

**SE
TU**

Ollscoil
Teicneolaíochta
an Oirdheiscirt

South East
Technological
University

CONTINUOUS ASSESSMENT REPORT

Distributed Systems - 92021 - [2022-2023]

Assignment #3

Live Streaming Service - Facial Recognition Server

Course: Bachelor of Science (Honours) in Software Systems Practice

KaiYu Chen

20100199

Table of Contents

| | |
|--|-----------|
| 1. INTRODUCTION..... | 1 |
| 2. REQUIREMENTS | 1 |
| 2.1 FUNCTIONAL REQUIREMENTS..... | 1 |
| 2.2 NON-FUNCTIONAL REQUIREMENTS | 1 |
| 3. ENVIRONMENT..... | 2 |
| 4. SYSTEM ARCHITECTURE..... | 2 |
| 5. DIRECTORY STRUCTURE..... | 2 |
| 6. TECHNOLOGIES | 3 |
| 7. INSTALLATION | 3 |
| 7.1 PROBLEMS | 4 |
| 7.1.1 <i>Could not find version that satisfies requirement cv2 OpenCV.....</i> | <i>4</i> |
| 7.1.2 <i>socketio</i> | <i>4</i> |
| 7.1.3 <i>Other errors.....</i> | <i>4</i> |
| 8. IMPLEMENTATION | 4 |
| 8.1 IMPLEMENTATION | 5 |
| 8.1.1 <i>Use separate threads for camera reading, model inference, and image display</i> | <i>6</i> |
| 8.1.2 <i>Skip some frames</i> | <i>7</i> |
| 8.1.3 <i>Compress the frames (to about 25% of the original size) before feeding them into the models</i> | <i>8</i> |
| 8.2 ERRORS | 10 |
| 8.2.1 <i>cv2.error</i> | <i>10</i> |
| 8.2.2 <i>socket.gethostbyname(hostname) error</i> | <i>10</i> |
| 9. CONCLUSION | 11 |
| 10. BIBLIOGRAPHY | 11 |
| 11. APPENDIX..... | 12 |

Table of Figures

| | |
|---|----|
| FIGURE 1 SYSTEM ARCHITECTURE | 2 |
| FIGURE 2 FOLDER TREE OF THE PROJECT DIRECTORY | 2 |
| FIGURE 3 LOADS THE FACE IMAGES | 5 |
| FIGURE 4 COMPUTES THE FACE ENCODINGS FOR THESE KNOWN IMAGES | 5 |
| FIGURE 5 FACE_DISTANCE_TO_CONF FUNCTION | 6 |
| FIGURE 6 CREATE A CAMERA OBJECT | 6 |
| FIGURE 7 USE SEPARATE THREADS FOR CAMERA READING, MODEL INFERENCE, AND IMAGE DISPLAY | 6 |
| FIGURE 8 SKIP SOME FRAMES | 7 |
| FIGURE 9 PROCESS_FRAME FUNCTION | 7 |
| FIGURE 10 COMPRESS THE FRAMES (TO ABOUT 25% OF THE ORIGINAL SIZE) | 8 |
| FIGURE 11 GENERATES THE PROCESSED CAMERA FRAME SEQUENCE | 8 |
| FIGURE 12 RUN SERVER | 8 |
| FIGURE 13 ROUTER | 9 |
| FIGURE 14 CLIENT WEB PAGE | 9 |
| FIGURE 15 SKIP THE WRONG FILE AND EMPTY IMAGES | 10 |
| FIGURE 16 GET THE LOCAL IP ADDRESS BY USING GOOGLE PUBLIC DNS | 10 |

1. Introduction

Live streaming has gradually become a very important market in the online video industry in recent years, with more and more people sharing their lives, work, games etc. via live streaming. From the dedicated live streaming platform twitch to the video site YouTube, all have launched live streaming services, from which we can see how popular live streaming platforms have become. This report is an Assignment #3 report for the continuous assessment of Distributed Systems - 92021 - [2022-2023]. This CA looks at live streaming and face recognition, with the aim of building a live streaming site that allows hosts to stream their own cameras and share them for others to see. Additionally, the program used for sharing the camera supports facial recognition, which identifies the faces in front of the camera. If a recognized face is present in the training set, the program displays the name of the person in front of the camera. If the face is not recognized, the program will prompt "Unknown".

