pyftpsync

Specification

Table of Contents

Overview

Pyftpsync is an open source project that allows to synchronize directories over FTP(S) and file system access.

- This is a command line tool…
- … and a library for use in your Python projects.
- Upload, download, and bi-directional synchronization mode.
- Allows FTP-to-FTP and Filesystem-to-Filesystem synchronization as well.
- Architecture is open to add other target types.

The project is maintained on GitHub: https://github.com/mar10/pyftpsync.
Feedback and contributions are welcome.

Requirements

Design goals and derived decisions.

Extensibility

pyftpsync is designed as a Python library in the first place. The command line interface is an application use case that builds on this.
This approach also allows for easy automated testing.

The architecture should be easy to extend, for example add new target types (think TFTP, WebDAV, or Google Drive API).

Performance

The synchronization process should be reasonably fast.

FTP servers (in general) don't support sending etags or CRC checksums with a dir listing.
Since calculating CRCs on a remote server would require slow downloads, we rely on file sizes and modification times to detect changes.
This implies that MLST support is a requirement for FTP servers.

Robustness

We need to deal with some special scenarios

- Local or remote files may be modified, added, or removed by users at any time, so the metadata stored by pyftpsync becomes invalid.
  → we have to detect invalidated metadata
- Files may be modified without changing the file size.
  → we will not rely on this value to check for equality (but use it to pre-check for inequality).
- It may even be possible that a file content has changed without changing the modification time, but this would require some explicit file time manipulation by the user (or occur in very unlikely cases, where the system clock is set).
  → we will still rely on the file time to detect modifications, but maybe add an option for binary comparisons / CRCs in the future.
- Files may be changed on both targets between two synchronizations:
  → We need to identify conflicts and offer different conflict resolution strategies.
- System clock of server and client may be out of sync by a few seconds or minutes.
  Server and client may also use different time zones.
  → always use GMT time, probe the server for a time delta, and use an epsilon when comparing times.

- One local folder may be synchronized with different remote targets. Likewise, one remote target may be used by different clients.
  → metadata must be stored per peer id
- Different pyftpsync jobs may run at the same time.
  For example two clients synchronize with the same remote target (or one remote target is a sub folder of the other). This may lead to corrupt data.
  → some sort of locking would help
- A pyftpsync job may be interrupted by Ctrl-C, FTP server problems, missing permissions, programming errors, network errors, etc.
  → We should try to prevent or repair inconsistent source files, metadata, or stale locks.

File System Encoding and Restrictions

FTP servers and file systems may use different encodings for folders and file names.
Targets may also define a maximum length for resource paths or have different sets of disallowed characters.

We should try to handle those potential conflicts in a transparent and predictable way. See `File System Encoding and Restrictions` below.

Compatibility and Requirements

We want to support as many platforms and impose as few pre-conditions or restrictions as possible.

There should be no need to have a watchdog service running on the server or client.

Usability and Consistency

Synchronization is technically tricky, but may also be confusing even if implemented correctly.
The interface should be clear and provide transparent information to avoid accidental misuse:

- Use terms “Local” / “Remote” consistently.
- Support dry-run mode.
- 'upload' mode never modifies local target. Likewise 'download' mode never modifies remote target.
- Use defensive defaults.
- Display understandable information on conflicts.

Algorithm

1 Instantiate two target objects (local and remote), derived from the `_Target` class.

2 Define a synchronizer (upload, download, or bi-directional), derived from the `_BaseSynchronizer` class.

3 Call `synchronizer.run()`

  3.1 Walk both target trees and find matching/new/missing entry pairs.
  Also optionally apply inclusion/exclusion filter patterns.

  3.2 Classify entries (Existing? Modified since last sync?)

  3.3 Classify entry pairs and call handler, for example
  `synchronizer.sync_older_local_file(local, remote)`

4 Dump statistics

File Entry Properties and Metadata

For every entry we need to know

1. Current file size and file modification time as reported by the file system
2. timestamp of last successful, not-dry-run synchronization (if any)
3. file size and file modification time at the time of last synchronization
   This values may be None if the file did not exist at that time.

The information of 2.) and 3.) is stored as additional metadata.
This also allows to detect files that have been deleted since last synchronization, because they will still appear in the metadata.

