Streaming Client and Server Model

A Senior Project
Computer Engineering Program
California Polytechnic State University, San Luis Obispo

by
Austin Alan Diec
June 2011

Acknowledgments

Dr. Seng, thank you being my first computer-engineering professor at Cal Poly. Much of my success as an undergraduate is due to the drive and motivation that started in CPE101. Thank you.

My parents, Daniel and Ellen, thank you for supporting me through these intense four years.

Tracy Akashi, thank you for all of your help with this report. I know proofreading it was tough.

Abstract

A few years ago, several Cal Poly undergraduates started the Autonomous Robot Platform (ARP). Interest in a revision of the ARP requires a new layer of communication. This paper describes the design and partial implementation of the communication layer to be used with the revised ARP. The communication layer would allow the autonomous robot platform to connect with a client over the wireless local area network (WLAN) for instructions and commands. It is important to note that currently, the assumed revision of the ARP is nonexistent, so many of the design decisions are based solely on the assumptions of ARP work. Therefore, this senior project only serves as a loose model of the communication layer for the revision of the ARP.

Contents

List of Figures

INDEX TERMS
autonomous robot platform (ARP), server, client, wireless local area network, node, IplImage, OpenCV, FFmpeg, VLC, capture

1. INTRODUCTION

A well-designed autonomous robot platform (ARP) would allow its users to perform a vast variety of tasks. Driving around campus or performing tasks that place human lives at risk are only two examples of its diverse capabilities. In order for any specific task to be accomplished, the ARP is designed with an external sensor to gather information about the surrounding environment. With this information, the ARP could easily determine its next course of action.

The initial design and implementation of the ARP at Cal Poly was created using an electrical wheel char as a base and, and server racks as a controller. Unfortunately, the ARP was too large and heavy, which in turn, limited its mobility and function. Dr. Seng suggested the revision of a significantly smaller and lighter chassis makes improvements on both functionality and mobility [1].

Despite the revisions, the ultimate goal of the new ARP remains the same: create a robot with the ability to navigate and drive around campus completely autonomously. Instead of using primitive methods of navigation such as infrared sensors to follow lines, or GPS, which relies on additional devices to provide functionality, the ARP will use the WLAN and image processing to determine its current location and next course of action. More specifically, the ARP will eventually utilize an integrated web camera onboard the ARP. It will then transmit the video stream to a client that is also located on the WLAN to process the image and issue a new command for the ARP.

As such, the goal of this project was to design and implement a model for the video transmission using the client-server architecture, in which each ARP is to be a server that communicates with a client to process its video stream and receive new commands.

2. PROJECT BACKGROUND AND GOALS

Currently, the revision of the ARP does not exist. Because of this, both the architecture and functionality of cannot truly be determined until the revision is created. For now, the functional goal of functionality for the ARP seems relatively simple: navigate the Cal Poly campus autonomously. Adding a communication layer enables the ARP to be fully autonomous.

2.1 ASSUMPTIONS

Since the revision of the ARP has no permanent specifications, the communication layer was designed with a number of assumptions in mind. First, the server, or robot, will be operating on a structurally sound chassis that supports the weight of a laptop, which will act as the server. The laptop must house an integrated web camera to stream video to a client. Additionally, the server is expected to be operating on a UNIX-based operating system that is supports all of the libraries used in the communication layer (standard C libraries, jpeglib and OpenCV). The client also runs on an UNIX-based operating system that has all of the libraries used to implement it. Lastly, both the server and client should both be on same WLAN, otherwise the quality of the video stream will be unpredictable.

Since all of the networking is done within the local area network, all clients and servers are assumed to 'play nice'. Moreover, all malicious behavior from external sources is assumed to be non-existent and highly improbable.

