CS 407 Design Document

Mercury – Team 11

Kai Tinkess (ktinkess@purdue.edu)
Kris Leungwattanakij (kleungwa@purdue.edu)
Alex Hunton (ahunton@purdue.edu)
Christopher Lee (lee3880@purdue.edu)
Leonard Pan (pan353@purdue.edu)

Purpose

When determining what project we wanted to focus on, we made the conscious decision to choose a project and architecture that afforded us opportunity for growth. This led us towards the idea of a decentralized desktop app to differentiate our portfolio from the web apps that we had done in the past. Designing a project explicitly in the free and open-source software field would move our priorities from profit margins and audience to utility.

Video streaming is a field that has exploded in probability over the last decade. After YouTube first took off with video hosting, live streaming entered the scene and exposed an additional part of the market. Live streaming tends to support younger users congregating around a particular niche or community. Video hosting also attracts those users, but additionally allows for tutorials and educational content.

The existing offerings of this domain have become progressively corporate and anti-consumer as they mature. Despite this, not many systems have been developed that can be self-hosted. The largest open-source competitor is Owncast, which allows for users to connect their own servers to video streams. However, it still routes through a configured website and serves web traffic. This prevents true privacy.

Mercury will solve all of the above problems by allowing for a self-hosted stream through a desktop application to send data application to application to another client. Because it will be FOSS, there will be no profit incentive or design made around principles beyond efficient streaming. Additionally, the open-source nature and direct communication ensures privacy. The intended user base consists of small friend and study groups who prioritize guaranteed performance and privacy. The peer-to-peer approach, especially over a LAN, is innately appealing to advanced internet users who desire more control over their online communication.

Design Outline

High Level Overview

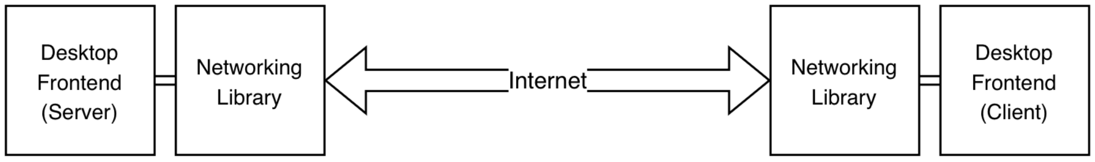
Mercury is an open source desktop app that allows users to host and connect to streams through peer-to-peer networking. The frontend is designed using Qt Widgets for a native and performant experience. The networking is done over application-level protocols we specify, built on top of Qt's TCP/UDP socket wrapper. These two components will be integrated using the slots and signals system design Qt uses.

Useful terminology:

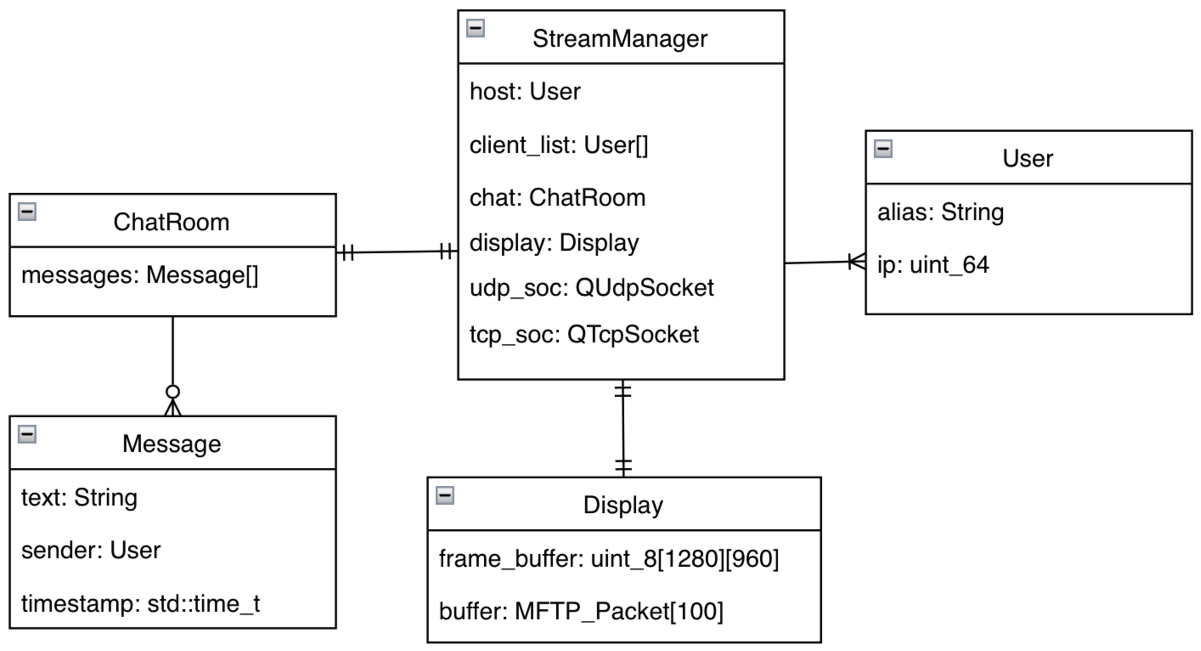
- **User** – anyone using Mercury. Every user has an **alias**, which acts as their display name.
- **Host** – a user who has started a server on their machine and connected an audio/video source to it
- **Client** – a user who has connected to someone else's server for the purpose of viewing their stream

System Components and Interactions

The two main internal components are the aforementioned frontend application and the underlying network library. The network library will be called upon for two purposes: to establish a connection for full-duplex data that requires reliability (such as the initial connection and messages), and to allow for audio/visual packets to be sent quickly for data that does not. In practice, one user will set their machine to act as a server. Any other user can then voluntarily become a client to that server and receive the data being streamed from it.



UML Structure



Design Issues

Functional Issues

How should a user specify the host they want to connect to?

A. Specific Users/Usernames
B. Web links
C. <u>ICANN Domain Names</u>
D. <u>IP Addresses</u>

Regardless of what information we want a client to be able to enter to connect, it is necessary that we have either an IP address, or something that can resolve to an IP address. In the future, it might be possible to encode this information through something like a web link, but for now, and for the purpose of as much anonymity and privacy as possible, only using IP/DNS is the best option.

Should Mercury include account services?

A. Yes - centralized database system
B. <u>No - viewer chosen aliases</u>

One of the core design principles underscoring this project is privacy and FOSS. A centralized account system would require us to violate both of those by distinguishing users who had registered with an authority, us. Consequently avoiding them entirely and not requiring any information on our users besides an alias is more consumer friendly and in-line with our philosophy.

How should hosts moderate viewers chat?

A. Hosts can block clients with specific IP addresses
B. Hosts can block clients with specific aliases
C. Moderation is unnecessary

Since we wanted to prioritize the safety and security of our users, we decided that hosts should be able to moderate their own stream, cutting out option C. Option B was the next thing we considered, but malicious clients could easily reconnect from the stream with a different alias. We ultimately decided to go with option A, as it would make it a lot more difficult for clients with bad intentions to reconnect to a stream.

Should Mercury prioritize a way for clients to engage with the host's video stream?

