Pepi Documentation

Release 3.0.0

Curtis West

Oct 01, 2017

Contents

"A Portable, Extensible Photogrammetry Instrument."

PEPI is a software platform that enables remote command and control of servers with connected cameras. PEPI is used for the purposes of acquiring stereo-photogrammetry imagery. It seamlessly supports any number of connected servers without any required manual setup.

PEPI is designed to be expanded upon. The implementation in this repo is purely Python-based for a Raspberry Pi with Camera Module, but it is built upon Apache Thrift. Therefore, any correct implementation in an Apache Thrift supported language should work out of the box. This means that as long as you can write a camera 'adapter' that adheres to the PEPI standard, you can use it with PEPI.

CHAPTER 1

Links

- PEPI is licensed under the Apache-2.0 license.
- PEPI's source code is available on GitHub.
- PEPI's documentation is available on ReadTheDocs.
- PEPI uses a continuous integration and testing system provided by Travis CI.

CHAPTER 2

Built With

- Python 2 & 3 - This implementation is built on Python 2.7 and should be compatible with Python 3.6. Python is not strictly, should the use of another language be desirable.
- Apache Thrift - A software framework for scalable, cross-language services
- PiCamera - A pure Python interface to the Raspberry Pi Camera Module
- Flask - A web microframework for Python, used for the included user interface
- ..and many more packages gracefully provided by the Python community.

CHAPTER 3

Authors & Acknowledgements

- Claudio Pizzolato - *Initial work and proof of concept*
- Curtis West- *Development from proof of concept through to version 2.0*
- Griffith University - *For supporting the project*

CHAPTER 4

Table of Contents

About PEPI

What is PEPI?

PEPI's is intended to be used as a method to acquire stereo-photogrammetry imagery. It should be noted that PEPI does not yet perform any of the photogrammetric analysis - that is reserved for existing software.

The implementation and design of PEPI was completed in a 12 week project as part of an undergraduate capstone project with Griffith University. The development of PEPI is detailed in the thesis, "A Low-cost Distributed Camera System for Stereo-photogrammetry", which provides more in-depth justifications than could reasonably fit here.

Terminology

In this documentation, we use the following terms:

Client the computer running the PEPI client/control software, which manages all connected servers

Server A camera-equipped computer running the PEPI server software

Raspberry Pi

RPi

Pi The Raspberry Pi is a low-cost, single-board computer produced by the Raspberry Pi foundation.

Camera Module

PiCamera The small camera accessory sold for the Raspberry Pi, or the Python library that interfaces with it.

Distro A distribution of an operating system eg. Ubuntu, Raspbian, etc.

PEPI's Structure

PEPI's structure is best described using the class diagram below:

PEPI is really just the PEPI core, which is two language-neutral interface definition files. This file defines the whole PEPI system and how all other components should be implemented. Of course, having this definition doesn't help when you need to *use* the system, because it doesn't actually do anything. To use the system, you'll need implementations of these interfaces on your servers so that a client may control the server.

We expect that there will–for the most part–only ever need to be one *base* server-side implementation of this interface *per language*. The base implementation for a language may be subclassed and modified to suit specific needs, because most functionality is shared between servers of a single language. In this repo, we provide a minimal implementation of a Python server (`BaseCameraServer`), as well as an example of extending that server for use on a Raspberry Pi (`RaspPiCameraServer`).

We provide a Python client with a user interface that can control *any* server on the network that implements the interface, no matter the language. While it would be valid to do so, we don't expect that there would be a need to write a new client–just use the provided one.

So to recap, PEPI is:

- The PEPI core, i.e. an interface definition file. The other components are merely implementations of this interface.
- A collection of server implementations in different languages, each addressing a different use case.
- A Python client, which should be applicable to all use cases (i.e. just use the provided client rather than writing your own).

Hardware

To run the provided implementation for the Raspberry Pi & Camera Modules, you will need:

- Raspberry Pi (any model) and Raspberry Pi Camera Module (any version) with matching cable
- 8GB or larger microSD card, preferably Class 10 for speed
- Quality power supply
- Router or switch – wireless if you wish to connect the RPi's over Wi-Fi
- A Unix-based computer to run the client software (may work on others - untested)

PEPI versions lower than v2.0 were intended for use solely with RPi's with the RPi camera only. V2.0 introduced a redesigned architecture that supports new server implementations and new cameras. This means that any server implementation should be able to use any camera, as long as that camera has an adapter written for it that conforms to the PEPI spec. Any server can be controlled by the provided Python client implementation.

New implementations do not need to be in Python but abstract base classes and contract tests are provided to simplify Python development. As communication between server and client occurs over Apache Thrift, support for cross-language, cross-platform implementations is built-in.

If you're looking to write a new implementation, refer to the *Extending Pepi* section.

Motivations

The design choices behind PEPI are motivated by several constraints and requirements. These have affected how PEPI has been designed and implemented.

Some of these motivations:

1. Indifferent to number of servers, so new servers can be brought online at any time without any manual setup (i.e., dynamic server discovery).
2. Rapidly deployable and easy to use
3. Extensible, well-documented and able to support future needs
4. Rapidly and reliably capture images across all cameras
5. Cross-platform, cross-language, so servers running new hardware (e.g. new computer, new cameras) can be implemented in the future without changes to the overall architecture.
6. Ultimately, produce a low-cost system that can scale.

Installation

Note: These installation notes are mainly for the provided PEPI implementations, running on the specified *PEPI's Structure*. Configurations outside this scope may require different setup steps, but will generally have similar steps.

Prerequisites

Python

The included client and servers target Python 2.7/3.6 or newer. Other versions may work, but are untested. You can check this in your terminal:

```
$ python --version
Python 2.7.13
$ python --version
Python 3.6.1
```

If you have an old version, or you get errors about Python not being recognised, you should follow the Python Install Guide.

Raspberry Pi

The provided *RaspPiCameraServer* implementation should run on any Raspberry Pi, but has only been tested on a RPi 3. The OS distro that you use shouldn't matter, but we suggest Raspbian (or Raspbian Lite for maximum performance). You'll need a microSD card - 8GB is perfect.

Unless you plan on plugging a keyboard, mouse and monitor into your Pis, you'll probably be using SSH to control them. It is important to note that, by default, SSH comes **disabled** unless you configure the flash SD card before boot. If you plan on SSH and you're running the Pi's over Wi-fi, you'll need to add your Wi-Fi details to the SD card before boot too.

Enabling SSH

1. Download the Raspbian image you wish to use.
2. Flash the Raspbian image to your SD card. You can use a number of tools for this, we recommend Etcher.

3. After flashing the SD card, eject it then reinsert it. You should see it mounted in your file explorer, usually called `BOOT`.
4. Open the SD card in your file explorer.
5. Create a new file in the root of the SD card called exactly `ssh`, **without any file extension** to enable SSH by default on this SD card.

Configuring Wi-Fi

1. Open your file exporer to the root of your flashed SD card.
2. Create a new file called `wpa_supplicant.conf`, containing the following details. Replace SSID and PASSWORD with your intended wireless network's details (keep the quotes).

```
network={
 ssid="SSID"
 psk="PASSWORD"
}
```

3. Save the file back to the SD card and eject the card.

Camera Module

The provided *`RaspPiCameraServer`* can use any `Camera` implementation, but is intended for use with a Raspberry Pi Camera Module. This module needs proper configuration and installation to work.

PiCamera has provided a great quickstart guide on this process. However, if you are using SSH (ie. a terminal) to connect to your Raspberry Pi, you cannot complete the last steps in that guide as it requires a GUI.

To enable the camera through the terminal:

```
$ sudo raspi-config
5 Interfacing Options   Configure connections to peripherals
P1 Camera       Enable/Disable connection to the Raspberry Pi Camera
Would you like the camera interface to be enabled? <Yes>
The camera interface is enabled. <OK>
<Finish>
```

You'll need to reboot after enabling the camera.

SSH Keys

You may wish to look into using SSH keys. This removes the need to type in a password when logging into the Pi over SSH – something that gets very annoying when you're wrangling dozens of them.

