pyftpsync

Specification

Table of Contents

Overview 1
Requirements 2
  Extensibility 2
  Performance 2
  Robustness 2
  Compatibility 3
  Usability and Consistency 3
Algorithm 4
  File Entry Properties and Metadata 4
    mtime on FTP Servers 5
    Glossary 6
  Classification of Entries and Pairs 6
  Determining the Sync Action 7
    Bi-Directional Synchronization 7
    Upload 7
    Download 8
Command Line Interface 8
Appendix 1: Sample Flow for 'update' command 9

Overview

Pyftpsync is an open source project that allows to synchronize folders over FTP.

- 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 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.

Compatibility

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 removed 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:

$size_{(R)}$	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

Except for status *existing*, we rely on metadata to figure out the entry status:

Existing	A file is *existing* if it currently exists on the file system.
Not existing	A file is *not existing* if it currently does not exists on the file system and we don't have metadata for it.
Unmodified	A file is *unmodified* if it is *existing* has metadata, but is not *modified*.
Modified	A file is *modified* if it is *existing*, has metadata, and `size ≠ pssize or mtime ≠ psmtime`
New	A file is *new* if it is *existing*, but we don't have metadata for it.
Deleted	A file is *deleted* if it is *not existing*, but we have metadata for it.

Based on the entry's status we can classify pairs:

Conflict	A pair of entries is *conflicted* if one entry is *modified* and the peer entry is *modified*, *new*, or *deleted*.

Determining the Sync Action

Bi-Directional Synchronization

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

Sync Action		Remote target				
		not existing	**new**	**unmodified**	**modified**	**deleted**
Local target	**not existing**	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)

Upload

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.

TODO:
How is --force different from --resolve=local?
What is the difference between new and unmodified?
If we have a pair

Download

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

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: Sample Flow for 'update' command

Assume we have a local folder „c:\data\“ that contains a few files.
We synchronize with a remote target „ftp://host/“

Time	Action	Files	Local Target Properties		Metadata (stored on local target)			Sync Status	Remote FTP Target Properties		
			size(L)	mtime(L)	pssize	psmtime	psutime		size(R)	mtime(R)	rtime(R)
01:00	Initial status	a.txt b.txt c.txt	100 100 100	01:00 01:00 01:00	- - -	- - -	- - -	New New New	- - -	- - -	- - -
02:00	$ pyftpsync update ...	a.txt b.txt c.txt	100 100 100	01:00 01:00 01:00	100 100 100	01:00 01:00 01:00	02:00 02:00 02:00	Equal Equal Equal	100 100 100	01:00 01:00 01:00	02:00 02:00 02:00
03:00	Manually edit 2 files on local	a.txt b.txt c.txt	105 105 100	03:00 03:00 01:00	100 100 100	01:00 01:00 01:00	02:00 02:00 02:00	Modified Modified Equal	100 100 100	01:00 01:00 01:00	02:00 02:00 02:00
03:01	Manually edit 2 files on remote	a.txt b.txt c.txt	105 105 100	03:00 03:00 01:00	100 100 100	01:00 01:00 01:00	02:00 02:00 02:00	Modified Conflict Modified	100 110 110	02:00 03:01 03:01	02:00 03:01 03:01
04:00	$ pyftpsync update ...	a.txt b.txt c.txt	105 105 100	03:00 03:00 01:00	100 100 100	01:00 01:00 01:00	02:00 02:00 02:00	Modified Conflict Modified	100 110 110	02:00 03:01 03:01	02:00 03:01 03:01