A. Yes, there should be an option to take "remote desktop" control of the stream
B. Yes, all clients should be able to draw "annotations" on the stream
C. Yes, the host should be able to enable drawing annotations for certain/all clients
D. No

We initially opted for option D but later decided against it, as we realized (through some research) that engagement was essential to streaming. This realization led us to option A, which we ended up cutting out, as there were already other tools on the market for remote desktop (i.e. teamviewer) and it did not align with the vision we had for Mercury. In the end, we decided on an "annotation" tool that would let the host/clients draw on the screen. After a long discussion, option B was cut out, as enabling annotations by default for all clients could result in a lot of confusion and chaos for larger streams, and so we opted for option C.

Should Mercury include an ethics agreement?

A. Yes
B. No

The peer-to-peer nature of Mercury means that it could easily be used in unethical and immoral ways which did not align with our goals and vision for the project. Therefore, we deemed that it was necessary to include an ethics agreement.

Non-Functional Issues

Which C++ framework should we use to develop the application?

A. Qt
B. wxWidgets
C. SDL and ImGui
D. Custom

For professional C++ multi-platform applications, Qt is easily the most widely supported, well documented, and helpful. WxWidgets is not as comprehensive, SDL is oriented at more rendering heavy graphics uses, and writing the entire project from scratch isn't feasible without our time constraint. Using Qt allows us to bypass some of the heavy lifting with frontend design. It also allows us to take advantage of other utilities they offer, such as their networking wrappers.

How should we accommodate our need for both connection-oriented and fast message-oriented communication?

A. Sacrifice speed and use a TCP connection
B. Sacrifice security and use UDP messaging
C. Establish a persistent TCP connection that sets up UDP messaging

Of the two transport protocols, UDP and TCP, neither of them are able to handle what we need. We require message-oriented communication for sending real-time audio/visual data. At the same time, we need connection-oriented communication to accommodate our requirement that connection requests and chat messages are delivered. As a result, we should first establish a TCP connection, and then send additional UDP datagrams when required.

Which audio/visual network protocol stack should we use?

A. UDP/RTP/Custom
B. UDP/RTP/RTSP
C. TCP/HTTP

Since we are not giving clients media control (e.g. pause, play, etc.), we do not need RTSP, eliminating option B. Option C was also eliminated as latency is a big part of our offerings, making TCP an unsuitable option. Therefore we need to use UDP and a generic implementation of some real-time protocol (RTP). The final application protocol for our stack will obviously be custom and specified later to meet the demands of the system.

Which start/stop connection network protocol should we use?

A. UDP/RTP
B. TCP/HTTPS
C. [✓ TCP/Custom + Chosen Security Protocol]

TCP will provide us with a stream-based, reliable connection that will allow two parties to communicate. As a result, both the viewer/host will reliably know when they have a connection between one another, and will be knowledgeable if either loses connection. We can then secure this connection with a chosen security protocol.

What security protocol would be the most appropriate for our use case?

A. SSL 3.0
B. [✓ TLS 1.3]
C. Custom

Security is one of the fields where reinventing the wheel comes with a high cost, as it sacrifices many of the complicated optimizations that researchers have developed. Consequently it behooves us to use the most recent and advanced security protocol, which in this case is TLS. SSL is outdated and not commonly used anymore.

Which implementation of SSL/TLS should we use?

A. Qt SSL Sockets
B. Qt Websockets
C. [✓ Custom with few external libraries]

Part of our goal for this project is to develop our understanding of essential components of the network layering stack. As a result, we feel that implementation of SSL/TLS would be beneficial for furthering our learning process.

For UDP/TCP sockets, which implementation should we use?

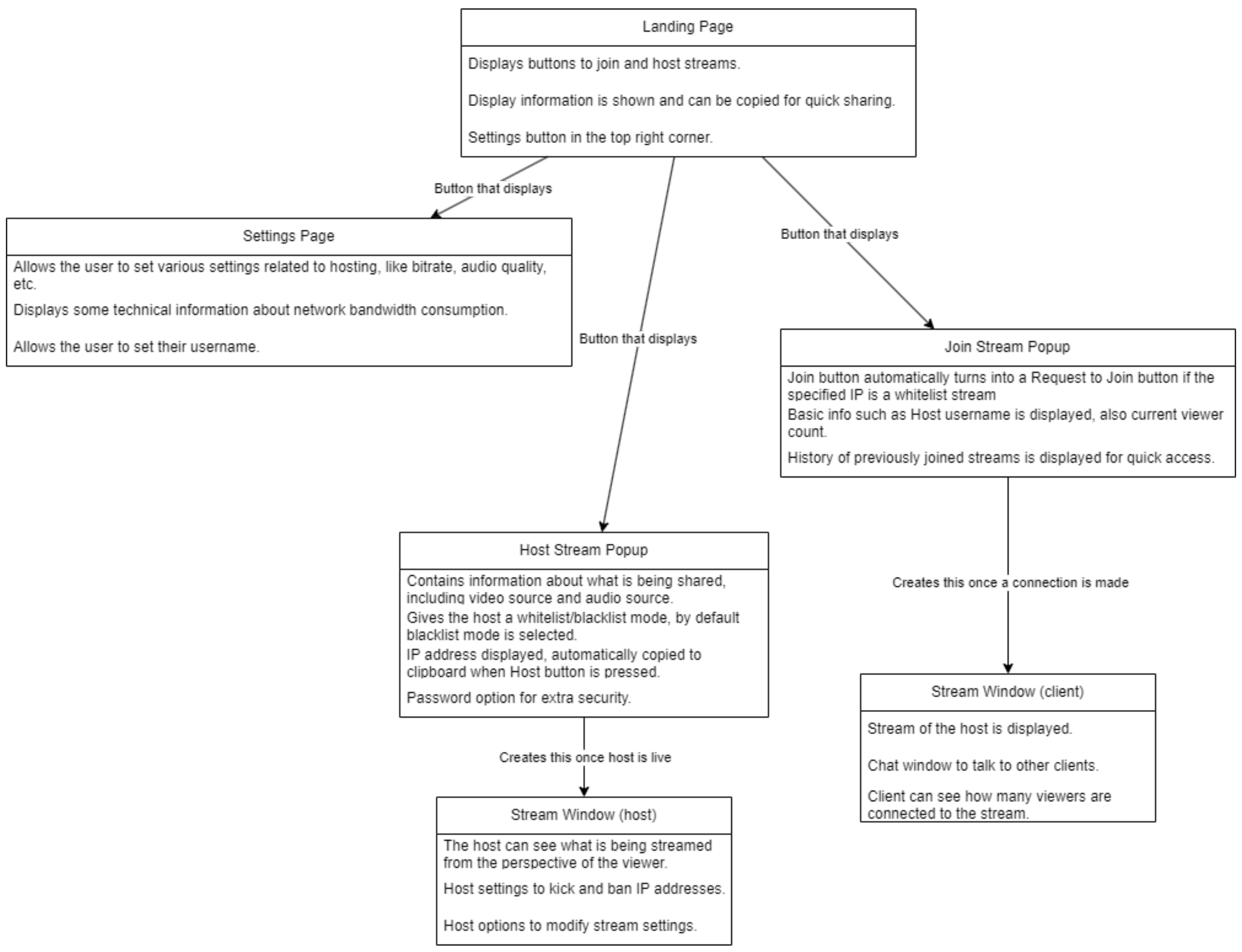
A. C Standard Library
B. Custom
C. [✓ QUdpSocket / QTcpSocket]

We decided to go with option C over A, as the Qt socket suite offered more interoperability than the C standard library, which would better suit our needs. We also considered building a custom socket library but decided against it, as we would be reinventing the wheel for a negligible difference in performance and functionality (for our needs).

