PROJECT REPORT

ORANGE PROJECT

DONE

Theia

Author(s):

Famura

Contents

1. Introduction

The Theia project is designed for using Android devices as monitors for Windows computers. It has been built with intentions of both using it and gaining experience here for developing another project which intends to use Android devices as VR headsets.

The project, thus the report is divided into two parts. Windows part and Android part. Each part had its own development, own analysis and own part in the report.

2. Windows

2.1. Project Technologies

2.1.1. Module(s)

- **Windows (Windows 10):**
  The entire project is designed to run in Windows systems. Did not tested in lower Windows versions like Windows 7. But tested in Windows 11 and running flawlessly.

2.1.2. Coding Language(s)

- **C++ (C++14 Standard) – Visual Studio 2022:**
  The entire project is written in C++ and is set to compile with the C++14 standard. This provides modern language features and compatibility with Visual Studio 2022.

2.1.3. Libraries & External Additions

- **Winsock2.h:**
  Provides the Windows Sockets API for network communication (TCP/IP).
- **targetver.h & stdafx.h:**
  Provides required coding for socket programming.
- **thread.h:**
  Using “Posix threads” for the parallel side of the communication.
- **shlobj.h:**
  Used for accessing special Windows folders (e.g., Downloads directory via SHGetKnownFolderPath).
- **Usbmmidd_v2:**
  An external project to create virtual screens. Used as deviceinstaller64.exe. References are given but unfortunately the creator was not found. Thus, does not exists in this report.
- **stb_image_write.h:**
  Used to write image data from memory to files in formats such as PNG, JPG, BMP, TGA, or HDR.
- **Janus:**
  Janus’s file sharing infrastructure is used in this project.

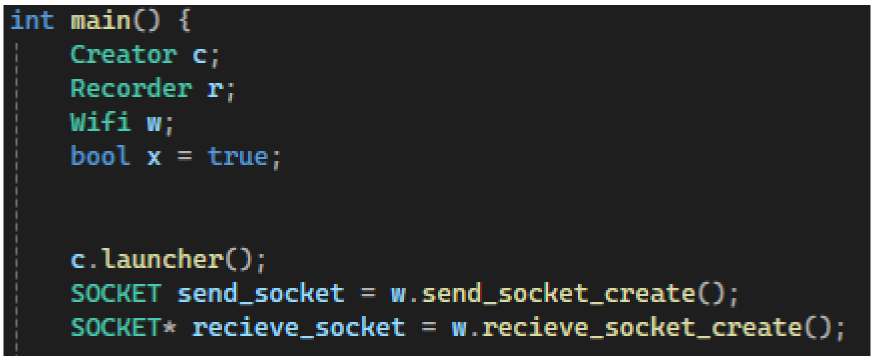
2.2. Project Architecture

This project is developed using modular programming. The design divides the project into four individual parts.

2.2.1. Main

Main class is the manager class of the application. All other classes are initialized, called, analyzed and fall into sequence here. The application works on the console right now, but if a frontend will be created in future it will be called and managed from here as well.

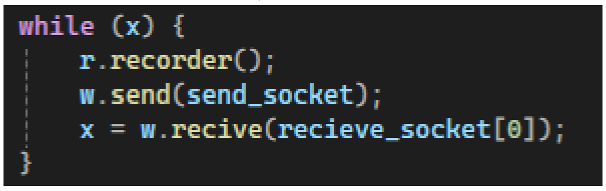
2.2.1.1. Initialize Everything



```
int main() {
    Creator c;
    Recorder r;
    Wifi w;
    bool x = true;


    c.launcher();
    SOCKET send_socket = w.send_socket_create();
    SOCKET* recieve_socket = w.recieve_socket_create();
```

At first, the import variables are created. And then a new screen is created using c.launcher(). After that, the process of using Janus' infrastructure begins with creating sockets using Wi-fi module.

2.2.1.2. Main Loop

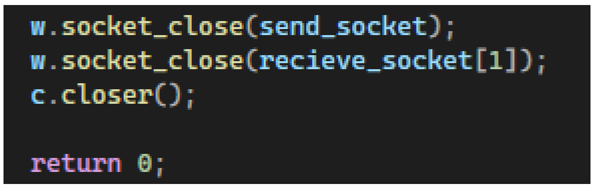


```
while (x) {
    r.recorder();
    w.send(send_socket);
    x = w.recive(recieve_socket[0]);
}
```

Main loop consists only 3 lines of code. On every loop occasion, recorder records the required screen. Then Wi-fi module sends the created image. And lastly waits for the receive message from the Android device. If it received correctly x stays true and loop keeps going. If it doesn't receive anything it returns false to x, and loop stops. This might sound like a costly process but in my tests, I found out that it works perfectly for daily usage.

2.2.1.3. Close Everything

Lastly, close everything thus virtual screen and sockets do not keep using resources after the termination of the program.



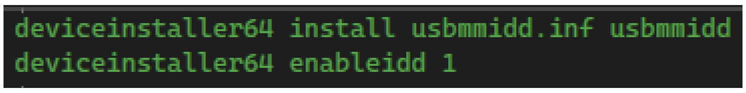
```
w.socket_close(send_socket);
w.socket_close(recieve_socket[1]);
c.closer();

return 0;
```

2.2.2. Screen Creator

Even if the name is only creator, this class both creates and stops the virtual screen process. It has some .bat files for this.

