Capstone: Autonomous Runway Detection for IoT

Student: Denis Baptista Rosas
Conclusion: December 2020

Contents

Project Overview

In this project we integrate a complete IoT system for assistance in autonomous landing of aircraft. We build a system capable of image filtering in real-time to spot the runway. This processed image is sent over a secure channel to a ground-based cloud system for further analysis by a high-performance backend to approve the safety or efficiency of the air craft positioning during the landing. In this project, the students are given various parts of program code, and the task is to integrate this code to form a working system.

Figure 1 Overview of the project shows the overview of the system with data in form of picture frames to be analyzed by the edge detection algorithm to spot the runway. The edge detection algorithm is being executed inside a FreeRTOS system on-aircraft, and provides real-time image frames of the environment. In this Project we provide these pictures in form of BMP images, while in a real-world scenario this would be provided by a camera connected to the real-time platform.

Once the picture frame is filtered, the frame matrix is sent to an encryption algorithm (also on-aircraft). This step provides the security for the frame to be transmitted to the ground cloud system. The secure frame is then forwarded to a TCP client, which transmits the frame to the cloud server. The cloud server interface is provided to the students in form of an IP address and its associated port number. The student must ensure that the filtered frame is successfully transmitted to the cloud server and verify that the un-encrypted frame is identical to the frame before encryption.

Since this capstone is completely based on free software and requires no additional hardware (other than a PC), the FreeRTOS real-time OS is run in a PC simulator. This environment is identical to a real-platform solution, but timing measurements cannot be completely accurate since the real-time OS is executed in a host OS as a pthread.

With this in mind, use the FreeRTOS timing measurement facilities to plan what would be needed if such a system was built in real hardware. Provide timings for:

- The edge detection algorithm with all provided BMP images
- The encryption algorithm for all the filtered frames
- The TCP connection to the cloud server

Additionally, the students shall investigate the timing requirements for transmitting the encrypted frames using a TCP connection via a wireless protocol of your choice to the cloud server. The student shall also provide a report for the confidence interval of TCP/IP transmission itself.

