ASCII Chat

Lucas Chen (lmc336), Danny Qiu (dq29), Chesley Tan (ct353)

System Description

Vision

For our final project, we will be building a terminal-based peer-to-peer video chat application with end-to-end encryption. It will feature text-based video rendering, audio support, and messaging support. We envision that our application will have diverse applications, such as usage in video conferencing or collaboration in bandwidth-constrained or graphically-constrained environments.

Features

Our program will support/have:

- Colorized text-based video-chatting with up to 4 clients
- End-to-end encryption of video, audio, and messages
- Ephemeral messaging capability - messages are not permanently stored and are encrypted during transmission
- Audio/Voice chat
- Cross-platform compatibility

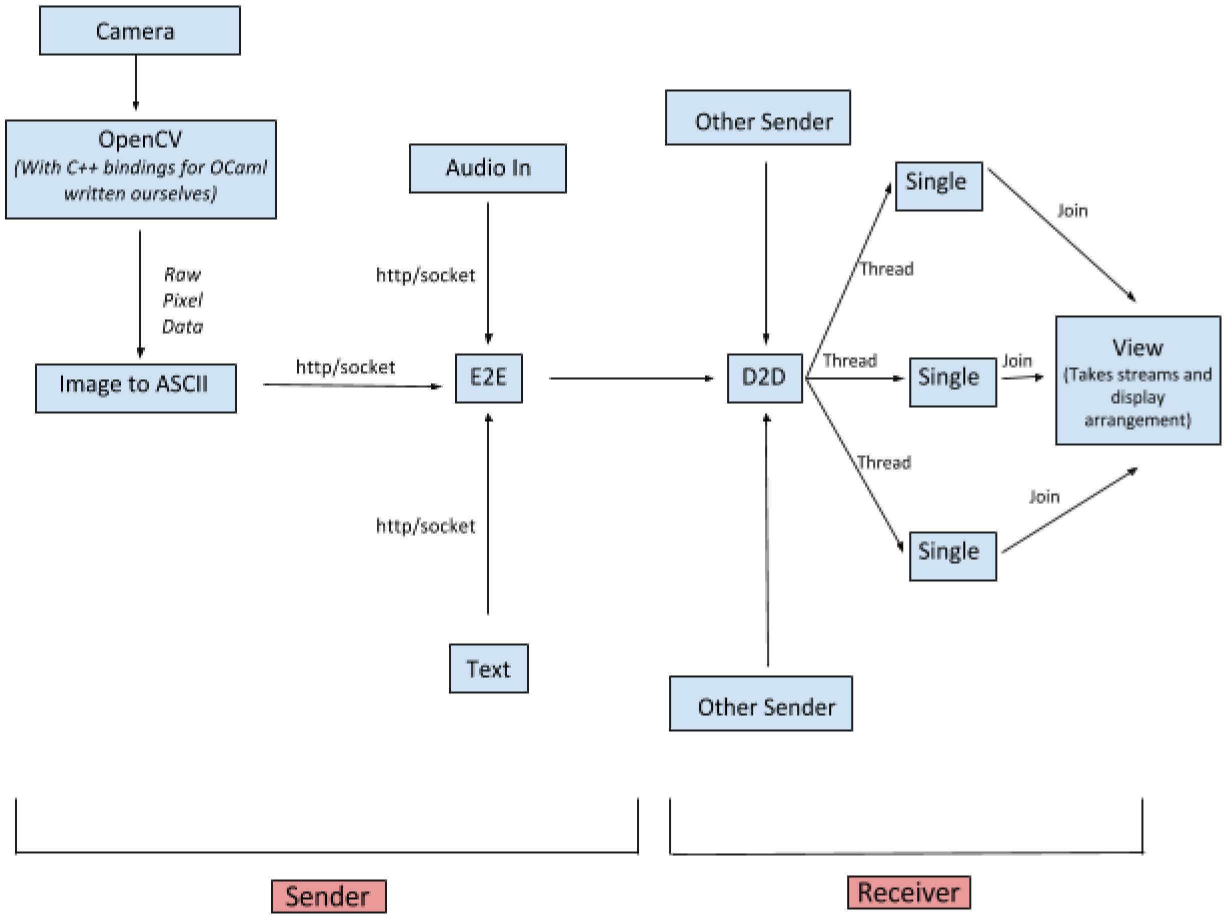
Description

To video-chat with peers, our program will accept an IP address or URL of the hosting user, who acts as the gateway for other users to join a video conference. Each user within the video conference will act as both a server and a client in that each user's server sends its video to other clients, as well as receives video from other clients. The video will be displayed in ASCII, where the variation of characters will be used to represent differing local features within the image. In addition, the color capabilities of the user's terminal will be utilized. For minimizing bandwidth, colors in the video will be constrained to a set of 256 predefined colors that span a wide range of the color spectrum. Each user's server will locally process the video input from the user's camera to generate an ASCII representation of the image, as well as compress the colors in the image and resize the image appropriately. To support video conferencing with up to 4 clients, the user interface will automatically re-orient itself to maximize screen usage and equalize screen space for each video stream. Users will also be able to send text messages to each other, as well as transfer audio from the user's microphone to other users. Finally, our program respects user privacy as all data that is transmitted will be end-to-end encrypted, and no data will be persistently stored.

Changes Since Initial Design

Due to the time constraints of the project, we were unable to implement audio streaming, and thus only video chat and text messaging is implemented. Similarly due to time constraints, we were not able to implement an established end-to-end encryption protocol, such as OTR (Off-the-Record) or the Signal protocol. However, we did add AES encryption, using a pre-shared key chosen by the clients, for all data transferred between clients. We initially planned on utilizing a terminal manipulation library, such as ANSITerminal or lambda-term, but we experienced difficulty in adapting it for the purposes of our program, which needed to print a large amount of text output many times a second while managing multiple threads. In addition, we found that lambda-term lacked functionality we needed, such as the ability to print arbitrary text, including ANSI escape sequences, into a text box and automatically wrap the text. As a result, we decided to implement the terminal manipulation ourselves within our view module.