Metadata for file status at the last synchronization time is always stored in a text file named `.pyftpsync-meta.json` inside the *local* folder, because it is more likely that we have write access here.

The metadata is used to detect conflicts, i.e. we want to tell if files have bee modified since the last synchronization. Because a target may be synchronized with different peers, we must maintain the data sets per peer.

Metadata is stored in JSON text format, normally in a compact version. For debugging a verbose version can be activated, which is indented and includes string formatted date fields.

Example of a Local .pyftpsync-meta.json After a Synchronization

(showing the verbose format):

```
{
    "_disclaimer": "Generated by https://github.com/mar10/pyftpsync",
    "_file_version": 2,
    "_time": 1470750669.0,
    "_time_str": "Tue Aug  9 16:51:09 2016",
    "_version": "1.1.0",
    "mtimes": {},
    "peer_sync": {
        "www.example.com/test_pyftpsync": {
            ":last_sync": 1470754266.689334,
            ":last_sync_str": "Tue Aug  9 16:51:06 2016",
            "a.txt": {
                "m": 1418577067.0,
                "mtime_str": "Sun Dec 14 18:11:07 2014",
                "s": 56,
                "u": 1470730157.237741,
                "uploaded_str": "Tue Aug  9 10:09:17 2016"
            },
            "b.txt": {
                "m": 1418577087.0,
                "mtime_str": "Sun Dec 14 18:11:27 2014",
                "s": 69,
                "u": 1470730157.452979,
                "uploaded_str": "Tue Aug  9 10:09:17 2016"
            }
        }
    }
}
```

mtime on FTP Servers

On FTP targets there is an additional global section in the metadata that holds the original modification times of the uploaded files.
This is required, because FTP servers will always set file time to the upload time.

This information is stored by the client on the *remote* server, whenever a file is uploaded.

We must discard those entries, when a file was modified by another client than pyftpsync.
In order to detect external changes, we also store the update time and size and check if the current size had changed, or if the current mtime is later than the last upload time.

Example of a .pyftpsync-meta.json on a Remote FTP Server After a Synchronization

(showing the verbose format):


```
{
    "_disclaimer": "Generated by https://github.com/mar10/pyftpsync",
    "_file_version": 2,
    "_time": 1470750669.0,
    "_time_str": "Tue Aug  9 16:51:09 2016",
    "_version": "1.1.0",
    "mtimes": {
        "a.txt": {
            "m": 1418577067.0,
            "mtime_str": "Sun Dec 14 18:11:07 2014",
            "s": 56,
            "u": 1470730157.237687,
            "uploaded_str": "Tue Aug  9 10:09:17 2016"
        },
        "b.txt": {
            "m": 1418577087.0,
            "mtime_str": "Sun Dec 14 18:11:27 2014",
            "s": 69,
            "u": 1470730157.452889,
            "uploaded_str": "Tue Aug  9 10:09:17 2016"
        }
    },
    "peer_sync": {}
}
```


Glossary

We refer to these properties as (assuming local filesystem and remote FTP server):

Local target:

$size_{(L)}$	Local file size as reported by the file system.
$mtime_{(L)}$	Local file modification time as reported by the file system.
pssize	Peer sync size: snapshot of size at the time of last copy operation. (Stored in a metadata file on the local target.)
psmtime	Peer sync modification time: snapshot of mtime at the time of the last copy operation. (Stored in a metadata file on the local target.)
psutime	Peer sync time: time of last copy operation. (Stored in a metadata file on the local target.)

Remote target:

(pssize, psmtime, and psutime are identical to the local target.)

$size_{(R)}$	Remote file size as reported by the FTP server.
$mtime_{(R)}$	File modification time of the uploaded file. Note that we store this as separate metadata on FTP servers, because FTP servers apply the upload time to all files during synchronization. In this case $mtime_{(R)}$ is this adjusted modification time of the original uploaded file. Defaults to $rtime_{(R)}$ if no metadata is available.

$rtime_{(R)}$	REAL remote file modification time as reported by the FTP server. This value is normally nearly identical with psutime (not mtime), because FTP servers cannot set or copy an explicit file time.

Classification of Entries and Pairs

If pyftpsync has never run on a folder, there will be no metadata file at all, so we can only do a simple classification:

Existing	A file is *existing* if it currently exists on the file system, but we have no metadata file at all.