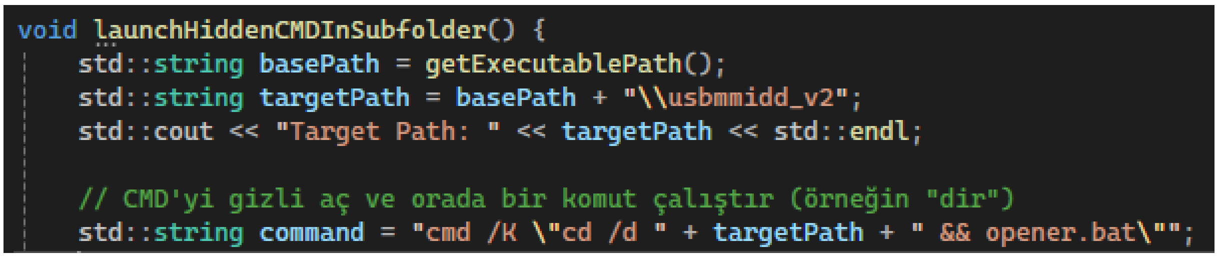
2.2.2.1. Reach Opener & Opener

This part uses the opener.bat. It is like the one in the image:



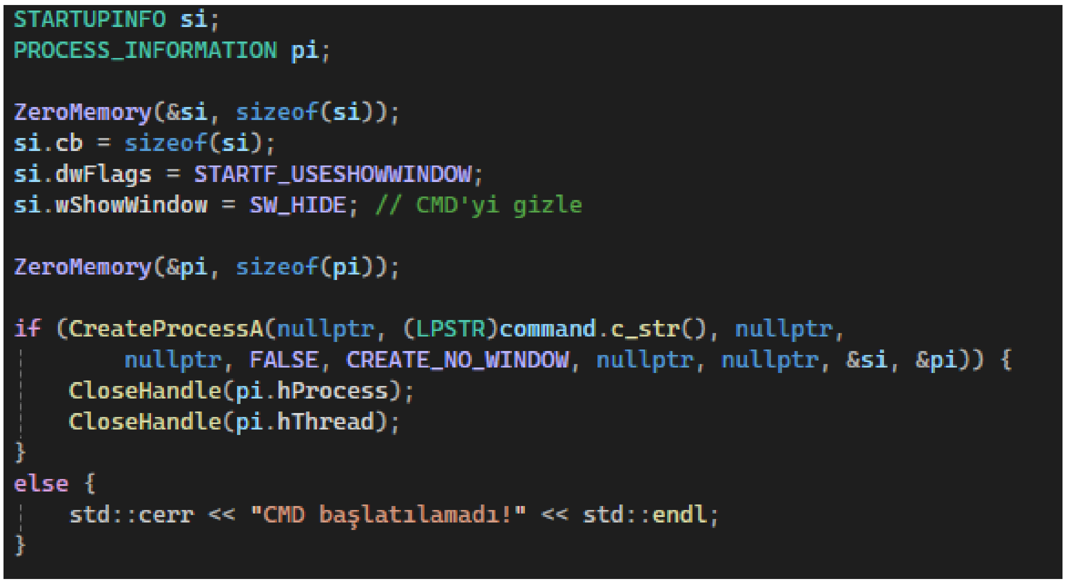
```
deviceinstaller64 install usbmmidd.inf usbmmidd
deviceinstaller64 enableidd 1
```

And the code reaches that .bat file like this:



```
void launchHiddenCMDInSubfolder() {
    std::string basePath = getExecutablePath();
    std::string targetPath = basePath + "\\usbmmidd_v2";
    std::cout << "Target Path: " << targetPath << std::endl;

    // CMD'yi gizli aç ve orada bir komut çalıştır (örneğin "dir")
    std::string command = "cmd /K \"cd /d " + targetPath + " && opener.bat\"";
```

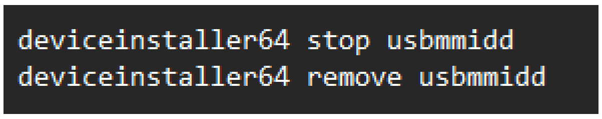
After reaching, the code creates a hidden command prompt in the background. And executes .bat file so it would be more usable to user.



```
STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
si.dwFlags = STARTF_USESHOWWINDOW;
si.wShowWindow = SW_HIDE; // CMD'yi gizle

ZeroMemory(&pi, sizeof(pi));

if (CreateProcessA(nullptr, (LPSTR)command.c_str(), nullptr,
        nullptr, FALSE, CREATE_NO_WINDOW, nullptr, nullptr, &si, &pi)) {
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
else {
    std::cerr << "CMD başlatılamadı!" << std::endl;
}
```