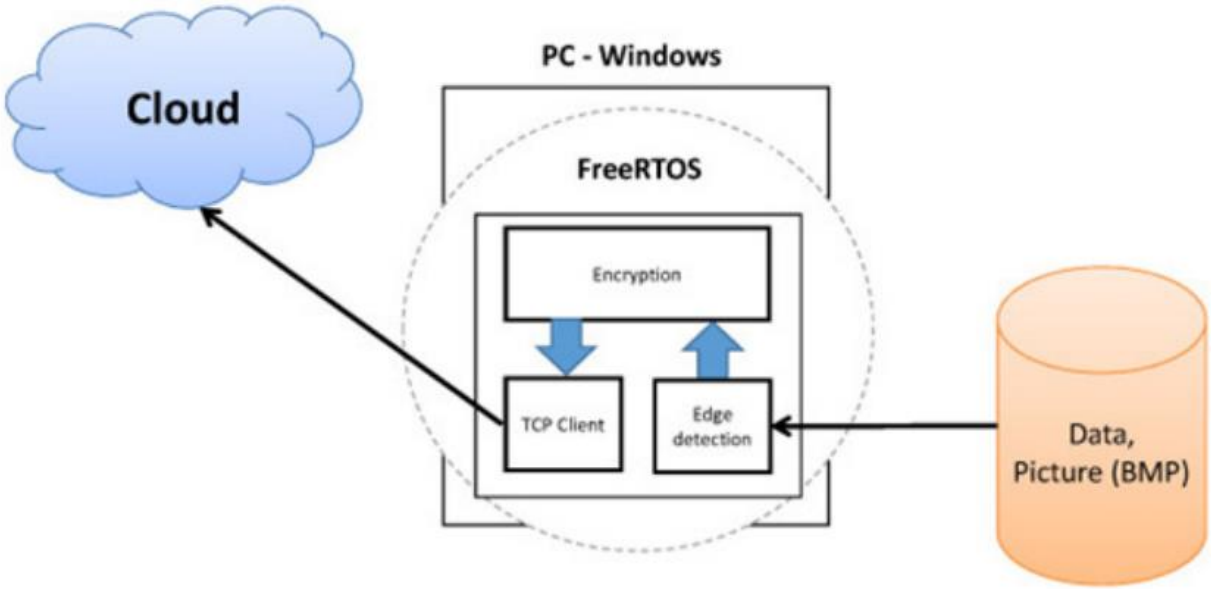


Figure 1 - Overview of the project

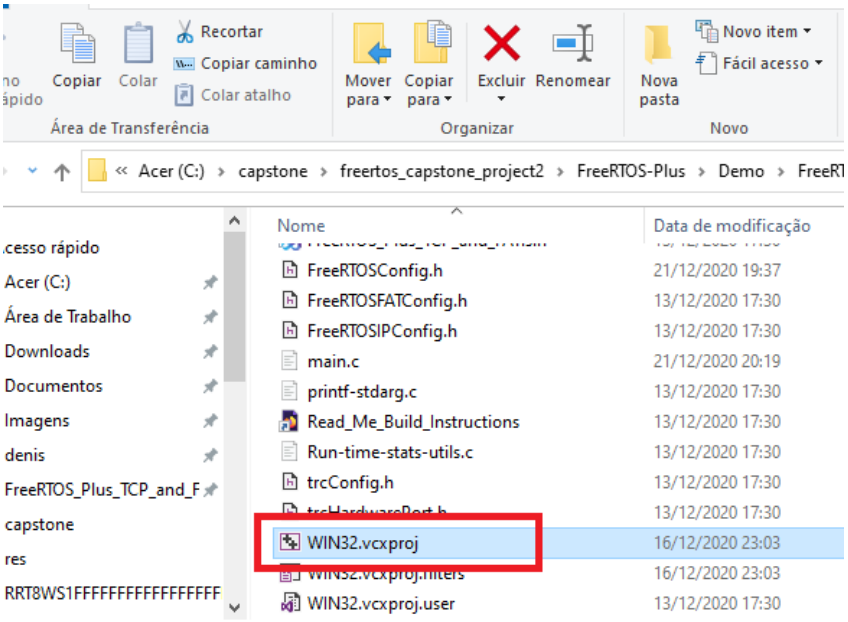
Instructions – How to Run the project

Pre-requirements:

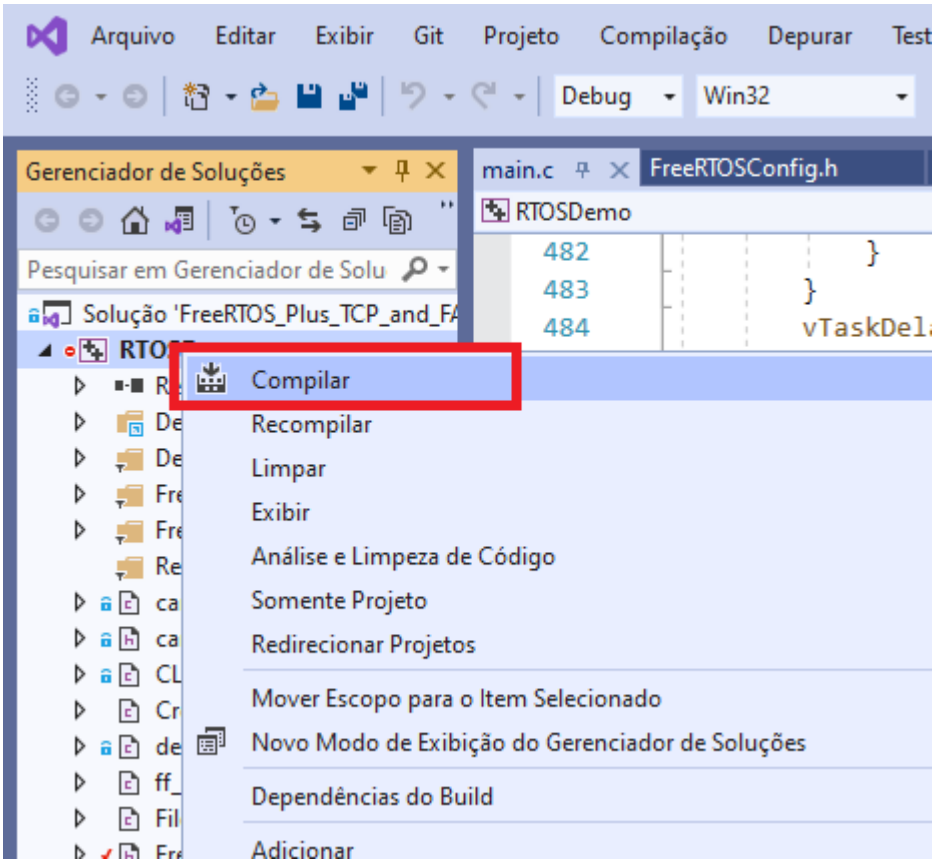
1. You must have Visual Studio Communuity 2019 Installed. If you don't install it from this link
2. You must have Python 3.X or later installed and mapped in your Windows PATH. You can download from this link

