PiCloud Documentation

Release

Paul Crook

November 23, 2015

Table of Contents:

CHAPTER 1

Introduction

PiCloud is intended as a multi-platform, client/server software syncing suite. Files may be synced between multiple computers over the internet.

CHAPTER 2

Installation

Although subject to change, current installation is relatively simple. Well, at least for linux users. Windows support may arrive in the future.

2.1 Prerequisites

First, if you wish to truly use the program, you need at least two devices. One to setup as a server, the other as the client. Next, download the source files in your desired directory:

```
git clone https://github.com/paulcrook726/PiCloud.git
```

Once all that is done, do it with your other devices. At this point, all source files are “installed”. Unfortunately, due to it still being in development, PiCloud has no installation scripts or fully-functioning cli or gui interfaces. Thus, please consider following the Usage section below.

CHAPTER 3

Usage

Follow these steps to set up and use the PiCloud source files.

3.1 Setting up the Server

Now for installing the server software. Starting from the location of the previously made command:

```
cd PiCloud/src
nano server.py
```

Of course, you may use which every editor you prefer instead of `nano`. These commands creates a file called `server.py` in the same directory of pycloud.py, which is the main module you need to import. Start by typing this into `server.py`:

```
from pycloud import *


def main():
        logging.basicConfig(format='%(asctime)s %(message)s',
                            filename='picloud.log',
                            level=logging.INFO)
        server = ServerSocket(46000)
        server.activate()



if __name__ == '__main__':
        main()
```

What this does is (1) creates a logging file for logging certain important information, (2) creates an instance of `ServerSocket`, (3) activates this socket by putting it into the main event loop. Next, save this file and exit. You may run this file now or later, your choice. Once you do, you have fully functioning server listening on port `46000`.

3.2 Setting up the Client

Setting up the client works quite similarly. Clone the repository, `cd` to `src`, and then this time make a file called `client.py`, or really whatever you feel like calling it. Put this in the file:

```
from pycloud import *


def main():
        logging.basicConfig(format='%(asctime)s %(message)s',
                            filename='picloud.log',
                            level=logging.INFO)
        c = ClientSocket(server_ip, 46000)
        f = pre_proc('your_file_here')
        send_file(c, f)
        while evaluate(c) == 0:
                pass


if __name__ == '__main__':
        main()
```

Add all this into the file except for `server_ip` and `'your_file_here'`. Instead of these, put in the actual ip of your device running the server, and the name of a file you would like to transfer in quotes and including the file extension. Once finished, simply make sure your server is running, and then run `client.py` and viola! your file has magically been copied to the other computer.

CHAPTER 4

Logs

After running your first time, you may notice that a `picloud.log` file has been created. If you open it up and view it in a text editor, it has documented the important parts of the interaction between server and client. If you ever run into problems, it is a good idea to check out the logs here.

CHAPTER 5

User Accounts

While still completely in development, a user account management system is sort of in place already. To create a "user", simply create an ID file in your client-side directory, and send it using the above method. Open an empty file, and name it whatever username you want to have, then with a .id file extension. Then write in the file your password. If you send this file to the server, the server automatically creates your user account.

CHAPTER 6

pycloud module

This module is the primary building block of the PyCloud infrastructure. Within it, the basic file transfer protocol is laid down.

class `pycloud.`**`ClientSocket`**(*host*, *port*)

Bases: `socket.socket`

This is a subclass of `socket.socket`. Introduced primarily for simplicity and the setting of pre-defined values.

class `pycloud.`**`ReceivedFile`**(*name*, *ext*, *sock*)

Bases: `object`

`evaluate`()

This method evaluates .cert, .id, and all other files with an extension.

`take_data`(*data*)

This method adds data to the associated filename and extension.

Parameters **`data`** (*byte str*) – Data to be added

class `pycloud.`**`ServerSocket`**(*port*)

Bases: `socket.socket`

This is a subclass of `socket.socket`. Creates a socket with a few pre-defined variables.

`activate`()

This activates the main server event loop for accepting incoming connections.

class `pycloud.`**`User`**(*name*, *pwd*, *sock*)

Bases: `object`

`check_pwd`()

This method checks the .pi_users file for `self.name`.

Returns Returns the corresponding password to the instance username. If it is not found, `1` is returned.

Return type str or int

`input_request`(*msg*)

This method sends a question to the client, and awaits the response.

Parameters **`msg`** (*str*) – This is the question that the client will see.

Returns Returns the answer to the question.

Return type str

login()
This method logs in the User instance with the instance password and username.

Returns Returns 0 on success. Returns 1 on failure to login.

Return type int

make_hashed(*salt=None*)
This method hashes the `self.pwd` into a salted hash.

Parameters **salt** (*str*) – Defaults to `None`. If so, a random salt is generated using SHA-512 hashing.

Returns The hashed password is returned (which includes the salt at the beginning.

Return type str

register()
This method registers the User instance with the username and password, and then logs in via `self.login().`

Returns Returns 1 on failure. Returns 0 on success.

Return type int

pycloud.**evaluate**(*sock*)
This is the primary function used in server/client communication evaluation.

Data is received and processed depending on whether it contains the primary delimiter for files, or whether it contains certain keywords used for server communication.

Parameters **sock** (*socket.socket*) – The socket by which data is received.

Returns Returns `0` on success.

Return type int

Returns Returns `1` on failure. This is especially important when determining when the server/client communication is over, or whether it is still going to go on. A return of `1` indicates that no more communication will go on. `0` indicates that communication will happen again, and consequently, `evaluate()` should be called again.

Return type int

pycloud.**get_cwu**()
This function gets the current logged-in user.

Returns Username from the .current_user file

Return type str

pycloud.**pre_proc**(*filename*, *is_server=0*)
This function processes a filename by whether or not it exists in the current working directory.

Parameters

- **filename** (*str*) – The filename of the file you want to process.
- **is_server** (*int*) – This acts as a flag for determining how exactly the function should work.

Returns Returns the file data if it is found in the local filesystem.

Return type byte str

Returns Returns the pre-processed filename and extension if the function acts as a client. (For requesting files) :rtype: byte str :return: Returns `FileError` if function is acting as a server, and the file could not be

found in the local filesystem.

Return type byte str

`pycloud.`**`proc_block`**(*client_sock*, *length*)

This is a helper function of `recv_all()`. It receives data according to `length`.

Parameters

- **`client_sock`** (*socket.socket*) – The socket by which data is received.
- **`length`** (*int*) – The length of the data to check for and receive.

Returns Returns `None` if no packets are received. Otherwise returns the block of data received.

Return type None or byte str

`pycloud.`**`recv_all`**(*client_sock*)

This function receives data on a socket by processing the data length at first.

Parameters **`client_sock`** (*socket.socket*) – The socket by which data is being received.

Returns Returns `None` if no data is received.

Return type None

Returns Otherwise returns the data received.

Return type byte str

`pycloud.`**`send_file`**(*sock*, *b_data*)

This function sends a byte string over a connected socket.

Parameters

- **`sock`** (*socket.socket*) – The socket by which data is sent.
- **`b_data`** (*byte str*) – The data to be sent.

Returns Returns `0` upon success.

Return type int

Returns Returns `None` on failure.

Return type None

CHAPTER 7

Index & Search

- genindex
- modindex
- search

Python Module Index

Index