.

- **INVALID_FILE_NAME** when the file name isn't valid
- **FILE_ALREADY_EXISTS** when the file to upload conflicts with an existing file (or directory)
- **NOT_A_DIRECTORY** when the destination directory doesn't exist
- **INVALID_FILE_SIZE** when the file size is invalid
- **INVALID_CHUNK_SIZE** when the chunk size is invalid

Another response include **BAD_REQUEST** error response if the destination directory isn't an absolute path.

4.9.2 The send chunk request

The **SEND_CHUNK** request is constructed with the chunk data which is a byte string with exactly length as initially defined.

Request example.

```
request = (Request.SEND_CHUNK, b'F\x8c1\xa4\xb5\xc7')
```

This will move the upload process one step forward by sending the next 6 bytes (if the chunk size) was set at 6 (unlikely). This writes the next 6 bytes to the uploaded bytes on the server side if the request is accepted.

Response example.

```
response = (Response.ACCEPTED, Reason.CHUNK_RECEIVED)
response = (Response.ACCEPTED, Reason.TRANSFER_COMPLETED)
```

The file size is given for the server to understand when the uploading process is completed. The chunk size defines how much data is sent per request-response and therefore will define how many of them.

Example:

```
request = {
    'type' : Request.UPLOAD_FILE,
    'destination' : '/my/directory',
    'file-name' : 'myfile',
    'file-size' : 1687365,
    'chunk-size' : 512
}
```

After the upload is initiated (the server responded with **TRANSFER_ACCEPTED**)

> and that means the server is now in **UPLOADING** state and ready to receive chunks. This is unless the server replies with one the following error response.

- INCORRECT_STATE
- NOT_A_DIRECTORY_ERROR
- FILE_EXISTS_ERROR

The next set of requests (and responses) are repeated **SEND_CHUNK** that carries the chunk data. Server reply with CHUNK_ACCEPTED.

The client is expected to send repeatedly chunks of the file data until the transfer is completed.

SEND_CHUNK ACCEPTED, CHUNK_ACCEPTED ACCEPTED, TRANSFER_COMPLETED

In case the client explicitly cancel the transfer, it sends CANCEL_TRANSFER and server replies with, ACCEPTED, TRANSFER_CANCELLED

In case client sends invalid chunk, the server response ERROR, BAD_REQUEST Beware, it will also cancel the current transfer and put the server back to IDLE state.

In case an error occurs, ERROR, UNKNOWN_ERROR

4.10 Download request-response cycle

Downloading can only happen when the file server is in the **IDLE** mode. Initiating a download will turn the server into **DOWNLOAD** state. Transfer can be interrupted in the middle because of an error on the server side, or can be explicitly interrupted on the client side.

The three requests involved in downloading files are.

- DOWNLOAD_FILE
- RECEIVE_CHUNK
- CANCEL_TRANSFER

The **DOWNLOAD_FILE** request initiates the downloading process and turns the server into **DOWNLOAD** state. The subsequent requests are either **RECEIVE_CHUNK** to receive the file data from the server, or **CANCEL_TRANSFER** to interrupt the transfer. When the file data is entirely received (when all data chunks are received) or the transfer interrupted, the server is put back into **IDLE** state.

4.10.1 The download file request

Long description.

Request example.

```
request = (Request.DOWNLOAD_FILE, args)
```

Long description.

Response example.

```
request = (Response.ACCEPTED, Reason.TRANSFER_ACCEPTED)
```

Long description.

4.10.2 The receive chunk request

Long description.

Request example.

```
request = (Request.RECEIVE_CHUNK, args)
```

Long description.

Response example.

```
request = (Response.ACCEPTED, Reason.CHUNK_SENT)
```

Long description.

4.10.3 The cancel transfer request

Long description.

Request example.

```
request = (Request.CANCEL_TRANSFER, args)
```

Long description.

Response example.

```
request = (Response.ACCEPTED, Reason.TRANSFER_CANCELLED)
```

Long description.

Note: The downloading state is akin to the uploading state.

The different with downloading is, instead of sending the file size information, it's received from the server.

directory = '/my-software' filename = 'Win7.iso' chunk_size = 512

request = (Request.DOWNLOAD_FILE, directory, filename, chunk_size)

Server returns the actual (optimal?) chunk size to be used. Long description.

response = (Response.ACCEPTED, Reason.TRANSFER_ACCEPTED, chunk_size)

4.11 Delete file request

To be written.

