Remote Desktop Voice-Control via Smartphone

IoT Voice Assistant

Miguel Gomez
Department of Computer Science
Binghamton University
Binghamton, NY, USA
mgomez4@binghamton.edu

Thomas Horowitz
Department of Computer Science
Binghamton University
Binghamton, NY, USA
thorowi1@binghamton.edu

ABSTRACT

Modern voice assistants are largely tied to their associated brand's ecosystem (Apple's Siri, Microsoft's Cortana, Amazon's Alexa, etc.). Cable-free communication and file transfer between devices is not intuitive nor accessible enough for the general public to set up and use in their day to day lives. This project, IoT Voice Assistant, aims to make communication between smart phones and home computers simple, fast, and adaptable. By having the computer running a server that listens for new requests via the internet, users can open up the application on their android device and issue voice commands to be executed on the server, as long as both devices are connected to the internet.

KEYWORDS

Internet of Things, Remote Desktop, Android, Voice Assistant, Voice Recognition, TCP, Network

1 Introduction

IoT Voice Assistant is a project that consists of a mobile application and a desktop server. The desktop server can be easily run and kept as a background process either throughout the day for home use or whenever one is about to step out of the house for remote use. Whenever one wishes to interact with their home desktop remotely, the app simply needs to be open, the network information needs to be entered, and a command must then be issued. The commands range from file retrieval to saving reminders for later (and even viewing them within the app after retrieving them).

2 Design

IoT Voice Assistant is the culmination of our work in exploring the Internet of Things subject and putting together many modular components to create a product that is both useful and modern. The portion of the project that most users will interact with is the Android application. The application was written in Java using Android studio and features a sleek UI that is minimalistic and relatively intuitive.
The ideal control flow for usage is as follows:

1. User initializes Desktop application on Port Number *N*. The default port number is 12000.
2. User opens the Android application.
3. User adds a 'new device' via the floating action button on the bottom right, and enters the IP and Port Number *N*.
4. User clicks on the newly added connection.
5. User clicks on the 'Wi-Fi' symbol to establish a connection and prompt for voice input.
6. User says aloud a command with the correct parameters and the application forwards the data to the server.
7. Server processes the data as a request and sends an appropriate message after attempting to execute the request.

Balancing the UI constraints and the functionality proved to be difficult yet possible by having the main desktop application being the source of computational power. The server is made to be a "set it and forget it" program that is always accessible and reliable, provided an internet connection is established. The server application is written in C++ and relies on scripts written in Python and Julia for command execution.

2.1 Hardware

The hardware used and required consists of only two components: and Android phone and a Windows 10 Desktop. The application was made using a Coolpad Legacy running Android version 9.0 Pie and having 3.0 GB RAM. The computer used changed various times throughout development, but Windows 10 is a must. Within the project directory, both the source code as well as a pre-compiled executable is present.

2.2 Development Tools Used

The majority of the Android application was written using Jetbrains' Android Studio. The language we opted for was Java, rather than Kotlin, simply due to our familiarity with Java. Despite starting off as Java novices, we both had a basic grasp on Java code syntax, structure, and practices.
The Desktop application was written using a random assortment of Visual Studio Code, Atom, and Notepad++. The Desktop application itself is compiled using the program "Make" which uses g++ to compile it with C++ version 17.

2.3 Network and Voice Libraries

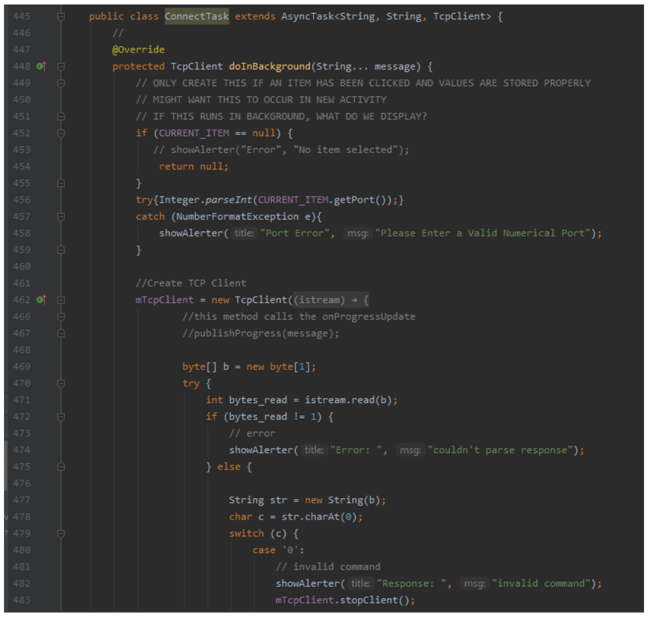
The network connections were chosen to use Transmission Control Protocol (TCP) to establish brief but secure two-way connection between the mobile Android application and the server. TCP allows for sequential request and response messages, as well as ordered delivery of bytes.