If you use SSH keys alongside the Raspberry Pi server implementation, you can place a copy of your private key in your cloned Git repo under `/raspi_server/keys`. This will allow you to use the provided utility scripts to push out new versions to all servers at once, which is super useful for development work.

While we recommend using a new SSH key-pair for PEPI with the name given below, it's not mandatory but you will need modify the utility scripts.

You can generate a new SSH key from most Unix terminals with:

```
$ ssh-keygen -t rsa -C "PEPI SSH Key"
```

You'll be asked to save the SSH key - save it somewhere easy like your home folder under the name pepi_rsa: `~/pepi_rsa`.

Now the problem is getting the SSH private key onto your Pi's. The easiest way is to use SSH itself (with a password this time):

```
$ cat ~/pepi_rsa.pub | ssh pi@<IP-ADDRESS> 'cat >> .ssh/authorized_keys'
```

You should now be able to SSH into the Pi without a password (you may be prompted about an unknown host, this is expected for the first usage):

```
$ ssh -i /path/to/your/pepi_rsa pi@<IP-ADDRESS>
```

Client Installation

1. Clone the Git repo to get the latest version of the PEPI client:

```
$ cd ~
$ git clone https://github.com/curtiswest/pepi.git
$ cd pepi
$ ls
LICENSE      README.rst   client       pepi.thrift   raspi_server server
↪test         unittest.cfg
```

2. Install the client's Python dependencies:

```
$ python --version
Python 2.7.13 (or Python 3.6.1)
$ cd client/
$ sudo pip install -r requirements.txt
```

3. Make the client executable:

```
$ chmod +x run.py
```

4. Run the client using either of the commands below:

```
$ python run.py
INFO:werkzeug: * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
$ ./run.py
INFO:werkzeug: * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

5. Open your internet browser to http://0.0.0.0:5000/ and you should see the PEPI user interface. See *Using PEPI* to learn more about this interface.

Raspberry Pi Server Installation

Note: Make sure you've followed the relevant steps in the *Prerequisites* section above before proceeding.

Setting up your first Pi is the slowest. After one is set up, you can simply *duplicate that SD card*.

Downloading & Installing

Firstly, we need to obtain the software:

1. SSH into your pi, e.g. `ssh pi@<IP-ADDRESS>` or if using *SSH Keys* (recommended), `ssh -i /path/to/pepi_rsa pi@<IP-ADDRESS>`
2. Clone the latest version of the software from Git

```
$ cd ~
$ git clone https://github.com/curtiswest/pepi.git
$ cd pepi
$ ls
LICENSE      README.rst   client       pepi.thrift   raspi_server server
↪test         unittest.cfg
```

Alternatively, if your Pi does not have internet access, you could download a *.zip of the repo* and use a flash drive to transfer it to the Pi.

1. Place a copy of your SSH key in the `raspi_server/keys` folder if you want to use the utility scripts:

```
$ cp /path/to/your/pepi_rsa raspi_server/keys
```

2. Install the server's requirements.

```
$ python --version
Python 2.7.13 (or Python 3.6.1)
$ cd raspi_server/
$ sudo pip install -r requirements.txt
```

3. Test that the server can launch:

```
$ python server.py
INFO:root:Starting RaspPiImagingServer
```

4. If you see the above, then the server is working fine. Stop the server with `CTRL + C`.
5. Now, we need to setup launching the software on boot. A script is included, `raspi_server/start_on_boot.sh`, that handles everything needed to launch the server (from the correct directory context etc). You can add this to the Pi's boot sequence by executing the following:

```
$ cd ..
$ pwd
/home/pi/pepi
$ sudo sed -i -e '$i \bash /home/pi/pepi/raspi_server/start_on_boot.sh &\n' /etc/
↪rc.local
```

6. Reboot your Pi with:

```
$ sudo shutdown -r now
```

7. The server should have started running automatically on boot. You can check this by looking for the `run.py` process:

```
$ ssh -i /path/to/pepi_rsa pi@<IP-ADDRESS>
$ ps aux | grep run.py
root       740  1.5  2.5 120552 22388 ?        Sl   20:58   0:02 python run.py
```

8. If everything works, congratulations! If not, try walking through these steps and double-checking the commands were entered correctly. Perhaps try checking that your script was added to the boot script correctly (sometimes you may need special permissions to edit the /etc/rc.local file) with:

```
$ cat /etc/rc.local | tail -5
fi

bash /home/pi/pepi/raspi_server/start_on_boot.sh &

exit 0
```

Duplicating SD cards

Warning: You will be reading and writing from raw disk partitions. **You could erase your computer** if you execute the commands below with the wrong parameters. Double-check your commands before executing.

Note: It is untested whether these image files are compatible across the different Raspberry Pi Models. That is to say, it is unclear whether a Raspberry Pi 3 image can be cloned onto an SD card intended for a Raspberry Pi Zero. If you try this, please update this documentation with the results and create a pull request.

Once you've verified that the card works exactly how you want, you can make an image of the SD card that will allow you to duplicate it onto other SD cards.

1. Insert the card into your card reader.
2. Find where the card is mounted by running `diskutil list`. Look for the device that matches your SD card, generally by the size of the disk is easiest. Here, a 8GB SD card is inserted and appears under `/dev/disk2/`.

```
$ diskutil list
/dev/disk0 (internal, physical):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        *121.3 GB   disk0
   2:          Apple_CoreStorage Macintosh HD            120.5 GB   disk0s2
/dev/disk2 (internal, physical):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:     FDisk_partition_scheme                        *7.9 GB     disk2
   1:             Windows_FAT_32 boot                    43.7 MB    disk2s1
   2:                      Linux                         7.9 GB     disk2s2
```

3. Remember the `/dev/diskx/` (where x = your disk's number, which above would be `/dev/disk2/`) location where you SD card is mounted.
4. Run the following command to unmount any mounted SD card partitions:

```
$ df -H
Filesystem      Size   Used  Avail Capacity iused      ifree %iused  Mounted on
/dev/disk1     112Gi   69Gi   42Gi    62% 1354336 4293612943    0%   /
/dev/disk2s1    41Mi   21Mi   20Mi    51%       0          0  100%   /Volumes/boot
$
$ sudo umount /dev/diskx*
$ df -H
Filesystem      Size   Used  Avail Capacity iused      ifree %iused  Mounted on
/dev/disk1     112Gi   69Gi   42Gi    62% 1354336 4293612943    0%   /
```

5. Now, we can image the SD card with the `dd` command:

```
$ dd if=/dev/diskx/ of=~/rpi.img bs=4M
$ ls -la ~/ | grep rpi.img
-rw-r--r--   1 root        staff  7948206080 20 Aug 23:23 rpi.img
$ sudo sync
```

   The `rpi.img` file contains a complete copy of the SD card. It is possible to shrink the image to copy it quicker using a GParted live boot disk, but you'll need to expand it again once copied across. If you have lots of cards to duplicate, you could look into building a Open Source Image Duplicator to allow you to duplicate several at a time.

1. Eject that SD card, and insert the new SD card you want to setup.
2. Locate where that disk is located (usually, it's the same–but not always!), in this case `/dev/disk2/`:

```
$ diskutil list
/dev/disk0 (internal, physical):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        *121.3 GB   disk0
   2:          Apple_CoreStorage Macintosh HD            120.5 GB   disk0s2
/dev/disk2 (internal, physical):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:             Windows_FAT_32                        *7.9 GB     disk2
```

3. Unmount the new SD card, if it has any mounted partitions:

```
$ sudo umount /dev/diskx*
```

4. Now we can copy the image back onto the SD card by simply reversing the `dd` command (notice the `if` and `of` arguments are now reversed):

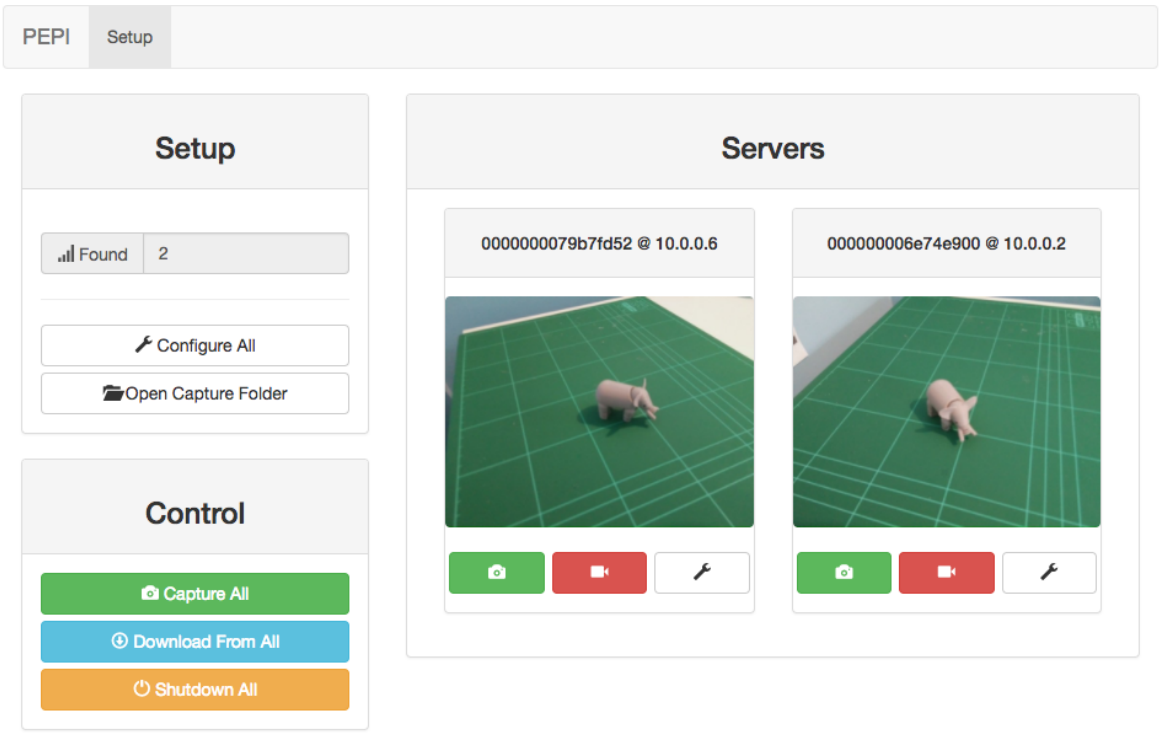
```
$ dd if=~/rpi.img of=/dev/diskx/ bs=4M
$ sudo sync
```

5. Eject the SD card, and repeat the above 3 steps for as many cards as you need. You should be able to put these cards directly into new Raspberry Pi's and have them work just as the first did.

Using PEPI

Once you've got your servers installed and running, and your client launches correctly (see *Installation* for more details), you're ready to use PEPI.

Open your browser and browse to http://0.0.0.0:5000/. You'll see the PEPI interface:



Hopefully you'll find that the buttons are relatively self-explanatory for the most part.

The user interface is responsive – if you scale the window, the user interface will resize appropriately.

You can access the user interface presented by the client from any device that is on the same network. Therefore, you can control PEPI from a smartphone or tablet by browsing to `http://<ip-of-your-PC>:5000/`. Images will still be downloaded to the *client PC* regardless of what device you use to access the user interface.

Setup Panel

The *Setup* panel holds information and controls for all the operations necessary before you start using the tool.

The *Found* box shows how many PEPI servers were found on the network. Note that some servers may have more than one camera attached to them, but this is not reflected in the count. This number should match the physical number of servers on your network – if the number is lower than expected, your server may not be properly configured or running. PEPI will automatically discover servers on the network every 5 seconds, but the count won't update until you refresh the page.

The *Configure All* button is not presently implemented (indeed, no configuration is implemented yet).

The *Open Capture Folder* button opens the folder on the computer running the client that contains images captured from the servers.

Control Panel

The *Control* panel holds the control for actually using the tool (i.e. capturing images) once the system is set up.

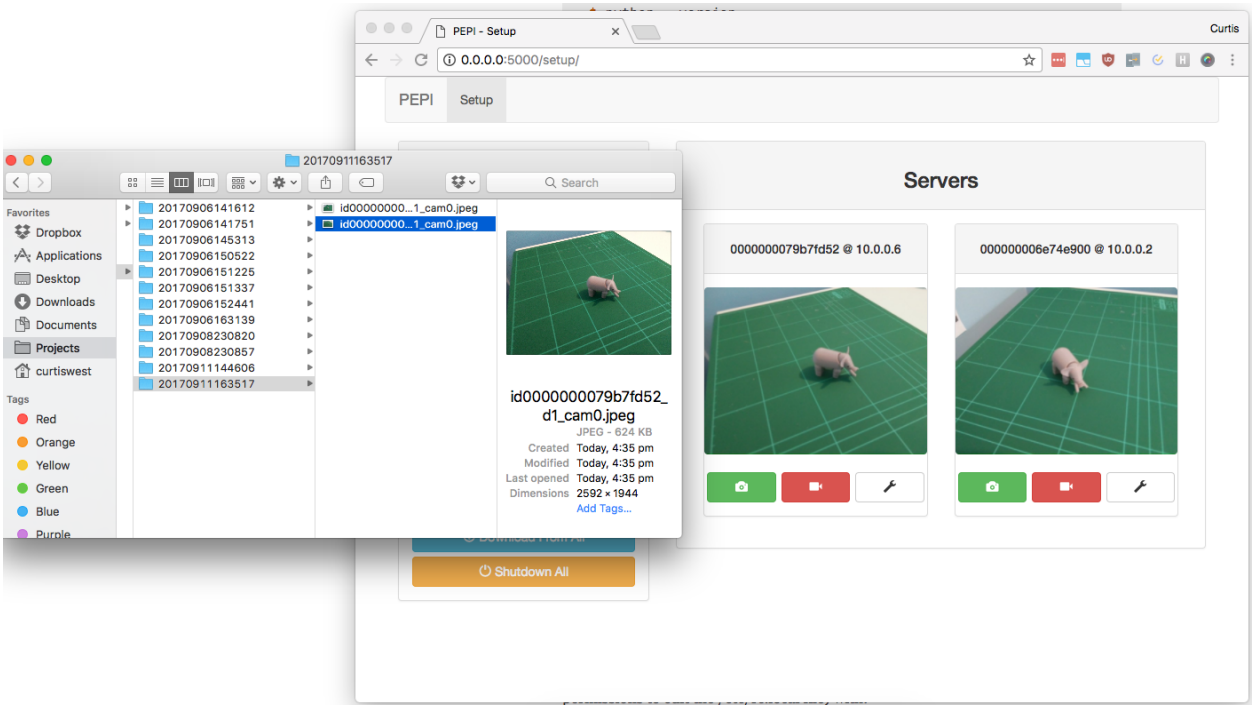
Note: The number of servers in the *Found* box may not be up-to-date as you must refresh the page to see the updated count.

However, all the controls here will still command *all* detected servers, as detecting servers is independent of the user interface's display.

The *Capture All* button sends a "take photo" command to all detected servers. The servers will capture an image and store in their local memory indefinitely. This allows for bulk captures without being delayed by waiting for downloads to occur image-by-image.

> **Warning:** There are no lockouts on the *Capture All* button. Pressing the button too frequently may cause server instability depending on the server's implementation. At the very least, you are likely to lose synchronisation between captured images.
>
> It is not possible to implement a timed lockout, as PEPI only understands "servers", not "cameras" and so it cannot understand how long each camera takes to capture an image.
>
> Therefore, we recommend that you understand how long it takes for your servers' to capture one image, and only press the *Capture All* button as quickly as the server can support.

The *Download All* button requests all captured images from the server as JPEGs. It is safe to press this even when new servers have come online, as the client remembers which server "owes" us images. Once the download is complete, your file explorer will open to the image's locations, as shown:



The images are saved with the following naming notation:

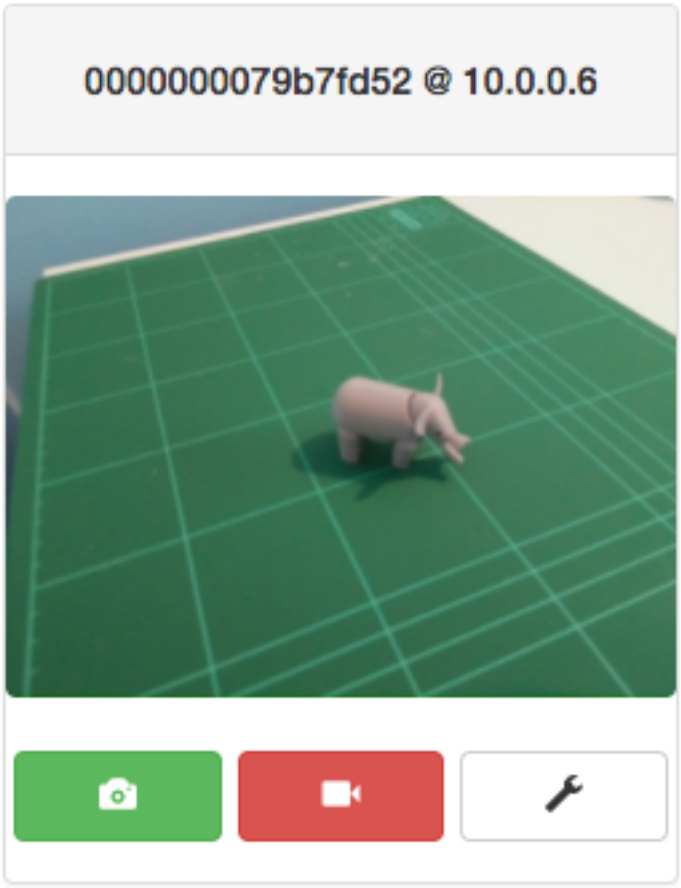
```
id{server_id}_d{unique capture #}_cam{server camera #}.jpeg

