Camera live streaming using MJPEG format

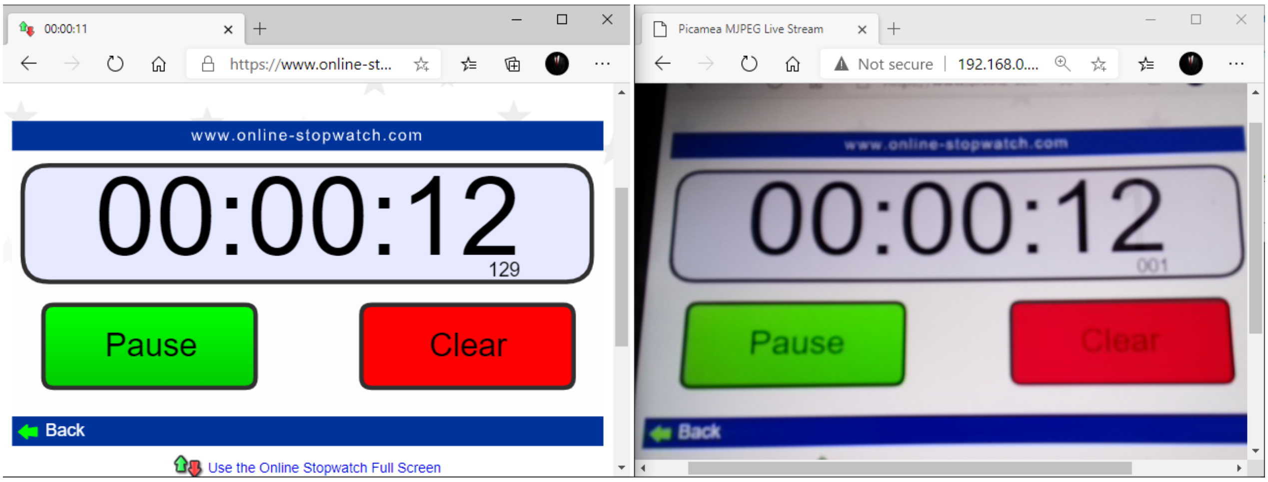
MJPEG streaming is a simple method to get low latency. It sends JPEG images over the network and display that sequence of images on the user webpage. However, it consumes much more bandwidth. A server can be easily made by Picamera and Python HTTP.

#pi #stream #camera #mjpeg #picamera

Last update: April 23, 2021

Table of Content

1. Record video to a stream
2. Frame buffer
3. Streaming Web server
4. Request Handler
5. Synchronize between threads
6. Some updates in the script
    6.1. Class variable
    6.2. Instance variable
    6.3. *args and **kwargs
    6.4. Lambda function
    6.5. Measure FPS



A low latency in MJPEG streaming

Source code available at https://github.com/vuquangtrong/pi_streaming

There are many choices to start streaming with MJPEG (MJPG) format. The basic method is to send a series of JPEG (JPG) image to the user's webpage and display it in an `<img>` tag. An example of streaming server is mjpg-streamer.

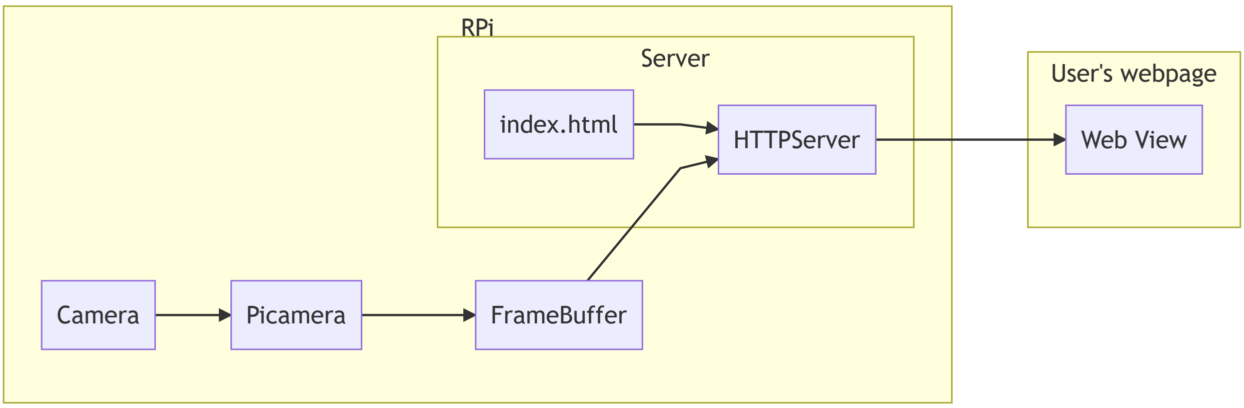
This post shows a method to develop a streaming system, starting with a Python package named picamera.