Steps to run the project:

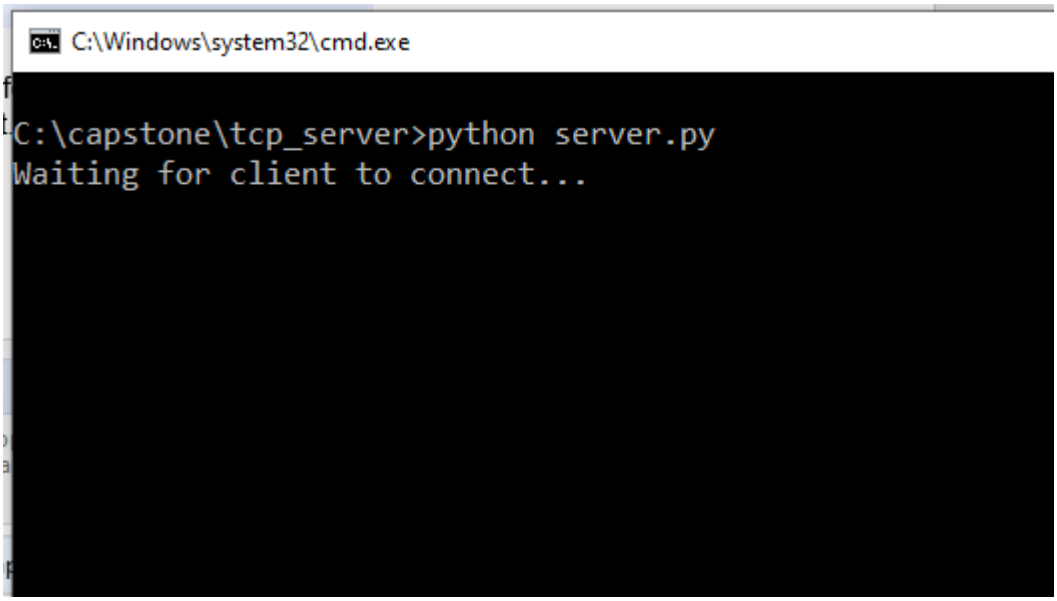
1. Uncompress the Capstone project
2. First step is to open the FreeRTOS Client on Visual Studio. Go to the folder ..\freertos_capstone_project2\FreeRTOS-Plus\Demo\FreeRTOS_Plus_TCP_and_FAT_Windows_Simulator
3. Double click the "WIN32.vcxproj" file to open the project
4. First start the server. Just double click on the



