EE3625/40 Video Streaming and Processing – Assignment – Jedrzej Fratczak (2207175)

Contents

Introduction

This report will present the development of a streaming video system between two RPis and between an RPi and a PC. The report will firstly go over the design stage of such systems, following by the development of the program code using python. The report will then showcase testing of both systems and go over the obstacles and how they were overcome.

System Description and Diagram

System Diagram

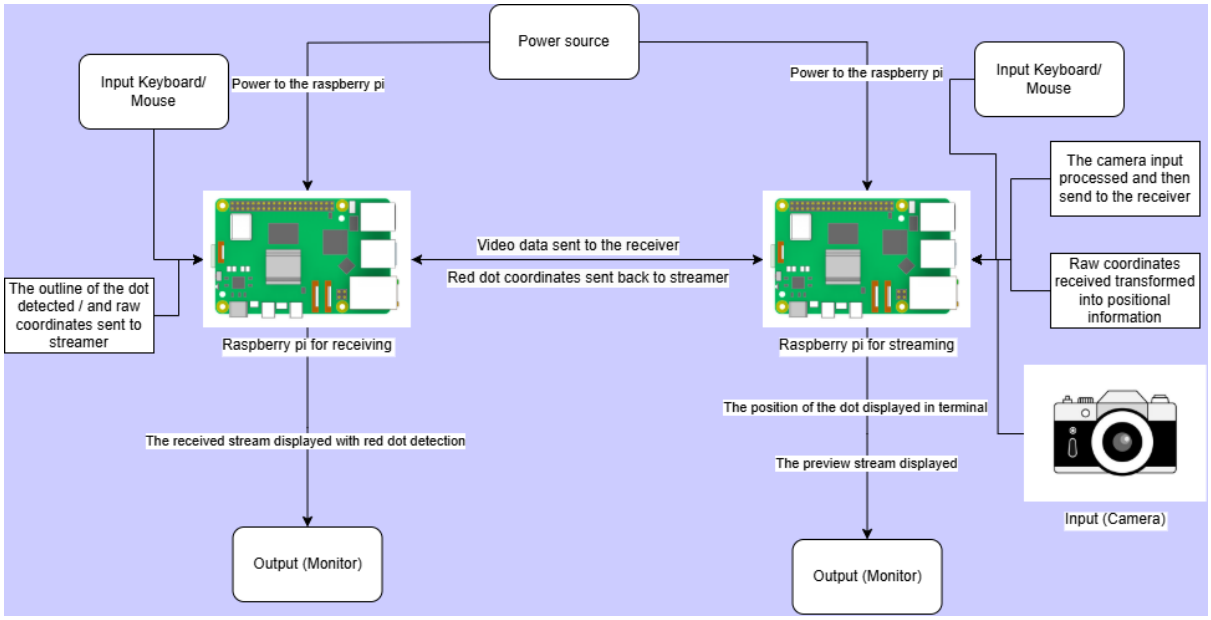

Figure 1 - Raspberry pi to Raspberry pi streaming

Figure 1 showcases the system diagram for RPi to RPi streaming (for pc replace the receiver RPi with a receiver PC). As seen in the diagram there are multiple input/output devices connected to each RPi as well as a power source. It can also be seen that for the streaming RPi there is a camera connected. For each pi we can also see the processes that take place to successfully achieve the requirements. Between the two RPis there can also be seen the data transferred and received by both.

Algorithm (Sequence of operation)

Below will be the sequence of operations that take place to achieve streaming between two RPis and between an RPi and a PC. This sequence of operations is after the devices are setup and each peripheral/component is connected and working.

1. Both RPis are setup for streaming and receiving data, this is done by creating a new UDP socket for each pi. Because there are two RPis and both are streaming and receiving data there will be four sockets total used.
2. The streamer is set up to get the input from the camera and send it over to the receiver.
3. The receiver is set up to detect if a red object appears on the stream and store the outline of that object into a “contours” variable.
4. This is then drawn onto the receiver feed to visualise any contours that may be detected.
5. The contours are the send back to the streamer.
6. The streamer processes the contours to display the position of the detected object relative to the centre of the frame and prints it in the terminal.

Hardware specifications/description

Is the hardware described? 4%

The hardware used to accomplish the RPi-to-RPi solution are:

- Raspberry Pi 4 – two of them one for streaming one for receiving
- Basic keyboard and mouse for input
- The camera used is a Logitech C270 HD webcam
- The RPis are connected to each other through Wi-Fi

For the RPi-to-PC solution the streamer side stays the same and for the receiver the “MacBook Air” is used.

Operating System & Python Software Description

Is the operating system and python software described? 4%

The Raspberry Pis use the Raspbian GNU/Linux 10 (buster) os, while the PC uses the “macOS Ventura 13.7.1”. Multiple libraries were used to ensure a complete, functional solution.

- OpenCV library was used for capturing the video from the webcam by the streamer and for processing the specific frames and detecting the red dot by the receiver.
- The socket library was used to facilitate the UDP to allow the RPis to send and receive data.
- NumPy library was used to ensure efficient array manipulations.
- Pickle library was used to help with serialising and deserializing data in order to transmit it over the network.
- Finally, the os library was used to help with handling processes.

These libraries were used to ensure that firstly the video footage from the camera is properly captured by the streamer. That footage is then sent to the receiver using UDP. After the receiver gets the footage the red dot detection process starts. When the receiver detects the contours of the dot, the X and Y coordinates are then transferred across the network back to the streamer. Finally with the coordinates the streamer can then calculate the position of the dot relative to the centre of the frame and display it in the terminal.