2. Requirements

2.1 Functional requirements

Functional requirements are the functions or features that a system must have in order to meet the needs of the user. For this application, the functional requirements include:

1. Live video stream processing: capturing a live video stream from the camera and processing the video frames.
2. Face detection: detection of face areas in video frames.
3. Face recognition: recognize detected faces, compare them to a pre-stored face library and determine if they match.
4. Online display: the processed video frames are displayed on the client's web page in real time, including drawing rectangular boxes around the recognized faces and displaying the recognized person's name and match.

2.2 Non-functional requirements

Non-functional requirements are measures of system performance, such as availability, reliability, maintainability, etc. For this real-time face recognition application, non-functional requirements include:

1. Response time: The application should generate a live video stream as soon as possible after receiving a client request to ensure user experience.
2. Real-time: the processing speed of face recognition should be fast enough to ensure a smooth real-time presentation.
3. System stability: the application should have certain error handling capability for abnormal situations such as image file corruption.
4. Ease of use: the user interface should be simple and clear and easy to use.
5. Maintainability: The code structure is clear and modular in design to facilitate later maintenance and upgrade.

3. Environment

macOS Ventura 13.3.1

Python 3.9.16

4. System Architecture

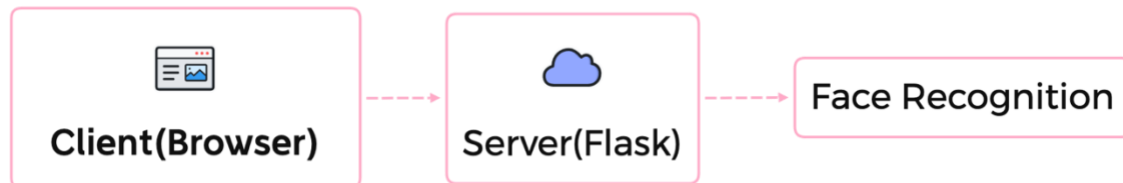


Figure 1 System Architecture

Fig.1 illustrates the basic system architecture of this program. Users access the server resource (this program) through the client (browser), and the server uses the Flask framework and provides services through Face Recognition. In other words, users can access the server resource through their browser, and the server shares the camera's live streaming content, which is streamed to the client. The streamed resources sent to the client are processed by the server and include the results of facial recognition. Therefore, the client can see the results of facial recognition. More specific details about the project structure will be introduced in the next section.

5. Directory Structure

Fig.2 shows the Directory Structure of this project, which contains two `.py` files and one `.html` file as seen in the Folder tree. As mentioned in the previous section, in this application, the server service is provided by `server.py`, which is responsible for setting up and running the WSGI server to host the Flask application in the production environment. Face Recognition is provided by `stream.py`, which is responsible for processing camera data streams, displaying and processing facial recognition tasks with clients, and setting up routes, etc. These tasks include loading face images and generating face encodings, etc.

```
> tree
.
├── README.md
├── __pycache__
│   └── stream.cpython-39.pyc
├── docs
│   └── Continuous Assessment Report.docx
├── faces
│   ├── ChengHaoXie.jpg
│   ├── GEM.jpg
│   ├── KaiYuChen.jpg
│   └── Taylor.jpg
├── requirements.txt
├── server.py
├── stream.py
├── templates
│   └── index.html
└── 5 directories, 11 files
```

A screenshot of a terminal window showing the directory structure of the project. The terminal title is '~//facialRecognitionServer' and the prompt is 'main'. The output of the 'tree' command shows a hierarchical structure with directories like 'docs', 'faces', and 'templates', and files like 'README.md', 'requirements.txt', 'server.py', 'stream.py', and 'index.html'. The total count is 5 directories and 11 files.