For example:
id000000006e74e900_d70_cam0.jpeg
id000000006e74e900_d70_cam1.jpeg
id000000006e74e900_d70_cam2.jpeg
id000000006e74e900_d71_cam0.jpeg
id000000006e74e900_d71_cam1.jpeg
id000000006e74e900_d71_cam2.jpeg
```

The *Shutdown All* button requests all detected servers to shutdown. Note that a server implementation may choose not to implement this functionality, but they will still accept the backing function call nonetheless.

Servers Panel

The *Servers* panel shows you all the detected servers and allows for individual control.

Each card in the panel represents one server. If the server supports a livestream from the camera, you'll see shown in the centre of the card.

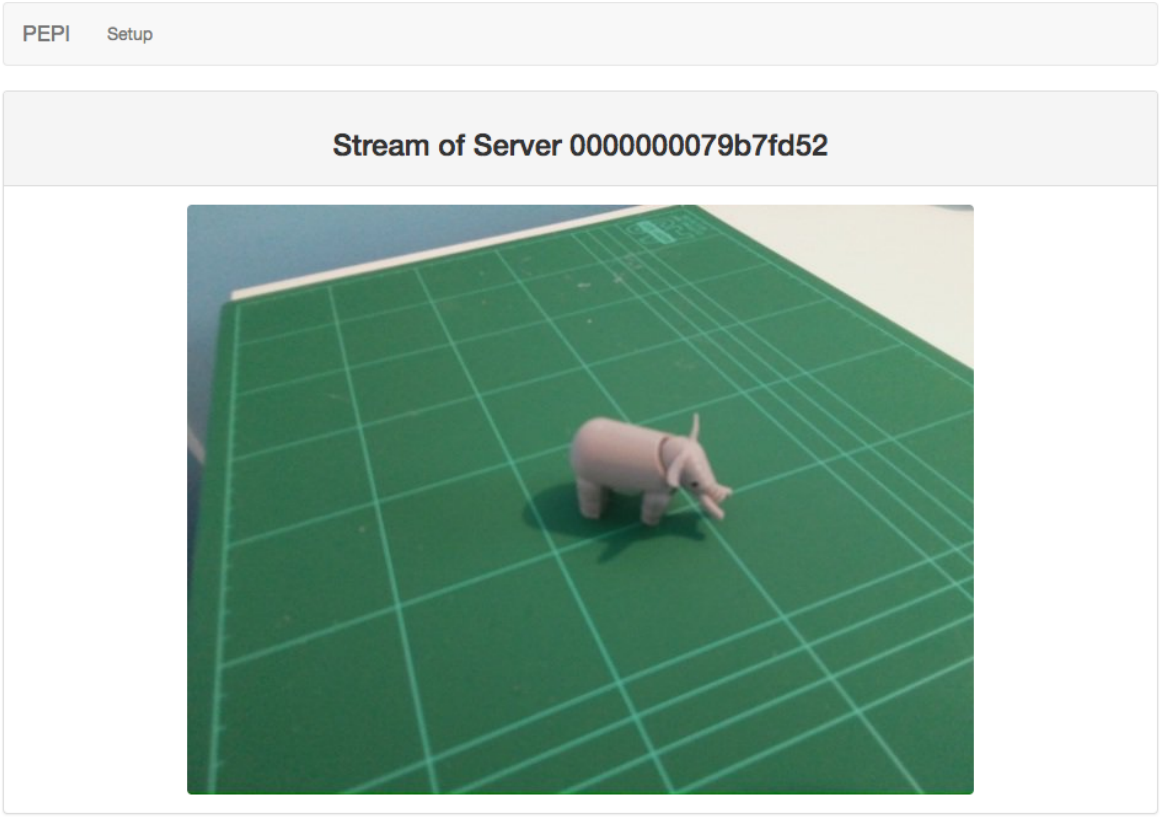


The header of each card contains it's server ID and it's network IP:

```
0000000079b7fd52 @ 10.0.0.6
{server id}      @ {server ip}
```

The green camera button captures an image from *only* this server.

The red camera button starts a full-screen stream from the servers camera, if supported:



The grey wrench button will be used for configuring an individual server in the future, but is currently not supported.

Extending PEPI

Generally, you'll want to extend PEPI by creating new servers and adding support for new cameras. The client-side software should work with any server out-of-the-box, but there are still some features that need to be added to the client.

PEPI Theory

PEPI can be divided into the client-side and the server-side. As discussed, the client-side doesn't really need to be extended–the server is where the interesting extensions can happen.

Servers can be divided into two components:

- The actual server itself (the `CameraServer`)
- Cameras connected to the server (the `Camera`)

We do not mandate that a `CameraServer` must take `Camera` objects, but it is **strongly** recommended unless you have a valid reason (e.g. very complicated hardware requirements). The provided Python implementations shows how this may be implemented.

Languages

PEPI is indifferent to which language you implement your server in, so long as it can be accessed over Apache Thrift. Thrift's language bindings include:

- C++
- C#
- Cocoa
- D
- Delphi
- Erlang
- Haskell
- Java
- OCaml
- Perl
- PHP
- Python
- Ruby
- Smalltalk
- ..plus others in the works or supported by third parties

Therefore, any of the above languages can be used to implement PEPI server-side components.

Writing New Servers

Interface Definition File

At the heart of PEPI is its interface definition file `pepi.thrift`. This defines the interface used to access servers and therefore specifies what functions you need to implement.