If a metadata file is available however, we can use this information to replace the classification *existing* with a more specific one:

Missing	A file is *missing* if it is not *existing* and we don't have metadata for it.
Deleted	A file is *deleted* if it is not *existing*, but we have metadata for it.
Modified	A file is *modified* if it is *existing*, has metadata, and $size \neq pssize \text{ or } mtime \neq psmtime$
Unmodified	A file is *unmodified* if it is *existing*, has metadata, but is not *modified*.
New	A file is *new* if it is *existing*, but we don't have metadata for it.

A pair's classification is defined by the combination of local and remote classifications.
Some examples:

(*existing*, *existing*)	A file name was found on remote and local target. Since we don't have a metadata file, we may need to compare the file sizes or content to find out if they are identical or if we have a conflict.
(*existing*, *None*)	A matching file does not exist on remote target, so we may upload the local file to remote or delete the local file.
(*unmodified*, *modified*)	We can download the modified remote file.
(*modified*, *modified*)	This is a conflict that we need to resolve in some way.

We can classify a pair as follows:

Equal	A pair of entries is *equal* if both are *existing* and $size_{(L)} = size_{(R)} \text{ and } mtime_{(L)} = mtime_{(R)}$ If metadata is available, we consider it *equal* if both entries are *unmodified*.
Conflict	A pair of entries is a *conflict* if both are *existing* and $size_{(L)} \neq size_{(R)} \text{ or } mtime_{(L)} \neq mtime_{(R)}$ If metadata is available, we consider it a *conflict* if one entry is *modified* and the peer entry is *modified*, *new*, or *deleted*.
Other	If only one entry is modified or only one entry exists, this may result in copy operations, depending on the synchronization mode.

Determining the Synchronization Action

Bi-Directional Synchronization

Standard Case: Metadata is Available

The synchronizer performs operations based on the preceding classification.
The standard operations are listed in the following table:

Sync Action		Remote target				
		missing	**new**	**unmodified**	**modified**	**deleted**
Local target	**missing**	n.a.	< Copy new	< Copy new	< Copy new	Only cleanup metadata
	new	Copy new >	(1) Need compare	(1) Need compare	(2) Potential conflict	Conflict
	unmodified	Copy new >	(1) Need compare	(1) Need compare(*)	< Replace modified	< Delete missing
	modified	Copy new >	(2) Potential conflict	Replace modified >	Conflict	Conflict
	deleted	Only cleanup metadata	Conflict	Delete missing >	Conflict	Only cleanup metadata

(1) Compare mtime of source and target.
$mtime_{(L)} < mtime_{(R)}$ → Use remote file
$mtime_{(L)} > mtime_{(R)}$ → Use local file
$mtime_{(L)} == mtime_{(R)}$ and $size_{(L)} == size_{(R)}$ → Nothing to do
$mtime_{(L)} == mtime_{(R)}$ and $size_{(L)} \neq size_{(R)}$ → Conflict.

(*) TODO: This should not be possible?

(2) Same as (1) but if the file that we would replace has status *modified*, we treat this as Conflict.

Additional options may be passed to modify the behavior:

- `--resolve`: Define a resolving strategy for conflicted pairs (skip, ask, use local, use remote, use newer, use older).

Special Case: No Metadata Available

If pyftpsync is run for the first time, or if a new folder was created by a user since last sync, we don't have metadata available. All we know is that an entry is *existing* or not. This limits the number of possible classifications:

Sync Action		**Remote target**	
		None	**existing**
Local target	**None**	n.a.	< Copy new
	existing	Copy new >	(1) Need compare

(1) Compare mtime of source and target.
$mtime_{(L)} < mtime_{(R)}$ → Use remote file
$mtime_{(L)} > mtime_{(R)}$ → Use local file
$mtime_{(L)} == mtime_{(R)}$ and $size_{(L)} == size_{(R)}$ → Nothing to do
$mtime_{(L)} == mtime_{(R)}$ and $size_{(L)} \neq size_{(R)}$ → Conflict.

The classification code will convert (*existing*, *existing*) pairs to combinations of *modified* and *unmodified*, such that the synchronizer will handle them accordingly.

Special Case: Directories

A directory is an entry, so we may have metadata available to decide whether it was created or removed since last sync.