Figure 2 Folder tree of the project directory

6. Technologies

The following technologies are mainly used in this application, they are:

- **opencv-python**: used to process images and video streams. In this application it is used to capture live video streams from the camera, resize images and plot the results of face recognition.
- **Flask**: used as a web server framework. In this application, Flask is used to create web applications and set up routing so that clients can access live video streams.
- **numpy**: used for numerical calculations. In this application, it is mainly used to find the index of the best match in the face recognition process.
- **python-socketio**: used for real-time communication. In this program it is used to create a socket.IO server.
- **Flask-SocketIO**: Integrates Socket.IO into a Flask application. In this application it is used to implement real time communication in the application.
- **waitress**: acts as a WSGI server in the production environment. In this application it is used to start a WSGI server to host the Flask application.
- **cmake**: is used to support the installation of dlib, a cross-platform C++ library for solving problems in machine learning, computer vision, image processing and more. In this application, cmake is used as a dependency to install dlib.
- **dlib**: is used to support the installation of face_recognition. In this application, dlib is the underlying library used to implement the face recognition functionality.
- **face_recognition**: used to perform face recognition. In this application it is used to detect faces in video frames, extract face features, compare face features and calculate matches.

7. Installation

You can install the required packages for this program by running the following command in the root directory:

```
pip3 install -r requirements.txt
```

or

```
pip install -r requirements.txt
```

I encountered some errors when configuring my environment, and you may encounter these problems as well. You may be able to try to resolve the problem by using these commands:

```
pip3 install --upgrade pip
```

```
python3 -m pip install --upgrade setuptools
```

Here are some of the other problems I have encountered:

7.1 Problems

7.1.1 Could not find version that satisfies requirement cv2 OpenCV

Error log:

Could not find version that satisfies requirement cv2 OpenCV

Solution:

Using **pip install opencv-python** instead of pip install cv2 [1]

7.1.2 socketio

Why error happen?

The package literally named "socketio" seems like it may simply be old and unsupported, while "python-socketio" seems alive and developed. [2]

Solution:

Using **python3 -m pip install python-socketio**

7.1.3 Other errors

Error log:

fatal error: 'Python.h' file not found / Could not build wheels / metadata-generation-failed / setuptools>=49.6.0, but you have setuptools 3.3

Solution:

pip3 install --upgrade pip

python3 -m pip install --upgrade setuptools [3]

8. Implementation

This section will describe the implementation of this program in detail. This section will first introduce the packages used, then the functional implementation with the implementation of the following three specific requirements:

- Compress the frames (to about 25% of the original size) before feeding them into the models. This already makes a huge difference, while recognition still works well!
- Skip some frames. Do the detection/recognition part only on every other frame.
- Use separate threads for camera reading, model inference, and image display.

8.1 Implementation

In **stream.py**, the program first loads the face images (Fig.3) stored in the faces folder via the **face_recognition.load_image_file** function and stores the images and the corresponding names (filenames) in the **face_images** and **face_names** arrays respectively.

```
21 # Get all the images in the faces folder
22 image_files = [f for f in os.listdir(face_images_path) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]
23
24 # Define some constants
25 SCALE_FACTOR = 0.25
26 UNKNOWN_THRESHOLD = 0.6
27
28 # Get all the images in the faces folder
29 for cl in image_files:
30     # Skip if not image
31     if not cl.lower().endswith(('.png', '.jpg', '.jpeg')):
32         continue
33     # Load image
34     current_image = face_recognition.load_image_file(f'{face_images_path}/{cl}')
35     # Skip if image is empty
36     if current_image is None or current_image.size == 0:
37         print(f"Unable to read image: {cl}")
38         continue
39     # Resize image
40     face_images.append(current_image)
41     # Get name only
42     face_names.append(os.path.splitext(cl)[0])
```

Figure 3 loads the face images

The **find_face_encodings** function (Fig.4) computes the face encodings for these known images. This function takes a set of images and calculates a face code for each image. The facial encoding is a 128-dimensional vector that represents the features of the face. These encodings will be used in subsequent facial recognition tasks. In this function, we traverse the list of input images, convert the images from BGR format to RGB format and then extract the face encodings using the **face_recognition.face_encodings()** function. Finally, these encodings are added to a list and returned.

```
57 # Function that returns a array of encodings for each image.
58 def find_face_encodings(images):
59     face_encodings_list=[]
60     for img in images:
61         # Convert the image from BGR color (which OpenCV uses) to RGB color (which face_recognition uses)
62         img= cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
63         # Get the face encodings for each face in each image file
64         encode=face_recognition.face_encodings(img)[0]
65         # Add face encoding for current image with corresponding label (name) to the training data
66         face_encodings_list.append(encode)
67     # Return the array of face encodings
68     return face_encodings_list