Design Details

Frontend Design

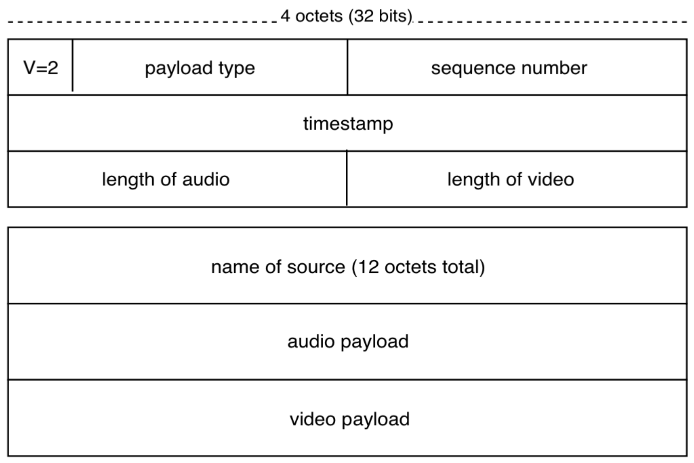
Mercury will be composed of one main window (the landing page) and several popups that form a hierarchical frontend. These are detailed at a high level in the following diagram. Qt takes care of precise formatting through their Grid styling, so the only real design decision is determining what callbacks (signals) are called when (slots).


Landing Page
Displays buttons to join and host streams.
Display information is shown and can be copied for quick sharing.
Settings button in the top right corner.
Button that displays
Settings Page
Allows the user to set various settings related to hosting, like bitrate, audio quality, etc.
Displays some technical information about network bandwidth consumption.
Allows the user to set their username.
Button that displays
Button that displays
Join Stream Popup
Join button automatically turns into a Request to Join button if the specified IP is a whitelist stream
Basic info such as Host username is displayed, also current viewer count.
History of previously joined streams is displayed for quick access.
Creates this once a connection is made
Host Stream Popup
Contains information about what is being shared, including video source and audio source.
Gives the host a whitelist/blacklist mode, by default blacklist mode is selected.
IP address displayed, automatically copied to clipboard when Host button is pressed.
Password option for extra security.
Creates this once host is live
Stream Window (client)
Stream of the host is displayed.
Chat window to talk to other clients.
Client can see how many viewers are connected to the stream.
Stream Window (host)
The host can see what is being streamed from the perspective of the viewer.
Host settings to kick and ban IP addresses.
Host options to modify stream settings.

Networking

Another important decision that needs to be made is how our networking protocols will work. Although we have decided on the protocol stack we will use underneath the application level, we need to specify what hosts and clients expect to receive from the other. As previously stated, we have two protocols: **MFTP (Mercury Fast Transfer Protocol)**, and **HSTP (Hermes Safe Transfer Protocol)**. The first will rely on UDP and a modified RTP to efficiently deliver audio/visual data, where loss/latency/jitter is acceptable. HTP, in contrast, will be used for everything else.

MFTP



MFTP will only require a single type of message. Notably, it will need a way to order the messages it receives in its buffer (sequence number and timestamp), and some way to identify the source. The rest is simply an audio and video payload. The V field corresponds to the protocol version.

HSTP

In contrast, HSTP will need to service a variety of needs. As a result, we will create a HSTP header that will prepend the actual message data – this header will identify the type of message. See:

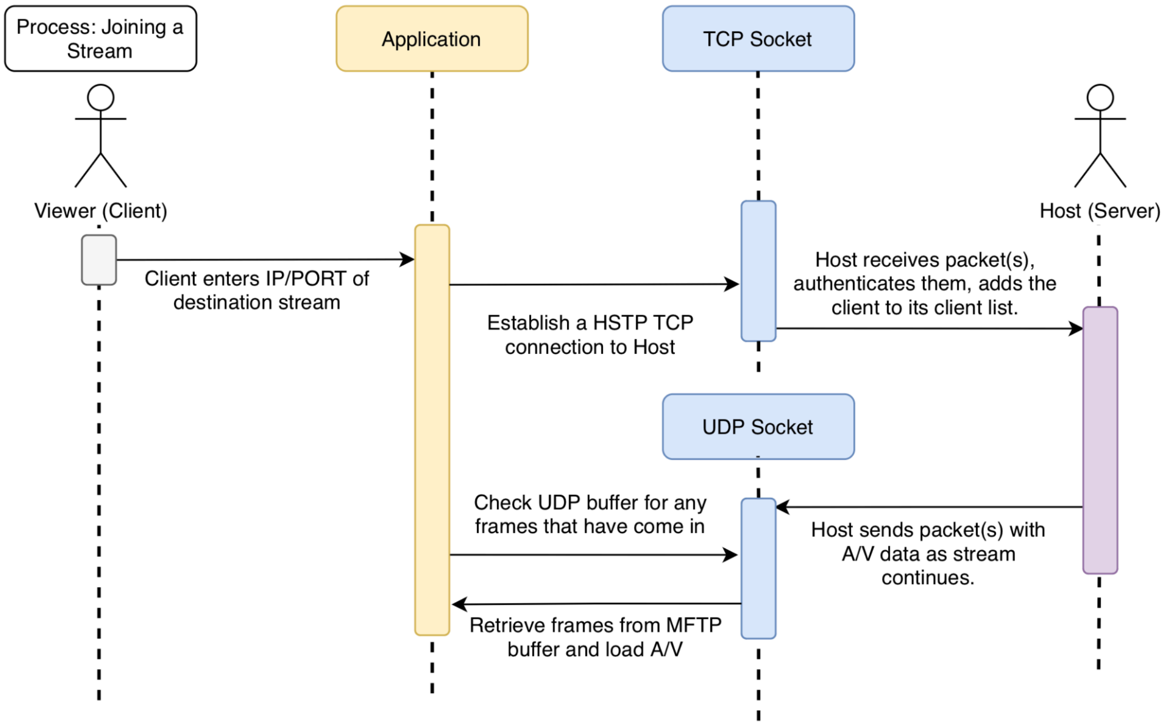
Type	Description	#
Establishing	The first and last message a client sends when connecting to a host. Allows the host to correctly maintain their viewer list.	1
Chat	The message sent whenever a client messages in the chat, or when the host forwards that message to every client.	2
Annotation	The message sent whenever a host or authorized client annotates the stream.	4

The fields these messages will each need are as follows:

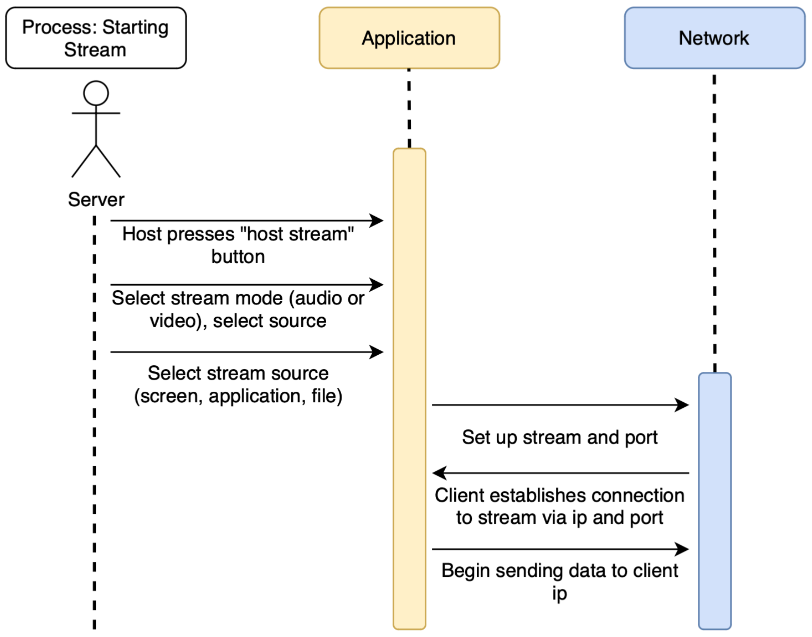
Header	Establishing	Chat	Annotation
• Type • Alias of sender	• Start or stop	• Message text • Timestamp	• Annotation Framebuffer

Sequence Diagrams

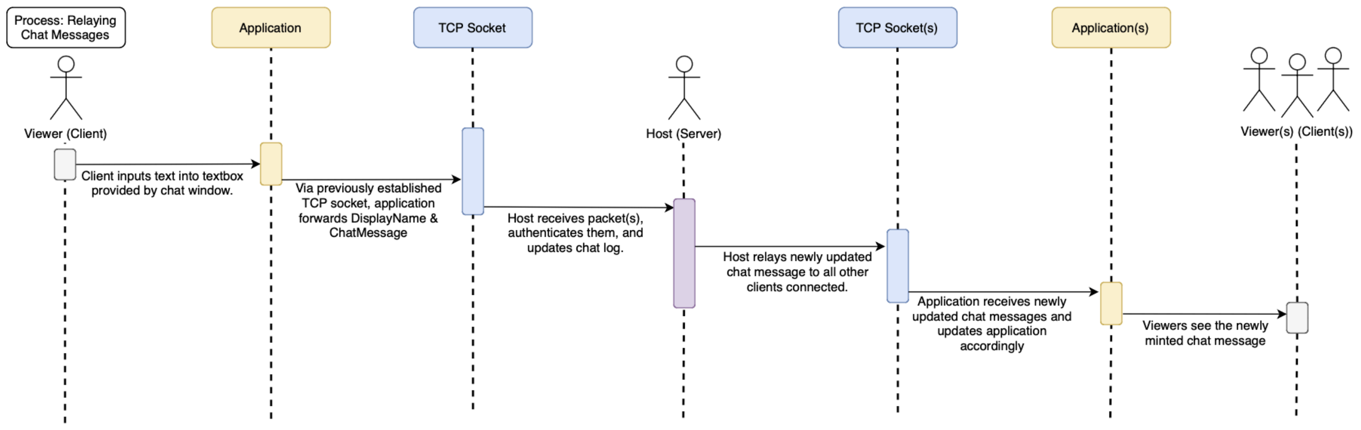
The client connecting to a host and receiving A/V frames:


Process: Joining a Stream
Application
TCP Socket
Viewer (Client)
Host (Server)
Client enters IP/PORT of destination stream
Establish a HSTP TCP connection to Host
Host receives packet(s), authenticates them, adds the client to its client list.
UDP Socket
Check UDP buffer for any frames that have come in
Host sends packet(s) with A/V data as stream continues.
Retrieve frames from MFTP buffer and load A/V

The host selecting a video source and streaming data to a list of clients:

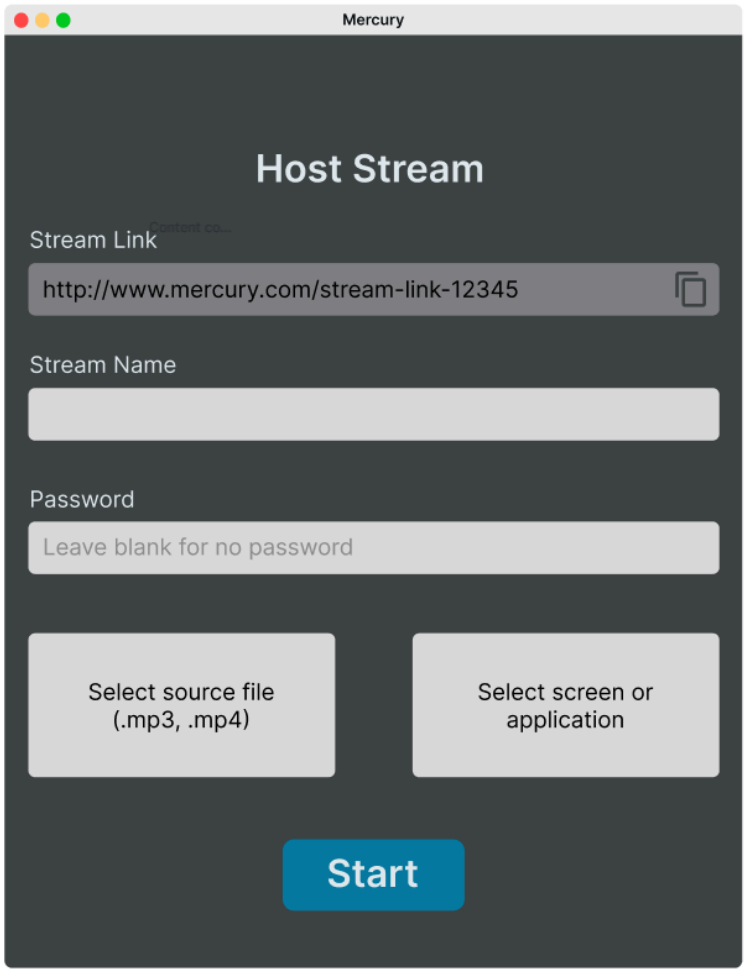

Process: Starting Stream
Application
Network
Server
Host presses "host stream" button
Select stream mode (audio or video), select source
Select stream source (screen, application, file)
Set up stream and port
Client establishes connection to stream via ip and port
Begin sending data to client ip

The host receiving a chat message from a client and displaying it, then forwarding it on to all other clients:


Process: Relaying Chat Messages
Application
TCP Socket
TCP Socket(s)
Application(s)
Viewer (Client)
Host (Server)
Viewer(s) (Client(s))
Client inputs text into textbox provided by chat window.
Via previously established TCP socket, application forwards DisplayName & ChatMessage
Host receives packet(s), authenticates them, and updates chat log.
Host relays newly updated chat message to all other clients connected.
Application receives newly updated chat messages and updates application accordingly
Viewers see the newly minted chat message