The windows API used for the network was Winsock, and the Java package used was Java Socket. The necessary headers are included within the package for Winsock.

The main library used for accessing the Android smartphone microphone and processing the vocal data into a string was the Google Voice Recognition Library.

3 Implementation

As stated earlier, the Android program was written in Java with the assistance of Android Studio. Android Studio allows for the correct folders and files to be created and ordered so that it can build the final package. Sample images of the source code and development in Android studio are shown in figures 1 and 2. Figures 3-6 shows the built application running in the Android Studio emulator.



```
public class ConnectTask extends AsyncTask<String, String, TcpClient> {
    //
    @Override
    protected TcpClient doInBackground(String... message) {
        // ONLY CREATE THIS IF AN ITEM HAS BEEN CLICKED AND VALUES ARE STORED PROPERLY
        // MIGHT WANT THIS TO OCCUR IN NEW ACTIVITY
        // IF THIS RUNS IN BACKGROUND, WHAT DO WE DISPLAY?
        if (CURRENT_ITEM == null) {
            // showAlerter("Error", "No item selected");
             return null;
        }
        try{Integer.parseInt(CURRENT_ITEM.getPort());}
        catch (NumberFormatException e){
             showAlerter( title: "Port Error",  msg: "Please Enter a Valid Numerical Port");
        }

        //Create TCP Client
        mTcpClient = new TcpClient((istream) → {
                //this method calls the onProgressUpdate
                //publishProgress(message);

                byte[] b = new byte[1];
                try {
                    int bytes_read = istream.read(b);
                    if (bytes_read != 1) {
                        // error
                        showAlerter( title: "Error: ",  msg: "couldn't parse response");
                    } else {

                        String str = new String(b);
                        char c = str.charAt(0);
                        switch (c) {
                            case '0':
                                // invalid command
                                showAlerter( title: "Response: ",  msg: "invalid command");
                                mTcpClient.stopClient();
```

Figure 1 (Above): Android Studio Layout and Application Source Code.

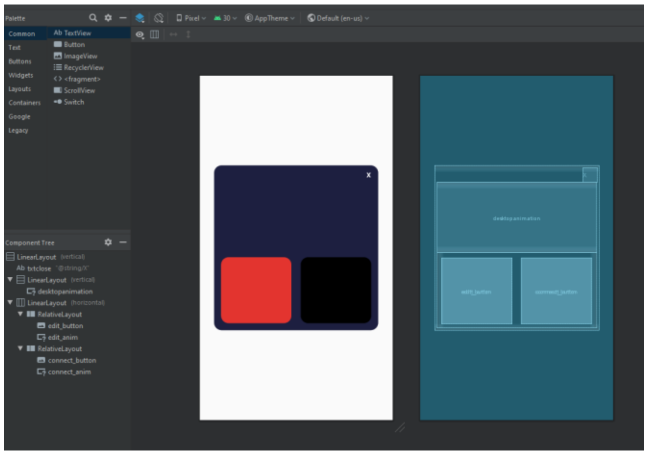


Figure 2 (Above): Android Studio XML Designer for Layout Files.

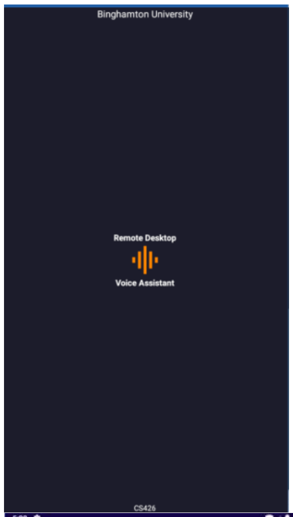
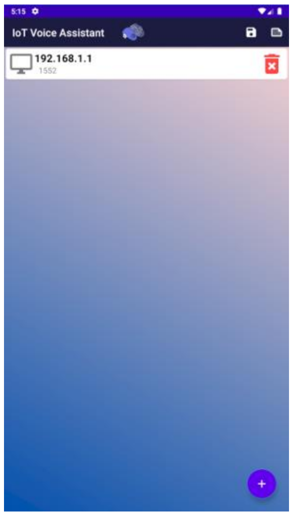


Figure 3 (Left): Android Application Loading Splash Screen shown on startup.