```

```
113 # Get the face encodings for each face in each image file
114 face_encodings_list_known= find_face_encodings(face_images)
115 print("encoding complete!")
```

Figure 4 computes the face encodings for these known images

face_distance_to_conf (Fig.5) function is used to convert the face distance calculated during face recognition to a confidence score. This score represents the level of confidence of the face match. This function takes two parameters: **face_distance** and **face_match_threshold**, with a default threshold of 0.6.


```

44 # Function that convert the face distance calculated during face recognition process (i.e., face_distance) into a confidence score
45 def face_distance_to_conf(face_distance, face_match_threshold=0.6):
46     # Below is a simple linear function that converts the distance to a confidence score
47     if face_distance > face_match_threshold:
48         range = (1.0 - face_match_threshold)
49         linear_val = (1.0 - face_distance) / (range * 2.0)
50         return linear_val
51     # Otherwise we linearly interpolate the confidence to a more logarithmic scale
52     else:
53         range = face_match_threshold
54         linear_val = 1.0 - (face_distance / (range * 2.0))
55         return linear_val + ((1.0 - linear_val) * ((linear_val - 0.5) * 2) ** 0.2)

```

Figure 5 face_distance_to_conf function

The **VideoCapture** class from OpenCV is used to create a **camera** object for capturing the real-time video stream from the webcam. (Fig.6)

```

117 # video capture
118 camera = cv2.VideoCapture(0)

```

Figure 6 create a camera object

8.1.1 Use separate threads for camera reading, model inference, and image display

The **CameraThread**, **ProcessThread**, and **DisplayThread** classes are used for asynchronously processing video frames. (Fig.7) The **CameraThread** is responsible for capturing video frames from the camera, the **ProcessThread** is responsible for processing the captured video frames (including face detection and recognition), and the **DisplayThread** is responsible for displaying the processed video frames.

```

121 class CameraThread(threading.Thread):
122     def __init__(self, camera, frame_queue):
123         threading.Thread.__init__(self)
124         self.camera = camera
125         self.frame_queue = frame_queue
126         self.stop_flag = False
127
128     def run(self):
129         frame_counter = 0
130         while not self.stop_flag:
131             success, img = self.camera.read()
132             if not success:
133                 break
134             self.frame_queue.put(img)
135
136     def stop(self):
137         self.stop_flag = True
138
139 class DisplayThread(threading.Thread):
140     def __init__(self, processed_queue):
141         threading.Thread.__init__(self)
142         self.processed_queue = processed_queue
143         self.stop_flag = False
144
145     def run(self):
146         while not self.stop_flag:
147             img = self.processed_queue.get()
148             cv2.imshow("Webcam", img)
149             if cv2.waitKey(1) == ord('q'):
150                 break
151             camera_thread.stop()
152             process_thread.stop()
153             camera.release()
154             cv2.destroyAllWindows()
155
156     def stop(self):
157         self.stop_flag = True
158
159 class ProcessThread(threading.Thread):
160     def __init__(self, frame_queue, processed_queue):
161         threading.Thread.__init__(self)
162         self.frame_queue = frame_queue
163         self.processed_queue = processed_queue
164         self.stop_flag = False
165
166     def run(self):
167         frame_counter = 0
168         while not self.stop_flag:
169             img = self.frame_queue.get()
170             if frame_counter % 2 == 0: # Only process every other frame
171                 img = process_frame(img)
172                 self.processed_queue.put(img)
173             frame_counter += 1 # Increment the frame counter
174
175     def stop(self):
176         self.stop_flag = True

```

Figure 7 Use separate threads for camera reading, model inference, and image display

8.1.2 Skip some frames

In the **ProcessThread**, we can see that the **Skip some frames** requirement is achieved by counting the frames in the loop using **frame_counter** (Fig.8). This helps to reduce the computational demand while maintaining vstreaming.

```
while not self.stop_flag:
    img = self.frame_queue.get()
    if frame_counter % 2 == 0: # Only process every other frame
        img = process_frame(img)
    self.processed_queue.put(img)
    frame_counter += 1 # Increment the frame counter
```

Figure 8 Skip some frames

The **process_frame** function (Fig.9) is used to detect faces in a video frame and compare them with known face encodings. If a matching face is found, a rectangular box is drawn around the face with the name and match score displayed.