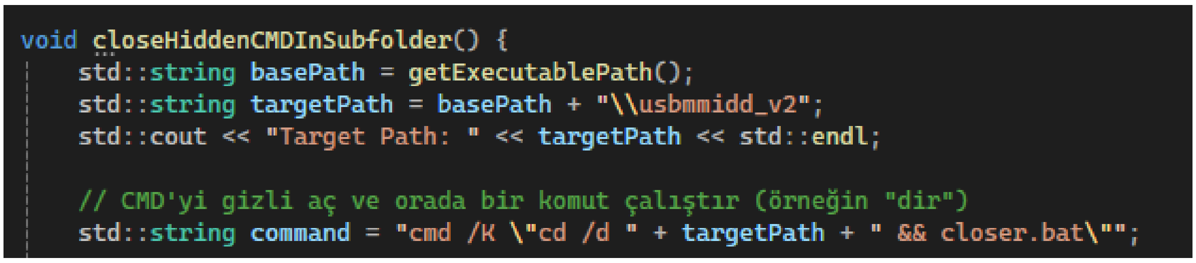
2.2.2.2. *Reach Closer & Closer*

On a similar approach, there is a closer.bat that has code on the right side:



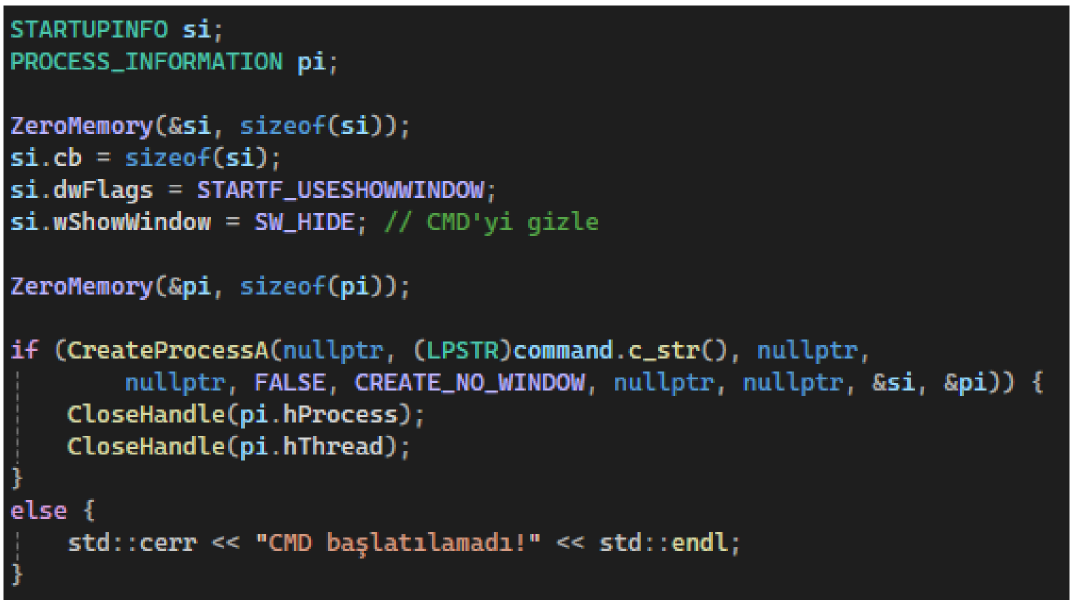
```
deviceinstaller64 stop usbmmidd
deviceinstaller64 remove usbmmidd
```

After executing this, the virtual screen gets removed. And as like on the launcher, it first reaches .bat file:



```
void closeHiddenCMDInSubfolder() {
    std::string basePath = getExecutablePath();
    std::string targetPath = basePath + "\\usbmmidd_v2";
    std::cout << "Target Path: " << targetPath << std::endl;

    // CMD'yi gizli aç ve orada bir komut çalıştır (örneğin "dir")
    std::string command = "cmd /K \"cd /d " + targetPath + " && closer.bat\"";
```

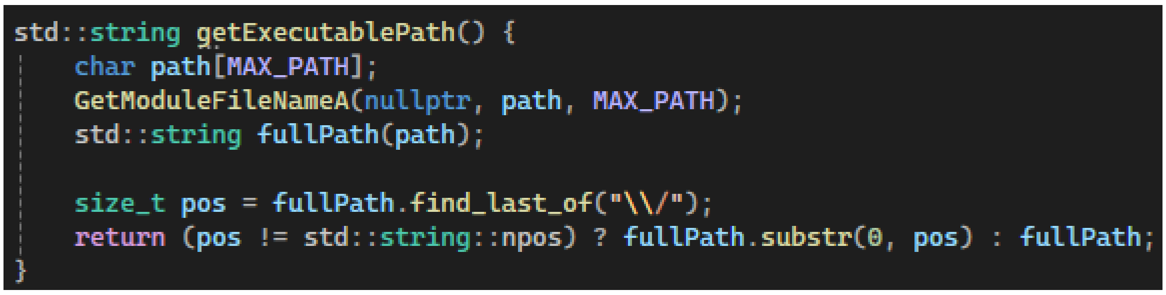
Then executes the .bat file on a hidden cmd:



```
STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
si.dwFlags = STARTF_USESHOWWINDOW;
si.wShowWindow = SW_HIDE; // CMD'yi gizle

ZeroMemory(&pi, sizeof(pi));

if (CreateProcessA(nullptr, (LPSTR)command.c_str(), nullptr,
        nullptr, FALSE, CREATE_NO_WINDOW, nullptr, nullptr, &si, &pi)) {
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
else {
    std::cerr << "CMD başlatılamadı!" << std::endl;
}
```

2.2.2.3. *Path Helper*

Both of these methods use help of a familiar method that is used in the Janus, getExecutablePath(). This method allows C++ to recognize the given filepath as a filepath and not as functions of std::String class. Further explanation about this method can be found under the 4th page of the Janus report.