Architecture



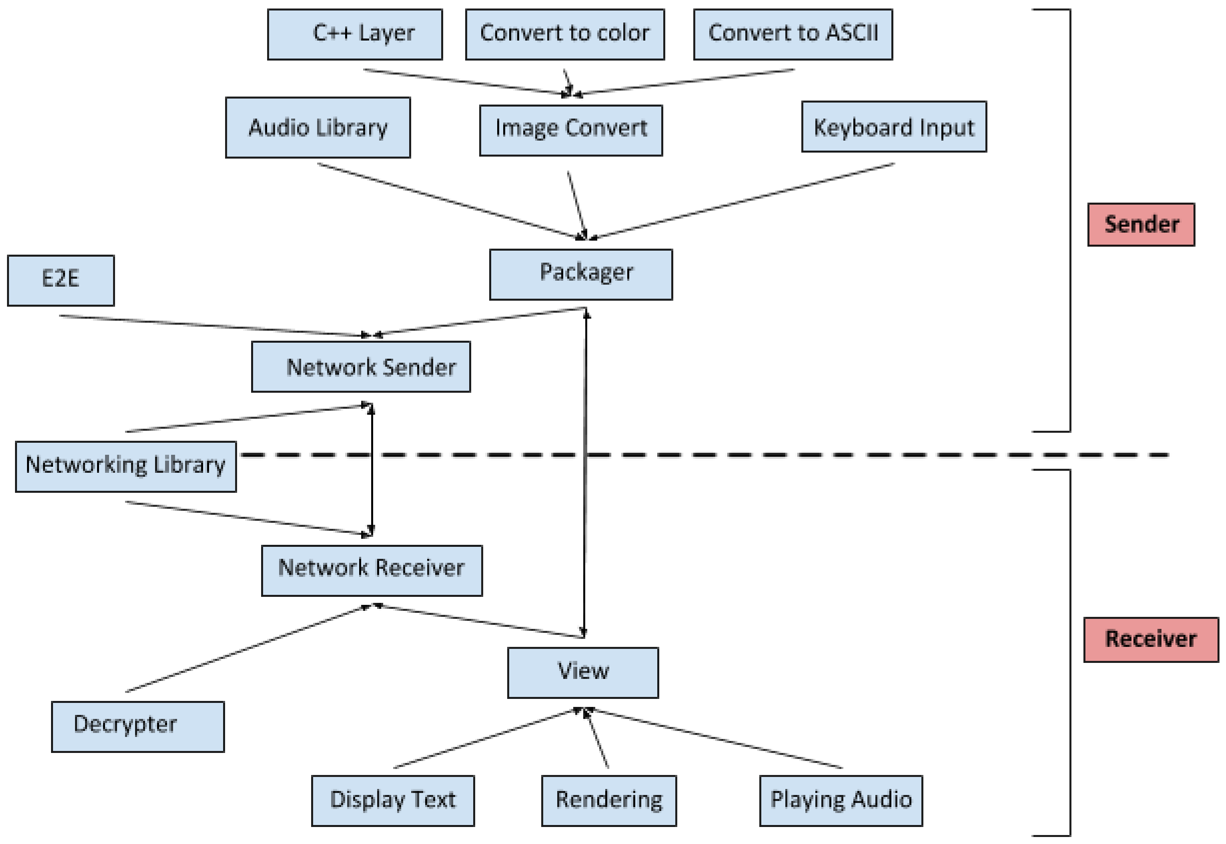
Our system uses strong MVC design, in which we make sure to decouple different components and put similar functionality together. The model is represented by the video, text, and audio data. The controller is represented by the E2E (end-to-end encryption) which takes the model data and encrypts it for sending, and the D2D (end-to-end decryption) which decrypts the encrypted data and reconstructs the model. The view is represented by the component which combines the decrypted data and displays it on the terminal. This partition of the system's components allows us to develop modules for one component in parallel with those of another component, because their interfaces will be loosely coupled. In addition, this enables us to test the implementation of one component, without relying on the implementation and correctness of other components. It also allows the system to be scalable, since each client will act as both a sender and a receiver, leading to the desired decentralized ASCII video chat. This peer to peer architecture emphasizes strong networking principles to lower the bandwidth used and decrease the latency for smooth video chatting. It will also use the pipe and filter model to ensure that the client has no performance delay.

System Design

The general functional components of this system include:

- **Convert to ASCII:** Renders the the raw pixel data as ASCII characters
- **Convert to Color:** Assigns color to each of the ASCII characters
- **Audio LIbrary*:** Takes in constant audio feed from system microphone
- **Image Convert:** Combines all the layers and feeds the stream of ASCII characters
- **Keyboard Input:** Takes text input from user keyboard.
- **Packager:** Combines all audio, visual, and text information.
- **E2E:** Encrypts all the data that is sent
- **Network Sender:** Sends data to host
- **Network Receiver:** Receives data from the network socket
- **Decrypter:** Decrypts data that is received
- **Viewer:** Processes received information and passes it on as text, visual data, or audio
- **Display Text:** Displays the next available text from all clients in chatbox.
- **Rendering:** Displays the next available image from all clients
- **Playing Audio*:** Passes audio onto system speakers.

*Audio was not implemented as part of this project. See *Changes to Initial Design*



Module Design

`cv.ml` - bindings for OpenCV image retrieval function, functions and modules for converting an image to ASCII and colorizing an image

`main.ml` - controller in MVC structure, accepts keyboard input from user and updates state accordingly; Entry point for the program

`messaging.ml` - functions for updating the chat history with new messages received

`network.ml` - client and server implementations for users, handles setting up peer-to-peer connections between participants and sending and receiving packaged data

`package.ml` - functions for packing and unpacking data into a single package, handles serialization/deserialization, compression/decompression, and encryption/decryption of packaged data `state.ml` - model in MVC structure, functions and data structures for maintaining the internal state of the program