```
/**************************************************************************
 * File: pepi.thrift
 * Author: Curtis West
 * ----------------------------
 * Interface definition file for Apache Thrift.
 **************************************************************************/

/**************************************************************************
Thrown when a requested image is not available on the server
**************************************************************************/
exception ImageUnavailable {
    1: string message,
}

/**************************************************************************
A Camera provides images from a physical camera in the form of RGB arrays.
**************************************************************************/
service Camera {
```

```
  /* still
   * description: returns a still image capture from the camera at the currently
   *              set resolution
   * returns: multidimensional array of row, column, RGB representing the image
   */
   list<list<list<i16>>> still()

  /* low_res_still
   * description: gets a 640 x 480px still from this camera for previewing
   * returns: multidimensional array of row, column, RGB representing the image
   */
   list<list<list<i16>>> low_res_still()

  /* still
   * description: gets the maximum resolution supported by this camera
   * returns: a list of length 2 representing the resolution i.e. (x, y)
   */
   list<i16> get_max_resolution(),

  /* get_current_resolution
   * description: gets the current resolution of this camera
   * returns: a list of length 2 representing the resolution i.e. (x, y)
   */
   list<i16> get_current_resolution(),

  /* set_resolution
   * description: if supported, sets the resolution of the camera
   */
   oneway void set_resolution(1:i16 x, 2:i16 y)
}

/*****************************************************************************
A CameraServer serves as a wrapper around a camera and provides a number of
utility functions for managing the server and camera.
*****************************************************************************/
service CameraServer {
  /* ping
   * description: Used to ping the server.
   * returns: True, always
   */
  bool ping(),

  /* identify
   * description: Gets this server's unique identifier.
   * returns: String containing this server's identifier
   */
  string identify(),

  /* stream_urls
   * description: Gets a list of URL where the stream of this server's cameras
                  may be accessed, if they exist.
   * returns: List of strings containing URLs, or an empty list.
   */
  list<string> stream_urls(),

  /* ping
   * description: Shuts down this server. This does not need to be implemented,
   *              but the server must accept the function call.
```

```
   */
  oneway void shutdown(),

  /* start_capture
   * description: Captures a still from this server's camera(s) and stores in
                  internally under the given `data_code` for later retrieval.
   */
  oneway void start_capture(1:string data_code),

  /* retrieve_stills_png
   * description: Retrieves .png images that were captured using `start_capture`
                  under the specified `data_code` (if they exist), encoded as
                  PNGs.
   * throws: ImageUnavailable
   * returns: a list of strings, where each string contains one image encoded as
              a PNG file. Each string should be able to be dumped directly to
              the disk and still form a valid PNG file.
   */
  list<string> retrieve_stills_png(1:string with_data_code) throws(1:ImageUnavailable␣
↪unavailable),

  /* retrieve_stills_jpg
   * description: Retrieves images that were captured using `start_capture`
                  under the specified `data_code` (if they exist), encoded as
                  JPEGs.
   * throws: ImageUnavailable
   * returns: a list of strings, where each string contains one image encoded as
              a JPEG file. Each string should be able to be dumped directly to
              the disk and still form a valid JPEG file.
   */
  list<string> retrieve_stills_jpg(1:string with_data_code) throws(1:ImageUnavailable␣
↪unavailable),

  /* enumerate_methods
   * description: Returns a dictionary of the methods supported by this server.
                  This is currently not used for any function as of v3,
                  so you may choose to just return an empty dictionary,
                  but be aware that this may change in the future versions.
   * returns: A dictionary with:
               key: method name
               value: a list of argument names that the method takes
   */
  map<string, list<string>> enumerate_methods()
}
```

Depending on the language you choose to implement your new server/camera, the exact format of how you implement these functions will vary, but generally you'll just write the functions exactly as listed (but in the syntax of your language).

From the perspective of writing a server implementation, there are no special requirements from Thrift; you don't need to return Thrift types or use Thrift objects. Your server won't even know its been called from Thrift (sometimes it won't be). Instead, treat component implementations as *handlers* that are called when in response to a Thrift requests, with Thrift managing all the necessary type conversions and network transports.

Python's BaseCameraServer

PEPI provides a minimal implementation of a CameraServer under the class `BaseCameraServer`.

In most cases, *BaseCameraServer* can be used without modification as long as you can provide the `Camera` you'd like to use. However, you may wish to override some methods to better suit your use case. For example, *RaspPiCameraServer* overrides the `identifier()` method to use the Raspberry Pi's CPU serial number as the ID.

If your server is being implemented in another language, it is still beneficial to refer to this implementation to understand how certain operations are accomplished.

```
class BaseCameraServer(object):
    """
    BaseCameraServer is the minimal Python implementation of a CameraServer
    as defined in ``pepi.thrift``. CameraServers are used in with the
    Apache Thrift protocol to provide RPC mechanisms, allowing control
    of this server over RPC, if it is launched with Thrift.

    A CameraServer subclassing BaseCameraServer may override any of these
    methods to better reflect their use. However, care must be taken to
    ensure that the side effects of the subclass's methods do not affect
    other methods. For example, if you were to change the capture method
    to store images in a list for whatever reason, you would need to change
    the image retrieval methods.

    A CameraServer's use-case is to provide a server that controls a
    number of cameras to be controlled in a consistent manner. This allows
    for a client to seamlessly control all implementations of CameraServer's,
    over Thrift without needing to concern themselves with what cameras are
    attached, the procedure call names, etc.

    This BaseCameraServer implementation supports multiple connected cameras,
    that are transparent to the connecting client. When retrieving images, a
    list of encoded images are returned. The order of this list remains
    consistent across procedure calls.
    """

    StreamInfo = collections.namedtuple('StreamInfo', 'port, folder, streamer')
    STREAM_PORT = 6001

    def __init__(self, cameras, stream=True):
        # type: ([AbstractCamera], bool) -> None
        """
        Initialises the BaseCameraServer.

        :param cameras: a list of AbstractCamera objects
        :param stream: True to start streams for all cameras, False to not.
        """
        # self.cameras = CameraManager(cameras)
        self.cameras = cameras
        self._stored_captures = dict()
        self.streams = dict()
        self.identifier = str(uuid.uuid4().hex)

        if stream:
            StreamInfo = collections.namedtuple('StreamInfo', 'port, folder, streamer,
↪ capturer')
            for count, camera in enumerate(cameras):
                port_ = self.STREAM_PORT + count
                folder_ = tempfile.mkdtemp()
                streamer_ = MJPGStreamer(folder_, port=port_)
                capturer = CameraTimelapser(camera=camera, folder=folder_, interval=0.
↪5)
```

```
                capturer.start()
                self.streams[camera] = StreamInfo(port=port_, folder=folder_,␣
↪streamer=streamer_, capturer=capturer)

        def cleanup():  # pragma: no cover
            """
            Cleans up after this server by destroying connected cameras
            and their streams, and erasing the stored images.
            """
            logging.info('Cleaning up RaspPiCameraServer')
            self._stored_captures = None
            self.cameras = None
            self.streams = None
            logging.info('Cleanup complete for RaspPiCameraServer')
        atexit.register(cleanup)

    def ping(self):
        # type: () -> bool
        """
        Ping the server to check if it is active and responding.

        :return: True (always)
        """
        logging.info('ping()')
        return True

    def identify(self):
        # type: () -> str
        """
        Get the unique identifier of this server.

        :return: the server's unique identifier string
        """
        logging.info('identify()')
        return self.identifier

    @staticmethod
    def _current_ip():
        # type: () -> str
        return IPTools.current_ips()[0]

    def stream_urls(self):
        # type: () ->  [str]
        """
        Get the a list of URLs where the MJPG image stream of each camera
        connected to this server may be accessed.

        The order of the returned images is consistent, e.g. Camera #1, #2
        .., #x returned in that order.

        :return: a list of the stream URLs as a string
        """
        logging.info('stream_urls()')
        out_urls = []
        for _, stream_info in viewitems(self.streams):
            out_urls.append('http://{}:{}/stream.mjpeg'.format(self._current_ip(),␣
↪stream_info.port))
        return out_urls
```

```
    def shutdown(self):
        # type: () -> None
        """
        Shutdown the server (i.e. power-off).

        Subclasses may choose to
        ignore calls to this function, in which case they should override
        this function to do nothing.

        :return: None
        """
        logging.info('shutdown()')
        os.system('shutdown now')

    def start_capture(self, data_code):
        # type: (str) -> None
        """
        Immediately starts the process of capturing from this server's Camera(s),
        and stores the captured data under the given unique data_code.

        Note: the received `data_code` is assumed to be unique. Subclasses may
        choose to implement better isolation methods, but this is not
        guaranteed nor required.