Output Presentation for RPi-to-RPi solution

Is the output presented showing the requested results on RPi and RPi solution? 4%

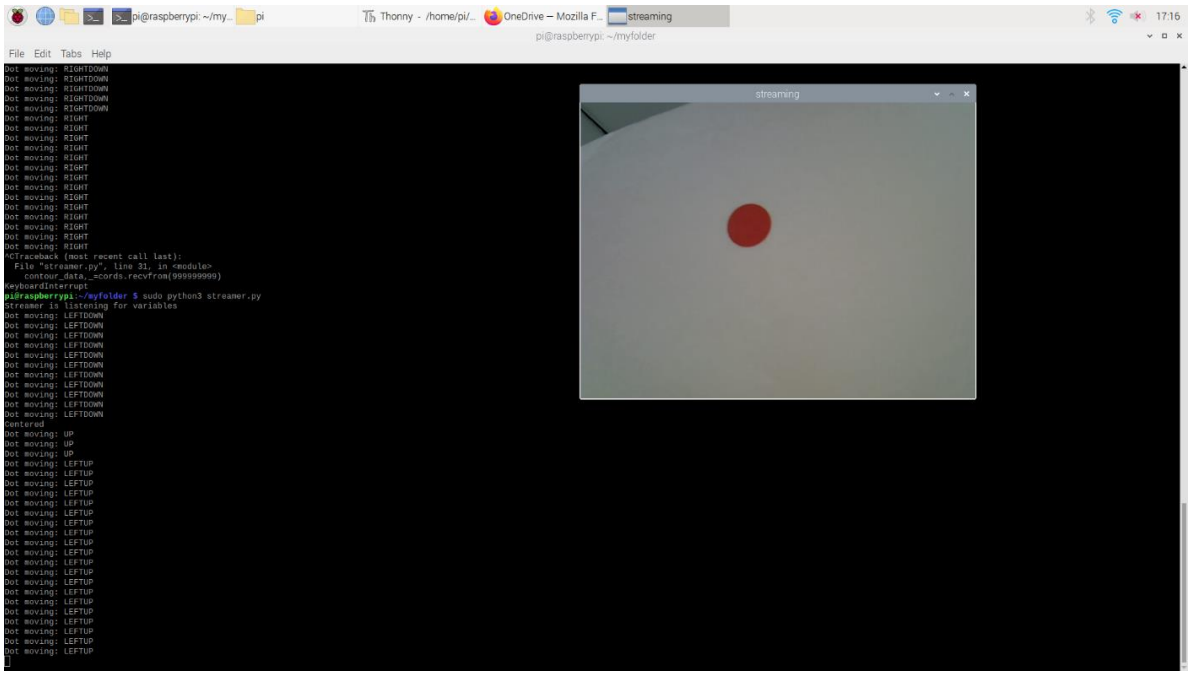

Figure 2 - Streamer output 1

Figure 2 showcase the streamer output for the RPi-to-RPi solution. As seen in the image the streamer feed can be seen as well as the position of the dot relative to the centre of the frame is printed.

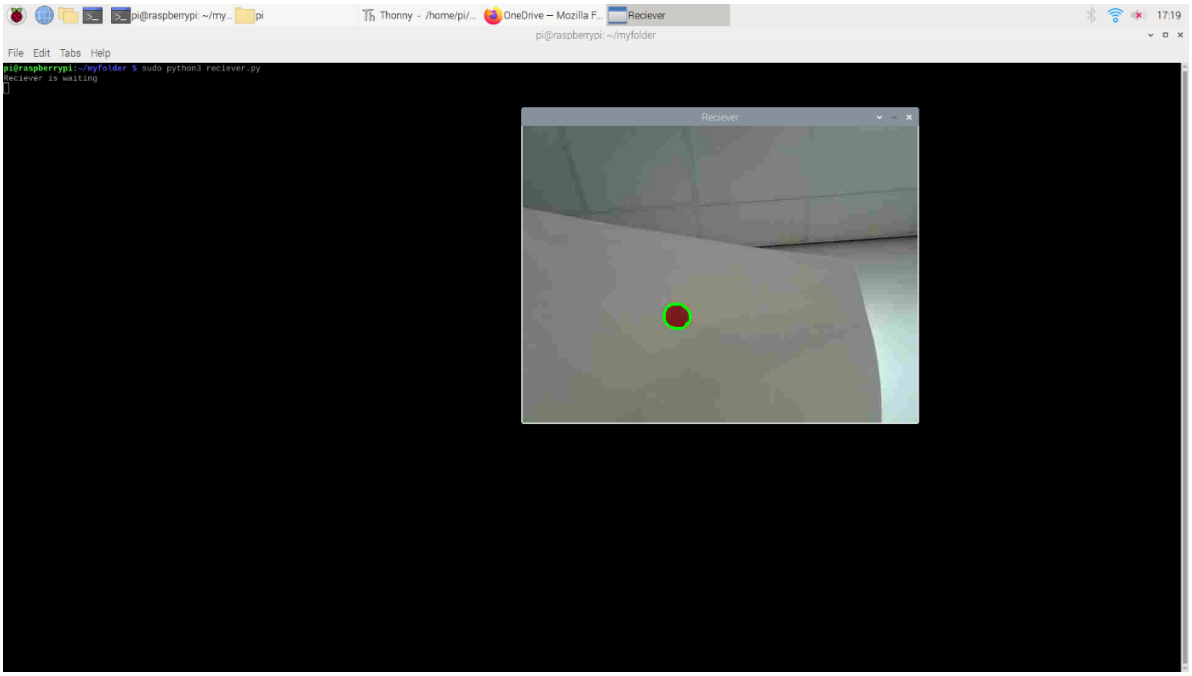

Figure 3 - Receiver output 1

Figure 3 showcases the receiver output for the RPi-to-RPi solution, in the image the outline of the red object can be seen highlighted in green on the receiver feed.

Output Presentation for RPi to PC solution

Is the output presented showing the requested results on RPi and PC solution?