`test_main.ml` - entry point for OUnit tests

`test_messaging.ml` - unit tests for the Messaging module

`test_state.ml` - unit tests for the State module

`utils.ml` - Utility functions for global use, includes bindings to C interface functions for executing code that must be run at a lower level

`view.ml` - view in the MVC structure, handles structuring output and rendering it with dynamically rendered custom layouts depending on the number of users in the chat

Data

In accordance with the privacy-oriented design of the application, no data is persistently stored by the application to storage outside of memory. Because the video chat implementation is text-based, videos are represented as a sequence of frames, which are represented as encrypted strings (using the end-to-end encryption scheme) when being transferred in a peer-to-peer fashion between clients. Textual data is transferred along with the ASCII video and encrypted together in a single package, which is serialized before being encrypted for transfer over the network. Audio will be transferred between clients in a binary format.

We used a hash table as a method of storing the most recent data package associated with a given peer, so that we can perform a constant-time indexing operation given the peer whose data we want to look up. We used arrays to store image data so that we can process (convert to text) them with reasonable efficiency, and we also implemented a faster string module whose buffer was backed by a byte array.

Our efficient implementation of multi-peer chats, which uses a decentralized network where clients are all connected to each other, allows for stability in the video chat program. Each client acts as a server to notify other clients of the change in the network state. If a client's connection is dropped, the network will respond immediately to this change.

External Dependencies

We leveraged the OpenCV library in order to read camera images from the user's webcam. We chose to use OpenCV, because of its cross-compatibility, portability, and performance. Because there does not exist any OpenCV bindings for OCaml, we needed to write a small API in C++ to read frames from the user's camera and return the image data as a primitive data type, and then use the ocaml-ctypes library to call the C++ function from within OCaml. We purposefully avoid doing a large amount of work within the C++ component, where we have the benefit of OpenCV's extensive API, in order to focus on using OCaml. In addition, we needed to utilize ocaml-ctypes to create interfaces to access lower-level C functions that OCaml lacked, such as those to interact with the system's network devices. These bindings can be found in `utils.ml`. We used a number of packages available to OPAM, such as Yojson to serialize data packages into strings so that they could be sent over the network, Lwt to handle cooperative and preemptive multithreading, cryptokit to perform encryption and compression, and oUnit to perform unit testing.

Testing Plan

We tested our code by writing unit tests using OUnit. Specific functionality we tested was the messaging module, and the state module. In the messaging module, we tested the interface functions exposed by the `messaging.mli` in order to ensure that the functions it provided produced the expected behavior for tracking the chat history given varying timestamps for text messages. We also used unit tests to test the functionality of the state module and its exposed functions in `state.mli`, in order to ensure that the logic for updating the data structure mapping data packages to unique users was correct, and successfully updated the stored data when new data was received. As of the time of submission, there are no known bugs.

Division of Labor

Our work was divided as follows:
Chesley Tan: Wrote the CV module, which contains the OpenCV bindings and requisite ocaml-ctypes interfaces to get images from the user's webcam, the main module which contains the main application loop and initialization logic, the package module, which contains the functions for serializing, deserializing, compressing, decompressing, encrypting, and decrypting data packages, the state module, which contains the functions and data structures for storing the internal state of the application, and the view module, which contains the functions for rendering the custom UI layouts of the images and textual data. Also, worked on Makefile to support building shared libraries. Added installation and submission generation scripts. Estimated time worked: 65 hours.
Danny Qiu: Wrote the networking module, which contains the functions for establishing a peer-to-peer network between chat participants, the client and server, and the functions for sending and receiving packaged data. Also worked on the Makefile to support cross-compatibility builds across Unix systems. Estimated time worked: 30 hours.
Lucas Chen: Wrote the messaging module, which contains the functions and data structures for combining the received and user-input messages to obtain the chat history, and wrote unit tests to check for edge cases. Estimated time worked: 20 hours.