1b. Live Video Streaming Application Category 1: Building New Internet Applications

Sahithi Kodali
kodali1@purdue.edu

ABSTRACT

This project presents a comprehensive live video streaming platform (web-based GUI and Android application) that facilitates seamless communication between remote servers and mobile devices through real-time video transmission. The system comprises a robust server-side infrastructure and cutting-edge mobile application developed using Python and the Kivy framework. Moreover, we integrate a new design based on a compression technique that improves the network performance compared to the baseline design implemented, and evaluate their performance.

1 PROJECT TOPIC AND BACKGROUND

In our interconnected world, mobile devices have become indispensable communication and information-sharing tools. Despite their widespread use, there is a growing demand for sophisticated applications that enable real-time video streaming. This project addresses this need by focusing on the development of a cutting-edge live-streaming application for smartphones. The application is designed to seamlessly transmit live video captured by the phone's camera to remote servers, bridging the gap between users and their intended audience or recipients. Commencing as a web-based Graphical User Interface (GUI) implementation, the project has evolved to encompass the creation of a dedicated Android application. This transformation reflects the dynamic nature of the technological landscape and the increasing preference for mobile-centric solutions.

1.1 Objectives

- *Develop a robust server-side infrastructure:* Establish a reliable server-side architecture capable of receiving, processing, and displaying live video streams transmitted from mobile devices.
- *Create user-friendly web GUI and mobile application:* Develop an intuitive web GUI using the tkinter library, providing users with a straightforward interface to initiate and terminate video streams. Concurrently, design a mobile application using the Kivy framework, incorporating functionalities such as capturing, processing, and transmitting video frames.
- *Deployment on Android devices:* Generate an Android Application Package (APK) file to facilitate effortless deployment on mobile devices. Ensure the APK satisfies all dependencies, requirements, and permissions necessary for the application's seamless functionality.
- *Explore designs to improve network performance:* Enhance the existing algorithm to optimize network performance by implementing advanced techniques such as data compression, dynamic bitrate adjustment, and other innovative approaches to enhance the overall efficiency of the live video streaming platform.

In section 2, the design and implementation details of the server and client-side architecture of web GUI and Android applications are discussed in detail, along with elaboration on the new design algorithm.

2 DESIGN AND IMPLEMENTATION

This section discusses the design and implementation details of server-client architecture implemented in this project.

2.1 Server Architecture

The server-side architecture is designed to efficiently receive, process, and present live video streams from mobile devices. Employing the User Datagram Protocol (UDP) for communication between the server and client reflects a strategic decision to optimize the efficacy of live video streaming. UDP, distinguished from the more robust Transmission Control Protocol (TCP), operates as a connectionless and inherently unreliable protocol. Despite lacking the intrinsic mechanisms for reliability and error recovery inherent in TCP, UDP stands out due to its advantages, particularly in real-time streaming scenarios such as live video transmission.

Key among these advantages is the expeditious transmission of data facilitated by UDP's connectionless nature, thereby excelling in providing swift and responsive content delivery to end-users. Furthermore, UDP minimizes overhead, as it excludes error-checking and acknowledgment mechanisms present in TCP. This reduction in overhead enhances the overall efficiency of data transmission, making it particularly well-suited for this project's objectives.

Although UDP inherently lacks reliability, the server-side architecture can mitigate this limitation by implementing robust measures at the application layer to ensure dependable data transfer. Since this live streaming is not confidential

and need not be reliable, like emails and messages, we can use the UDP framework for efficient communication.

While the above are the design justifications, we shall now discuss the implementation of the server side. A UDP socket is created, binding it to a specific IP address and port (in this case, IP: "10.0.0.155", port: 6666) based on the network the server is running on. The server then enters a loop and receives video data frames (as a sequence of images) from the client. The data, transmitted as bytes, undergoes an unpickling process, transforming it into an image for display on the server end. For this purpose, the OpenCV [1] library is used for its well-known image/video processing functions. The server also tracks performance metrics, including latency, total data received, and network usage. The server continually checks for a stop signal from the client to stop streaming and close the established socket connection. Finally, the server-side implementation concludes by computing and displaying average latency and average received data, providing insights into the server's performance during the video streaming.

2.2 Client Architecture

The client-side architecture, though, has a similar design for Web GUI and Android Applications; they differ in how the GUI is built. The client also leverages OpenCV functions to process the video stream.

2.3 Web GUI-based implementation

The Web GUI client, crafted with the tkinter library, provides users an accessible platform to initiate and terminate video streams effortlessly. The interface boasts functionalities like starting and stopping streams, allowing users to engage in real-time video communication effortlessly. A snap of the GUI with the start and stop stream buttons can be seen in Figure 1.