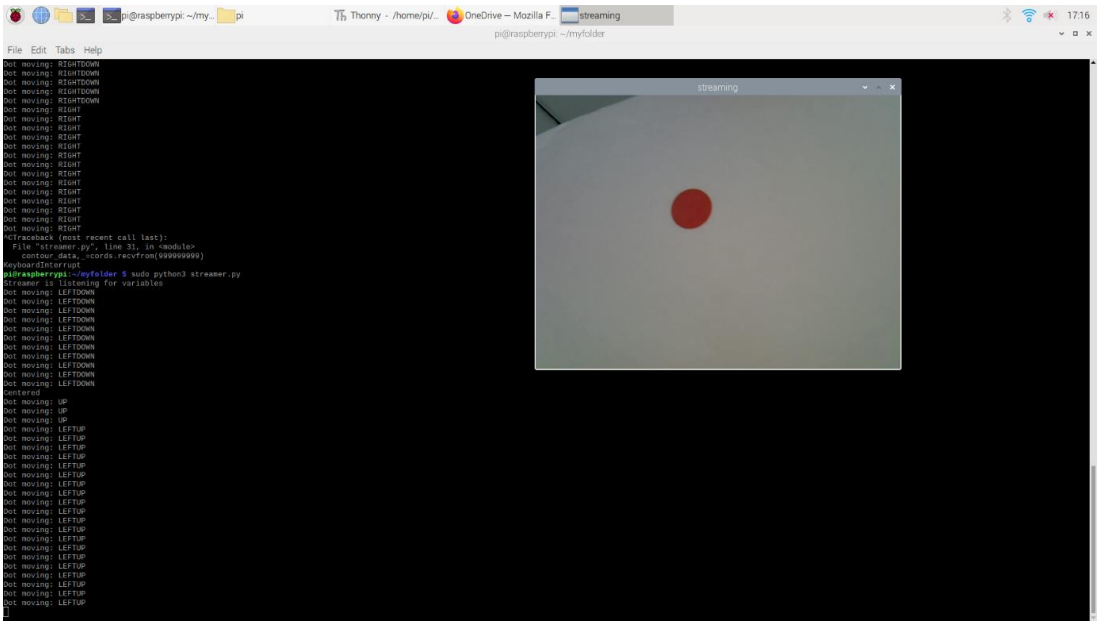

Figure 4 - Streamer output 2

Figure 4 presents the output of the streamer for the RPi-to-PC solution. As we can see the output is identical to the previous streamer as nothing changes for the streamer between the two different solutions.

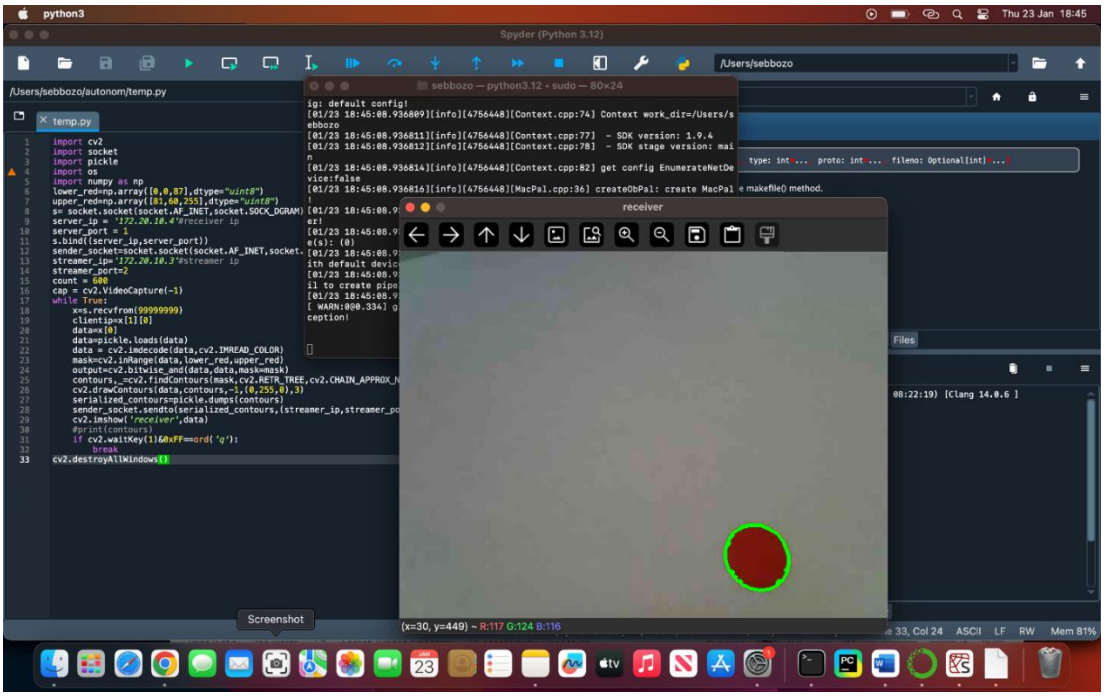

Figure 5 - Receiver output 2

Figure 5 showcases the receiver output for the RPi-to-PC solution. Similarly to the previous solution we can see the outline of the red object highlighted in green onto the receiver feed.