Figure 4 (Below): Android Application Main Item List Screen.



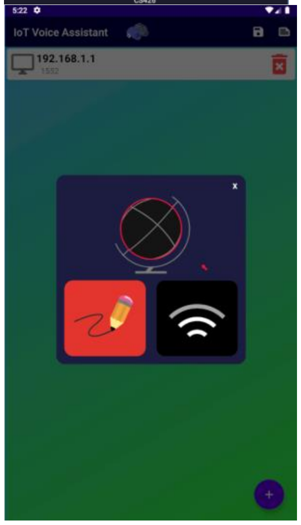
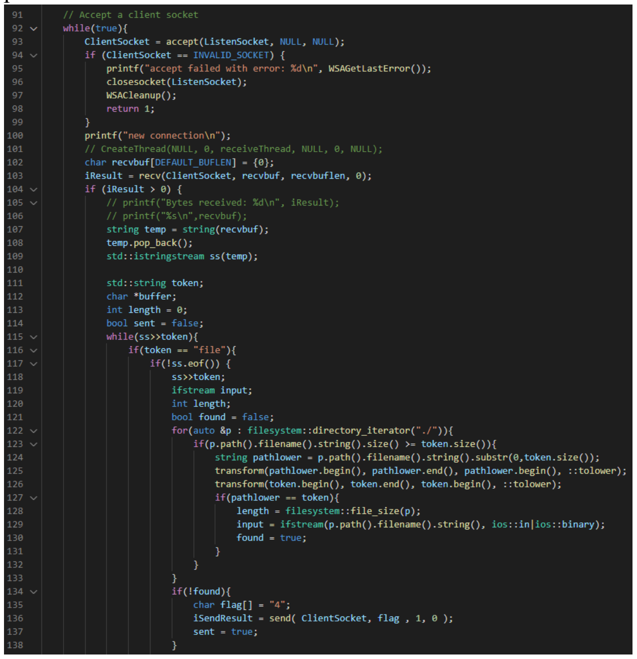


Figure 5 (Above): Item Options Menu. Left Button (RED) edits the item fields. Right button (BLACK) establishes a connection and prompts user for a command.

In Figure 4, the '+' button on the bottom right allows the user to add a new connection. The 'Note' Symbol at the top right allows the user to view the memo's saved locally (see commands section). The 'Save' Button allows the user to save the local data of the application so it does not lose the memo data or item list on exit. The red 'garbage can' located on the item is the delete button to remove that specific item from the list.

In regard to the Desktop application, the C++ program acts as a server for the mobile app to connect to. Once started, the server runs indefinitely, listening for new connections (figure 6). Once a connection request is encountered, it binds it to a socket and waits for a command to be sent. The vocal data in string form is received from the Android app via a socket. It then parses the command by using white space as a delimiter and cross referencing each token with the list of eligible commands. If no matching command is found, an error message is sent back to the Android phone over the network to let the user know that an invalid command was given. It should be noted that each command has its own connection request sent before, and the connection is terminated once the command result is received. This allows for more network stability due to fewer temporal points of failure.



```cpp
    // Accept a client socket
    while(true){
        ClientSocket = accept(ListenSocket, NULL, NULL);
        if (ClientSocket == INVALID_SOCKET) {
            printf("accept failed with error: %d\n", WSAGetLastError());
            closesocket(ListenSocket);
            WSACleanup();
            return 1;
        }
        printf("new connection\n");
        // CreateThread(NULL, 0, receiveThread, NULL, 0, NULL);
        char recvbuf[DEFAULT_BUFLEN] = {0};
        iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
        if (iResult > 0) {
            // printf("Bytes received: %d\n", iResult);
            // printf("%s\n",recvbuf);
            string temp = string(recvbuf);
            temp.pop_back();
            std::istringstream ss(temp);

            std::string token;
            char *buffer;
            int length = 0;
            bool sent = false;
            while(ss>>token){
                if(token == "file"){
                    if(!ss.eof()) {
                        ss>>token;
                        ifstream input;
                        int length;
                        bool found = false;
                        for(auto &p : filesystem::directory_iterator("./")){
                            if(p.path().filename().string().size() >= token.size()){
                                string pathlower = p.path().filename().string().substr(0,token.size());
                                transform(pathlower.begin(), pathlower.end(), pathlower.begin(), ::tolower);
                                transform(token.begin(), token.end(), token.begin(), ::tolower);
                                if(pathlower == token){
                                    length = filesystem::file_size(p);
                                    input = ifstream(p.path().filename().string(), ios::in|ios::binary);
                                    found = true;
                                }
                            }
                        }
                        if(!found){
                            char flag[] = "4";
                            iSendResult = send( ClientSocket, flag , 1, 0 );
                            sent = true;
                        }
```

