PI Webcam Server

Network Centric Programming

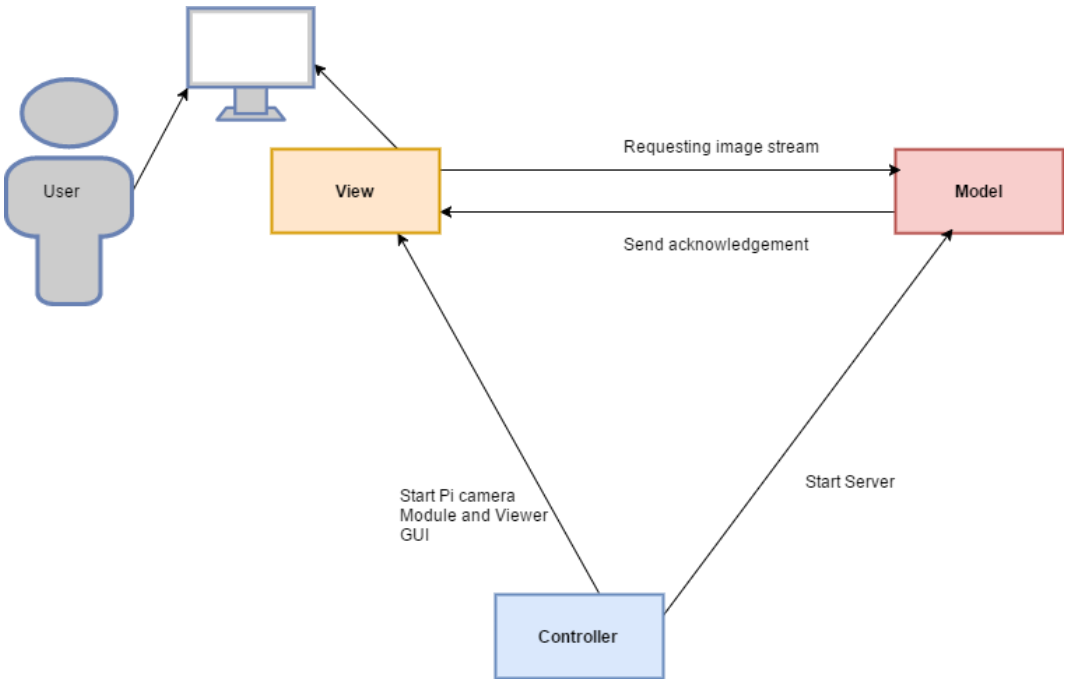
5/8/17

Alejandro Aguilar

Brian Faure

System Components

Our system can be easily divided into 3 components, using the Model View Controller Architecture we break from just a client-server interaction. We created a GUI that allows a viewer to interact with the client in an easy fashion. While the main functionality of the the our user interface is to display live streams, the interface has buttons to stop, play and pause streaming. The Controller of our webcam server are the python scripts, executing the codes allows for the GUI to start up and for our webcam to start taking continuous pictures. The controller the works behind the scenes to provide a function whenever a button is pushed on the GUI.The model is the server, when the client sends the request for an image, the server verifies the request, parses it and begins to send the binary code for the image. This image is then received by the controller where it will display the image to the user unless they stop the steaming or pause it.



Protocols

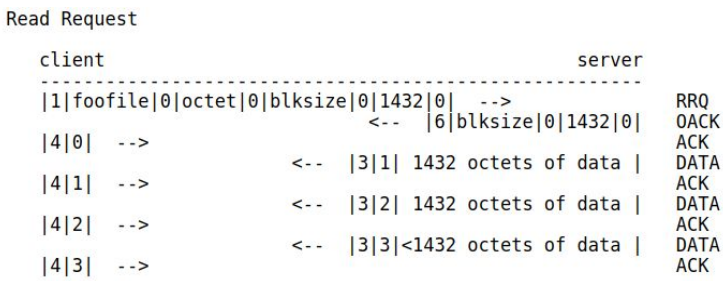
Our server and viewer transmit all information over UDP connected sockets using a more recent version of the TFTP protocol we saw in the TFTP course assignment. We have implemented RFC 2348[1], which enables a variable block size option, to allow for significantly faster transfer times of our frame images. The RFC specifies that the designated block size must be included in the first connection request, so either in the initial RRQ or WRQ. The format of this modified packet can be seen below.

```
+-------+---~~---+---+---~~---+---+---~~---+---+---~~---+---+
|  opc  |filename| 0 |  mode  | 0 | blksize| 0 | #octets| 0 |
+-------+---~~---+---+---~~---+---+---~~---+---+---~~---+---+
```