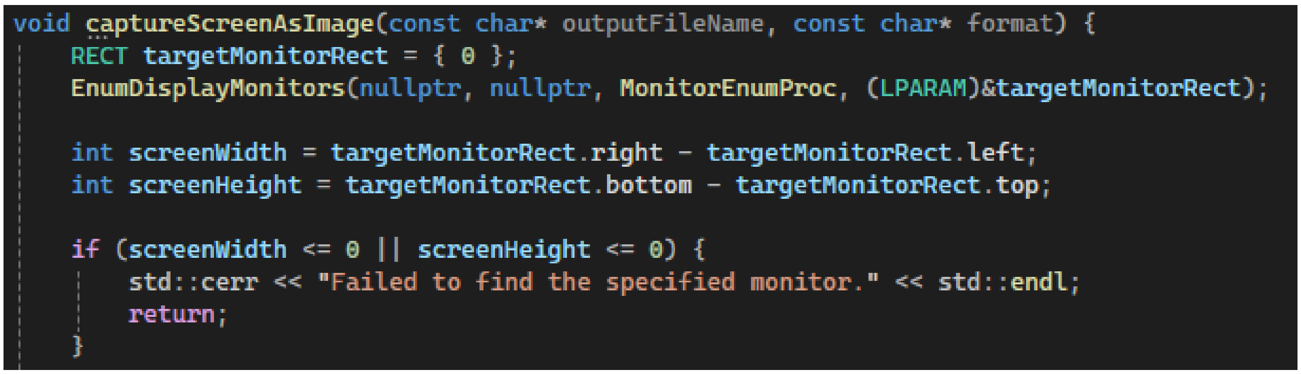
```
std::string getExecutablePath() {
    char path[MAX_PATH];
    GetModuleFileNameA(nullptr, path, MAX_PATH);
    std::string fullPath(path);

    size_t pos = fullPath.find_last_of("\\/");
    return (pos != std::string::npos) ? fullPath.substr(0, pos) : fullPath;
}
```

2.2.3. Screen Recorder

The main purpose of this class is to get screen data as BitMap from Windows library functions and write it as an image file so that when it has been sent, it can be shown to the user on Android. This process also gets divided into five parts.

2.2.3.1. *Initialize & Find Monitor on the Code*

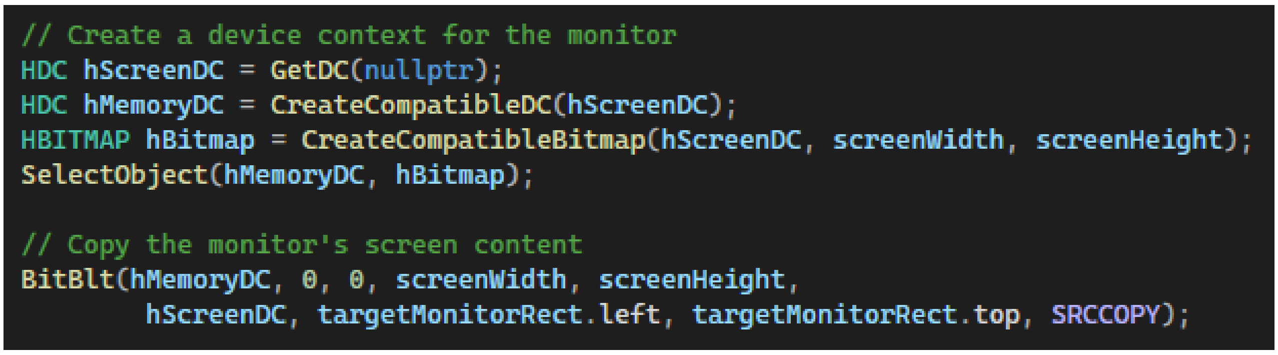
The code finds the required monitor using Windows's own libraries. And then it takes screen width and height so that it would be used later for BitMap



```
void captureScreenAsImage(const char* outputFileName, const char* format) {
    RECT targetMonitorRect = { 0 };
    EnumDisplayMonitors(nullptr, nullptr, MonitorEnumProc, (LPARAM)&targetMonitorRect);

    int screenWidth = targetMonitorRect.right - targetMonitorRect.left;
    int screenHeight = targetMonitorRect.bottom - targetMonitorRect.top;

    if (screenWidth <= 0 || screenHeight <= 0) {
        std::cerr << "Failed to find the specified monitor." << std::endl;
        return;
    }
```