UI Mockups

Hosting Stream

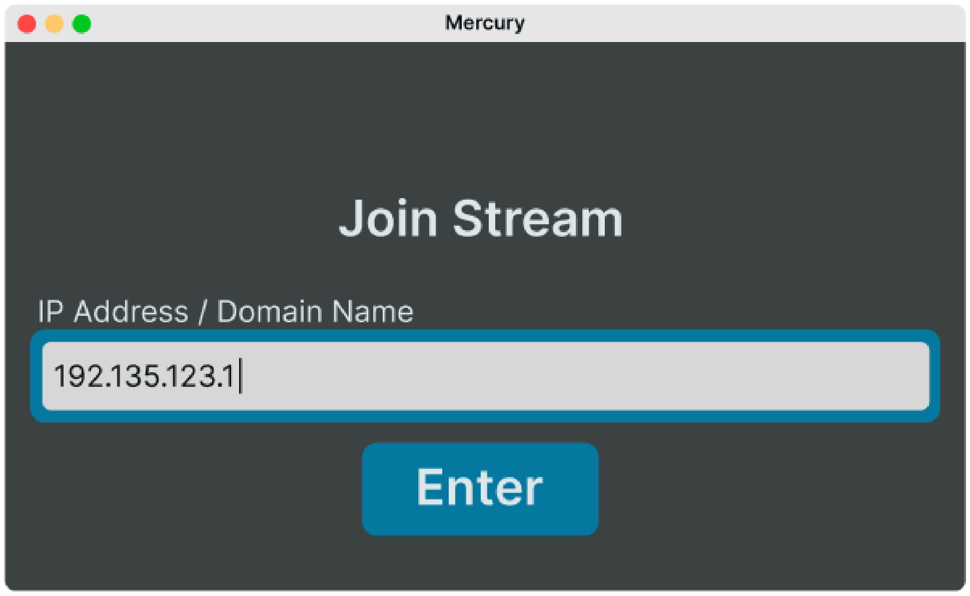


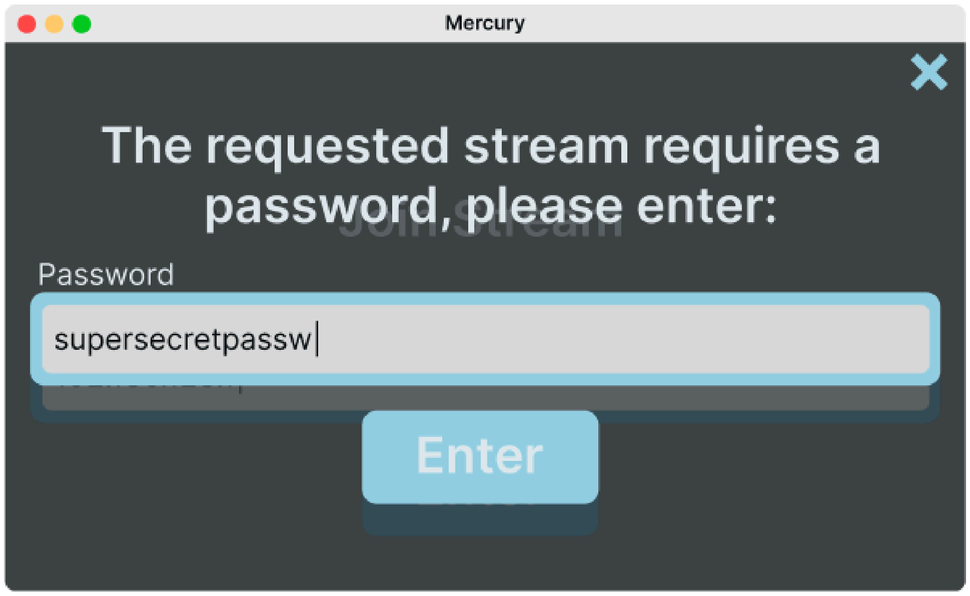
Stream Link

http://www.mercury.com/stream-link-12345

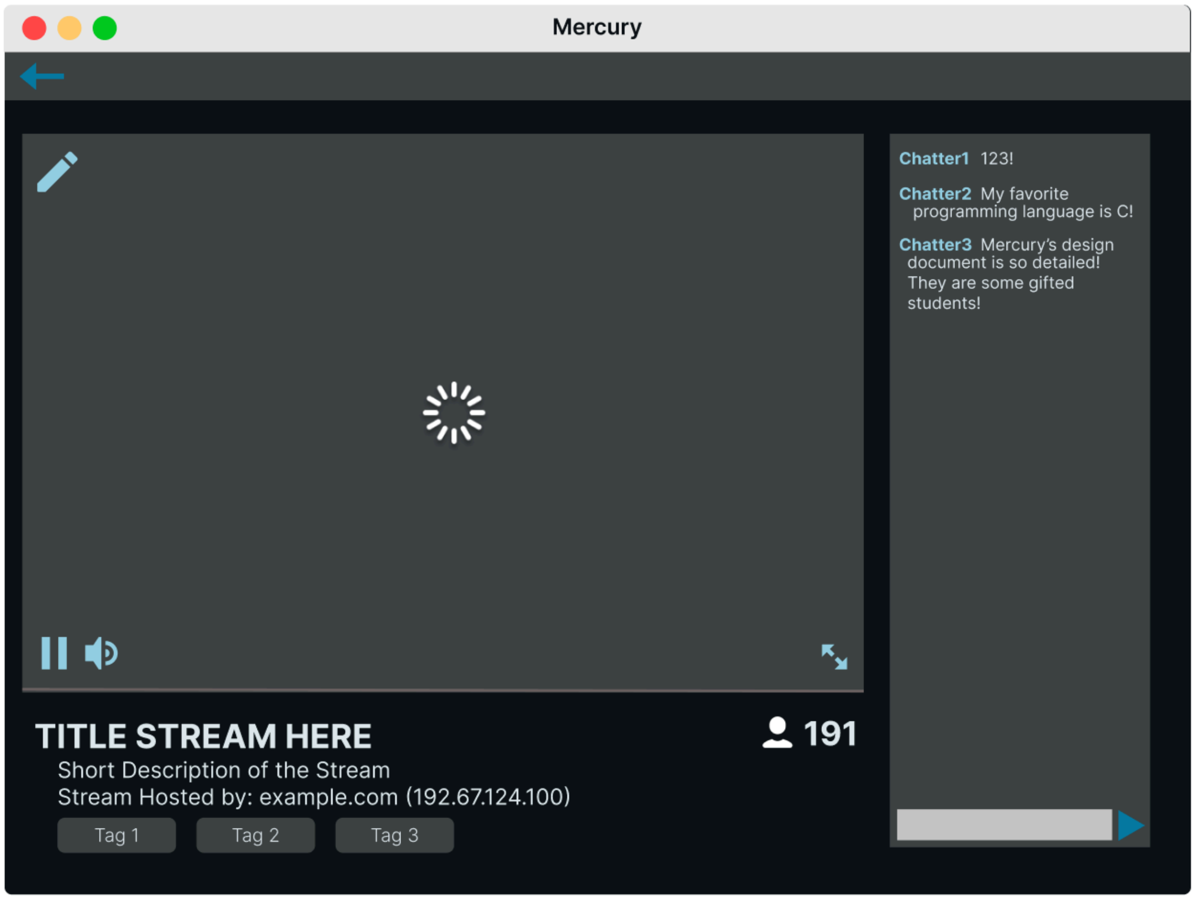
Mercury
Host Stream
Stream Name
Password
Leave blank for no password
Select source file (.mp3, .mp4)
Select screen or application
Start

Joining Stream





Client-side Annotations Enabled (by host)



Chatter1 123!

Chatter2 My favorite programming language is C!

Chatter3 Mercury's design document is so detailed! They are some gifted students!