Description of Systems Hardware and Software

Flowcharts

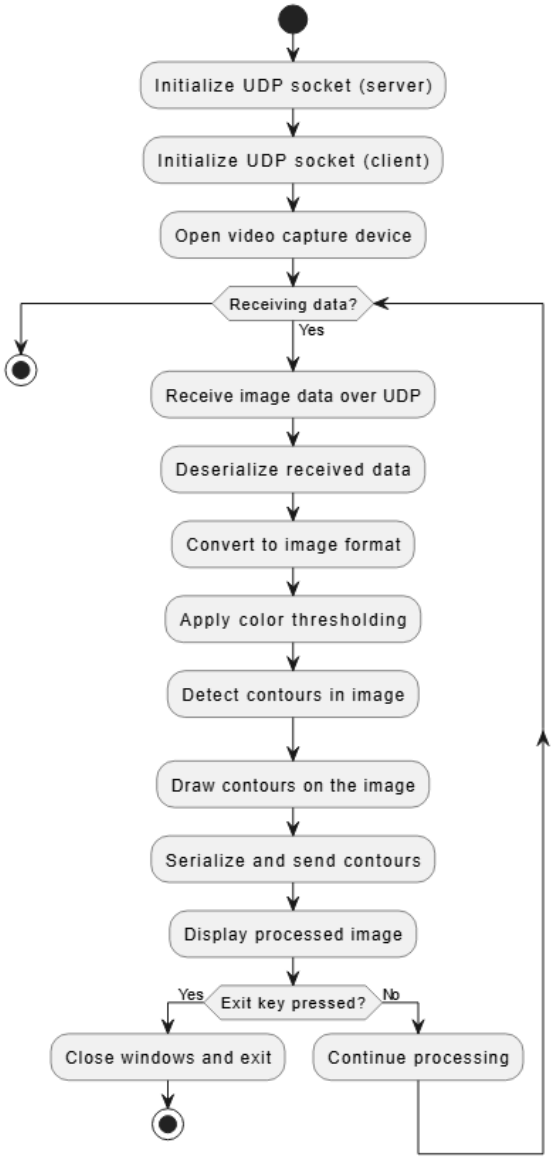

Figure 6 - Flowchart for receiver

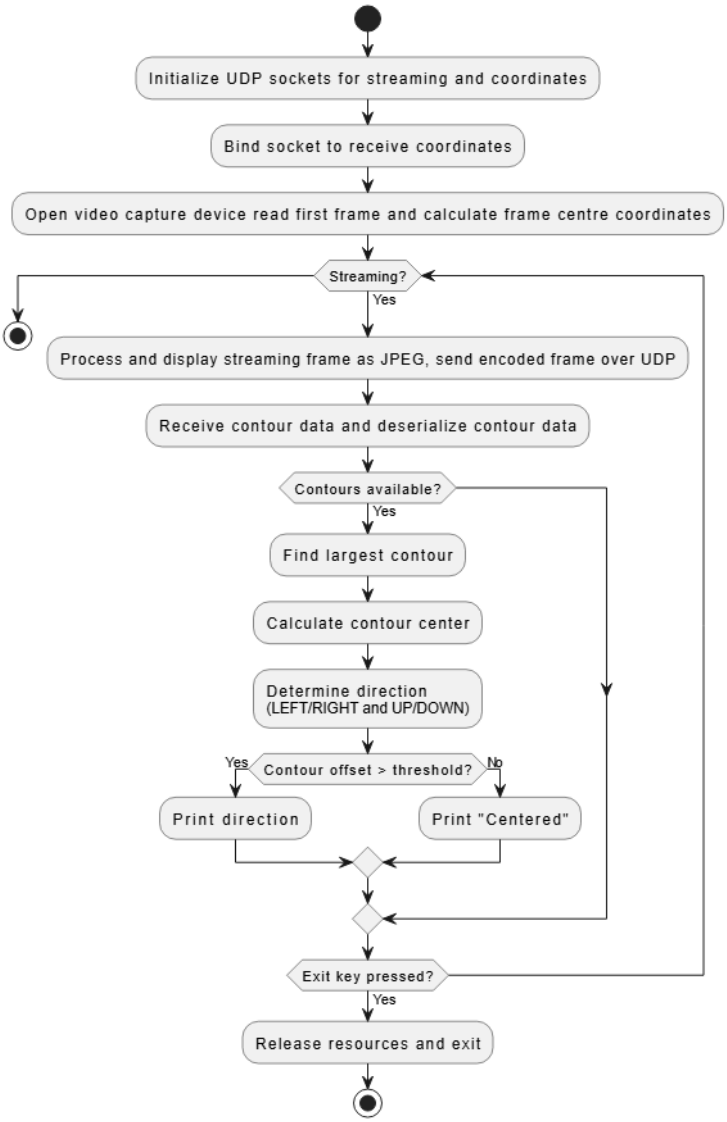

Figure 7 - Flowchart for streamer

In figure 6 and 7 the flowcharts for the receiver and streamer can be seen. From the receiver flowchart it can be seen that firstly the sockets are assigned for streaming and receiving data, afterwards the video capture device is open, and an infinite loop is started based on if the data from the streamer is received. Within the loop the video is converted to images which are then analysed for any red content. If the colour red is detected, then it is sent to the streamer. For the streamer the start is similar with the assignment of UDP sockets, afterwards the loop is started based on video being received. Within the loop the video is processes and transmitted to the receiver. Afterwards with the contours received the streamer calculates the position of the red object relative to the frame and display it in the terminal.