2.2.3.2. *Create a Device Context*

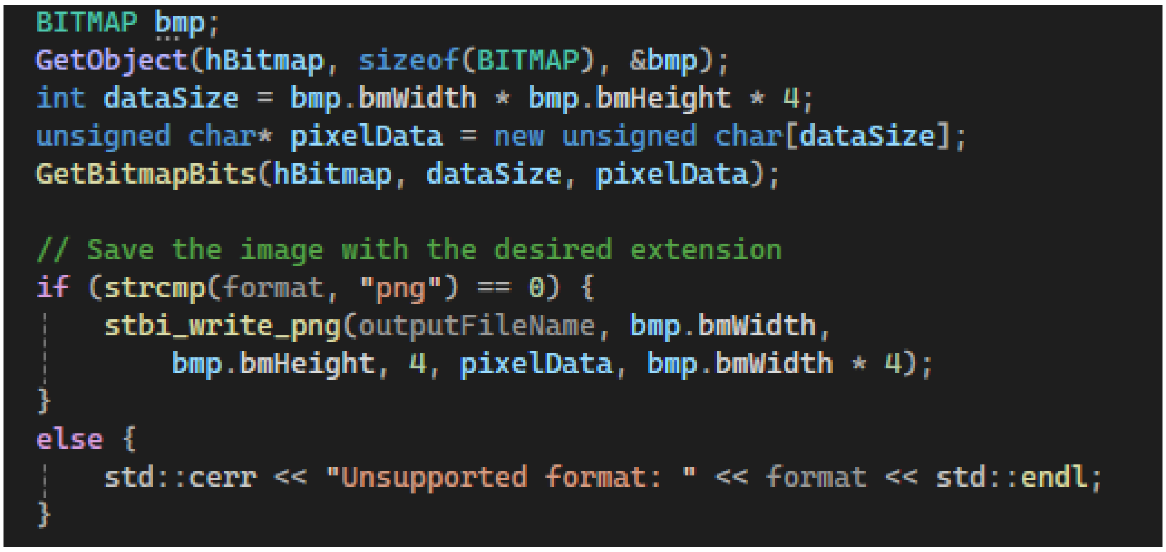
After getting this monitor data, the screen content gets created so that it can be transferred as BitMap and then from BitMap to photograph.



```
// Create a device context for the monitor
HDC hScreenDC = GetDC(nullptr);
HDC hMemoryDC = CreateCompatibleDC(hScreenDC);
HBITMAP hBitmap = CreateCompatibleBitmap(hScreenDC, screenWidth, screenHeight);
SelectObject(hMemoryDC, hBitmap);

// Copy the monitor's screen content
BitBlt(hMemoryDC, 0, 0, screenWidth, screenHeight,
       hScreenDC, targetMonitorRect.left, targetMonitorRect.top, SRCCOPY);
```

2.2.3.3. *Get the Data as Bitmap*

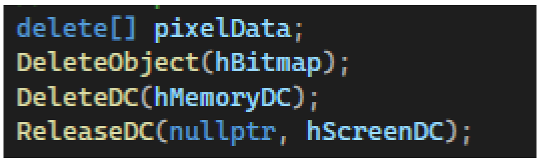
Then the monitor gets saved as photograph



```
BITMAP bmp;
GetObject(hBitmap, sizeof(BITMAP), &bmp);
int dataSize = bmp.bmWidth * bmp.bmHeight * 4;
unsigned char* pixelData = new unsigned char[dataSize];
GetBitmapBits(hBitmap, dataSize, pixelData);

// Save the image with the desired extension
if (strcmp(format, "png") == 0) {
    stbi_write_png(outputFileName, bmp.bmWidth,
        bmp.bmHeight, 4, pixelData, bmp.bmWidth * 4);
}
else {
    std::cerr << "Unsupported format: " << format << std::endl;
}
```

2.2.3.4. *Cleanup*

After all the process, everything gets cleared so that it won’t waste resources of the computer.



```
delete[] pixelData;
DeleteObject(hBitmap);
DeleteDC(hMemoryDC);
ReleaseDC(nullptr, hScreenDC);
```

2.2.4. Wi-Fi Handler

This part is using fully Janus. For this part, please check Janus’ report which can be found here:

Janus.docx

2.3. Project Analysis

2.3.1. Test

The tests were done on Windows computers. All tests were successful. And as for scalability, the code shows promising results. It seems that we can send around 80 photos per second.

2.3.2. Upgradable Things

The process basically comes down to taking screenshots and sending it. This is a very primal way to approach this problem. A more complicated and more optimized way should be used.

3. Android

3.1. Project Technologies

3.1.1. Module(s)

- **Android (Android 13+):**
  Project’s android side has been designed to run in the Android 13+. It shouldn’t have any issues running in Android 7+ devices but there is no device to test it, thus it is considered as uncertain.

3.1.2. Coding Languages

- **Java (Java 15) – Android Studio:**
  The entire project is written in Java and targets Java 15. It leverages modern language features such as text blocks, records, and enhanced pattern matching, ensuring compatibility and optimal performance within Android Studio.

3.1.3. Libraries & External Additions

- **java.net.ServerSocket & java.net.Socket:**
  Provides TCP/IP-based network communication. ServerSocket listens for incoming connections, while Socket handles data exchange with the client.

- **java.io.DataInputStream & java.io.DataOutputStream:**
  Used for structured binary data transmission over sockets. Supports reading and writing primitive data types, enabling a custom protocol (e.g., image length + image bytes).

- **android.graphics.BitmapFactory:**
  Decodes image data from a byte array into a Bitmap object. Supports formats like PNG and JPEG, allowing dynamic image rendering in the UI.

- **android.widget.ImageView & android.widget.TextView:**
  UI components for displaying images and status messages. ImageView renders the received bitmap, while TextView updates connection status and error messages.