        :param data_code: the requested data_code to store the capture under
        :return: None
        """
        logging.info('start_capture(data_code: {})'.format(data_code))
        captures = []
        # TODO: parallelize capture from all cameras
        for camera in self.cameras:
            try:
                captures.append(Image.fromarray(np.array(camera.still(), dtype=np.
↪uint8)))
            except (AttributeError, TypeError, ValueError) as e:
                logging.warn('Could not construct image from received RGB array: {}'.
↪format(e))
                continue
        if captures:
            self._stored_captures[data_code] = captures
        logging.info('Stored_captured after start_capture(): {}'.format(self._stored_
↪captures.keys()))

    def _retrieve_and_encode_from_stored_captures(self, data_code, encoding, quality):
        # type: (str, str, int) -> [str]
        try:
            image_list = self._stored_captures.pop(data_code)
        except KeyError:
            raise ImageUnavailable('No images are stored under the data_code "{}"'.
↪format(data_code))
        else:
            out_strings = []
            for image in image_list:
                image_buffer = BytesIO()
                image.save(image_buffer, encoding, quality=quality)
                out_strings.append(image_buffer.getvalue())
            return out_strings
```

```
    def retrieve_stills_png(self, with_data_code):
        # type: (str) -> [str]
        """
        Retrieves the images stored under `with_data_code`, if they exist, and
        encodes them into a .png str (i.e. bytes).

        The order of the returned images is consistent, e.g. Camera #1, #2
        .., #x returned in that order.

        :param with_data_code: the data_code from which the image will be retrieved
        :raises: ImageUnavailable: when image requested with an invalid/unknown data_
↪code
        :return: a list of strings with each string containing an encoded as a .png
        """
        logging.info('retrieve_stills_png(with_data_code: {})'.format(with_data_code))
        return self._retrieve_and_encode_from_stored_captures(with_data_code, 'PNG',␣
↪quality=3)

    def retrieve_stills_jpg(self, with_data_code):
        # type: (str) -> [str]
        """
        Retrieves the images stored under `with_data_code`, if they exist, and
        encodes them into a .jpg str (i.e. bytes).

        The order of the returned images is consistent, e.g. Camera #1, #2
        .., #x returned in that order.

        :param with_data_code: the data_code from which the image will be retrieved
        :raises: ImageUnavailable: when image requested with an invalid/unknown data_
↪code
        :return: a list of strings with each string containing an encoded as a .jpg
        """
        logging.info('retrieve_stills_jpg(with_data_code: {})'.format(with_data_code))
        return self._retrieve_and_encode_from_stored_captures(with_data_code, 'JPEG',␣
↪quality=85)

    def enumerate_methods(self):
        # type: () -> [(str, str)]
        """
        Retrieves a map of the methods available on this server. This is useful
        for clients to verify the methods it can expect to be able to call
        if being called remotely.

        :return: dict of <method_name: [arguments]>
        """
        import inspect
        methods = inspect.getmembers(self, predicate=inspect.ismethod)
        output_dict = dict()
        for _tuple in methods:
            name, pointer = _tuple
            args = inspect.getargspec(pointer).args
            try:
                args.remove('self')
            except ValueError:  # pragma: no cover
                pass
            output_dict[name] = args
```

```
        return output_dict
```

Writing New Cameras

If you choose to use a `Camera` object with your server, then you should implement your camera according to the `Camera` interface.

In Python, *`AbstractCamera`* is provided as an abstract class with some implemented methods. It is an abstract class rather than a base class (like `CameraServers`) because it's impossible to cater for all possible connected hardware.

The most important (and tricky) method in a `Camera` is it's `still()` method that returns a multi-dimensional array of 0-255 RGB pixels (row, column, RGB). For example, `MyConcreteCamera` is implemented in Python and captures RGB images at a 4-by-3 pixel resolution:

```
>>> camera = MyConcreteCamera()
>>> image = camera.still()
>>> print(type(image))
<type 'numpy.ndarray'>
>>> print(image.shape)
(3, 4, 3)
>>> image
    array([[[244, 213,  53],  # Row 1, Col 1
            [141, 130, 195],  #         Col 2
            [229, 156,  94],  #         Col 3
            [204,  19, 191]], #         Col 4

           [[105, 202, 239],  # Row 2, Col 1
            [183, 109, 243],  #         Col 2
            [164, 190,   1],  #         Col 3
            [216, 191,  63]], #         Col 4

           [[160, 232, 240],  # Row 3, Col 1
            [ 86, 186, 252],  #         Col 2
            [ 19, 212, 221],  #         Col 3
            [253, 143,  29]]], dtype=uint8)  # Col 4
```

Most physical cameras don't provide a RGB array. The easiest way to transform from JPG or PNG (preferred) files is to use a library such as Pillow (previously, PIL). In Python, we provide a utility class *`server.abstract_camera.RGBImage`* based on Pillow that can do some of these conversions for you.

```
from server import RGBImage
import numpy as np

class MyConcreteCamera(AbstractCamera):
   def __init__(self):
      self._camera = MyDSLRCamera()

   def still(self):
      png = self._camera.get_png()
      return np.array(RGBImage.fromstring(png))
```

Alternatively, if you wish to use Pillow directly:

```
from io import BytesIO

from PIL import Image
```

```
import numpy as np

class MyConcreteCamera(AbstractCamera):
   def __init__(self):
      self._camera = MyDSLRCamera()

   def still(self):
      png_buffer = BytesIO()
      png_buffer.write(self._camera.get_png())
      png_buffer.seek(0)
      image = Image.open(png_buffer)
      return np.array(Image.open(png_buffer))
```

Testing Your Camera Implementation

PEPI includes tests that you can run against your new camera implementation to see if it returns the correct values both natively in Python and over Apache Thrift. Note that this isn't an exhaustive test of your camera implementation and how it handles errors etc., but instead just a test to check the correct values are returned.

To setup these tests against your server, you'll need to define a few pytest fixtures that is used to "inject" your camera into the tests:

```
import pytest

from server.tests import AbstractCameraContract, AbstractCameraOverThrift
import MyConcreteCamera

class TestMyConcreteCamera(AbstractCameraContract):
   @pytest.fixture(scope="module")
   def camera(self):
      return MyConcreteCamera()

class TestDummyCameraOverThrift(AbstractCameraOverThrift):
 @pytest.fixture(scope="module")
 def local_camera(self):
     return MyConcreteCamera()
```

Refer to the *Testing* section for more details on testing in PEPI.

Raspi Server Implementation

A subclass of *BaseCameraServer* is provided with PEPI for use with Raspberry Pi's, *RaspPiCameraServer*. This serves as a useful example on how to extend a language's base implementation to customize functionality.

```
class RaspPiCameraServer(BaseCameraServer):
    """
    An implementation of a BaseCameraServer for a Raspberry Pi.
    """
    def __init__(self, cameras, stream=True):
        super(RaspPiCameraServer, self).__init__(cameras, stream)

        # Set identifier based on the RPi's CPU serial number
        try:
            with open('/proc/cpuinfo', 'r') as f:
                for line in f:
```

```
                if line[0:6] == 'Serial':
                    self.identifier = line[10:26]
        except IOError:
            pass
```

As we have previously discussed, in most cases the procedures implemented in *BaseCameraServer* can be used for new servers. Simply subclass *BaseCameraServer* and your new server will inherit all of these implementations, which have been thoroughly tested and refined.

RaspPiCameraServer does exactly this: subclasses *BaseCameraServer* and overrides the `identify()` procedure to better suit its use case by using the CPU serial number to identify the server (to allow rapid deployment without manual setup on each server). This demonstrates just how easy extendnig servers are (at least in Python, but generally you'll only need one base server to extend written for each language).

Testing

Components need to adhere properly to their respective interface definition to maintain compatibility in the system. To check this, we provide test cases implemented in Python (based on the Pytest framework).

Given Python's duck-typing mechanism, it can natively test implemetations in any language, so we only need one test definition. To 'inject' your component, no matter the language, you simply need to subclass and provide an instance of your to-be-tested object.

Running Tests

Setting up to run the test cases requires a few steps:

1. CD into the PEPI base directory: `cd pepi`
2. Install the test's Python package requirements: `pip install -r test/requirements.txt`

And then running the tests simply requires a single command: `PYTHONPATH=$PWD:$PYTHONPATH py.test`

Testing Camera Components

Python Cameras

Python native components are the simplest to test. We provide tests for both local Python object components, and Python components being served over Thrift. In both cases, you just need to provide an instance of your component class in the below test fixtures and the rest will be handled.