Figure 6: Desktop Application Source Code. Main loop that listens for connections.

4 Commands

Listed below are the baseline supported commands

4.1.1 File Transfer. Command = "file <filename>". Looks for a file in the windows app directory named <filename> and sends it to the phone. If <filename> is equal to "memo", then the memo file is sent to the app (see 4.1.4 and 4.1.5). If no file is found, then sends back an error message. The user is prompted to enter a file name and extension in the format <filename>.<extension> on file reception.

4.1.2 Ping. Command = "ping". Runs the "ping.py" Python script. Requires Python to be installed and in the PATH. Pings google.com 10 times and sends the results to the Android phone, where it is stored as a text file in the storage.

4.1.3 Clipboard transfer. Command = "clipboard". Runs the "clipboard.jl" Julia script. Requires Julia to be installed and in the PATH. Sends the contents of the desktop's clipboard to the Android phone, where it is stored as a text file in the storage.

4.1.4 Open Application. Command = Opens any application provided in the windows system PATH and returns a success/failure message.

4.1.5 New Memo. Command = "new memo <voice memo>". Creates/Overwrites memo file located in the "windowsApp" directory. The message given by <voice memo> is then written to the first line of the folder. The user can retrieve this file and have it displayed in the notes section via the command "file memo".

4.1.6 Add to Memo. Command = "memo <voice memo>". Appends the contents of <voice memo> to a new line in the memo file located in the "windowsApp" directory. Creates a new memo file if one does not exist. The user can retrieve this file and have it displayed in the notes section via the command "file memo".

4.1.7 Run Script. Command = "run <script name>". Runs a script located in the "windowsApp/commands" directory. Script name should not include file extension. Returns a success/failure message.

5 Limitations and Requirements

The hardware requirements are currently firm and unchangeable. The mobile application is designed only for Android and the desktop application uses Windows Native libraries and API. Conversions in the desktop application to support Linux and MAC, however, are relatively simple and could be quickly done in further pursuit of expanding this work.

The network limitations are also unavoidable. Network failures are unavoidable and could potentially cause the system to crash and need to be restarted, but the design choices made and mentioned in section 3 aim to minimize this risk. Additionally, due to the nature of the TCP connections, there is another major limitation. In order for the mobile application and desktop application to communicate when connected on different networks, the port that is used for the server must be port forwarded on the desktop machine beforehand. Otherwise, the application only works when the two systems are on the same network.

CONCLUSION AND ACKNOWLEDGMENTS

IoT Voice Assistant started off as a simple idea that would solve the small inconveniences in our day to day lives and quickly became a practical tool that people may make use of on a routine basis. The ability to communicate and essentially control a desktop computer via voice commands without needing to be at home could prove priceless in a technologically centered and driven society.

Since our group was new to both Windows-based Network Programming and Android Programming, we sought out public resources for learning the basics and fundamentals of these two topics. Open source projects and tutorials helped us build our knowledge foundation so that we could implement our project successfully. Additionally, we would like to thank Professor Mo Sha and course T.A. Junyang Shi for being reliable resources over the course of the semester.

REFERENCES

[1] https://developer.android.com/guide

[2] https://docs.oracle.com/en/java/

[3] https://docs.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2

[4] https://docs.julialang.org/en/v1/

[5] https://developer.android.com/studio

[6] https://code.visualstudio.com/

[7] https://atom.io/