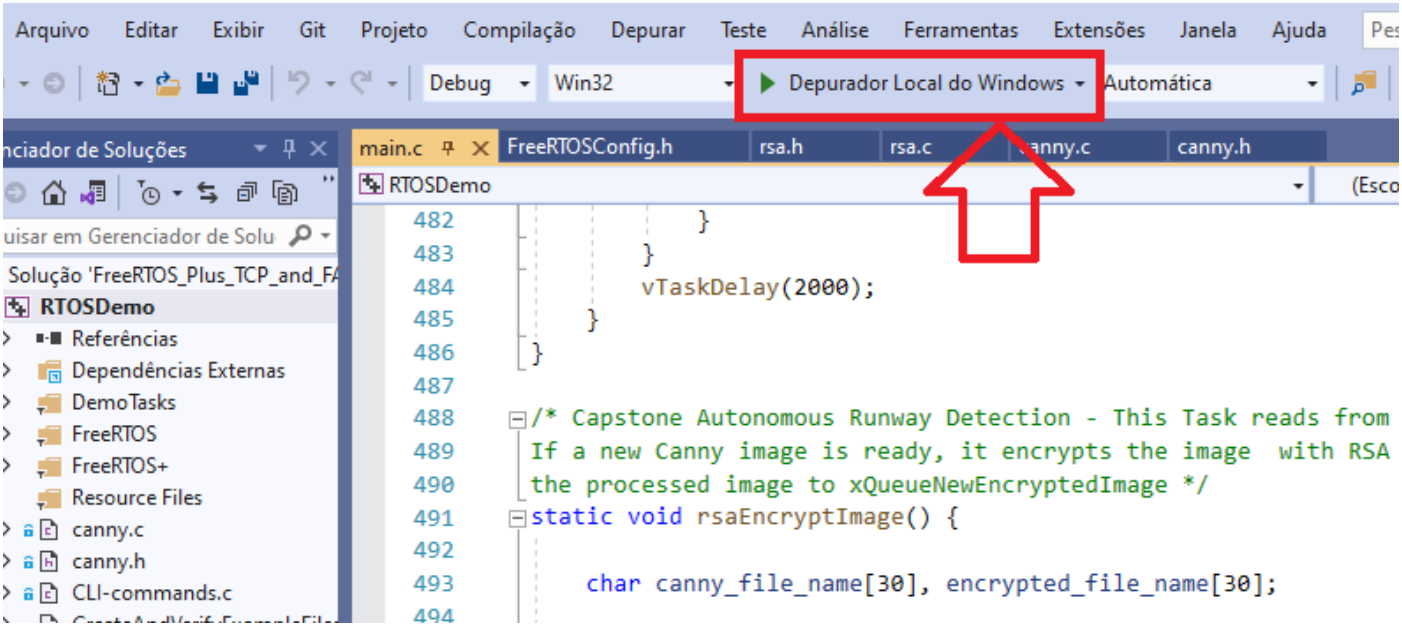
5. Right click in RTOSDemo and click in "Compile"



6. Now lets start the server. Go to the folder ..\capstone\tcp_server
7. Double click on the runme.bat script. You should see the command line like the image below:



8. Now to start the FreeRTOS Client to start the simulation click on the “Start Debugging button”



Project Code Flow

FreeRTOS Client

new_runway_image_task()

xQueueNewRunwayImage

canny_process_image_task()