Sequence Diagrams

Does it show a sequence diagram of software 5%

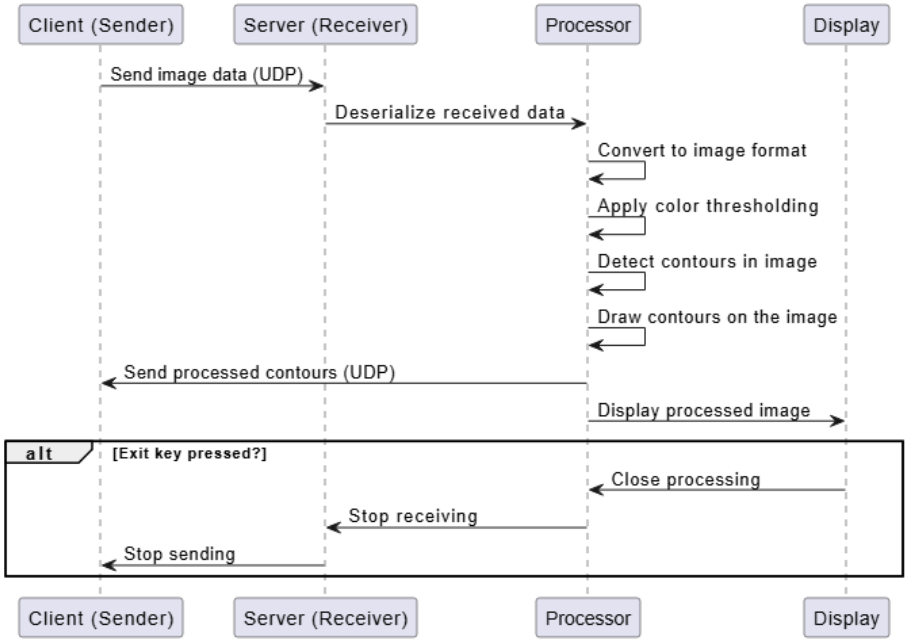

Figure 8 - Sequence Diagram for receiver

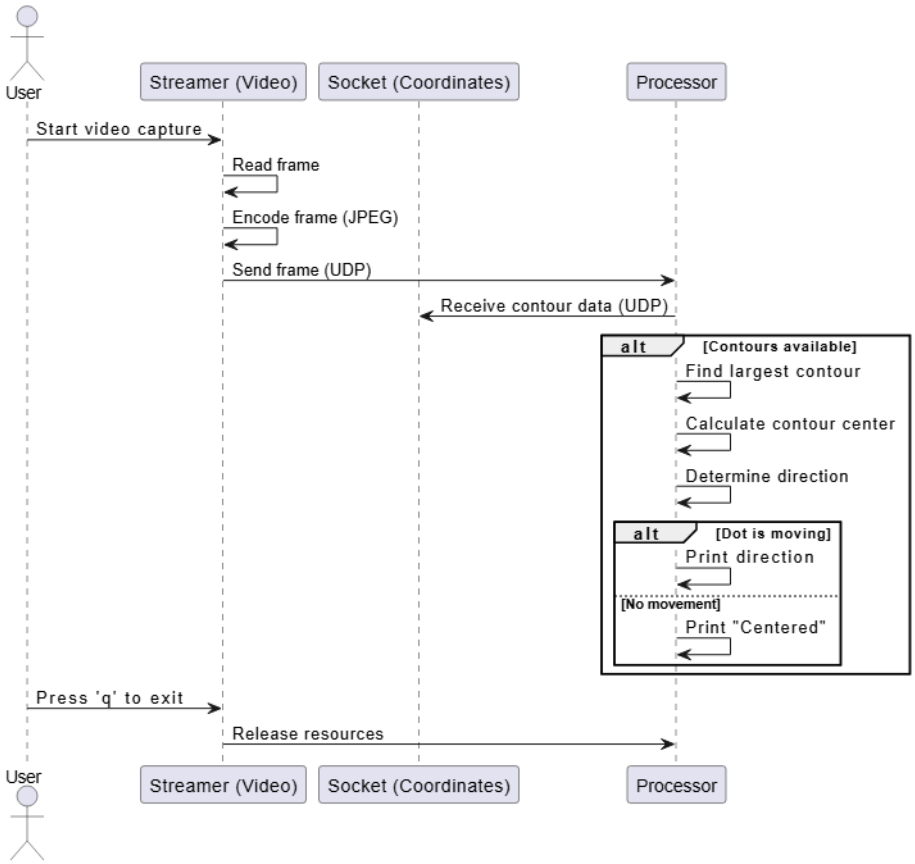

Figure 9 - Sequence diagram for streamer

Figure 8 and 9 showcases the sequence diagrams for the receiver and streamer respectively.

Code analysis

Receiver Code

```
import cv2
import socket
import pickle
import os
import numpy as np

lower_red = np.array([0,0,87],dtype="uint8")
upper_red = np.array([81,60,255],dtype="uint8")


s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
server_ip='172.20.10.3' #Its own ip for recieving video
server_port = 30 #Port for recieving video
s.bind((server_ip,server_port))

cords = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
server_ip2 = '172.20.10.6' #Streamer ip for sending coordinates
server_port2 = 6969 #Port for streaming coordinates

count=600
cap = cv2.VideoCapture(-1)
print("Reciever is waiting")

while True:
    x = s.recvfrom(999999999)
    clientip = x[1][0]
    data=x[0]
    data = pickle.loads(data)
    data = cv2.imdecode(data,cv2.IMREAD_COLOR)
    mask = cv2.inRange(data,lower_red,upper_red)
    output = cv2.bitwise_and(data,data,mask=mask)
    _,contours,_=cv2.findContours(mask,cv2.RETR_TREE,cv2.CHAIN_APPROX_NONE)
    cv2.drawContours(data,contours,-1,(0,255,0),3)
    serialized_contours=pickle.dumps(contours)
    cords.sendto(serialized_contours,(server_ip2,server_port2))
    cv2.imshow('Reciever',data)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the camera and close all windows
vc.release()
cv2.destroyAllWindows()
```

Figure 10 - Code for the receiver

Firstly, five different modules are imported, “OpenCv” for capturing the video from the streamer, “socket” to create channels for the data to be transferred, “pickle” for loading the data received and “numpy” and “os” for crating the red dot detection arrays and managing the code respectively. After the modules are imported two numpy arrays are created for the red colour detection. These arrays “lower_red” and “upper_red” specify the range of red colour that will be detected in RGB colour space. Following the arrays being set up, two UDP sockets are created to facilitate the communication between the receiver and the streamer. The first socket called “s” is set up for receiving incoming data while the 2nd socket “cords” is set up for sending data. Afterwards the video capture is set up using the “cap=cv2.VideoCapture{-1}” line. Then an infinite loop is entered, in the loop the variable “x” receives a 1mb packet from the UDP socket s. Then that data is deserialised and received using the “pickle.loads()” line. The data is then decoded. Finally, after the data is decoded a red colour mask is created using the “lower_red” and “upper_red” bounds. Lastly the red regions are extracted from the frame using the “cv2.bitwise_and()” function. From the mask created previously the contours are extracted using the “cv2.findContours()” function. Then the extracted contour are drawn onto the original frame and serialised using the “pickle.dumps()” function. Then the serialised contours are sent to the specific “streamer_ip” and “streamer_port” over the “cords” UDP socket. Finally, the processed image is displayed in the receiver window. Then a function to stop the program is implemented when the button “q” is pressed.