Setup PiCamera

To setup `picamera`, please see more in Setup Camera

Picamera has an example to stream MJPEG at Web streaming section. Just copy and run their code to see how it works.

The basic structure of this streaming server is as below:



1. Record video to a stream

This is a basic step to write a video stream to a buffered memory. Python has `io` package which provides binary I/O which expects bytes-like objects and produces `bytes` objects. No encoding, decoding, or newline translation is performed.

```
from io import BytesIO
from picamera import PiCamera

# create in-memory stream
stream = BytesIO()

# create camera object (instance)
camera = PiCamera()
# config camera
camera.resolution = (640, 480)
# start recording to stream
camera.start_recording(stream, format='h264', quality=23)
# wait
camera.wait_recording(15)
# stop recording
camera.stop_recording()
```

That's very easy.

2. Frame buffer

Next step is to create a custom output to used in `PiCamera.start_recording()` method. Refer to Custom outputs:

A file-like object (as far as picamera is concerned) is simply an object with:

- a `write(b)` method which must accept a single parameter consisting of a byte-string, and which can optionally return the number of bytes written.
- a `flush()` method with no parameters, which will be called at the end of output.

In `write` method, it can implement code that reacts to each and every frame. `write` method is called so frequently, its implementation must be sufficiently rapid that it doesn't stall the encoder.

Let's write a class `FrameBuffer()` like below:

```
import io

class FrameBuffer(object):
    def __init__(self):
        # store each frame
        self.frame = None
        # buffer to hold incoming frame
        self.buffer = io.BytesIO()
```

```python
    def write(self, buf):
        # if it's a JPEG image
        if buf.startswith(b'\xff\xd8'):
            # write to buffer
            self.buffer.seek(0)
            self.buffer.write(buf)
            # extract frame
            self.buffer.truncate()
            self.frame = self.buffer.getvalue()
```

Note that `FrameBuffer.frame` will be used to send the frame to user's webpage.

Then, use the FrameBuffer instead of the buffered memory:

```python
# create buffer
frame_buffer = FrameBuffer()

# write to framebuffer
camera.start_recording(frame_buffer, format='mjpeg')
```

3. Streaming Web server

Python has built-in a simple HTTP Server, which is ready to run by providing a server address and a request handler class.

```python
from http.server import HTTPServer, BaseHTTPRequestHandler

def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

Now, look at some pre-defined Request Handler classes:

`class http.server.BaseHTTPRequestHandler(request, client_address, server)`

> This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). `BaseHTTPRequestHandler` provides a number of class and instance variables, and methods for use by subclasses.
>
> The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method *SPAM*, the `do_SPAM()` method will be called with no arguments. All of the relevant

information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

`class http.server.SimpleHTTPRequestHandler(request, client_address, server, directory=None)`

This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests. A lot of the work, such as parsing the request, is done by the base class `BaseHTTPRequestHandler`. This class implements the `do_GET()` and `do_HEAD()` functions.

`class http.server.CGIHTTPRequestHandler(request, client_address, server)`

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in `SimpleHTTPRequestHandler`.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the cgi_directories path.

Let's start with `SimpleHTTPRequestHandler` which has some implemented features.

4. Request Handler

Based on `SimpleHTTPRequestHandler`, create a new class `StreamingHandler` and only override `do_GET()` method to just print requested `path` and then call the base method as it is already implemented.

```
from http.server import SimpleHTTPRequestHandler

class StreamingHandler(SimpleHTTPRequestHandler):
    def do_GET(self):
        print(self.path)
        # call to the base method implemented in SimpleHTTPRequestHandler
        super().do_GET()
```

The `SimpleHTTPRequestHandler` will serve files in *GET* requests, and it will looking for `index.html` for the homepage.

To display image, create an `<img>` tag which will request a file named `stream.mjpg`.

```
<html>
    <head>
        <title>Picamea MJPEG Live Stream</title>
    </head>
    <body>
        <!-- Request MJPEG stream -->
        <img src="stream.mjpg"/>
    </body>
</html>
```

There is no actual `stream.mjpg` file!. When the web page request `stream.mjpg`, web serve should return a stream, not a single file, therefore a special sequence is needed to handle this special request of `stream.mjpg` file in the `do_GET()` method:

1. Send response with HTTP Status Code 200 (Successful responses)

   read more at HTTP response status codes

2. Send header with information to notify web client about type of responded content

   read more at Content-Type

3. Send the content in a stream format (loop forever!): send content type of each frame, then send the actual `frame` data

