Raspberry Pi - Camera live streaming using H264 format

H.264 is a well-known video compression standard for high-definition digital video. One of its advantages is having Network Abstraction Layer (NAL) units which is easily transferred on the internet for video streaming. This post guides to use PiCamera to capture and make H264 video stream by sending H264 NAL units over the internet via a websocket. The decoder used in user's browser is Broadway.js library.

#raspberry-pi #stream #camera #h264 #python #picamera

Last update: 2021-08-04 17:31:07

Table of Content

1. Stream video

Live streaming requires very low latency with acceptable quality and bandwidth. MJPEG Streaming has low latency but high bandwidth. HLS/DASH Streaming is not realtime. Therefore people have to find a method to transfer encoded video in realtime.

An example of streaming real video (not frame by frame) is pistreaming which uses `mpeg1video` format. The video stream is sent to user's browser via a `websocket`, and is decoded by JSMPEG javascript library.

This post will show a method similar to both MPEG stream and MJPEG images: send video using H264 Network Abstract Layer (NAL) units and decode those units to display video.

2. Broadway.js - H264 decoder

The h264-live-player is used for streaming an Android screen to a webpage. That player uses Broadway.js library to decode the video stream. It also has a streaming server for Raspberry Pi using `raspivid`, `nodejs`, and `websocket`.

The method used in that player is quite similar to MJPEG Streaming: video stream is split into NAL units (Video Control Layer (VCL) or non-VLC packages), then transported using a websocket, and finally decoded by the Broadway.js library.

Broadway.js provides `Player.js`, `Decoder.js`, `YUVCanvas.js`, and `avc.wasm`, with very simple usage: create a new Player object; then put the player's canvas to an element to display the video; and call the decode function with the stream data.

```
var player = new Player({<options>});
playerElement = document.getElementById(playerId)
playerElement.appendChild(player.canvas)
player.decode(<h264 data>);
```

3. Create a webpage

The webpage firstly loads necessary libraries and requests to open a websocket connection, then feeds Broadway decoder with a streaming data chunk by calling `player.decode()` method.

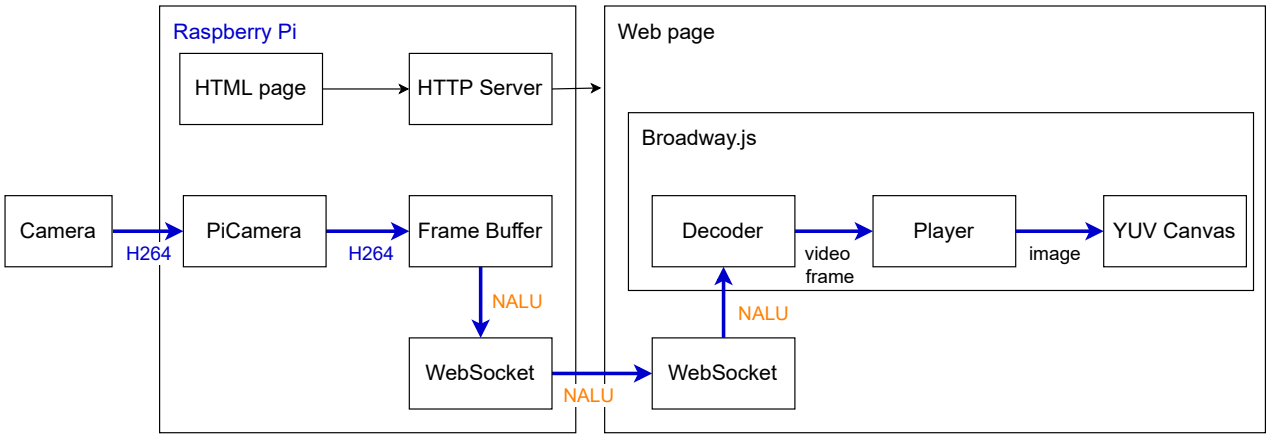
index.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset='utf-8'>
  <title>PiCamera H264 Streaming</title>
</head>
```

```
<body>
  <h1>PiCamera H264 Streaming</h1>
  <div id='viewer'></div>
  <script src='Decoder.js'></script>
  <script src='YUVCanvas.js'></script>
  <script src='Player.js'></script>
  <script>
    // player
    window.player = new Player({
      useWorker: true,
      webgl: 'auto',
      size: { width: 848, height: 480 }
    })
    var playerElement = document.getElementById('viewer')
    playerElement.appendChild(window.player.canvas)
    // Websocket
    var wsUri =
      window.location.protocol.replace(/http/, 'ws')
      + '//' + window.location.hostname + ':9000'
    var ws = new WebSocket(wsUri)
    ws.binaryType = 'arraybuffer'
    ws.onopen = function (e) {
      console.log('Client connected')
      ws.onmessage = function (msg) {
        // decode stream
        window.player.decode(new Uint8Array(msg.data));
      }
    }
    ws.onclose = function (e) {
      console.log('Client disconnected')
    }
  </script>
</body>