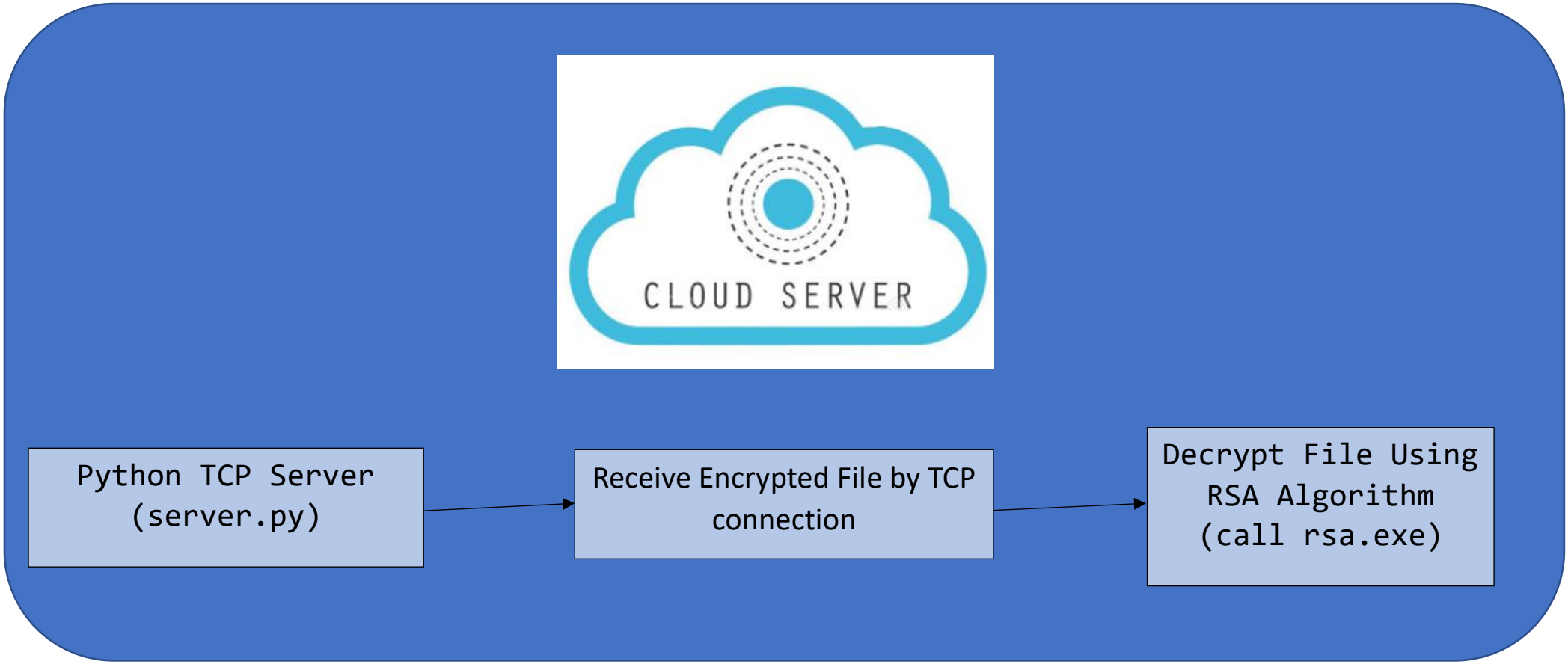
xQueueNewCannyImage

rsaEncryptImage()

xQueueNewEncryptedImage

Cloud Storage

FreeRTOS Server Code Flow



Code Organization

The communication between Tasks is done by Queues. 3 Queues were created so that the Tasks can communicate between each other.

- xQueueNewRunwayImage
- xQueueNewCannyImage
- xQueueNewEncryptedImage

And 4 Tasks were created, each with a proper task in the project:

- newRunwayImageReady() - This Task adds the new runway images to the xQueueNewRunwayImage

```
static void newRunwayImageReady() {

    while (1) {
        if (xQueueNewRunwayImage != 0)
        {
            printf("New runway image ready!!\n");
            fflush(stdout);
            /* Send the vector c to the queue */
            if (xQueueSendToBack(xQueueNewRunwayImage, RUNWAY_FILE_NAME, NO_WAIT)!= pdPASS)
            {
                printf("Fail to send item to queue. Queue is Full! \n");
            }
            printf("Sendind complete!!\n\n");
            fflush(stdout);
        }
        vTaskDelay(2000);
    }
}
```

- cannyProcessImage() - This Task reads from the xQueueNewRunwayImage. If a new image is ready, it processes the image with Canny algorithm and sends the processed image to xQueueNewCannyImage

```
static void cannyProcessImage() {

    char runway_file_name[30], canny_file_name[30];
    char *char_pointer;

    while (1) {

        if (xQueueReceive(xQueueNewRunwayImage, runway_file_name, NO_WAIT) == pdPASS)
        {
            printf("Debug - cannyProcessImage() - runway_file_name: %s\n", runway_file_name);

            //run canny algorithm on the runway image
            char_pointer = cannymain(RUNWAY_FILE_NAME);
            if (char_pointer == 0) //canny failed. Return
                return;

            //now we copy the string to a local pointer, as canny main will erase it when run again.
            strcpy(canny_file_name, char_pointer);
            if (xQueueSendToBack(xQueueNewCannyImage, canny_file_name, NO_WAIT) != pdPASS) {
                printf("Fail to send item to queue. Queue is Full! \n");
            } else {
                printf("Debug - cannyProcessImage() - New canny image sent to queue: %s \n", canny_file_name);
            }
        }
        vTaskDelay(2000);
    }
}
```