```
from http.server import SimpleHTTPRequestHandler

class StreamingHandler(SimpleHTTPRequestHandler):
    def do_GET(self):
        if self.path == '/stream.mjpg':
            # response
            self.send_response(200)
            # header
            self.send_header('Age', 0)
            self.send_header('Cache-Control', 'no-cache, private')
            self.send_header('Pragma', 'no-cache')
            self.send_header('Content-Type', 'multipart/x-mixed-replace;
boundary=FRAME')
            self.end_headers()
            try:
                while True:
                    frame = frame_buffer.frame # need frame_buffer as global
                    self.wfile.write(b'--FRAME\r\n')
                    self.send_header('Content-Type', 'image/jpeg')
                    self.send_header('Content-Length', len(frame))
                    self.end_headers()
                    self.wfile.write(frame)
                    self.wfile.write(b'\r\n')
            except Exception as e:
                print(str(e))
        else:
            super().do_GET()
```

Finally, wrap them up by creating instances of FrameBuffer, PiCamera, HTTPServer to start streaming:

```
frame_buffer = FrameBuffer()

camera = PiCamera(resolution='640x480', framerate=24)
camera.start_recording(frame_buffer, format='mjpeg')

server_address = ('', 8000)
handler_class = StreamingHandler # alias
try:
    httpd = HTTPServer(server_address, handler_class)
    httpd.serve_forever()
finally:
    camera.stop_recording()
```

Bug: Hangup stream

When run the above code, the web page shows up but with only one frame displayed, CPU is locked up at 100%, because the block `while True:` loop causes the problem.

Need to find a way to synchronize between camera thread and web server thread: send a frame only when it is availabe.

5. Synchronize between threads

Python has implemented a lock mechanism between threads:

`class threading.Condition(lock=None)`

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread. If the `lock` argument is given and not `None`, it must be a `Lock` or `RLock` object, and it is used as the underlying lock. Otherwise, a new `RLock` object is created and used as the underlying lock.

`wait(timeout=None)`

Wait until notified or until a timeout occurs. If the calling thread has not acquired the `lock` when this method is called, a `RuntimeError` is raised.

This method releases the underlying `lock`, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the `lock` and returns.

`notify_all()`

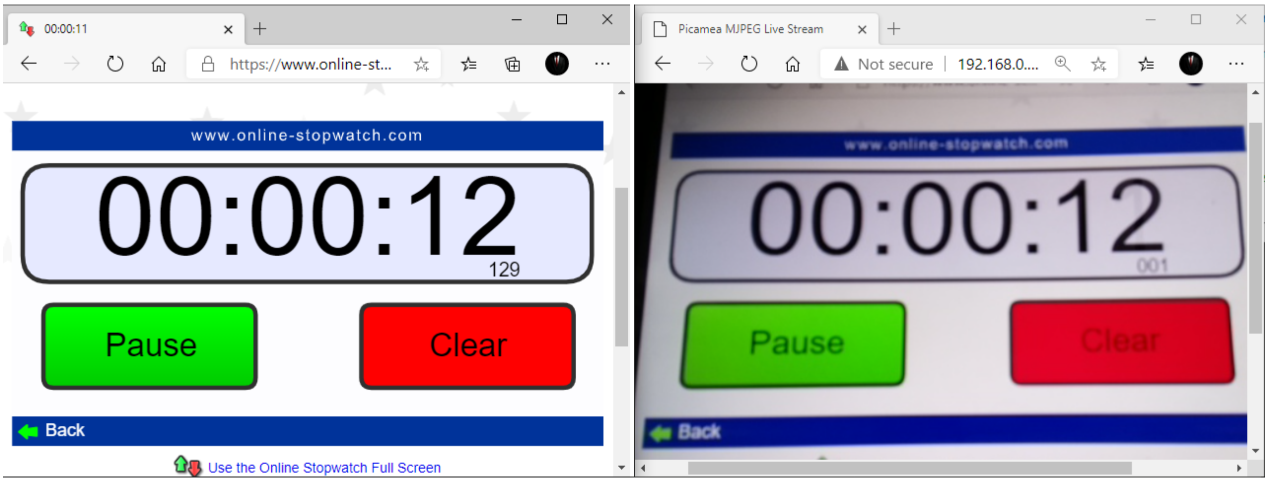
> Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the `lock` when this method is called, a `RuntimeError` is raised.

Then add a `Condition` object in `FrameBuffer`, and use it in `StreamingHandler`:

```
from threading import Condition

class FrameBuffer(object):
    def __init__(self):
        self.frame = None
        self.buffer = io.BytesIO()
        # synchronize between threads
        self.condition = Condition()