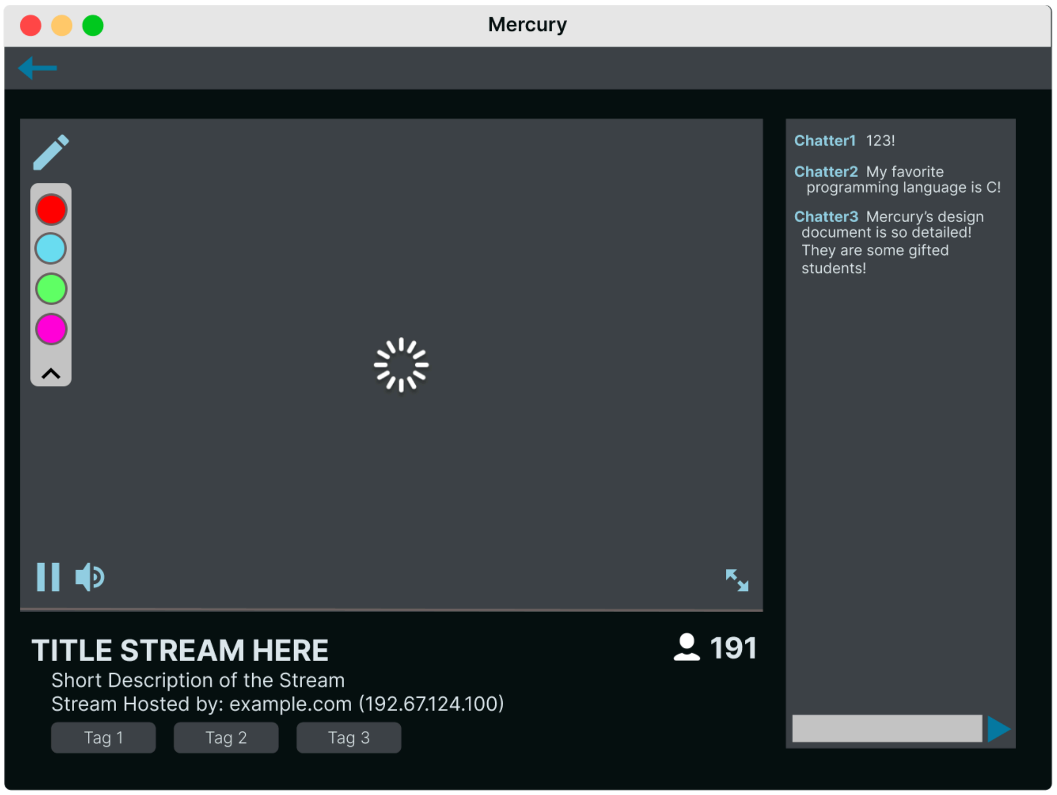
TITLE STREAM HERE

191

Short Description of the Stream

Stream Hosted by: example.com (192.67.124.100)

Client-side Annotations Palette Selection



Mercury

Chatter1 123!

Chatter2 My favorite programming language is C!

Chatter3 Mercury's design document is so detailed! They are some gifted students!

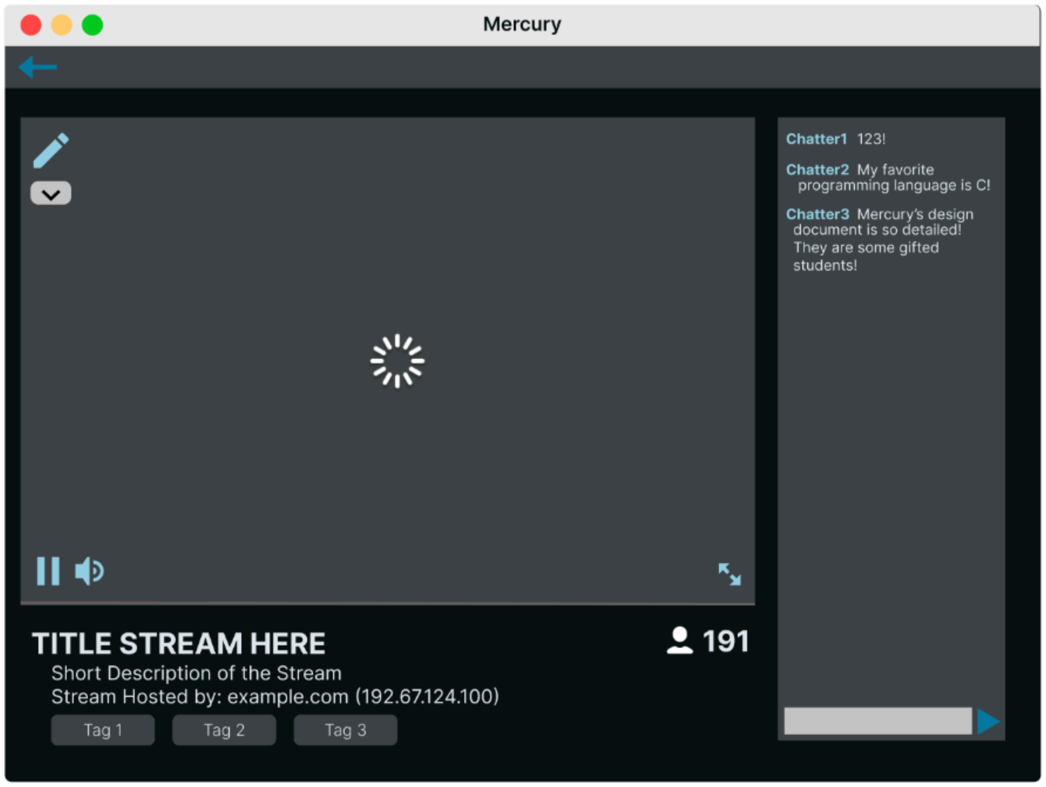
TITLE STREAM HERE

191

Short Description of the Stream

Stream Hosted by: example.com (192.67.124.100)

Tag 1 Tag 2 Tag 3



Mercury

Chatter1 123!

Chatter2 My favorite programming language is C!

Chatter3 Mercury's design document is so detailed! They are some gifted students!