```
import pytest

from server.tests import AbstractCameraContract, AbstractCameraOverThrift
import MyConcreteCamera

class TestMyConcreteCamera(AbstractCameraContract):
    @pytest.fixture(scope="module")
    def camera(self):
        return MyConcreteCamera()

class TestDummyCameraOverThrift(AbstractCameraOverThrift):
    @pytest.fixture(scope="module")
```

```
    def local_camera(self):
        return MyConcreteCamera()
```

Non-Python Cameras

It is still possible to test non-Python components with the provided Python test classes, but you'll need to launch the component's Thrift server yourself (perhaps with some script, depending on your language).

Suppose your server is accessible at *192.168.1.10:6000*, then we can utilise duck-typing to convince the Python local object test cases that they are just working with a local object. This achieved as follows:

```
import pytest

from server import pepi_thrift
from server.tests import AbstractCameraContract, AbstractCameraOverThrift
from thriftpy.rpc import client_context

class TestMyNonPythonConcreteCamera(AbstractCameraContract):
   @pytest.fixture(scope="module")
   def camera(self):
      with client_context(pepi_thrift.Camera, '192.168.1.10', 6000) as c:
         return c
```

Testing Servers

Testing servers is much the same as testing cameras – just targeting different test fixtures.

Python Servers

```
import pytest

from server import pepi_thrift
from server.tests import MetaCameraServerContract
from thriftpy.rpc import client_context

import MyPythonServer

class TestMyNonPythonServer(MetaCameraServerContract):
   @pytest.fixture(scope="module")
   def server(self):
      return MyPythonServer()

class TestMyPythonServerOverThrift(MetaCameraServerContractOverThrift):
   @pytest.fixture(scope="module")
   def local_server(self):
      return MyPythonServer()
```

Non-Python Servers

Again, non-Python servers can still be tested, but you'll need to launch the launch the component's Thrift server yourself. Suppose your sever is accessible at *192.168.1.10:6000*:

```
import pytest

from server import pepi_thrift
from server.tests import MetaCameraServerContract
from thriftpy.rpc import client_context