Streamer Code

```python
import cv2
import socket
import numpy as numpy
import pickle5 as pickle
import os

s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_SNDBUF,1000000)
server_ip='172.20.10.2' #Reciever ip (for streaming)
server_port=28 #Streaming video port
#Count=600
cap=cv2.VideoCapture(-1)

cords=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
server_ip2 = '172.20.10.3' #Its own ip for recieving coordinates
server_port2 = 30 #Coordinate reciving port
cords.bind((server_ip2,server_port2))

print("Streamer is listening for variables")

ret,frame= cap.read()
frame_centre_x = frame.shape[1]//2
frame_centre_y = frame.shape[0]//2

while True:
    ret,buffer = cap.read()
    cv2.imshow('streaming',buffer)
    ret,buffer = cv2.imencode('.jpg',buffer, [int(cv2.IMWRITE_JPEG_QUALITY),30])
    x_as_bytes = pickle.dumps(buffer)
    s.sendto((x_as_bytes),(server_ip,server_port))
    contour_data,_=cords.recvfrom(999999999)
    contours=pickle.loads(contour_data)
    if contours:
            largest_contour = max(contours, key=cv2.contourArea)
            x,y,w,h = cv2.boundingRect(largest_contour)
            dot_centre_x = x + w // 2
            dot_centre_y = y + h // 2
            direction = ''
            if dot_centre_x < frame_centre_x -20:
                    direction += "LEFT"
            elif dot_centre_x > frame_centre_x + 20:
                    direction += "RIGHT"

            if dot_centre_y < frame_centre_y -20:
                    direction += "UP"
            elif dot_centre_y > frame_centre_y + 20:
                    direction += "DOWN"

            if direction:
                    print(f"Dot moving: {direction.strip()}")
            else:
                    print("Centered")

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cv2.destroyAllWindows()
cap.release()
```

Figure 11 - Code for the streamer

The imports for the streamer code are identical to the receiver and serve similar purposes in both codes. Then two UDP sockets are created for the data to be send and received. One socket is used for sending 1mb packets of video while the other is used to receive the coordinates. Following the initialisation of the socket a frame is captured from the video using the “cap.read()” function, the image is stored in the variable called “frame”. With this frame the centre point can be calculated by dividing the height and width by 2. After the frame centre is established the program while enter an infinite loop. In this loop the frame of a video is constantly stored in the “buffer”, then that frame is displayed on the streamer monitor as a preview. The frame is then compressed into a JPEG format using the “cv2.imencode” function, this helps reduce the size for better transmission. Finally, the frame is serialised using “pickle.dumps()” and sent over to the specific ip and port using the UDP socket established previously. After the video is streamed the program can get ready for receiving coordinates of any red object. The coordinates received will come from the “cords” socket using the “recvfrom()” function. Then the data received will be deserialised using the “pickles.loads()”. Then if a contour is detected the largest one is identified using the “max()” function. With the largest contour identified the bounding box coordinates are found to determine the centre of the detected object. Finally, the object centre is compared to the centre of the frame and if it deviates by more than 20 pixels appropriate movement direction will be added into the “direction” string. This string is then printed to the console for the user to see. Lastly, similarly to the receiver there is a function to stop the infinite loop by pressing “q”.

Wireshark Performance Results

Wireshark output

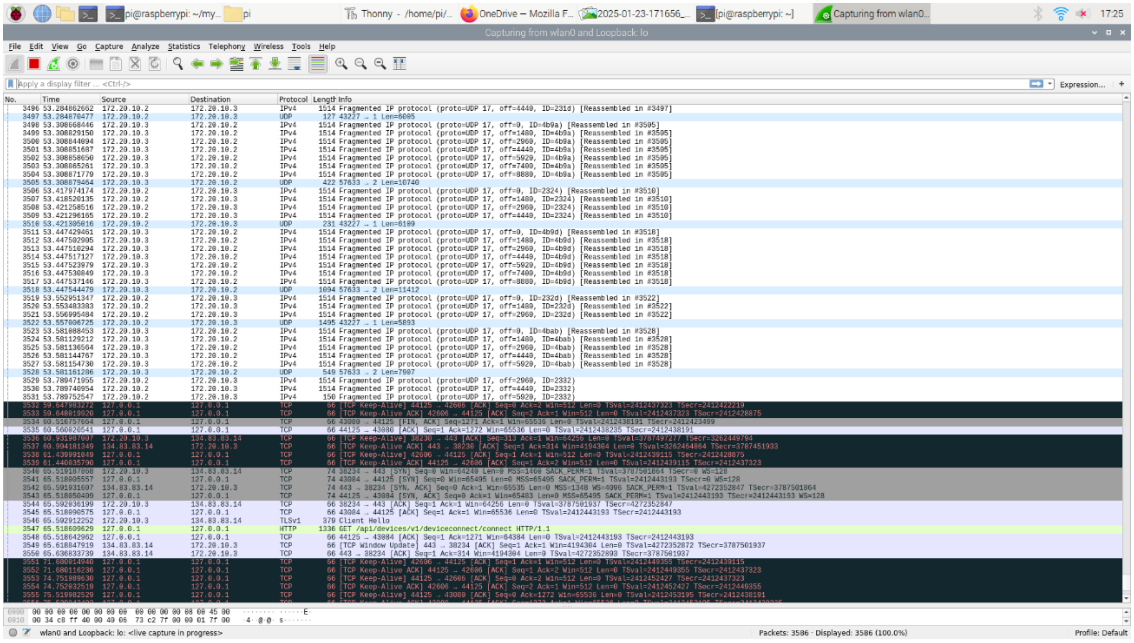

Figure 12 - Wireshark output with no filter

Figure 12 showcases the Wireshark output with no filters, from this it can be seen that there is a lot of traffic coming in and out, below in figure 13 the output will be filtered to only show the UDP traffic which will be the traffic between the streamer and the receiver.