For comparison, our original implementation of the TFTP server, following RFC 1350[2] specified that RRQ and WRQ packets include only the first three sections (*opcode, filename, mode*). The first addition to the end in this updated version, *blksize*, is used to tell the receiver how to interpret the *next* section, in this case telling the server to use the following as the number of sets of octets (bytes) to include in each transferred data packets. Both the *blksize* and *#octets* sections are ASCII encoded, such that the *#octets* value must be converted to an integer or numerical format before any computations can be performed upon it. After receiving a RRQ or WRQ request of this format, the server's next job is to return with an *OACK* packet, of the following format.

```
+-------+---~~---+---+---~~---+---+---~~---+---+---~~---+---+
|  opc  |  opt1  | 0 | value1 | 0 |  optN  | 0 | valueN | 0 |
+-------+---~~---+---+---~~---+---+---~~---+---+---~~---+---+
```

The opcode for these messages is simply an integer 6, and the (opt1,value1) up to (opt*n*,value*n*) pairs following designate the certain option being acknowledged, as well as the accepted value, '*blksize*' & *'#octets*' being the only tuple in our case. To enable this functionality we also had to partially implement RFC 1782[3] which first introduced these modified *OACK* responses. The image on the right shows a simplified model of the client-server communication under this protocol, specifically an RRQ request.



To simplify the implementation, we have currently set our viewer.py file (the client) to request a blocksize of 8000 bytes, and our server.c file (the server) is expecting that exact value, such that we can use statically 8000-sized buffers and avoid having to re-write larger portions of our TFTP server code. The options to control the socket connections are all hidden from the user, when interfacing with the GUI, so we don't foresee this being a problem. On the client-side, our GUI code (viewer.py) uses a Python library called Tftpy[4], which makes implementing TFTP extremely simple. This module supports all three RFCs explained above, at least to the degree which we required them in this program. The following lines of code display how we request an image through tftpy.

```
client = tftpy.TftpClient(self.ip_address,self.port_num,options={'blksize':DEFAULT_BLK_SIZE})
try:
    client.download(SERVER_DIR+"/"+FRAME_FILE,CLIENT_DIR+"/"+FRAME_FILE,timeout=MAX_WAIT_TIME)
    self.parent.current_frame_file=CLIENT_DIR+"/"+FRAME_FILE
    self.update_gui.emit()
except:
    print("Transmission Error")
    num_transmission_errors+=1
```

Note that the ‘SERVER_DIR’ and ‘CLIENT_DIR’ correspond to ‘server_frame_buffer’ and ‘client_frame_buffer’ directories, respectively, and the ‘FRAME_FILE’ is set to ‘frame.png’. Our ‘DEFAULT_BLK_SIZE’ is, as explained above, 8000 bytes. We have specified a maximum wait time, which sets the socket timeout settings for the request, to be 0.2 seconds to prevent the connection thread from hanging too long if it gets caught up on a single frame.

The decision to implement variable block sizes has substantially increased the speed at which we can refresh the image shown to the user, leading to a much more fluid video experience. With 8000 Byte blocks, as opposed to the original 512 Byte, we have increased our transfer speed nearly 16 times, though the speed-up isn’t a full 16x because we must account for the larger transfer sizes and slow-downs on having to re-transmit large packets. Now, for most files in the range of 100kB-300kB we can complete the entire transfer, in initial testing, in ~0.05 seconds. This video[5] shows a transfer over wifi of the a sample video, split into .png frames and simulating output from the Raspberry Pi.

Our server.c file acts as the server in the TFTP connection and provides access to the outputs of the Raspberry Pi camera to the client. The port the server listens to is provided in the command line parameters to the server.o (compiled server.c) process and, as of now, we are using *15213* as our standard and this value is hard-coded into the receiver end (see *self.port_num* in the client code above).