CHAPTER 5

Design Decisions

This document presents **Remofile** from the creator perspective and justifies the software design decisions that were taken all along the development.

- Reinventing the wheel
- A simpler concept
- File ownership, permissions and timestamp
- Not tuned for performance
- Upcoming improvements

5.1 Reinventing the wheel

Transferring files is not new; it's an essential need since the beginning of the Internet. Our options when it comes to transferring files are FTP and their variant (or even HTTP), usually tunneled through SSH for security. But those are ancient, sophisticated, and overly complicated solutions in most situations. For instance, if you are creating a client-server software and need to transfer files between multiple endpoints, how do you use FTP (and its hundreds of tools and libraries out there) in without making the code of your software ugly and over-complicated. No surprise that most application prefer roll their own mini-solution by re-implementing basic file transfer operations on top of other protocols.[4][7]

A modern and lightweight alternative to FTP would be a better fit when simply uploading and downloading a couple of files, or synchronizing directories is needed. Doing these essential file operations should be effortless in 2018, either at a command-line level or programming level. This is why Remofile "reinvents the wheel", with a simpler implementation and a much nicer interface for both sides, client and server.

[4] Gitlab Runner, Buildbot, Jenkins and most CI services have custom code to transfer source code back and forth.

[7] Joe Armstrong, creator of Erlang, complains about FTP and write his own quick solution: http://armstrongonsoftware.blogspot.com/2006/09/why-i-often-implement-things-from.html

5.2 A simpler concept

Remofile is not much different from FTP in terms of concept; it jails a given directory on the server side and exposes it to the client. But unlike FTP, it does it with few differences.

- There is no concurrent connections
- There is no complex authentication system
- There is no changing file owners and permissions
- It's done with less performance and optimizations concerns
- There is no multiple communication channels

It results into a simpler tool closer to the real-world needs. Less headache as it's easier to grasp, and also results in saner and more maintainable code on the developer side. Let me elaborate.

One, and only one, client at a time can interact with the remote directory. Not only it **greatly** simplifies the protocol and the implementation as it doesn't have to deal with possible unpredictable conflicts, but it also removes a non-safe practice. Think about it, how can you reliably make changes and ensure correctness if someone else is allowed to make changes (at the same time) that can possibly mess with yours; you'd rather wait until they disconnect before making your changes. The concept of concurrent access is by nature confusing and flawed (if it doesn't come with a higher access policy).

Instead of a users with password authentication system, it simply uses a passphrase, referred as **token**, which the client must know to access the remote directory. In other words, it's a sort of unique and global password. If you think a minute, multiple user authentication is hardly-ever needed because if a folder is shared, users are trusted and we are aware of the consequences. And most of the time it's not even shared across different users, but rather across different services, often owned by a single user. Tokens are easier to work with and is closer to real-life needs. There's not even a username to remember! And if the token is compromised, just reset it and redistribute it.

5.3 File ownership, permissions and timestamp

This simpler concept jostles a bit with the traditions when transferring files and it has to do with file ownership, permissions and timestamps. In fact, those "details", who aren't always important, are needed. But since Remofile isn't using the server's underlying OS system users to authenticate clients (unlike FTP), what happens when a file is transferred, who owns it, what permissions it gets, and how about the timestamp ?

To keep things simple, and to avoid bloating the interface, both clients and servers are responsible for their local file-system for the reading and writing access of their files on each side. When a file is transferred from one point to another, it gets the local user ownership. A Remofile server can be started with a given system user and will assume it has access to all files present in the directory it's serving.

As for file permissions, a file always is readable and writable by the user, but not executable (unless it's a directory of course). The group and public permissions are defined by the configuration of the client or the server.

5.4 Not tuned for performance

My primary focus when writing Remofile was **reliability** because how would it be like if files are corrupted, and **maintainability** because dealing with transferring files and an internet protocols is actually difficult in the sense that it can become tricky. The other objectives were to achieve scriptability and embeddability easiness. The rest, such as performances and optimized implementation can be improved later.