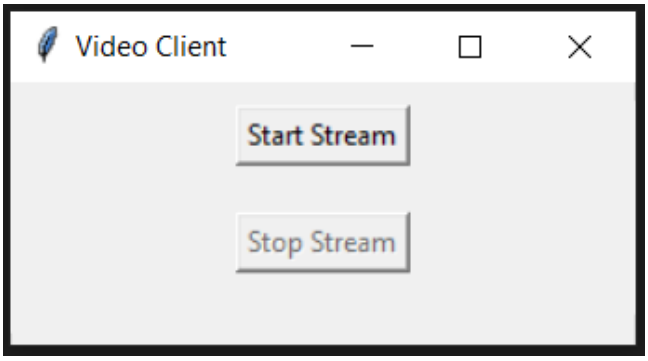


Figure 1: Snapshot of the web-based GUI

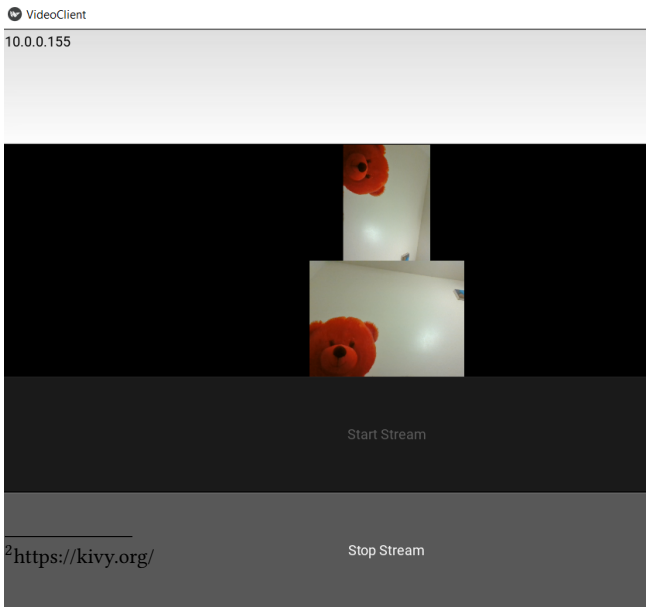
[1]https://opencv.org/

Upon initiating the stream by clicking the start stream button, a UDP socket is initialized for data transmission to the server, and the camera starts to capture video using OpenCV's VideoCapture module, given the camera index. The client captures video frames from the device's camera, encodes them as JPEG images, and pickles them into bytes to transmit the data to the server via the socket, and the GUI is updated. When the stop stream button is clicked, the stream is stopped, and the GUI is closed. Finally, the client-side ends by computing and displaying average latency and average received data, giving the performance metrics in real-time.

2.4 Android app implementation

The Android application client architecture is similar to the web-based GUI implementation. The GUI design is the main difference between the web-based client architecture and the application. Android app-based client GUI is built using the Kivy [2] – an open-source Python framework – which provides cross-platform functionality, python integration, touch-based functionalities, and excellent development workflow, making it ideal for building applications leveraging its capabilities. Moreover, it also provides compatibility with the buildozer tool, simplifying the process of deploying Kivy-based applications on mobile devices, discussed in detail in 2.6.

The GUI of this application has a user-friendly interface featuring "Start Stream" and "Stop Stream" buttons for intuitive control. The application also offers visual feedback, displaying the live video feed on the device's screen providing users with a real-time glimpse into the transmitted content. The app establishes a connection with the server when the user enters the server IP address. Figure 2 shows the snapshot of the Android application GUI.



[2]https://kivy.org/

Figure 2: Snapshot of the Android application GUI

Once the connection is established, when the user clicks the start stream, the application captures frames from the device camera, encodes them as JPEG images, pickles them, and converts them to bytes to transmit the data to the server via a UDP socket. The GUI is updated periodically for every 1/30th of a second as scheduled by the schedule interval of the Clock module. Once the stream is stopped, the server ends, the video stream is stopped, and the connection is closed. The Android client meticulously monitors performance metrics and provides insights into latency and total data sent during the streaming process.

2.5 New design implementation

The newly implemented compression technique addresses optimizing network usage in the real-time video streaming platform. The system can reduce the amount of data transmitted over the network by leveraging the "zlib" library, renowned for its efficient data compression capabilities.

Upon capturing video frames, the client employs the "zlib. compress" function to compress the pickled frame data before dispatching it to the server. This compression step plays a pivotal role in minimizing the payload size, thereby conserving bandwidth and enhancing the overall efficiency of data transmission.

A corresponding mechanism is in place at the server end to handle the compressed data. The received data undergoes decompression using the "zlib.decompress" function, restoring the original data structure. Subsequently, the bytes are decoded to reconstruct the video frames, ensuring the visual content is accurately presented. Finally, the performance metrics are computed to compare with the baseline algorithm and evaluate the performance, discussed more in the section 3.

By integrating the "zlib" library into the compression process, the platform attains a streamlined approach to data optimization. This mitigates network congestion, lowers latency, and provides a more responsive and seamless real-time video streaming experience. The implementation aligns with contemporary best practices in data compression and reflects a commitment to enhancing the platform's performance, particularly in scenarios where network resources are limited or need to be conserved.