TITLE STREAM HERE

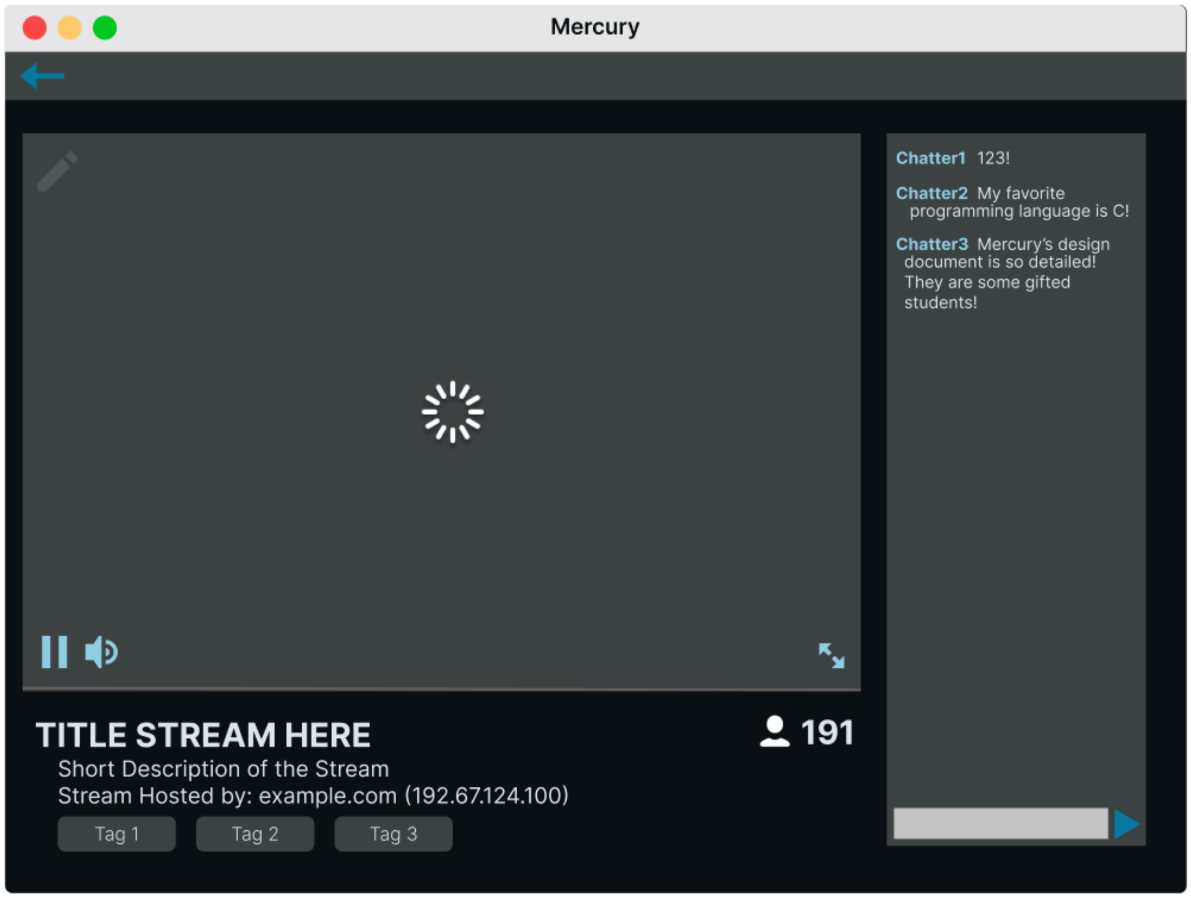
191

Short Description of the Stream

Stream Hosted by: example.com (192.67.124.100)

Tag 1 Tag 2 Tag 3

Client-side Annotations Disabled (default)



Chatter1 123!

Chatter2 My favorite programming language is C!

Chatter3 Mercury's design document is so detailed! They are some gifted students!

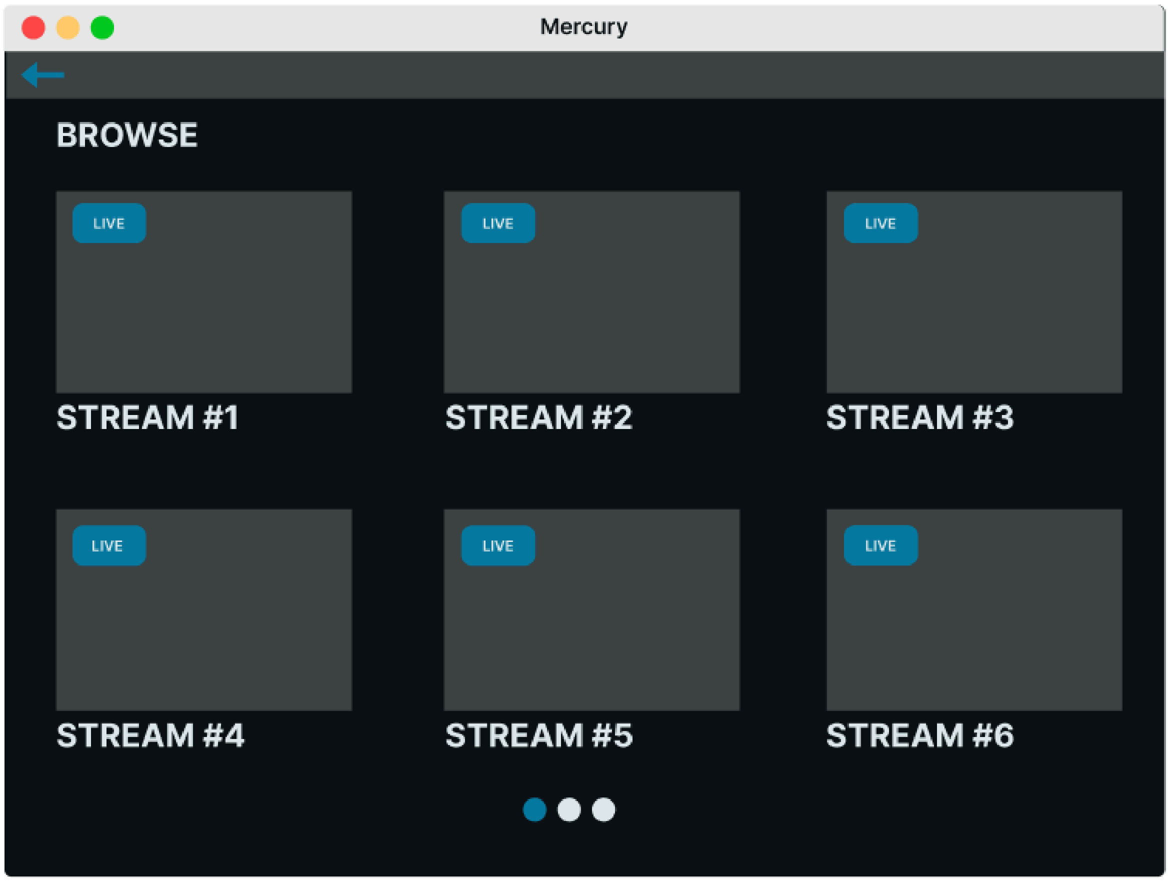
TITLE STREAM HERE

191

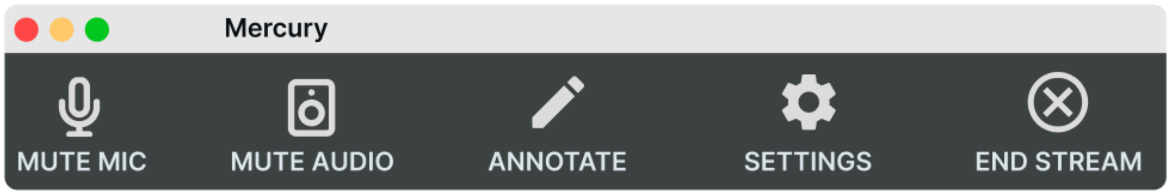
Short Description of the Stream

Stream Hosted by: example.com (192.67.124.100)