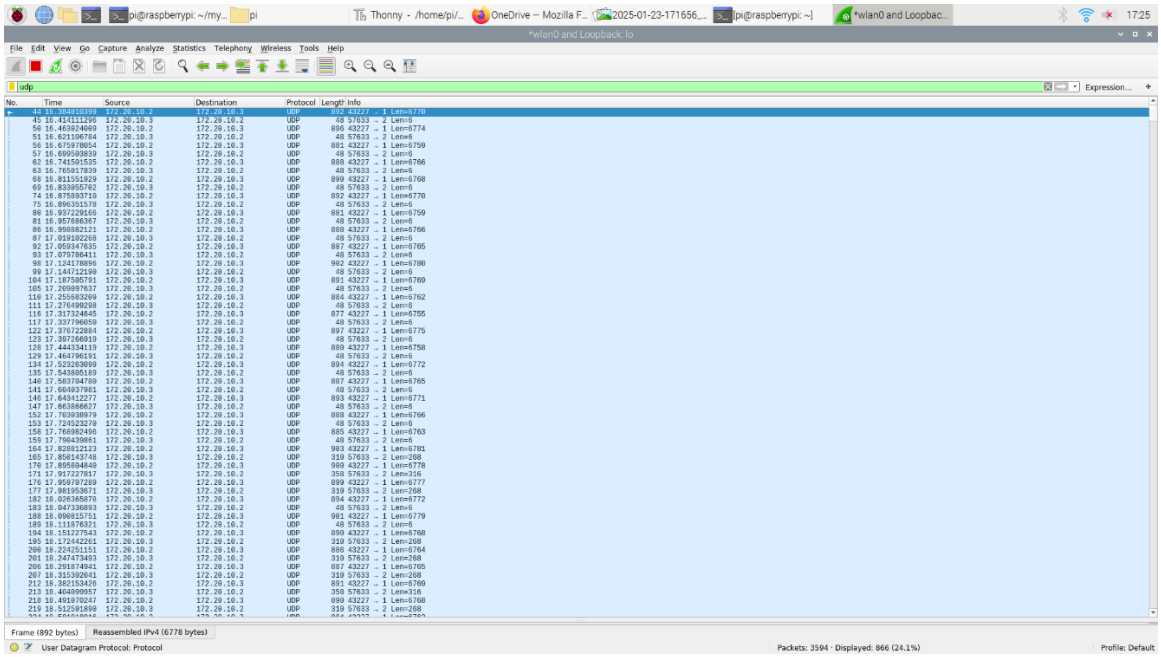

Figure 13 - Wireshark output with UDP filter

As seen in figure 13 when filtered for UDP all the transmissions between the receiver and the streamer can be seen.

Packet Size Analysis

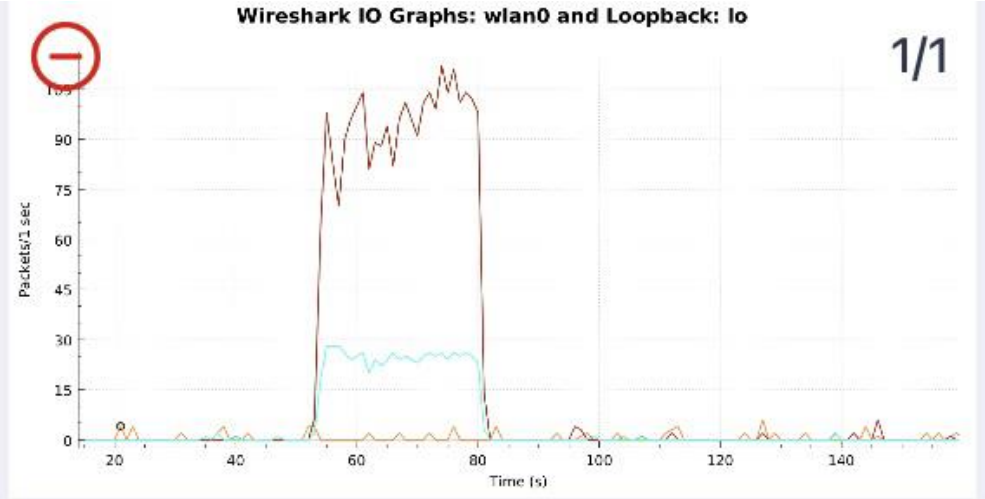

Figure 14 - packets per second graph

Figure 14 showcases the packet per second graph. The packets/s for the video transmission can be seen in the orange colour, meanwhile the packets/s for coordinates transmission can be seen in the blue. From comparing these two lines it can be seen that there is a higher number of packets per second needed for video transmission compared to the coordinate transmission.

Source	Destination	Protocol	Length
172.20.10.2	172.20.10.3	UDP	892
172.20.10.3	172.20.10.2	UDP	48
172.20.10.2	172.20.10.3	UDP	896
172.20.10.3	172.20.10.2	UDP	48

Figure 15 - UDP packet size comparison

Figure 15 showcases the packet length for the UDP sockets. It can be seen that from the streamer with ip “172.20.10.2” to the receiver under the ip “172.20.10.3” the packet length is 892, however when switching the source and destination the packet length is only 48. This is as the first packet sends over the video frame compared to the second packet which is sending the coordinates back hence higher length is needed for the first packet.