- rsaEncryptImage() - This Task reads from the xQueueNewCannyImage. If a new Canny image is ready, it encrypts the image with RSA algorith and and sends the processed image to xQueueNewEncryptedImage

```
static void rsaEncryptImage() {

    char canny_file_name[30], encrypted_file_name[30];

    while (1) {
        printf("rsaEncryptImage()\n");
        if (xQueueReceive(xQueueNewCannyImage, canny_file_name, NO_WAIT) == pdPASS)
        {
            printf("Debug - rsaEncryptImage() - Canny image received! - canny_file_name: %s\n",
                canny_file_name);
            strcpy(encrypted_file_name, canny_file_name);
            encrypted_file_name[4] = 'e'; encrypted_file_name[5] = 'n'; //change 'de'crypted to 'en'crypted
            RSAEncryptFile(canny_file_name, encrypted_file_name);

            if (xQueueSendToBack(xQueueNewEncryptedImage, encrypted_file_name, NO_WAIT) != pdPASS) {
                printf("Fail to send item to xQueueNewEncryptedImage. Queue is Full! \n");
            }
            else {
                printf("Debug - rsaEncryptImage() - New encrypted image sent to xQueueNewEncryptedImage: %s \n",
                    encrypted_file_name);
            }

        }
        else {
            printf("Debug - rsaEncryptImage() - xQueueNewCannyImage is empty. No new images to process\n");
        }
        vTaskDelay(2000);
    }
}
```

- tcpSendFileToServerTask() - This Task reads from the xQueueNewEncryptedImage. If a new Encrypted image is ready, it sends the encrypted image to the Python server

```
static void tcpSendFileToServerTask()
{
    Socket_t xClientSocket;
    struct freertos_sockaddr xRemoteAddress;
    size_t xLenToSend;
    FILE* input_file;
    char buffer[BUFFER_SIZE];
    char encrypted_file_name[30];
    int read_count = 0;
    static const TickType_t xTimeOut = pdMS_TO_TICKS(1000);

    while (1)
    {
        //check if there is a new file in queue
        if (xQueueReceive(xQueueNewEncryptedImage, encrypted_file_name, NO_WAIT) == pdPASS) {

            /* Set the IP address (192.168.0.200) and port (1500) of the remote socket
            to which this client socket will transmit. */
            xRemoteAddress.sin_port = FreeRTOS_htons(1500);
            xRemoteAddress.sin_addr = FreeRTOS_inet_addr_quick(127, 0, 0, 1); //server is on localhost

            // Create a socket.
            xClientSocket = FreeRTOS_socket(FREERTOS_AF_INET, FREERTOS_SOCK_STREAM, FREERTOS_IPPROTO_TCP);

            //setting the timeouts
            FreeRTOS_setsockopt(xClientSocket, 0, FREERTOS_SO_RCVTIMEO, &xTimeOut,  sizeof(xTimeOut));
            FreeRTOS_setsockopt(xClientSocket, 0, FREERTOS_SO_SNDTIMEO, &xTimeOut, sizeof(xTimeOut));

            //binding to the PORT 1500
            FreeRTOS_bind(xClientSocket, &xRemoteAddress, sizeof(xRemoteAddress));

            // Connect to the server.
            if (FreeRTOS_connect(xClientSocket, &xRemoteAddress, sizeof(xRemoteAddress)) == 0)
            {
                input_file = fopen(encrypted_file_name, "rb");
                if (input_file == NULL) {
                    printf("Error opening Source File.\n");
                    exit(1);
                }

                printf("tcpSendFileToServerTask() - file sending STARTED\n");

                //first read before entering the loop
                read_count = fread(buffer, sizeof(char), BUFFER_SIZE, input_file);
```

Hardware Description

First let's calculate the bandwith required:

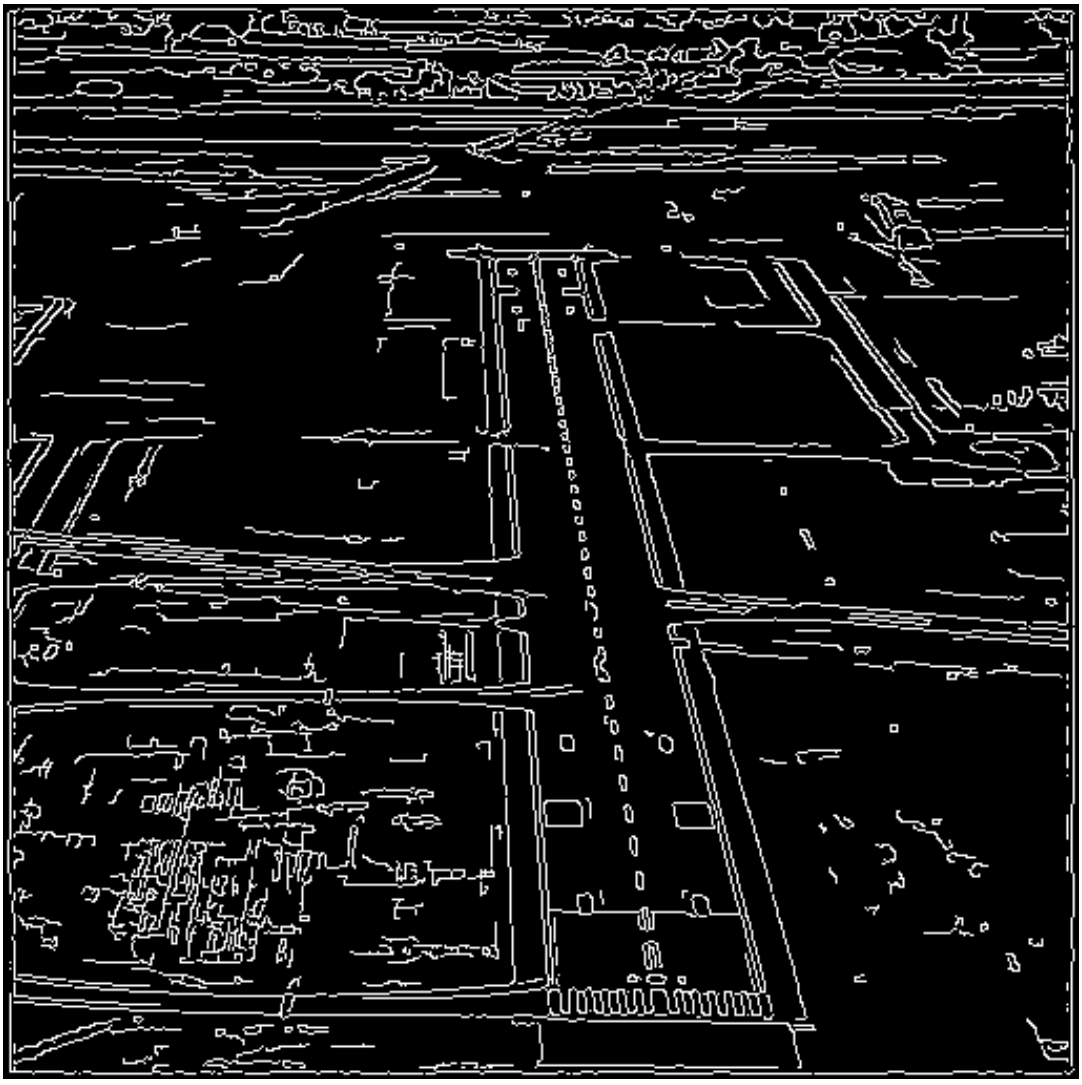
Bandwith:
640*480 pixels/frame * 25 frames/second * 1byte/pixel = 7.7 MBps or 61.4 Mbps

This project requires:

- A camera to get capture the images of the runway during the landing
- A main board to process the image and send to the server
- A long-distance connection between the airplane and a server, assuming the land

Camera Resolution

A VGA Camera (640x480) should be enough to generate good quality images to process on the server side, like the image below.



Power Consumption

The power consumption with high bandwith in LTE should be around 1 Watt.

Chosen Hardware and Operational System

Chosen Board: Qualcomm Snapdragon, because it has low power consumption, good portability to FreeRTOS, is largely used in the market and

Connection technology: LTE because of it's high availability and in lower frequencies (~900MHz) it can reach around 10Km which is required to communicate, and can offer a compatible throughput.

Costs

	Fixed Cost (US$)	Monthly Cost (US$)
Qualcomm Snapdragon board	200	
LTE Shield	100	
LTE International Plan (Verizon)		55
Amazon Cloud Plan		1000
Total	300 US$	1055 US$ / month