This function is responsible for processing captured camera frames. It takes an image (camera frame) as input and performs the following operations:

- Resizes the image to speed up processing.
- Converts the image from BGR colour space to RGB colour space.
- Detects the position of the face in the image.
- Calculates the feature code for each detected face in the image.
- Compare the feature codes of the faces in the current frame with the known face feature codes to find the best match.
- Draws a rectangular box on the image showing the face name and the confidence level of the match.
- Return the processed image.

```
70 # Detecting faces in a video frame and comparing them with known face encodings
71 def process_frame(img):
72     # Skip if image is empty
73     if img is None or img.size == 0:
74         return img
75     # Resize image
76     resized_frame = cv2.resize(img, (0, 0), None, SCALE_FACTOR, SCALE_FACTOR)
77     # Convert the image from BGR color (which OpenCV uses) to RGB color (which face_recognition uses)
78     resized_frame = cv2.cvtColor(resized_frame, cv2.COLOR_BGR2RGB)
79     current_frame_faces = face_recognition.face_locations(resized_frame)
80     current_frame_encodings = face_recognition.face_encodings(resized_frame, current_frame_faces)
81
82     # Loop over each face found in the frame to see if it's someone we know.
83     for face_encoding, face_coordinates in zip(current_frame_encodings, current_frame_faces):
84         # See if the face is a match for the known face(s)
85         face_matches = face_recognition.compare_faces(face_encodings_list_known, face_encoding)
86         # Use the known face with the smallest distance to the new face
87         face_distances = face_recognition.face_distance(face_encodings_list_known, face_encoding)
88         # Get the best match index
89         best_match_index = np.argmin(face_distances)
90
91         # If there is a match
92         if face_matches[best_match_index]:
93             # Get the name to display on the video
94             matched_name = face_names[best_match_index].upper()
95             # Calculate the match percentage
96             matchPerc = round(face_distance_to_conf(face_distances[best_match_index]) * 100)
97         else:
98             # Set the name to display on the video
99             matched_name = "Unknown"
100             # Set the match percentage to 0
101             matchPerc = 0
102
103     y1, x2, y2, x1 = face_coordinates
104     y1, x2, y2, x1 = y1 * 4, x2 * 4, y2 * 4, x1 * 4
105     cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)
106     cv2.rectangle(img, (x1, y2 - 35), (x2, y2), (0, 255, 0), cv2.FILLED)
107     cv2.putText(img, f'{matched_name} {matchPerc}%', (x1 + 6, y2 - 6), cv2.FONT_HERSHEY_COMPLEX,
108
109     return img
```

Figure 9 process_frame function

8.1.3 Compress the frames (to about 25% of the original size) before feeding them into the models

The compression of the **frame** is also implemented in the **process_frame** function by using the **SCALE_FACTOR** constant, which controls the compression ratio of the frame. (Fig.10)

```
24  # Define some constants
25  SCALE_FACTOR = 0.25
26  UNKNOWN_THRESHOLD = 0.6

75  # Resize image
76  resized_frame = cv2.resize(img, (0, 0), None, SCALE_FACTOR, SCALE_FACTOR)
```

Figure 10 Compress the frames (to about 25% of the original size)

The **gen_frames** function (Fig.11) then generates the processed camera frame sequence for real-time display on the webpage.