As such, I preferred to sick with a "dumb" and straight-forward implementation that assumes the file-system isn't changing by a third process, and relies on existing high-level tools to do the job. For instance, it uses **ZeroMQ** for TCP communication and more precisely the REP-REQ pattern even if it's far from the most efficient to transfer files across a network.[5] It uses the Python standard library for its high-level API (the `pathlib` module) to deal with path and files, as well as its ability to serialize and de-serialize Python objects (see `pickle` and `marshal` module) and thus simplifies dealing with data sent across the network.

Implementing a FTP-like solution (that actually does more[6]) is a lot of work for a single person and this is why I didn't focus on performance. Luckily, in 2018, with our powerful machines and fast lines, this is not a problem for most scenarios, and is a very acceptable solution. From another perspective, even if not tuned for performance, we can say it's faster as it costs less to implement and maintain Remofile code.

Also note that the implementation will be improved over time to compress data and evolves into a more optimized solution.

5.5 Upcoming improvements

Initially, I created the protocol and programming interface of Remofile as part of another software which needed file transfer features. And because I felt like this is reinventing the wheel, it slowly evolves into a project on its own. Here are two important features which weren't needed by the former software but would enhance greatly Remofile.

- Resuming interrupting file transfers
- Direct read/write file in Python code

See the *roadmap* document for more information about features and improvements.

[5] Usually, when it comes to transferring files, one would use a lower-level solution that directly deals with streams of bytes.
[6] See its synchronization features and its ability to resume interrupted file transfers.]

CHAPTER 6

Roadmap

While Remofile already achieves its objectives and does perfectly what it claims to be, I still have a few things in mind. This document lists the improvements I'd like to make and the features I'd like to add.

Note: If you're using Remofile in production and need to have a feature added or need a more efficient implementation, you can still reach me and we will see what we can do.

Here is the list of features and improvements altogether (sorted by priority).

- Warn when the server is in used
- Resume interrupted file transfers
- Direct remote file I/O operations
- Compress large files when transferring
- More efficient transfer implementation
- Depythonization of the Remofile protocol

I'm elaborating on each feature and improvement down below. Feel free to add your grain of salt on the issue tracker (there's a ticket for each of them). You can even implement a feature yourself if want to contribute.

6.1 Warn when the server is in used

By design, simultaneous access to the Remofile server is not allowed. However, connecting to the server while it's currently use by somebody else will result in the connection hanging.

There should be a proper authentication system that warns user when trying to connect when the server is in used.

6.2 Resume interrupted file transfers

Interrupted file transfers are annoying when dealing with large files. Even though implementing this feature isn't technically a challenge, it does require to adapt the protocol. Also various decisions must be taken such as where do the interrupted file data resides (left in the remote directory itself, or saved in a temporary directory?) and others. As soon as I've got some time for that, I'll get down to it.

6.3 Direct remote file I/O operations

Another feature that could be useful is direct opening of remote file in Python code. Basically, one would open a remote file using the *open()* function and reading and writing would happen transparently. Also, with the new *pathlib* module, we can also imagine implementing a *RemofilePath*.

6.4 Compress large files when transferring

Optimizing transfer of large files by compressing them is one (easy) step towards a more efficient implementation.

The idea came to me after looking at the FTP client options. I should investigate the *-z*, *–compress* and *–compress-level=NUM* that allows to compress file data during the transfer and explicitly set a compression level.

6.5 More efficient transfer implementation

The current implementation is dumb. In fact, it's really dumb because I was rushing an implementation that works, and was focusing on a quality programming interface quality that wouldn't change. Luckily, with the high-bandwidth connection we have nowadays, the current implementation is probably acceptable for most use cases. But some effort could be made in that direction.

For now, all the work is entirely done with a single pair of REQ-REP socket for simplicity. Transferring large files will result in a huge amount of unnecessary requests from the client (see the protocol specifications document to understand why). The protocol could be updated to use a pair of streaming socket (or whatever their name, I'm not a ZeroMQ expert) after a large file is 'accepted' and its transfer initiated, to transfer the file chunks over.

6.6 Depythonization of the Remofile protocol

Check out the protocol specifications document and you'll quickly see it relies on the Python bindings of ZeroMQ which itself relies on serialization Python objects.

I'd like to make some improvements to the protocol to make it more 'binary' oriented (and thus, not specific to a given programming language), so one day, somebody (me ?) could make Remofile available to C/C++ level.

CHAPTER 7

Commands List

CLI Reference

This document is the reference of the command-line interface provided by **Remofile** to start a server and perform various file operations from a shell. Commands are divided into client-related commands and server-related commands.