</html>
```

4. Create server

Here is the structure of a H264 streaming system use in the post. PiCamera will capture a H264 stream and write to FrameBuffer each NALU package which will be sent to the Broadway.js via a websocket. The decoded video frame will be drawn on a canvas to show in the webpage. The webpage is provided via a HTTP server which will load the Broadway.js and setup the decoder and a websocket client.



H264 streaming server structure

4.1. Frame buffer

The `FrameBuffer` is implemented as an output of Picamera which store each H264 Network Abstraction Layer (NAL) unit from H264/AVC or HEVC video stream. There is a `Condition` object to synchronize between `FrameBuffer` and `WebSocketServer`.

For more detail of how to construct FrameBuffer class, refer to Streaming using MJPEG

```
import io
from threading import Condition

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

    def write(self, buf):
        if buf.startswith(b'\x00\x00\x00\x01'):
            with self.condition:
                self.buffer.seek(0)
                self.buffer.write(buf)
                self.buffer.truncate()
                self.frame = self.buffer.getvalue()
                self.condition.notify_all()
```

4.2. HTTP Server

The web interface server is served by `ThreadingHTTPServer` with `SimpleHTTPRequestHandler` to serve requested files (`index.html` , `*.js` , etc.).

```
from http.server import SimpleHTTPRequestHandler, ThreadingHTTPServer
from threading import Thread
```

```
httpd = ThreadingHTTPServer(('', 8000), SimpleHTTPRequestHandler)
httpd_thread = Thread(target=httpd.serve_forever)
```

4.3. Websocket Server

One of good WebSocket packages for Python is ws4py which supports both Python 2 and Python 3 (while websockets requires Python ≥ 3.6.1).

From the package `ws4py`, use module `wsgiref` as a Web Server Gateway Interface to make a websocket server.

The function `make_server()` needs to know the port, and some classes to initialize a server, those can be built-in objects in `ws4py` such as `WebSocketWSGIRequestHandler`, `WebSocketWSGIApplication`, and base `WebSocket`.

Finally, a client manager should be created in the websocket server, to use broadcasting function later.

```
from wsgiref.simple_server import make_server
from threading import Thread

websocketd = make_server('', 9000, server_class=WSGIServer,
    handler_class=WebSocketWSGIRequestHandler,
    app=WebSocketWSGIApplication(handler_cls=WebSocket))
websocketd.initialize_websockets_manager()
websocketd_thread = Thread(target=websocketd.serve_forever)
```

4.4. Main thread

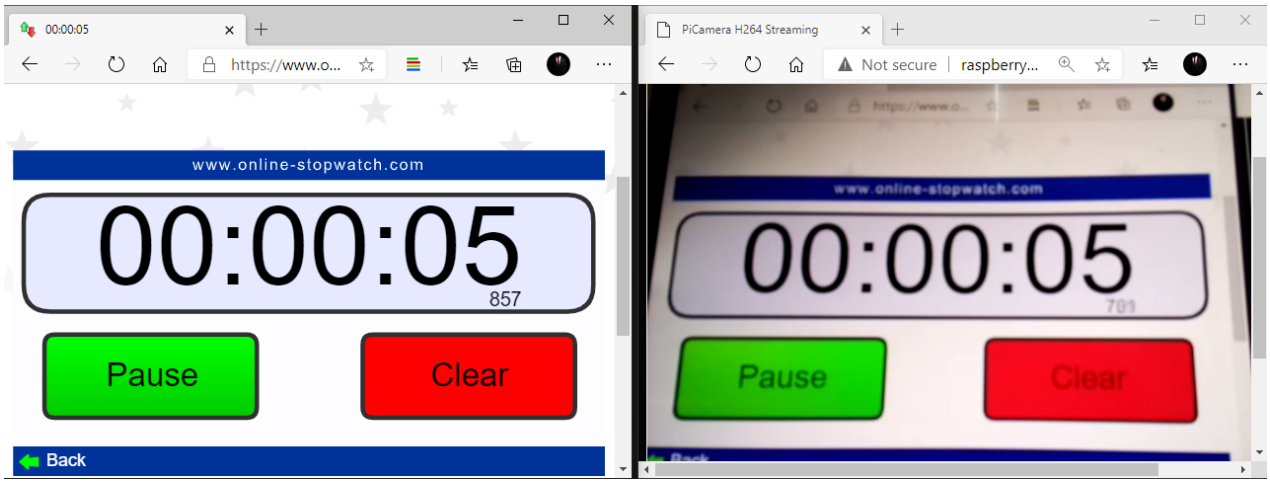
The main application will start PiCamera and write output video in `h264` encode. As noted in Broadway.js, it only supports H264 **Baseline** profile, therefore, set `profile="baseline"` when starting video record.

```
import picamera

with picamera.PiCamera(resolution='640x480', framerate=24) as camera:
    broadcasting = True
    frame_buffer = FrameBuffer()
    camera.start_recording(frame_buffer, format='h264', profile="baseline")
```

The main loop should broadcast H264 NAL units to all connected clients, after it starts threads for HTTP Server and WebSocket Server.

```
try:
    websocketd_thread.start()
    httpd_thread.start()
```

```
    while broadcasting:
        with frame_buffer.condition:
            frame_buffer.condition.wait()
            websocketd.manager.broadcast(frame_buffer.frame, binary=True)
```



Low latency in H264 streaming

There may be some delay before the video shows up in user webpage because the Player has to wait for a IDR Frame (keyframe) to be able to start decoding.

Some lines of code to handle exception are also needed, for full source code, please download by clicking on the download button at the beginning of this post.