- **androidx.appcompat.app.AppCompatActivity:**
  Base class for Android activities using the AppCompat support library. Manages lifecycle events such as onCreate and onDestroy.

- **android.os.Bundle:**
  Used to pass initialization parameters and restore state during activity creation.

- **Thread (Java/Kotlin standard):**
  Enables background execution of the server logic to avoid blocking the main UI thread. Ensures responsive user experience.

- **@Volatile annotation:**
  Ensures visibility of the running flag across multiple threads. Prevents caching issues in concurrent environments.

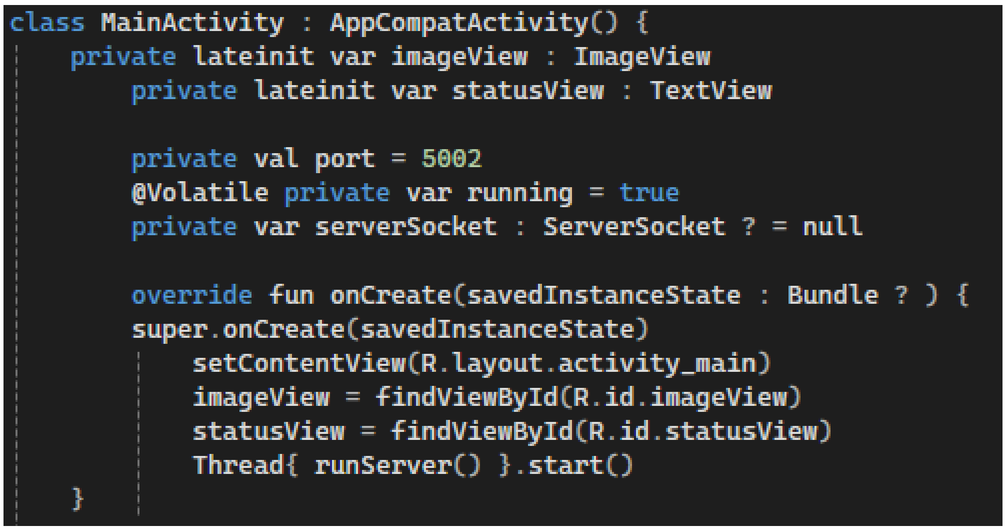
- **R.layout.activity_main, R.id.imageView, R.id.statusView:**
  References to XML-defined UI layout and components. Used to bind Java code to visual elements.

3.2. Project Architecture

The mobile side architecture is actually pretty simple. Since most of the work is handled by the Windows side, the Android side has a huge lift-off regarding workload. Here is the design of it:

3.2.1. Activity Lifecycle Manager

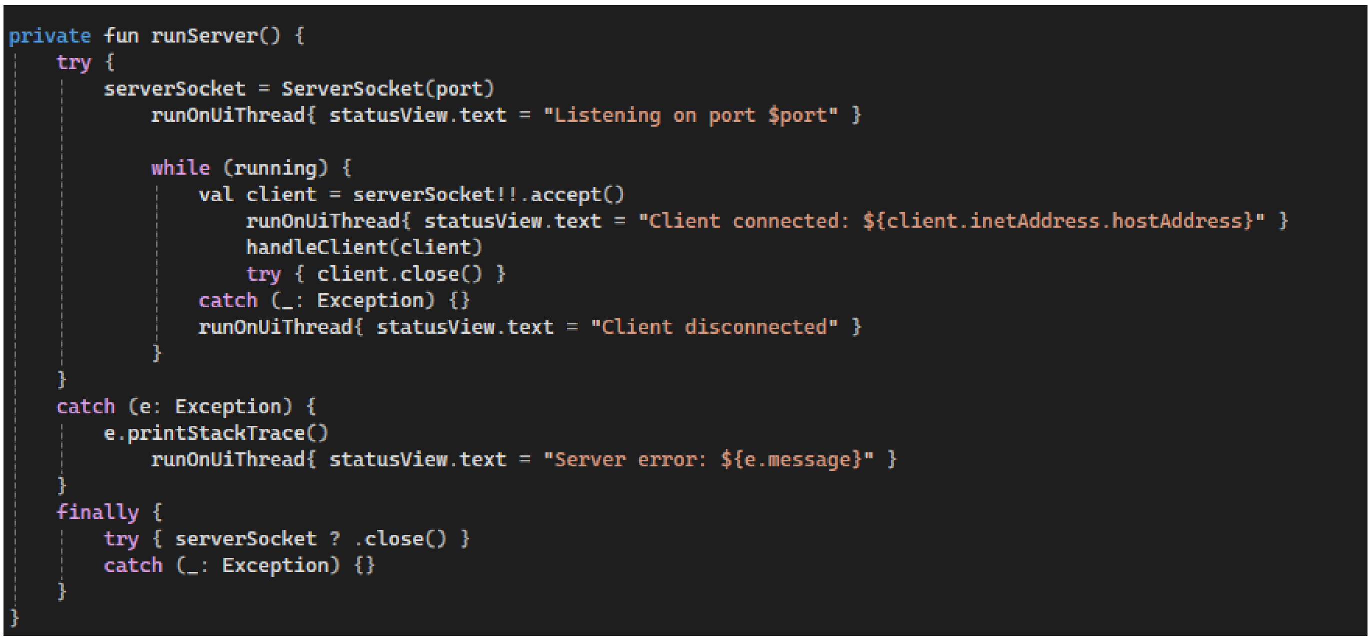
Acts as the entry point of the application. Initializes UI components (ImageView, TextView) and starts the background server thread. Handles lifecycle events (onCreate, onDestroy) to manage server startup and shutdown cleanly.



```kotlin
class MainActivity : AppCompatActivity() {
    private lateinit var imageView : ImageView
        private lateinit var statusView : TextView

        private val port = 5002
        @Volatile private var running = true
        private var serverSocket : ServerSocket ? = null

        override fun onCreate(savedInstanceState : Bundle ? ) {
        super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_main)
            imageView = findViewById(R.id.imageView)
            statusView = findViewById(R.id.statusView)
            Thread{ runServer() }.start()
    }
```