This function is used to generate a sequence of processed camera frames for live display on a web page. It performs the following operations:

- Fetches the processed image from the `processed_queue` queue.
- Encodes the image into JPEG format.
- Converts the image to a byte string.
- Generates a sequence of frames and sends it to the client via an HTTP response.

```
208 # gen_frames() function generates frames to be displayed on the webpage
209 def gen_frames():
210     while True:
211         # Capture frame-by-frame
212         img = processed_queue.get()
213         # encode OpenCV raw frame to jpg and displaying it
214         ret, buffer = cv2.imencode('.jpg', img)
215         # convert the image to bytes and yield as output
216         img = buffer.tobytes()
217         #stream video frames one by one
218         yield (b'--frame\r\n'
219               + b'Content-Type: image/jpeg\r\n\r\n' + img + b'\r\n')
```

Figure 11 generates the processed camera frame sequence

Then in **server.py**, the **waitress** library is used to host the Flask application and run the WSGI server. (Fig.12)

```
23 #Get local ip address
24 ip_address = get_local_ip()
25 #Number of threads
26 thread = 12
27 #SocketIO
28 sio = socketio.Server()
29 #Wrap Flask application with SocketIO's middleware
30 app_server = socketio.WSGIApp(sio, app)
31
32 #SocketIO event handler
33 if __name__ == '__main__':
34     print("Server running at: http://" + ip_address + ":8080 or http://localhost:8080,\nThreads: " + str(thread))
35     #Run server
36     serve(app_server, host='0.0.0.0', port=8080, url_scheme='http', threads=thread)
```

Figure 12 run server

Finally, in **stream.py**, the routes are bound by using the **@app.route** decorator, providing web page services to the users. (Fig.13)

```
221 # route for video streaming
222 @app.route('/video_feed')
223 def video_feed():
224     # return the response generated along with the specific media
225     return Response(gen_frames(), mimetype='multipart/x-mixed-replace; boundary=frame')
226
227 # route for home page
228 @app.route('/')
229 def index():
230     #Streaming Page
231     return render_template('index.html')
```

Figure 13 router

The **index.html** file in the **templates** folder defines the structure of the client web page. By using **{{ url_for('video_feed') }}** in the **src** attribute of the **** tag, the video stream can be transmitted from the server to the client in real-time. (Fig.14)

```
1 <body>
2 <head>
3     <meta charset="UTF-8">
4     <title>Kaiyu Chen's DS CA</title>
5 </head>
6 <h1>Live Streaming</h1>
7 <div class="container">
8     <div class="row">
9         <div class="col-lg-8 offset-lg-2">
10             
11         </div>
12     </div>
13 </div>
14 </body>
```

Figure 14 client web page

8.2 Errors

8.2.1 cv2.error

error log:

cv2.error: OpenCV(4.7.0) /Users/runner/work/opencv-python/opencv-python/modules/imgproc/src/color.cpp:182: error: (-215:Assertion failed) !_src.empty() in function 'cvtColor'

solution:

Skip the wrong file and empty images. (Fig.15)



```
28 # Get all the images in the faces folder
29 for cl in image_files:
30     # Skip if not image
31     if not cl.lower().endswith(('.png', '.jpg', '.jpeg')):
32         continue
33     # Load image
34     current_image = face_recognition.load_image_file(f'{face_images_path}/{cl}')
35     # Skip if image is empty
36     if current_image is None or current_image.size == 0:
37         print(f"Unable to read image: {cl}")
38         continue
39     # Resize image
40     face_images.append(current_image)
41     # Get name only
42     face_names.append(os.path.splitext(cl)[0])
```

Figure 15 Skip the wrong file and empty images

8.2.2 socket.gethostbyname(hostname) error

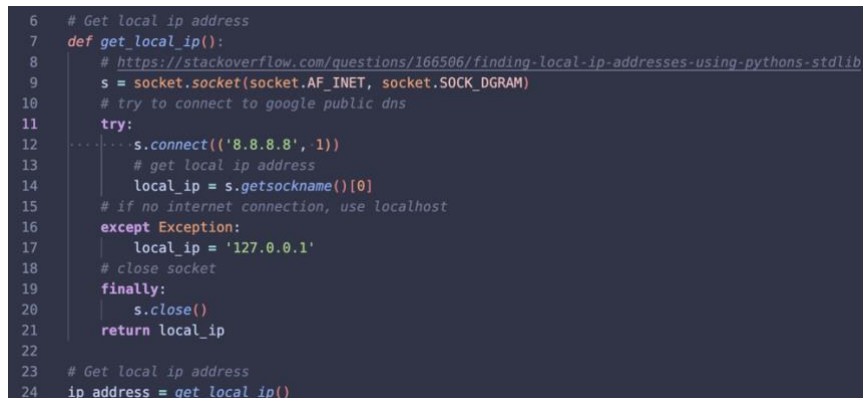
error log:

socket.gaierror: [Errno 8] nodename nor servname provided, or not known

This error is mainly caused by the attempt to obtain the local IP address of the computer through `socket.gethostbyname(hostname)`. Specifically, there may be problems with resolving the hostname. This issue may be caused by DNS resolution issues or firewall settings in the operating system (for example, macOS may have measures that prevent me from obtaining the local IP address). The solution is to try another method to obtain the local IP address.