However we don't know if it was *modified*, because there is no meaningful `mtime` or `size` value. Figuring out would require a deep traversal, which we currently do not do for performance reasons. Instead we consider directories *unmodified*, so we will **not** detect conflicts when a folder was deleted on one target and modified on the peer.
We **do** detect changes and conflicts of child entries, since we will traverse folders.

Sync Action		**Remote target**			
		missing	**new**	**unmodified**	**deleted**
Local target	**missing**	n.a.	< Copy new	< Copy new	Only cleanup metadata
	new	Copy new >	(1) Nothing to do	(1) Nothing to do	Conflict
	unmodified	Copy new >	(1) Nothing to do	(1) Nothing to do	< Delete local
	deleted	Only cleanup metadata	Conflict	Delete remote >	Only cleanup metadata

(1) Nothing to do: Simply traverse child entries.

Upload Synchronization

Basically a subset of Bi-Directional synchronization where the local target is treated as read-only (except for writing to the metadata file .pyftpsync-meta-json).

In Upload (and Download) mode we only replace files with newer versions (and only if they are not conflicted).
Additional options may be passed to modify this behavior:

- `--force`: always replace files on remote, even if local version is older (but still only if not conflicted).
- `--delete`: remove files on remote if they don't exist on local.
- `--delete-unmatched`: remove files on remote if they don't match the custom filter (implies `--delete`).
- --resolve: provide a conflict resolution strategy.

Download Synchronization

Basically a subset of Bi-Directional synchronization where the remote target is treated as read-only.

File System Encoding and Restrictions

(New with v3.0.)

This topic is known and rooted in a broken FTP specification.

From https://wiki.filezilla-project.org/Character_Encoding (2016-08-28):

> *FTP is a rather old protocol and things we take for granted now were not even considered when it was designed. One of these things is support for non-English characters in filenames.*
> *When the FTP protocol was designed, computers mostly spoke English and were unable to display any non-English characters. As such, the FTP protocol was designed to be used with English characters only, namely 7-bit ASCII.*
>
> *The problem is that many FTP clients and servers purposely violate the FTP specifications in order to support other, non-standard character sets.*
> *Which of these character sets are used is not subject to any negotiation. For any character set in existence, you can find a server using it with no way of detecting the proper encoding.*
>
> *The result: non-English characters are not transferred correctly.*
>
> *To solve this problem, the FTP protocol has been extended in a backwards compatible way to use UTF-8 as the character set. (This solution is backwards compatible only with servers in compliance with the original specifications.)*
>
> *If you have problems with filenames containing any foreign characters, this can have two reasons:*
>
> 1. *The server or client follows the original specifications by the letter and rightfully rejects those filenames*
> 2. *The server or client violates the specifications and uses a custom encoding that the other party does not understand*
>
> *Both FileZilla Client and Server are fully compliant with the updated specifications and use UTF-8. FileZilla will not break FTP specifications by supporting non-standard encodings in order to accommodate the user.*
> *If you have problems with other clients or servers, please upgrade (or ask the server to upgrade) to FTP software capable of UTF-8 or refrain from using foreign characters. Anything else is in violation of the FTP specifications and will only work if you manually ensure that the server and client use the same character encoding (which may not even be possible).*

So we assume that

1. FTP servers store and retrieve names as binary strings (don't care about encoding).

(https://stackoverflow.com/a/3120633/19166)
Modern implementations of FTP servers and clients use UTF-8.
This is backwards compatible with the FTP specification in theory, since previously only 7-bit ascii was officially allowed.
However it is known that some servers out there still deliver any other encoding.
(https://wiki.filezilla-project.org/Character_Encoding)

2. Even if UTF-8 is the most reasonable encoding for byte strings, ISO-8859-1 is probably the 2nd most used form that we meet (the latest HTML standard requires ISO-8859-1 to be interpreted as CP-1252). (https://stackoverflow.com/a/3120633/19166)
   So if UTF-8 decoding fails, we should assume CP-1252.
   This can be overridden using `–remote-encoding=CODING`.

3. File System targets

   1. On Python 2 we pass unicode paths to the os API , so we get unicode in return.
   2. Local File-System-Targets use `sys.getfilesystemencoding()` as default (UTF-8 if not detectable).