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            with self.condition:
                self.buffer.seek(0)
                self.buffer.write(buf)
                self.buffer.truncate()
                self.frame = self.buffer.getvalue()
                # notify other threads
                self.condition.notify_all()

class StreamingHandler(SimpleHTTPRequestHandler):
    def do_GET(self):
        if self.path == '/stream.mjpg':
            ...
            try:
                while True:
                    with frame_buffer.condition:
                        # wait for a new frame
                        frame_buffer.condition.wait()
                        frame = frame_buffer.frame # access global variable, need
to change later
```

Wow, it works!!! The latency is just about 200ms which is unachievable with HLS/ MPEG-DASH

However, the CPU usage is quite high, Pi Zero W *only can handle 6 clients* at the same time with video quality at 640x480 @25fps.



A low latency in MJPEG streaming

6. Some updates in the script

The instance `frame_buffer` is used as a global variable in the `StreamingHandler`, it is *not good* if there is another `FrameBuffer` used for another stream in a same script.

Here is an advanced method to have multiple frame buffers by passing an instance of `FrameBuffer` into an instance of `StreamingHandler`. It can be done by adding an **Instance variable** that holds reference to an instance of `FrameBuffer`, but can *not* be done using **Class variable**.

Let's check how they work.

6.1. Class variable

Class variable is shared by all instance, therefore it acts like a global static attribute of the class.

```
class StreamingHandler(SimpleHTTPRequestHandler):
    # class variable refers to an instance of FrameBuffer
    my_frame_buffer = None

    def do_GET(self):
        ...
        frame = self.my_frame_buffer.frame

# create an instance of FrameBuffer
frame_buffer = FrameBuffer()
handler_class = StreamingHandler # alias

# assign class variable
handler_class.my_frame_buffer = frame_buffer

# all instance will share class variables
```

```
first_handler = StreamingHandler()
second_handler = StreamingHandler()

# first_handler.my_frame_buffer will be the same as
second_handler.my_frame_buffer
```

6.2. Instance variable

Instance variables are for the data unique to each instance, they are create in the `__init()__` constructor of that class:

```
class StreamingHandler(SimpleHTTPRequestHandler):
    def __init__(self, frame_buffer, request, client_address, server,
directory=None):
        self.my_frame_buffer = frame_buffer
        super().__init__(request, client_address, server, directory)

    def do_GET():
        ...
```

However, with this modification, script cannot use `StreamingHandler` to initialize `ThreadingHTTPServer` anymore, because it expects to call a request handler with only required positional arguments `(request, client_address, server)`, without a new argument `frame_buffer`.

Therefore, write a function that convert expected params list to new params list:

```
frame_buffer = FrameBuffer()

def getStreamingHandler(request, client_address, server):
    return StreamingHandler(frame_buffer, request, client_address, server)

httpd = ThreadingHTTPServer(address, getStreamingHandler)
```

Well, it works, but the convert function actually drop the param `directory` which is an optional param in original constructor of `SimpleHTTPRequestHandler`. To solve this problem, let's use special `*args` and `**kwargs` params.

6.3. `*args` and `**kwargs`

The special `*args` and `**kwargs` params allow to pass multiple arguments or keyword arguments to a function. Read about them in here.

So, change the param list `(request, client_address, server, ...)` to `*args` in code, then it looks better:

```
class StreamingHandler(SimpleHTTPRequestHandler):
    def __init__(self, frame_buffer, *args):
        self.my_frame_buffer = frame_buffer
        super().__init__(*args)

frame_buffer = FrameBuffer()

def getStreamingHandler(*args):
    return StreamingHandler(frame_buffer, *args)

httpd = ThreadingHTTPServer(address, getStreamingHandler)
```

6.4. Lambda function

Python and other languages like Java, C#, and even C++ have had lambda functions added to their syntax, whereas languages like LISP or the ML family of languages, Haskell, OCaml, and F#, use lambdas as a core concept. Read more in here

So, reduce the function `getStreamingHandler` to a lambda function which can be declared in-line when creating `ThreadingHTTPServer` instance:

```
frame_buffer = FrameBuffer()
httpd = ThreadingHTTPServer(address, lambda *args: StreamingHandler(frame_buffer,
*args))
```

6.5. Measure FPS

In the while loop of sending frames, use `frame_count` variable to count the number of processed frames. With `time` package, it is easy to calculate FPS over a defined period, for example, 5 seconds in below code:

```
try:
    # tracking serving time
    start_time = time.time()
    frame_count = 0
    # endless stream
    while True:
        with self.frames_buffer.condition:
            # wait for a new frame
            self.frames_buffer.condition.wait()
            # it's available, pick it up
            frame = self.frames_buffer.frame
            # send it
            ...
            # count frames
            frame_count += 1
            # calculate FPS every 5s
            if (time.time() - start_time) > 5:
                print("FPS: ", frame_count / (time.time() - start_time))
```

```
                frame_count = 0
                start_time = time.time()
...
```

Some lines of code to handle exception are also needed, for full source code, please download by clicking the below button.

Download stream_picamera_mjpeg

Comments