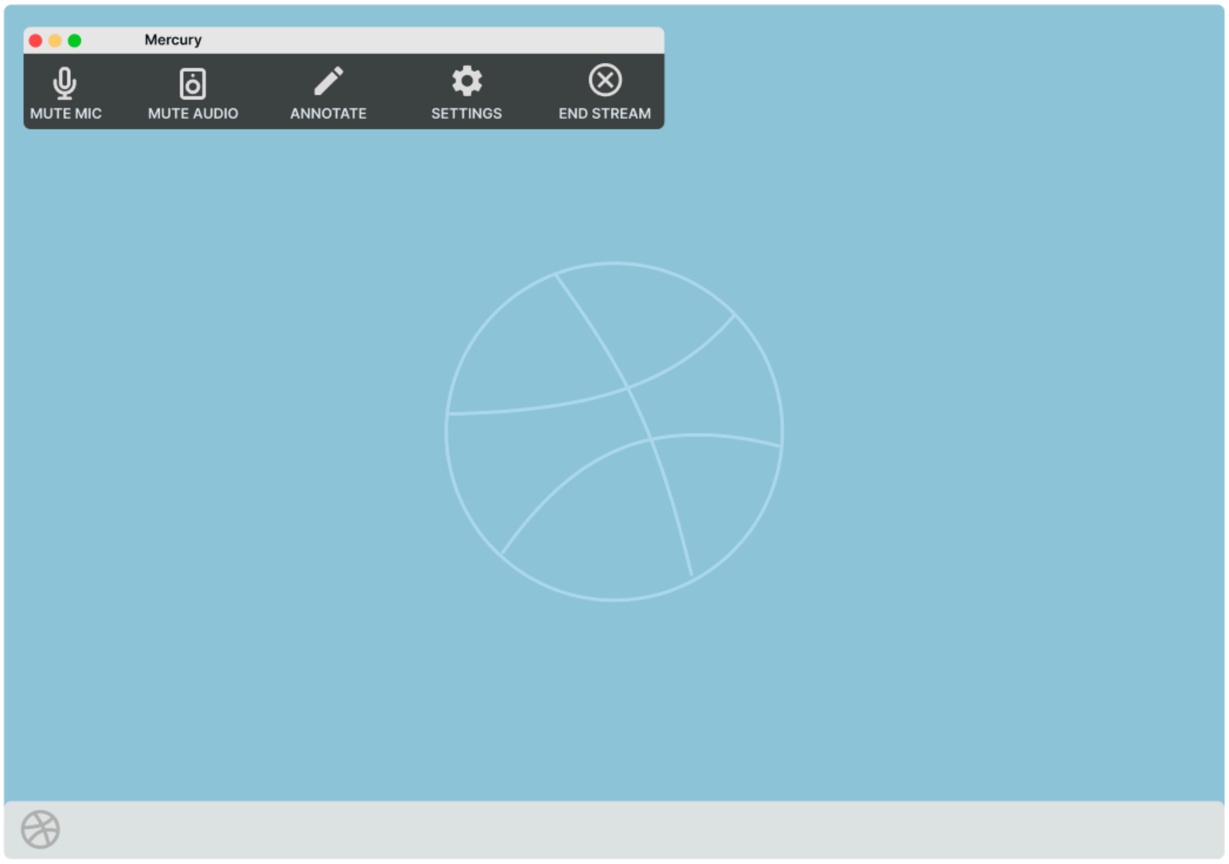
Stream Showcase



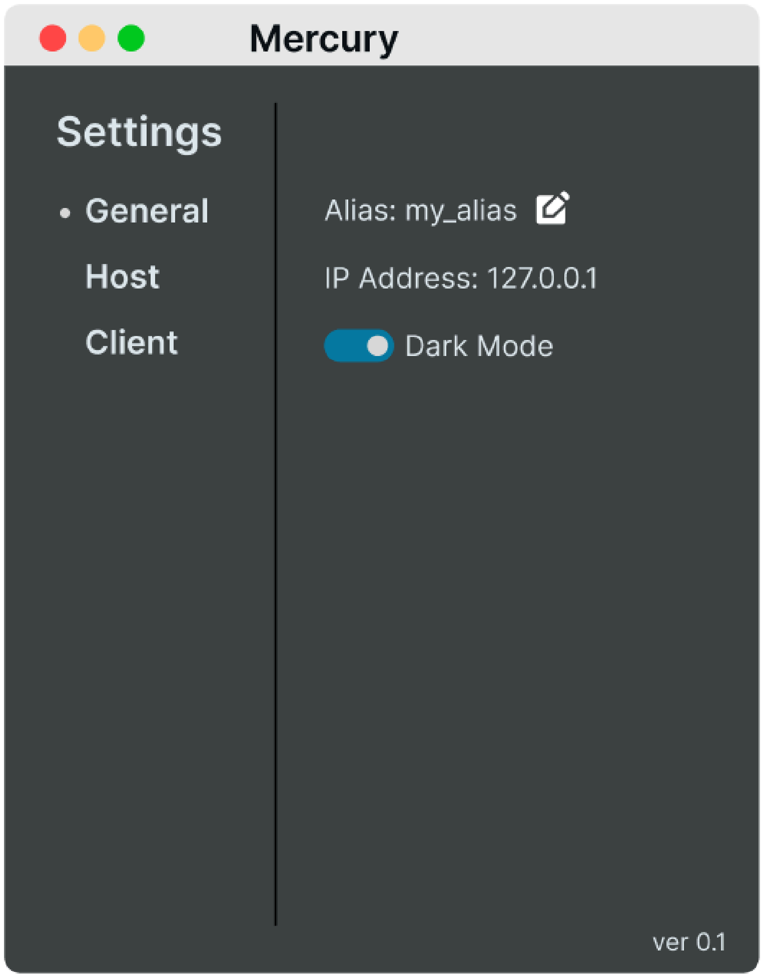
Host Overlay



Host Overlay in Streaming Mode



General Settings



Mercury

Settings

- General

Host

Client

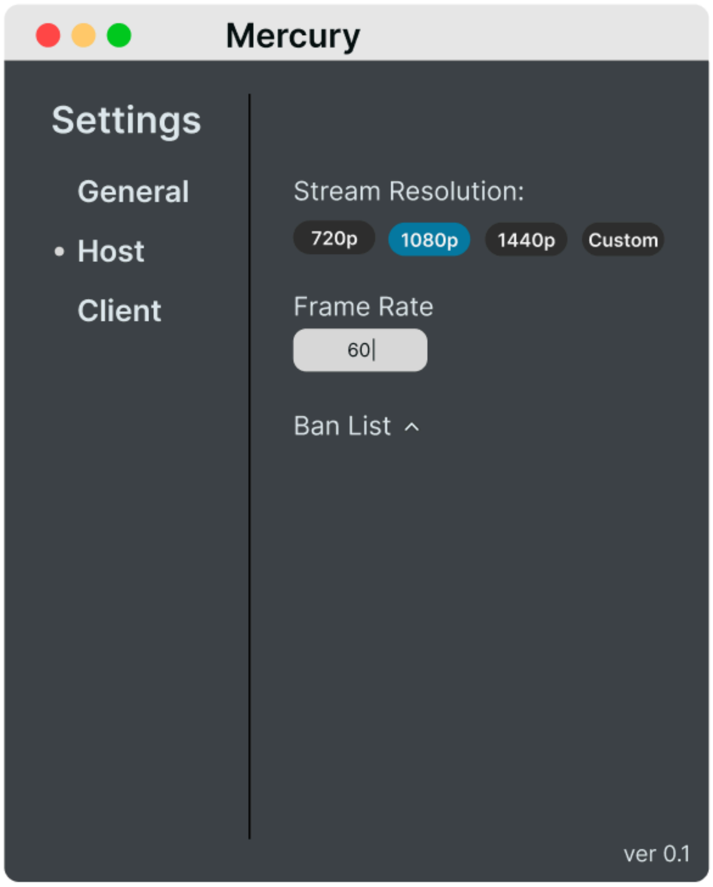
Alias: my_alias

IP Address: 127.0.0.1

Dark Mode

ver 0.1

Host Settings



Mercury

Settings

General

• Host

Client

Stream Resolution:

720p [✓ 1080p] 1440p Custom

Frame Rate

60

Ban List ^

ver 0.1