solution:

Get the local IP address by using Google Public DNS. [4] (Fig.16)



```
6 # Get local ip address
7 def get_local_ip():
8     # https://stackoverflow.com/questions/166506/finding-local-ip-addresses-using-pythons-stdlib
9     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
10    # try to connect to google public dns
11    try:
12        s.connect(('8.8.8.8', 1))
13        # get local ip address
14        local_ip = s.getsockname()[0]
15        # if no internet connection, use localhost
16    except Exception:
17        local_ip = '127.0.0.1'
18    # close socket
19    finally:
20        s.close()
21    return local_ip
22
23 # Get local ip address
24 ip_address = get_local_ip()
```

Figure 16 Get the local IP address by using Google Public DNS

9. Conclusion

This report summarises and analyses my Facial Recognition Server application which recognises faces of known people from live video streams and displays the recognition results. The application was built using the Flask framework and SocketIO library and written in Python. The file structure of the project contains face images, Python scripts, web templates and the required dependency libraries.

The facial recognition functionality is mainly implemented through the **process_frame** function in the **stream.py** file. This function utilizes the **face_recognition** library to detect faces in input images and compares each detected face with the facial features of known individuals. If a match is found, a rectangular box is drawn on the original image, with the matched individual's name and matching percentage displayed below the box.

In the **server.py** file, a web server is launched using the **waitress** library, which associates the generated real-time video stream with the Flask application. Meanwhile, the **index.html** file provides users with a web interface for displaying the live video stream.

The project's dependencies are listed in the **requirements.txt** file and can be installed by running **pip install -r requirements.txt**. To enhance system performance, the project employs multithreading for processing real-time video streams, as implemented in the **CameraThread**, **ProcessThread**, and **DisplayThread** classes in the **stream.py** file.

Within the project, the **faces** folder stores face images of known individuals, which are used to recognize individuals in the video stream. New facial images can be added to this folder, allowing the application to recognize more individuals. Currently, the project includes facial images of four known individuals.

In conclusion, this project implements a real-time facial recognition application capable of identifying and labeling known individuals' faces. The web application, built using the Flask framework and SocketIO library, offers a user-friendly interface, while multithreading technology ensures smooth processing of the real-time video stream and enhances performance.

10. Bibliography

- 1] [B. Hadzhiev, "Could not find version that satisfies requirement cv2 OpenCV | bobbyhadz," [Online]. Available: <https://bobbyhadz.com/blog/python-could-not-find-version-that-satisfies-requirement-cv2-opencv#:~:text=The%20error%20%22Could%20not%20find,not%20supported%20by%20the%20package..>
- 2] [ti7, "• python - Installing socketio module on python3 seems to be corrupting pip - Stack Overflow," 26 Feb 2021. [Online]. Available: <https://stackoverflow.com/questions/63385158/installing-socketio-module-on-python3-seems-to-be-corrupting-pip>.
- [BigchainDB, "How to Install the Latest pip and setuptools - BigchainDB 0.6.0

-
- 3] documentation," [Online]. Available:
<https://bigchaindb.readthedocs.io/projects/server/en/v0.6.0/appendices/install-latest-pip.html>.
- 4] [UnkwnTech, "networking - Finding local IP addresses using Python's stdlib - Stack Overflow," 3 Oct 2008. [Online]. Available:
<https://stackoverflow.com/questions/166506/finding-local-ip-addresses-using-pythons-stdlib>.

11. Appendix

GitHub: [cky008/facialRecognitionServer: Live Streaming Service - Facial Recognition Server](https://github.com/cky008/facialRecognitionServer)
(<https://github.com/cky008/facialRecognitionServer>)