Source	Destination	Protocol	Length
13.107.138.10	172.20.10.3	TCP	66
172.20.10.3	13.107.138.10	TLSv1.2	105
172.20.10.3	13.107.138.10	TLSv1.2	90
172.20.10.3	13.107.138.10	TCP	66

Figure 16 - TCP packet size

Figure 16 showcases the length of the TCP packet. This packet is between the user’s machine and other internet services, as seen from the image the packet length for TCP is consistently 66. Therefore, it can be concluded that for video streaming the UDP packets have higher length on average compared to TCP however for coordinated streaming TCP has higher length on average compared to UDP.

Packet Delay Analysis

Figure 17 - Delay analysis

Figure 17 showcases the attempt to find the delay between the steamer and the receiver. However, after zooming in with the time interval of 1 sec no delay could be found between the two. This can be observed as the lines are overlapping no matter how zoomed in the graph would be. From this it can be deduced that there is not enough delay to appear on the graph. To see the delay the graph would need to be more precises however software limitations do not allow for that to be possible. Hence for the purpose of the assignment it can be assumed that the delay can be ignored.

Jitter Performance Analysis

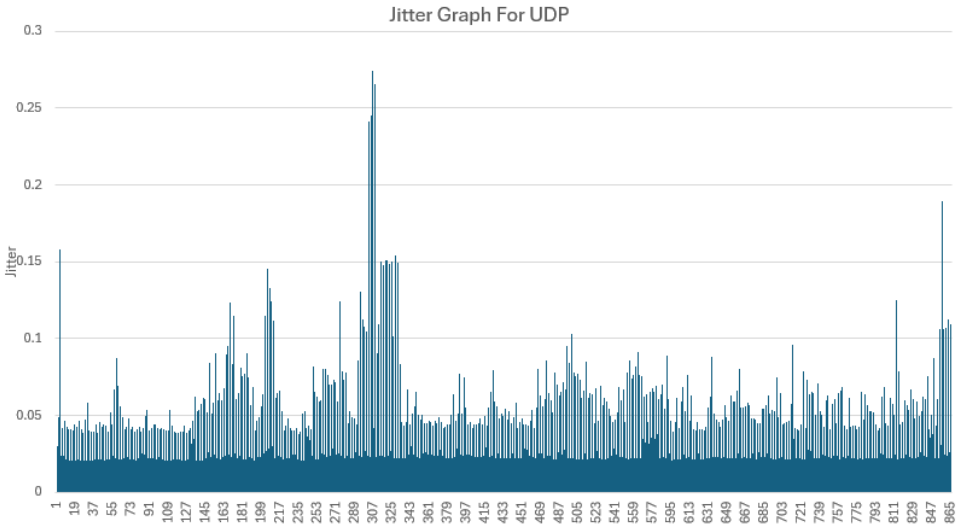

Figure 18 - Jitter graph for UDP made in excel

Figure 18 showcases the jitter graph made in excel for the UDP packets. From the figure it can be seen that for most of the time the jitter is lower than 0.1 Ms. However, on some packets it spikes to even above 0.25. This may be due the instability of the network as the hotspot was used or due to interference as the tests were done in the library with a lot of people present. Overall, we can say that the jitter for the UDP packets performed well within range.

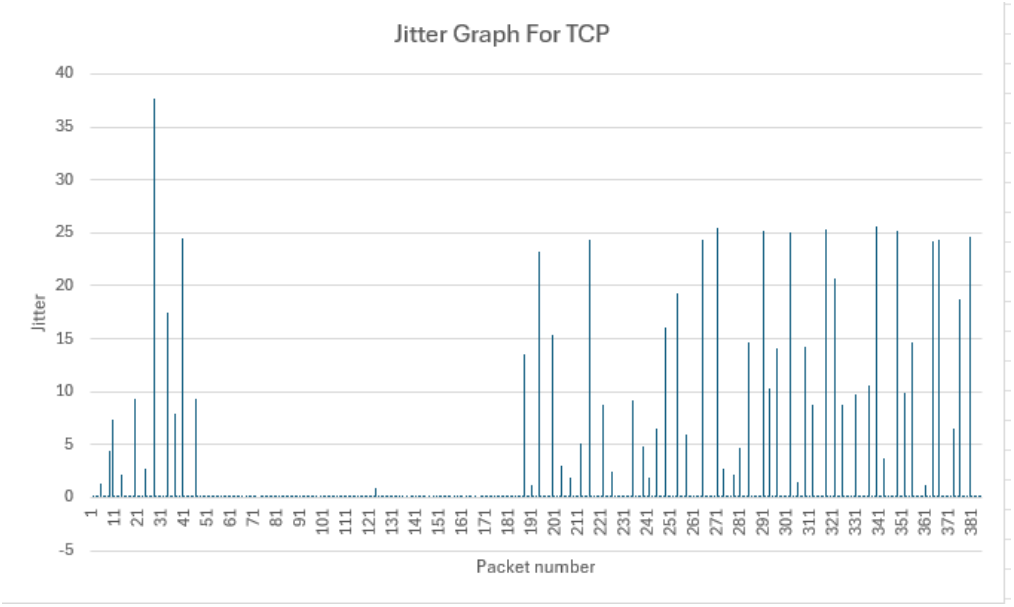

Figure 19 - Jitter graph for TCP created using excel

Figure 19 showcases the jitter graph for all TCP packets made using excel. It can be seen that on average the jitter is significantly larger compared to the UDP packets. It can also be seen that some results are missing as multiple TCP packets errored and the jitter was not registered. Overall, it can be seen that the average jitter for a TCP packet is about 15 Ms while the maximum can go above 35 Ms. This shows that TCP packets are more laggy compared to UDP packets and therefore not a good method to transfer over sensitive data such as video.

Video of working Systems

Video For RPi-to-RPi solution

Video included in the wiseflow appendix.

Video For RPi-to-PC solution

Video included in the wiseflow appendix.