4. Python's `ftplib`

   1. On Python 2
      Expects native path names (i.e. binary). If `unicode` is passed in, ASCII is always used to convert, which fails for special characters.
      → We always pass already encoded binary strings to the API.
      The API produces native (i.e. binary) name listings, in the encoding that the server sent.
      → We can immediately convert API results to unicode using utf-8/cp1252 as described above.

   2. On Python 3
      Expects native path names (i.e. unicode).
      The default encoding is "latin-1", but can be ocverridden.
      → We always pass unicode to the API and set `ftplib.ftp.encoding` to "utf-8".
      API produces native (i.e. unicode) name listings, that have been decoded from the server's binary response.
      → this should work well if `ftplib.ftp.encoding` was set correctly.

5. Metadata files
   We store metadata in dicts that have the file names ans paths as keys. Those dicts are persisted as JSON files.

   1. Python's `json` loader converts binary keys to unicode using UTF-8 (needed for sorting). This may fail for ISO-8859-1 encoded strings.
      → We use native `str` for paths internally (i.e. unicode or utf-8). We also convert the

keys to unicode before saving as JSON on Python 2.

2. Lookups may fail if the key format does not match:
   if utf8_path in dict_with_unicode_keys
   may not find the entry.
   → We convert all keys to native `str` after reading from JSON.

6. Conclusions

1. In our code, we store all paths in a canonical format that supports unicode characters. However, we use the platforms native format, so it is easier to code string operations. This means on Python 2: UTF-8 encoded `str`, whereas on Python 3 it's `unicode`.

2. The `.pyftpsync-meta.json` files use UTF-8 always.
   We pass `` `ensure_ascii=False` ``, so non-ascii characters will not be escaped as `\x`*NN* or `\u`*NNNN*.
   For pretty-printing we pass `` `sort_keys=True` ``. The JSON writer converts to unicode assuming UTF-8 for sorting, so passed-in binary strings must be encoded as UTF-8.

3. **TODO**: If this heuristic does not work, we need to handle this exceptions and skip the node, or ask the user?
   Should we issue a warning instead of falling back to CP-1252 for single files?
   If the server does not send UTF-8, should we switch a CP-1252 mode altogether, i.e. also send our files in that encoding?
   ==> NO. It should be enough to warn/skip. User can use the encoding option then.

4. **TODO**: How should we handle the case that local uses UTF-8, and remote uses CP-1252?
   ==> we still send UTF-8 to the FTP server, so it will store it in that form.
   Unless `--remote-encoding` was set otherwise.

Command Line Interface

The command line interface is a front end to the pyftpsync library, that adds this functionality:

- Allow to instantiate and configure the synchronizer and targets, based on URL strings
- Allow to pass flags to configure the synchronization process and define filter patterns
- Maintain credentials in the system keyring
- Display progress status and statistics
- Provide a dry-run mode
- Prompt for resolution if conflicts are detected

Appendix 1: Test Fixture

This test fixture is used by some of the unit tests.

```
                        Local           Remote
-------------------------------------------------------------------------
file1.txt               12:00           12:00        (unmodified)
file2.txt               13:00           12:00
file3.txt                 x             12:00
file4.txt               12:00           13:00
file5.txt               12:00             x
file6.txt               13:00           13:00:05     CONFLICT!
file7.txt               13:00:05        13:00        CONFLICT!
file8.txt                 x             13:00        CONFLICT!
file9.txt               13:00             x          CONFLICT!

folder1/file1_1.txt     12.00           12:00        (unmodified)
folder2/file2_1.txt     13.00           12:00
folder3/file3_1.txt       x             12:00        (folder deleted)
folder4/file4_1.txt       x             13:00        (*) undetected CONFLICT!
folder5/file5_1.txt     12:00           13:00
folder6/file6_1.txt     12:00             x          (folder deleted)
folder7/file7_1.txt     13:00             x          (*) undetected CONFLICT!

new_file1.txt           13:00             -
new_file2.txt             -             13:00
new_file3.txt           13:00           13:00        (same size)
new_file4.txt           13:00           13:00        CONFLICT! (different
size)
new_file5.txt           13:00           13:00:05     CONFLICT!
new_file6.txt           13:00:05        13:00        CONFLICT!

NOTE: (*) currently conflicts are NOT detected, when a file is edited on one
        target and the parent folder is removed on the peer target.
        The folder will be removed on sync!
```