class TestMyNonPythonServer(MetaCameraServerContract):
   @pytest.fixture(scope="module")
   def server(self):
      with client_context(pepi_thrift.CameraServer, '192.168.1.10', 6000) as c:
         return c
```

pepi

raspi_server package

Submodules

raspi_server.raspi_camera module

raspi_camera.py: Provide a PEPI-compatible Camera backed by a connected Raspberry Pi Camera Module.

class raspi_server.raspi_camera.**RaspPiCamera**(*resolution=(2592, 1944)*)

Bases: *server.abstract_camera.AbstractCamera*

RaspPiCamera is a concrete AbstractCamera that uses the Raspberry Pi Camera Module v1/v2 to obtain imagery. It is capable of taking pictures in various resolutions, but defaults to the maximum resolution of 2592x1944. It essentially serves as a convenient wrapper around PiCamera, but in the PEPI format.

MAX_RESOLUTION = (2592, 1944)

SUPPORTS_STREAMING = True

get_current_resolution()

Gets the resolution of this PiCamera

get_max_resolution()

Gets the maximum supported resolution of this PiCamera

low_res_still()

Captures a 640x480 still from PiCamera natively.

set_resolution(*x*, *y*)

Sets the resolution of this camera for all future captures from it, if the provided resolution is valid.

Parameters

- **x** – the x-dimension of the desired resolution
- **y** – the y-dimension of the desired resolution

still()

Captures a still from PiCamera with its current setup.

raspi_server.raspi_server module

class raspi_server.raspi_server.**RaspPiCameraServer**(*cameras*, *stream=True*)
Bases: *server.base_camera_server.BaseCameraServer*

An implementation of a BaseCameraServer for a Raspberry Pi.

raspi_server.run module

Launcher for the Raspberry Pi server and camera implementations.

Module contents

server package

Subpackages

server.tests package

Submodules

server.tests.test_camera module

class server.tests.test_camera.**AbstractCameraContract**
Bases: object

Tests a Camera object against the defined Camera contract, essentially proving that it is compatible with all servers that use this defined contract. It uses a PyTest fixture which you must override to use.

Example

class MyCamera(AbstractCamera):

def init(): self._camera = MagicalCamera()

def still(): return self._camera.capture()

...

class TestMyCamera(AbstractCameraContract): @pytest.fixture(scope="module") def camera(self):

return MyCamera()

camera()

test_low_res_still(*camera*)

test_resolutions(*camera*)

test_still(*camera*)

server.tests.test_dummy_camera module

class server.tests.test_dummy_camera.**TestDummyCamera**
Bases: *server.tests.test_camera.AbstractCameraContract*

camera()

class server.tests.test_dummy_camera.**TestDummyCameraOverThrift**
Bases: server.tests.test_camera_over_thrift.AbstractCameraOverThrift

local_camera()

server.tests.test_iptools module

class server.tests.test_iptools.**TestIPTools**
Bases: object

Unit tests for IPTools module, mainly checking that it extracts the correct IPs from the netifaces package and that they are correctly formatted.

test_current_ip(*monkeypatch*)

test_current_ip_for_multiple(*monkeypatch*)

test_current_ips_without_gateway(*monkeypatch*)

test_gateway_ip(*monkeypatch*)

test_get_first_digits_from()

test_get_subnet_from()

test_no_best_candidate_ip_no_gateway(*monkeypatch*)

test_no_best_candidate_no_gateway(*monkeypatch*)

test_no_gateway_ip(*monkeypatch*)

server.tests.test_server module

class server.tests.test_server.**MetaCameraServerContract**
Bases: object

server()

test_capturing_to_jpgs(*server*)

test_capturing_to_pngs(*server*)

test_enumerate_methods(*server*)

test_identify(*server*)

test_image_unavailable(*server*)

test_ping(*server*)

test_stream_url(*server*)

server.tests.test_server_over_thrift module

class server.tests.test_server_over_thrift.**MetaCameraServerOverThrift**
Bases: *server.tests.test_server.MetaCameraServerContract*

local_server()

port()

run_server(*local_server*, *port*)

server(*run_server*, *port*)

server.tests.test_stream module

server.tests.test_stream.**test_jpeg_generator**(*tmpdir*)

server.tests.test_stream.**test_newest_file_in_folder_generator**(*tmpdir*)

server.tests.test_stream.**test_response**(*tmpdir*)

Module contents

Submodules

server.abstract_camera module

abstract_camera.py: Holds the Python definition of a `Camera` as defined in `pepi.thrift`.

class server.abstract_camera.**AbstractCamera**
Bases: object

AbstractCamera is an abstract base class that defines the interface required from an `Camera`. `Camera` objects are used to capture imagery a physical camera. The means in which this image is obtained does not matter, as long as presented interface is consistent.

`Camera` objects are intended to be used by `BaseCameraServer` subclasses.

A concrete `Camera` should subclass AbstractCamera and *must* implement all methods marked with @abstract-method.

get_current_resolution()
Gets the current resolution of this camera.

Returns a list of length 2 representing the resolution i.e. (x, y)

get_max_resolution()
Gets the maximum resolution supported by this camera.

Returns a list of length 2 representing the resolution i.e. (x, y)

low_res_still()
Captures a still from the camera and returns it as 3-dimensional RGB array representing the image.

Returns [[[R, G, B]] NumPy array of Numpy.uint8 0-255 values.

set_resolution(*x*, *y*)
If supported, sets the resolution of the camera.

Parameters

- **x** – the x component of the desired resolution
- **y** – the x component of the desired resolution

still()
Captures a still from the camera and returns it as 3-dimensional RGB array representing the image.

Returns [[[R, G, B]] NumPy array of Numpy.uint8 0-255 values.

class server.abstract_camera.**RGBImage**(*array*)
Bases: object

A utility object to convert images of different formats to RGB arrays

array
The array RGB array that represents this RGBImage :return:

classmethod frombytes(*mode*, *size*, *bytes_*)
Construct a RGBImage from the pixel data in a buffer.

Parameters

- **mode** – the image mode, see https://pillow.readthedocs.io/en/3.1.x/handbook/concepts.html#concept-modes
- **size** – the image size
- **bytes** – a byte buffer containing raw data for the given mode

Raises ValueError: when the bytes are malformed or not an image

Returns RGBImage

classmethod fromfile(*file*)
Construct a RGB image from the given file

Parameters **file** – a filename (str) or file object.

Raises ValueError: when the file object is malformed or not an image

Returns RGBImage

classmethod fromstring(*string*)

list
RGBImage as a native Python list.

Returns Returns this RGB as a native Python list.

low_res
Returns this RGB image as a numpy array.

Returns this RGB image as a numpy array

server.base_camera_server module

base_camera_server.py: Holds the Python implementation of a CameraServer as defined in pepi.thrift.

class server.base_camera_server.**BaseCameraServer**(*cameras*, *stream=True*)
Bases: object

BaseCameraServer is the minimal Python implementation of a CameraServer as defined in pepi.thrift. CameraServers are used in with the Apache Thrift protocol to provide RPC mechanisms, allowing control of this server over RPC, if it is launched with Thrift.

A CameraServer subclassing BaseCameraServer may override any of these methods to better reflect their use. However, care must be taken to ensure that the side effects of the subclass's methods do not affect other methods. For example, if you were to change the capture method to store images in a list for whatever reason, you would need to change the image retrieval methods.

A CameraServer's use-case is to provide a server that controls a number of cameras to be controlled in a consistent manner. This allows for a client to seamlessly control all implementations of CameraServer's, over Thrift without needing to concern themselves with what cameras are attached, the procedure call names, etc.

This BaseCameraServer implementation supports multiple connected cameras, that are transparent to the connecting client. When retrieving images, a list of encoded images are returned. The order of this list remains consistent across procedure calls.

STREAM_PORT = 6001

class StreamInfo(*port*, *folder*, *streamer*)
Bases: tuple

folder
Alias for field number 1

port
Alias for field number 0

streamer
Alias for field number 2

BaseCameraServer.**enumerate_methods**()
Retrieves a map of the methods available on this server. This is useful for clients to verify the methods it can expect to be able to call if being called remotely.

Returns dict of <method_name: [arguments]>

BaseCameraServer.**identify**()
Get the unique identifier of this server.

Returns the server's unique identifier string

BaseCameraServer.**ping**()
Ping the server to check if it is active and responding.

Returns True (always)

BaseCameraServer.**retrieve_stills_jpg**(*with_data_code*)
Retrieves the images stored under *with_data_code*, if they exist, and encodes them into a .jpg str (i.e. bytes).

The order of the returned images is consistent, e.g. Camera #1, #2 .., #x returned in that order.

Parameters **with_data_code** – the data_code from which the image will be retrieved

Raises ImageUnavailable: when image requested with an invalid/unknown data_code

Returns a list of strings with each string containing an encoded as a .jpg

BaseCameraServer.**retrieve_stills_png**(*with_data_code*)
Retrieves the images stored under *with_data_code*, if they exist, and encodes them into a .png str (i.e. bytes).

The order of the returned images is consistent, e.g. Camera #1, #2 .., #x returned in that order.

Parameters **with_data_code** – the data_code from which the image will be retrieved

Raises ImageUnavailable: when image requested with an invalid/unknown data_code

Returns a list of strings with each string containing an encoded as a .png

BaseCameraServer.**shutdown**()
Shutdown the server (i.e. power-off).

Subclasses may choose to ignore calls to this function, in which case they should override this function to do nothing.

Returns None

BaseCameraServer.**start_capture**(*data_code*)
Immediately starts the process of capturing from this server's Camera(s), and stores the captured data under the given unique data_code.

Note: the received *data_code* is assumed to be unique. Subclasses may choose to implement better isolation methods, but this is not guaranteed nor required.

Parameters **data_code** – the requested data_code to store the capture under

Returns None

BaseCameraServer.**stream_urls**()
Get the a list of URLs where the MJPG image stream of each camera connected to this server may be accessed.

The order of the returned images is consistent, e.g. Camera #1, #2 .., #x returned in that order.

Returns a list of the stream URLs as a string

class server.base_camera_server.**CameraTimelapser**(*camera*, *folder*, *interval*)
Bases: threading.Thread

CameraTimelapser is a utility class designed to call the low_res_still() method on a concrete AbstractCamera at a defined rate and save it to a given folder for the purposes of timelapsing or streaming from the camera.

run()

server.dummy_camera module

dummy_camera.py: A concrete *AbstractCamera* that generates random images of RGB noise for debugging purposes.

class server.dummy_camera.**DummyCamera**(*resolution=[1920, 1080]*)
Bases: *server.abstract_camera.AbstractCamera*

DummyCamera is a concrete *AbstractCamera* that generates random images of RGB noise.

MAX_RESOLUTION = [1920, 1080]

get_current_resolution()
Gets the current resolution of this camera.

Returns a list of length 2 representing the resolution i.e. (x, y)

get_max_resolution()
Gets the maximum resolution supported by this camera.

Returns a list of length 2 representing the resolution i.e. (x, y)

low_res_still()
Captures a low resolution still from the camera and returns it as 3-dimensional RGB array representing the image.

Returns multidimensional list of row, column, RGB values between 0-255.

resolution

set_resolution(*x*, *y*)
If supported, sets the resolution of the camera.

Parameters

- **x** – the x component of the desired resolution

- **y** – the x component of the desired resolution

still()
Captures a still from the camera and returns it as 3-dimensional RGB array representing the image.

Returns multidimensional list of row, column, RGB values between 0-255.

server.iptools module

Tools related to IP operations, mainly getting IP's and checking if servers exist at a certain IP:port combination.

class server.iptools.**IPTools**
Bases: object

Static methods that work on IPv4-based connections, such as getting current IP, gateway IP, etc.

static current_ips()
Gets the best-candidate IPs of this computer on the network.

static gateway_ip()
Gets the IP of this computer's gateway, if it exists.

static get_first_digits_from(*ip*, *num_digits*, *with_dot=True*)
Gets the given sets of digits from an IP, with or without the trailing period.

Note if num_digits exceeds 4, the ip is returned unchanged.

Parameters

- **ip** – the IP address string to trim
- **num_digits** – the number of digit sets to keep
- **with_dot** – True to keep the trailing period, False to not

static get_subnet_from(*ip*, *with_dot=True*)
Gets the first 3 sets of digits from an IP, with or without the trailing period.

Parameters

- **ip** – the IP address string to trim
- **with_dot** – True to keep the trailing period, False to not

server.pepi_thrift_loader module

Loads PEPI's Thrift interface definitions as Python objects accessible under pepi_thrift after importing this module.

server.stream module

class server.stream.**MJPGStreamer**(*img_path*, *ip='0.0.0.0'*, *port=6001*)
Bases: object

Starts a HTTP stream based on JPEG images obtained from the specified folder.

static jpeg_image_generator(*path*, *quality=85*, *resolution=(640*, *480)*)

Generates JPEG bytes from any image file in the given path based on the second newest file modified in the given *path*.

JPEG files (and other image formats) are compressed to a JPEG as they must be modified to be resized.

Args: path: path to the folder to check for new files quality: JPEG quality to compress to (0 lowest quality, 100 highest) resolution: resolution to yield the JPEGs as

static **newest_file_in_folder**(*path*, *delete_old=True*)
Generator that yields the second newest file by modified time in the given *path*. The second newest file is yielded so that files in the process of being written are not used before they are complete; this is generally not an issue that the second newest file faces.

Args: path: path to the folder to check for new files delete_old: True to delete all but the second_newest and newest files, False to not delete any

stream_handler_factory(*img_path*)
Create a MJPGStreamHandler with the *img_path* set inside of it.

This is necessary due to how BaseHTTPServer creates the BaseHTTPRequestHandler.

Args: img_path: the path to give to MJPGStreamHandler

class server.stream.**ThreadedHTTPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Bases: socketserver.ThreadingMixIn, http.server.HTTPServer

A multi-threaded HTTP server, i.e. creates a new thread to respond to each connection, so multiple connections can coexist.

allow_reuse_address = True

daemon_threads = True

Module contents

CHAPTER 5

Indices and Tables

- genindex
- modindex
- search

Python Module Index

r

s

Index