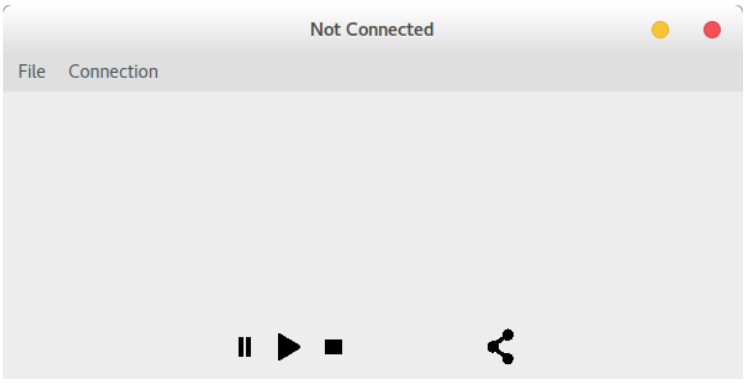
Message Formats

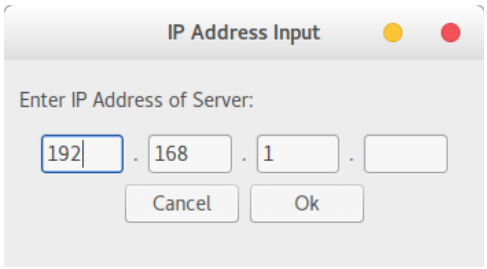
As described previously our server will send an OACK back to the client where the client will continue to receive the binary bytes from the server. A simple message format would be:
Client -: get /home/user/server_frame_buffer/frame.png
Server-: RRQ ‘frame.png’ octet from 127.0.0.1:35504
Client -: received Data <block = 1, 4 bytes >

The server will display the port number it is sending to and the file, in the our case the file will named image.png. The messages will only be displayed on the server side and the client will have the GUI to view the live stream. The raspberry pi will be sending the files and on a remote desktop, or laptop where the viewer can see anywhere the camera is placed.

User Interface

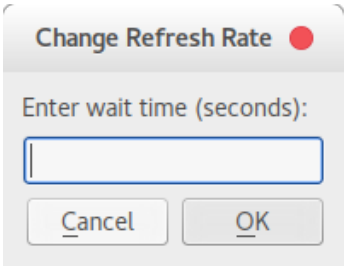
The main user interface of our system was built using PyQt, a GUI design toolkit providing Python bindings to the Qt framework. Below is the startup screen of our desktop application.





From here the user can choose to connect to a server, upon which they will be presented with the dialog window, above on the right, where they can input the IP address of the machine on which the video is being recorded.

After a valid IP address has been entered (recall, the port number is a static *15213*), the system will automatically begin requesting the frame.png file (*server_frame_buffer/frame.png*) which corresponds to the most recent image output by the Raspberry Pi camera. The rate at which the camera snaps pictures is not controllable from the desktop GUI but the user is able to control the rate at which images are requested, the image below shows the pop-up window which enables this feature.



By default, the refresh rate is set to 0.1 seconds, such that a new copy of *frame.png* will be requested after every tenth of a second.

Every time a valid transfer occurs, the thread which maintains the connection to the server notifies the main UI thread and provides the path to the new image, enabling the main UI thread to immediately load it into the display box. If the speed at which these images are received is a high enough rate, ~0.05 to ~0.1 wait time, the images become fluid and resemble a video.

The three buttons on the bottom-left of the main UI are used to control the video stream as follows; the pause button (two vertical lines) will send a signal to the connection thread to pause it's requesting of frames, the play

button (triangle) will send the opposite signal to the connection thread (informing it to begin requesting images), or, alternatively, if the user has not yet input the IP address of their source, the play button will open up the IP Address Input Dialog. The stop button (square) will tell the connection thread to perform the same function as the pause button, but will also delete the current IP address, such that the user will need to re-enter a new IP address if they wish to continue the stream. The last button on the right is used as a connection button, and will open the IP Address Input Dialog, whether or not the user has a currently running stream.

References

[1] http://www.faqs.org/rfcs/rfc2348.html
[2] http://www.faqs.org/rfcs/rfc1350.html
[3] https://tools.ietf.org/html/rfc1782
[4] http://tftpy.sourceforge.net/sphinx/index.html
[5] https://www.youtube.com/watch?v=Y6nr3eEEhso

Appendix

The source code for our project can be found at our public Github repository:

- https://github.com/bfaure/Pi_Webcam_Server

After downloading/cloning the repository, our client GUI can be opened with the following command:

- ‘python viewer.py’

The C server program can be initialized with the following series of commands:

- ‘make clean && make && ./server 15213’ (15213 specifies to port to listen at)

The Raspberry Pi Camera is controlled with the Python script ‘CamMod.py’, found under the ‘server_frame_buffer’ directory on the repository, and can be run using the following command:

- ‘python CamMod.py’

Note that Python 2.7 is the distribution this system has been developed under, Python 3+ has not been tested but we foresee no conflicts as the elements of 2.x we have implemented are all compatible with 3.x syntax.

The server.c file we built on top of was originally the IO-Multiplexing submission by Brian Faure but both Alejandro and Brian have added the code to it collaboratively which enables larger packet sizes and reusing of port numbers. The viewer.py file was mainly written by Brian and the CamMod.py hardware controller was written mainly by Alejandro.

If any questions/comments arise while reviewing our source code/project, feel free to send us an email at:

- Brian: bfaure23@gmail.com
- Alejandro: aa1363@scarletmail.rutgers.edu