SFA-CAST Documentation

1. Libraries (use pip to install necessary libraries)

Client Libraries:

* socket
  + The program only works by sending data over a network and the socket library is what allows that capability.
* zlib > decompress
  + Each frame is compressed before sending and decompressed at the client end
* sys
  + If the TCP client socket cannot connect to the server, a try/catch block will catch the ConnectionRefusedError and message will be printed followed by a sys.exit(1) call closing the program.
* os
  + The GUI’s use the OS module to make system calls like starting the TCP\_client.py file when the start button is pressed in the TCP client GUI
* pygame
  + Mostly used on the client side. Pygame creates the viewing window and shows the frames sent from the server. It enabled window resizing as well.
* threading > Thread
  + Threading makes the client and server GUI’s usable. The main thread runs the GUI while the broadcast and receive functionality is run on spawned children threads, so the GUI will continue working.
* datetime
  + Used on the client for the screenshot functionality. Each screenshot is saved where the current datetime (second specific) is most of the filename. This way the pictures can be named automatically.
* struct (for UDP server only – not TCP)

Server Libraries:

* threading > Thread
* zlib > compress
* socket
* os
* mss
  + Used on server side to capture the current frame.
* pygame (for UDP server only – not TCP)
* time (for UDP server only – not TCP)

1. Import script (copy/paste and run in terminal within folder containing files)

pip install mss

pip install pygame

All other libraries are built-in with Python.

1. Environment Variables

GUI’s will not work properly if you do not have *py* and *python3* setup as environment variables pointing to the python language on your computer. Both of these commands are run with the os.system() call to start the broadcasting or receiving programs depending on if you are the client or the server.

1. How to Use:
   1. Start the server GUI called SFACastGUI-SERVER.py and press start to begin sharing your screen.
   2. Start the client GUI called SFACastGUICLIENT.py and press the start button. Make sure to start the client after the server is started so you receive a stream properly.
   3. On the client you can take a screenshot of the broadcast. Press F12 with the client GUI as the active window to take a screenshot. It will be stored as a .jpg file in a subtree of your working directory.

./~/SFA-CAST/SFACAST-Screenshots/

* 1. Finally, you can hit the exit button on the server GUI to close the server socket and have it stop sending images. On the client you can press ESC to close the stream window and then the exit button to close out of the client.

1. How it works
   1. Server GUI

The GUI is operated with 2 buttons, a START and an EXIT. At the moment, the start button launches the *TCP\_server.py* program in a thread and initiates the server. The program is called from the command prompt on Windows and terminal on Mac. The exit button terminates the *TCP\_server.py* program and closes out of the GUI.

* 1. Client GUI

The Client’s GUI has 3 buttons. START, SCREENSHOT LIBRARY, and EXIT. The START button launches a thread to initialize the TCP\_client.py program from the command prompt on windows and terminal on Mac. The SCREENSHOT LIBRARY button supports both Mac and Windows and is used to open the folder where the screenshots are saved to. The GUI has a menu bar option in order to change the location of the screenshot folder if desired. When a different location is selected the SCREENSHOT LIBRARY button opens the folder in its new location. Finally, the EXIT button terminates the running program and exits the GUI.

* 1. TCP\_server.py

The *TCP\_server.py* program works by creating a TCP socket. That socket is bound to the local IP found by using the socket.gethostbyname() function. This helps automate the creation of the server by making the machine running the program be the host for clients to connect to.   
When a client connects the server accepts the connection and sends its screen resolution as a message to let pygame on the client’s side know what to expect. Then a thread is created to capture an image of the screen and send to the client.  
Capturing the screen is done by using the grab() function from MSS. Each image grabbed is then compressed using zlib’s compression function.

* 1. TCP\_client.py

The client connects to the server by connecting to the given host and port using socket’s .connect() function. After the connection is established the client then initializes a pygame window that is default to a window size of 1600x900. The client then receives the resolution sent from the server and uses that to set a surface for the image to be reconstructed. The client receives the information sent from the server and then uses zlib’s decompress function to decompress the buffered data. This decompressed data is passed to pygame’s image.fromstring() function to recreate the image onto the surface. If the window is resized the surface is then resized to be the same dimensions as the new window. Once the image is reconstructed it updates the screen to show the image. After it updates, the program loops until the pygame window is closed.   
Before the client receives data for the next image it goes through a queue for events. We have 3 different events it checks for. Those events are video resize, quit, and a key being pressed. When the window is resized, it changes the dimensions of what it is showing to match the new dimensions of the window. When a quit event is made, the loop that it is in breaks and the socket is closed. The last type of event it checks for is if a key is pressed. We only checked for 2 different keys, ESC and F12. If the pressed key is ESC, then the window is closed just like if it was a quit event. When F12 is pressed, the program calls pygame’s image.save() function to save the current frame in the SFACast-Screenshots folder from the GUI. The file name the image is saved as uses the current date and time.

* 1. udpserver.py

The UDP server is built off of the TCP server described above. The methods are mainly the same, but the way data is sent is what makes it very different. The socket created is a UDP socket. The server sends the resolution to the Multicast IP of 224.0.0.1. [RFC 5771](https://tools.ietf.org/html/rfc5771) shows the ranges of addresses used for IPv4 Multicast. We used 224.0.0.1 because it is designated to the Local Network Control Block. This means that protocol traffic is not forwarded outside of IP the router is assigned to. The server works the same way as TCP as in how it sends information of the image before sending the actual image data. When the server sends the image’s data it does so very differently from the other server. When you try to send very large amounts of data across a UDP socket, an error will get raised saying “A message sent on a datagram socket was larger than the internal message buffer or some other network limit, or the buffer used to receive a datagram into was smaller than the datagram itself”. We had troubles working around this and found a solution to send the large data.  
The solution was to chunk the image data and send 4KB size packets and have the client join the incoming packets into a buffer. This process is done by calculating how many chunks need to be sent and using a while loop to send segments of 4kb at a time and decrementing the calculated number until it reaches 0. When the counter reaches 0 it means the last segment of data was reached.   
This process is done repeatedly until the server is closed.

* 1. udpclient.py

The UDP client also works the same way as its TCP counterpart. The UDP socket created is bound to the local IP found by socket’s .gethostbyname(). There is a socket option names socket.IP\_ADD\_MEMBERSHIP used to let the kernel know that we want to receive packets from the server. If this option was not set, then the packets destined to the host would be discarded. After the socket is done setting up, it then receives information from the server. The information from the server is used to recreate the image being sent over. In order to collect all of the data sent by the server, it also uses a while loop. The number of packets expected to be sent is sent to the client to keep track of what data is belonging to the image. With this number, the client calls a method used to accumulate the data. The client’s socket receives the chunks in the same size that the server is sending. After it receives this data it then decompresses it and produces the image to be displayed.