Client-related commands

The client-related commands all relies on the shell environment variables to locate the remote server and do the authentication process. Improperly configured environment will result in a premature stop.

REMOFILE_HOSTNAME

This is the address of the Remote server.

REMOFILE_PORT

By default, it listens to 6768.

REMOFILE_TOKEN

Foobar.

REMOFILE_PUBLIC_KEY

Foobar.

The *list* command

NAME

remofile-list - To be written.

SYNOPSIS

remofile list [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a client-related command that does something. To be written.

OPTIONS

To be written.

EXAMPLES

To be written.

The *file* command

NAME

remofile-file - To be written.

SYNOPSIS

remofile file [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a client-related command that does something. To be written.

OPTIONS

To be written.

EXAMPLES

To be written.

The *directory* command

NAME

remofile-directory - To be written.

SYNOPSIS

remofile directory [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a client-related command that does something. To be written.

OPTIONS

To be written.

EXAMPLES

To be written.

The *upload* command

NAME

remofile-upload - To be written.

SYNOPSIS

remofile upload [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a client-related command that does something. To be written.

OPTIONS

To be written.

EXAMPLES

To be written.

The *download* command

NAME

remofile-download - To be written.

SYNOPSIS

remofile download [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a client-related command that does something. To be written.

OPTIONS

To be written.

EXAMPLES

To be written.

The *remove* command

NAME

remofile-remove - To be written.

SYNOPSIS

remofile remove [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a client-related command that does something. To be written.

OPTIONS

To be written.

EXAMPLES

To be written.

Server-related commands

Foobar.

The *run* command

NAME

remofile-run - Start a non-daemonized sever.

SYNOPSIS

remofile run [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a server-related command that start a non-daemonized server (not detached from the shell). The directory parameter is the root directory which will be served and therefore must be an existing directory. The server listens on port 6768 by default but it can be changed with the port parameter. If the token is not specified, it's generated and printed out to the console before the server starts running.

Additionally, the file size limit and the chunk size range can be altered. The file size limit and minimum chunk size must be both be greater than 0, and maximum chunk size must be greater or equal to minimum chunk size.

OPTIONS

--file-size-limit	Prevent transferring files that exceed the given file size limit.
--min-chunk-size	Prevent transferring files if the chunk size is too small.
--max-chunk-size	Prevent transferring files if the chunk size is too big.

EXAMPLES

You can quickly start a Remofile server that serves *my-directory/* on port **6768** with the following command-line.

```
` mkdir my-directory rmf run my-directory/ 6768 my-custom-token `
```

Refer to the client-related commands to start interacting with the served directory.

The *start* command

NAME

remofile-start - Start a daemonized sever.

SYNOPSIS

remofile start [OPTIONS] DIRECTORY [PORT] [TOKEN]

DESCRIPTION

This is a server-related command that start a daemonized server (detached from the shell). Unlike the run command, it accepts the *–pidfile* flag which tells the pidfile location. By default, the pidfile is created in the current working directory and named 'daemon.pid'.

Refer to the run command for more information.

OPTIONS

--pidfile	Location of the pidfile. By default, it assumes 'daemon.pid' in the current working directory.
--file-size-limit	Prevent transferring files that exceed the given file size limit.
--min-chunk-size	Prevent transferring files if the chunk size is too small.
--max-chunk-size	Prevent transferring files if the chunk size is too big.

EXAMPLES

You can quickly start a Remofile server that runs in the background (you can close the shell) and that serves *my-directory/* on port **6768**, with the following command-line.

```
` mkdir my-directory rmf run my-directory/ 6768 my-custom-token `
```

Refer to the stop command to stop the server.

The *stop* command

NAME

remofile-stop - Stop a daemonized server.

SYNOPSIS

remofile stop [OPTIONS]

DESCRIPTION

This is a server-related command that stop a daemonized server from its pidfile. By default, it expects the pidfile in the current working directory with the name 'daemon.pid' but it can be altered with the *–pidfile* flag.

OPTIONS

--pidfile	Location of the pidfile. By default, it assumes 'daemon.pid' in the current working directory.

EXAMPLES

You can stop a Remofile server that has previously been started with the start command in the same directory, with the following command-line.

```
` remofile stop `
```

With the *–pidfile* flag, you can run this command from any directory if you specify the pidfile location.