3.2.2. Server Handler

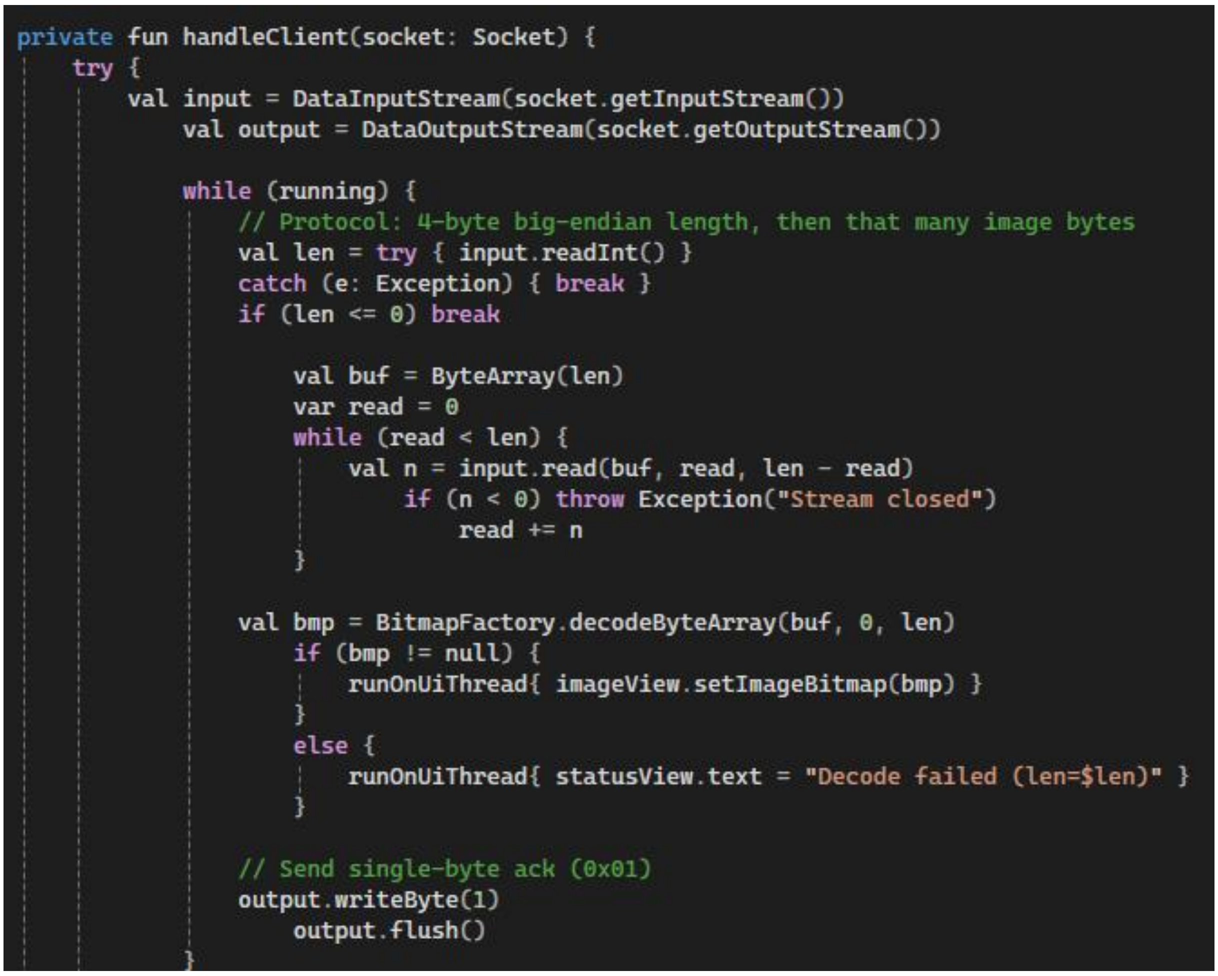
Creates a ServerSocket bound to port 5002. Listens for incoming client connections from the Windows application. For each connection: Displays connection status on the UI / Delegates image handling to handleClient() / Closes the client socket after communication ends.



```kotlin
private fun runServer() {
    try {
        serverSocket = ServerSocket(port)
            runOnUiThread{ statusView.text = "Listening on port $port" }

            while (running) {
                val client = serverSocket!!.accept()
                    runOnUiThread{ statusView.text = "Client connected: ${client.inetAddress.hostAddress}" }
                    handleClient(client)
                    try { client.close() }
                catch (_: Exception) {}
                runOnUiThread{ statusView.text = "Client disconnected" }
            }
    }
    catch (e: Exception) {
        e.printStackTrace()
            runOnUiThread{ statusView.text = "Server error: ${e.message}" }
    }
    finally {
        try { serverSocket ? .close() }
        catch (_: Exception) {}
    }
}
```