2.6 Deploying Android Application

The pivotal tool utilized for deploying the client code as an Android application is "Buildozer [3]," a powerful and versatile tool specifically designed to package and deploy Kivy applications on Android devices seamlessly. This deployment methodology ensures that the live video streaming application is easily accessible on mobile devices, providing users with a convenient and portable solution.

Buildozer operates as a bridge between the Kivy application and the Android platform, streamlining the complex task of creating standalone packages compatible with Android devices. Its capabilities extend beyond mere compilation, encompassing the handling of dependencies, permissions, and various configuration settings crucial for the successful deployment of the application.

Central to the Buildozer workflow is the "builddozer.spec" file, a configuration file that serves as a blueprint for the deployment process. The critical settings, such as required permissions, dependencies, and other application-specific parameters, should be specified within this file. Apart from the required libraries for this app to work, the permissions required for this application are access to the phone's camera, Internet, and external storage read/write. This configurability ensures that the Android package aligns with the intended functionality and environmental requirements.

Building Android Package: Commands

Executing the command "builddozer -v android debug" initiates the process of building the Android package (APK). This command, executed from the directory where the client code resides in the "main.py" file, triggers Buildozer to compile, package, and optimize the Kivy application for the Android platform. The "-v" flag provides verbose output, offering insights into the build process.

Once the APK (Android Application Package) file is generated, it can be installed on Android devices effortlessly. The Android Debug Bridge (ADB) is pivotal in this phase. ADB facilitates debugging by establishing a connection between the connected computer and the mobile device. Developers can leverage USB debugging, accessible through the device's Developer Mode settings, to enable real-time debugging and testing of the application directly on the device. The log generated via ADB when running the application on the phone facilitates easy debugging by providing a detailed crash/error report, aiding developers in identifying and addressing issues that may be causing problems in running the app. The first build takes about 3-4 hours, based on the application size and depending libraries.

3 PERFORMANCE EVALUATION

This section presents a comprehensive performance evaluation of the live video streaming platform. The evaluation aims to assess key metrics such as latency, network usage, and overall system efficiency. The performance of the Android application in both the cases with compression and without compression has been evaluated.

[3] https://buildozer.readthedocs.io/en/latest/

Figure 3 and 4 showcase the server and client performance metrics using the original without a compression-based algorithm. The average latency at the server is 0.00058594 seconds, and the average network usage at the server is 8223.5714 bytes. The average latency at the client is 0.000395 seconds, and the average network usage at the client is 8223.5714 bytes.

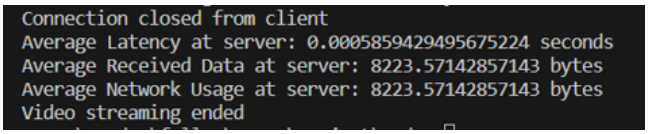


Figure 3: Performance metrics on server end using no-compression algorithm

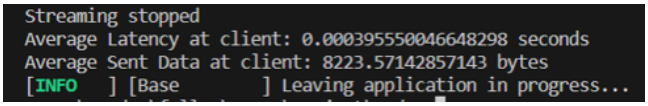


Figure 4: Performance metrics on client end using no-compression algorithm

Figure 5 and 6 showcase the server and client performance metrics using the new compression-based algorithm. The average latency at the server is 0.000519 seconds, and the average latency at the client is 0.000287 seconds. The average network usage at the server is 5135.486 bytes, while the average network usage at the client is 7493.900 bytes before compression. This shows that the number of bytes transmitted to the server from the client side has been compressed, leading to less network usage and, thus, reduced latency while streaming continuously.

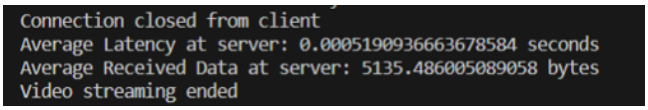


Figure 5: Performance metrics on server end using the new compression algorithm

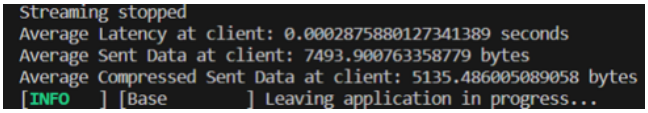


Figure 6: Performance metrics on client end using the new compression algorithm

While these are some performance metrics chosen for this project, other network performance evaluation metrics, like frames per second, bandwidth usage, etc., can also be considered for further evaluation.

4 FUTURE WORK AND CONCLUSION

In conclusion of this project, we have discussed the design and implementation of the server-client architecture, along with deploying an Android application on a mobile device for building a live video streaming platform, and evaluated the performance of different algorithms.

For future work, other network-enhancing algorithms like adaptive bit-rate streaming, dynamic frame rate adjustment, etc., along with data compression techniques, can be explored. Further, more advanced GUI features can be developed to enhance the video streaming experience for the users.