Warning: Remofile is still in development. Please give me your feedbacks to dewachter[dot]jonathan[at]gmail[dot]com.

Remofile is a **protocol**, a **Python library** and a **command-line interface** to transfer files back and forth from/to a remote server. It's a **quick** and **easy-to-use** alternative to FTP and other transfer files tools.

Note: Remofile doesn't claim to be better than FTP but rather offers a better solution for developers in most situations as it's purposely designed for easier use and integration, At the end, it does the same work but with a saner and more maintainable code on the developer side.

See this *document* for more details.

It's also properly documented and heavily tested. Check out the *user guide* to get you started with using Remofile.

CHAPTER 8

Remofile is... quick

It doesn't take much time and effort to get a Remofile server running.

```
mkdir my-shared-directory
remofile run my-shared-directory/ 6768 qRkVWJcFRqi7rsNMbagaDd
```

Directory *my-shared-directory/* is now jailed and accessible over the network on port 6768 with the token *qRkVWJcFRqi7rsNMbagaDd*.

Note: There are other ways to start a server and different options. They are all covered in the *documentation*.

CHAPTER 9

Remofile is... easy-to-use

It's straightforward to interact with the remote directory; no connection step is required, just configure the shell environment and you are ready to go.

```
export REMOFILE_HOSTNAME=localhost
export REMOFILE_PORT=6768
export REMOFILE_TOKEN=qRkVWJcFRqi7rsNMbagaDd

remofile upload --progress ubuntu-16.04.3-desktop-amd64.iso /
```

This uploads the local file *ubuntu-16.04.3-desktop-amd64.iso* to the remote directory. It may take time to complete the operation, this is why the **–progress** flag is set to display a progress bar.

For even more accessebility, add the first 3 lines to your *.bashrc* and you can now interact with the remote directory from anywhere and anytime.

Note: Replace localhost with the actual IP or the domain name to access the remote Remofile server. Ensure the port is open on the server side too.

CHAPTER 10

Remofile is... powerful

Remofile features all common file operations indeed. On top of that, it also comes with bidirectional synchronization. You can synchronize directories the same way you would do with *rsync*.

```
# synchronize the local 'foo' directory with the remote 'bar' directory
remofile sync local  foo /bar

# synchronize the remote 'bar' directory with the local 'foo' directory
remofile sync remote foo /bar
```

Remofile was purposely written to leave hundreds of features that we usually don't need. The result is an uncomplicated software that we're happy to work with.

CHAPTER 11

Remofile is... scriptable

With its sessionless command-line interface, Remofile becomes highly scriptable. See the bunch of commands available akin to *ls*, *touch*, *mkdir* and *rm*.

```
# list files the remote (root) directory
remofile list /

# create a directory then a file in the remote directory
remofile directory foo /
remofile file      bar /foo

# delete the file and the directory we just created
remofile remove /foo/bar
remofile remove /foo
```

The command-line interface might feel odds for now but it will likely change later to feel more natural. See the *commands list* learn how to use the command-line interface.

CHAPTER 12

Remofile is... embeddable

Remofile primarily is a **Python library** to run servers and interact with the remote directory from the code. It's an ideal solution when you write client-server software that needs to transfer files to multiple endpoints.

```
from remofile import Client, synchronize_upload

client = Client('localhost', 6768, 'qRkVWJcFRqi7rsNMbagaDd')
synchronize_upload(os.getcwd(), '/')
```

The remote directory is now synchronized with the current working directory in the most painless fashion.

Note: Because it's based on **ZeroMQ**, you can even configure the server with your own ZeroMQ socket and reduces the need to open an additional port.

CHAPTER 13

Remofile is... secure

Remofile is indeed secure. But it doesn't encrypt the communication by default. However, with little effort, it's easy to get it fully encrypted. With the following command, you can generate a pair (public + private) of keys.

```
remofile generate-keys

public key: aE8{cjoe?JPDxGuX^/d*5KyP(ZuxKwIHB{EM7o&H
private key: D3yb)c-Nxw}DY8kAy<IOUww5@A4:G[n)8*}0S01^
```

Now, you will keep the private key **secret** and configure the server with it. Then, just distribute the public keys across all clients.

Note: Configuring the client with the public key and the server with the private key is explained in the *documentation*.

Index

Symbols

B

C

D

F

G

I

L

M

N

P

R

S

T

U

V