3.2.3. Client Communication & Image Decoder

Reads image data from the socket using a custom protocol: First reads a 4-byte integer indicating image length / Then reads the image bytes into a buffer. Decodes the image using BitmapFactory.decodeByteArray. Displays the image on the ImageView if decoding is successful. Sends a single-byte acknowledgment (0x01) back to the sender. Handles stream errors and ensures socket closure.



```kotlin
private fun handleClient(socket: Socket) {
    try {
        val input = DataInputStream(socket.getInputStream())
            val output = DataOutputStream(socket.getOutputStream())

            while (running) {
                // Protocol: 4-byte big-endian length, then that many image bytes
                val len = try { input.readInt() }
                catch (e: Exception) { break }
                if (len <= 0) break

                    val buf = ByteArray(len)
                    var read = 0
                    while (read < len) {
                        val n = input.read(buf, read, len - read)
                            if (n < 0) throw Exception("Stream closed")
                                read += n
                    }

                val bmp = BitmapFactory.decodeByteArray(buf, 0, len)
                    if (bmp != null) {
                        runOnUiThread{ imageView.setImageBitmap(bmp) }
                    }
                    else {
                        runOnUiThread{ statusView.text = "Decode failed (len=$len)" }
                    }

                // Send single-byte ack (0x01)
                output.writeByte(1)
                    output.flush()
            }
```

3.3. Project Analysis

3.3.1. Test

As said before, this project only tested under Android 13+ devices. And worked flawlessly on them. In our test, I found out that code can show around 400 photographs per minute. But counting Windows performance and general Wi-Fi performance, it works with around 60 photos per minute.

3.3.2. Upgradable Things

The process basically comes down to taking screenshots and sending it. This is a very primal way to approach this problem. A more complicated and more optimized way should be used.

4. Project Comments

4.1. Possible Usages

The project was mostly designed for daily usage. It is not suitable for gaming. Only for office work.

4.1.1. To-Do's When Using

If you want to further develop this project, please be very careful with the deviceinstaller64.exe. It has such a power on your computer that I accidentally turned off all my computers' drivers one time. Poor Emily (my computer) never recovered from that until I eventually formatted for switching SSD.

4.2. Creator Comments

If this project is meant to be a step for using Android devices as virtual VR devices, we have a long way to go and have many more things to understand about streaming. Also, this project was meant to be developed by two people, Famura and Aybüke. But, as all of us already expected, she left mid-development without doing anything and this project was again developed alone.

Hooah

-Famura

5. Credits

- Famura Only Project

6. References

- Microsoft. (n.d.). Winsock2 Reference. Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2
- ISO/IEC. (2014). ISO/IEC 14882:2014 – Programming Languages – C++ (C++14 Standard). International Organization for Standardization. https://www.iso.org/standard/64029.html
- Microsoft. (n.d.). SHGetKnownFolderPath function (shlobj_core.h). Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/shlobj_core/nf-shlobj_core-shgetknownfolderpath
- Tanenbaum, A. S., & Wetherall, D. J. (2011). Computer Networks (5th ed.). Pearson.
- Stevens, W. R., Fenner, B., & Rudoff, A. M. (2004). Unix Network Programming, Volume 1: The Sockets Networking API (3rd ed.). Addison-Wesley.
- Microsoft. (n.d.). InetPton function (ws2tcpip.h). Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/ws2tcpip/nf-ws2tcpip-inetpton
- How To Create a Virtual Monitor - Windows [Video]. (2023, Mart ?). YouTube. https://www.youtube.com/watch?v=ybHKFZjSkVY
- Oracle. (n.d.). ServerSocket (Java Platform SE 15). Oracle Documentation. https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/net/ServerSocket.html
- Oracle. (n.d.). DataInputStream (Java Platform SE 15). Oracle Documentation. https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/io/DataInputStream.html
- Android Developers. (n.d.). BitmapFactory. Android Developer Guide. https://developer.android.com/reference/android/graphics/BitmapFactory
- Android Developers. (n.d.). AppCompatActivity. Android Developer Guide. https://developer.android.com/reference/androidx/appcompat/app/AppCompatActivity
- Microsoft. (n.d.). GetDC function (wingdi.h). Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getdc
- Microsoft. (n.d.). BitBlt function (wingdi.h). Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/wingdi/nf-wingdi-bitblt
- Microsoft. (n.d.). SHGetKnownFolderPath function (shlobj_core.h). Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/shlobj_core/nf-shlobj_core-shgetknownfolderpath
- GitHub. (n.d.). stb_image_write.h by nothings. GitHub Repository. https://github.com/nothings/stb/blob/master/stb_image_write.h

7. Disk Contents

- Project source code (whole Visual Studio Project and Android Studio project in .rar) contains this report on .docx format
- This report in PDF format
- Igra Rock’en’Roll Cela Jugoslavia by Elektricini Orgazam