2.2 REQUIREMENTS

The main focus of this project, as stated in the earlier sections, is enabling the ARP, or server, to communicate with a client over the WLAN. More specifically, the project will enable the server to provide a video stream from the integrated web camera to any client via WLAN. The client will continually receive this stream, process the images and issue a new command to the server. Additionally, the client is required to support connections to multiple servers and clients. Some additional shared features between both applications are:

- Persistent connections. The client will be able to automatically reconnect to any given server that was previously open, whenever that server goes back online.
- Display. All video streams will be displayed to be the user.
- Portability. Both the client and server applications should be able to compile and run on any UNIX-based operating system, assuming that all of the libraries are readily available.
- Flexibility. Both applications should be flexible to meet the needs of the ARP used.

2.3 PERSONAL GOALS

The goal of this senior project was to utilize the knowledge and experiences I have gained from previous courses (Operating Systems, Networks, Systems Programming, and Robotics) and develop a senior project that would combine these skills. However, in addition to the technical goals and requirements stated in the previous section, many of the tasks required underlying skills to complete the tasks. As such, at the completion of this senior project I have attained the achieved the following goals and skills:

- Documentation. Issuing a proposal, clearly stating the problem at hand and suggesting a viable solution.
- Research. The majority time spent in the design phase required extensive research to select the most suitable libraries and architecture.
- Project Dependencies. Using open source libraries had numerous interdependencies to trace.

- Applied Development. Integration of all the skills acquired from previous courses to design and implement the project.

3. DESIGN

From the requirements section, the implementation of the communication layer only has one certainty: it will follow the client-server model to transmit data and relay messages to one another. Selecting an image processing library, on the other hand, proved to be much more difficult. There are multitudes of preexisting libraries that are not only capable of capturing video from a web camera, but also establish video-streaming server for the video as well. The following section will describe some of the advantages that these libraries offer, and why OpenCV was the library of choice.

3.1 LIBRARIES CONSIDERED

Of all the video encoding, capturing, and streaming libraries and applications, only three truly stood out: VideoLAN, OpenCV, and FFmpeg.

Initially, the most appealing library was the VideoLAN application VLC media player, mainly because it already contained most of the functionality that was required for this project. However, after delving into the application and testing its uses, I discovered that the application is very limited in terms of flexibility. The VLC player is certainly capable of streaming from a capturing device over many protocols, including HTTP, and supports many different video encodings, but it does not allow enough flexibility user to alter the video capture properties. Particularly, the VLC application does not allow the user to specify the video stream resolution, which is a vital for image processing. The importance of video resolution lies in the ability of the client to process an image--the smaller the image, the less accurate the next instruction sent to the ARP will be. The figure below shows all of the optional parameters offered in the VLC application [3].

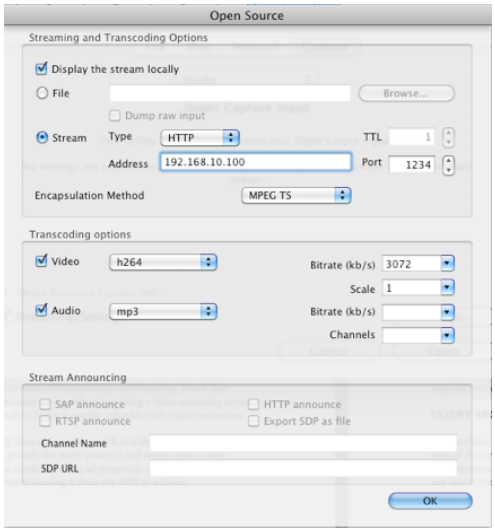

Figure 1: VLC User Interface for Video Streaming

The FFmpeg library is very similar in terms of functionality and flexibility. Much like VLC, the FFmpeg library and application are capable of providing a video-streaming server and supporting a plethora of video codecs, but it is equally inflexible with video resolutions.

3.2 OPENCV

The OpenCV library is often selected as the main image and vision-processing library for robotic applications. Although it does not have as rich of a feature set as either FFmpeg or VideoLAN, OpenCV provides the developer complete control of the video stream and video codecs. Additionally, complexity is reduced when a single library, such as OpenCV, for both video transmission and processing. For these reasons, OpenCV was chosen.

3.3 CLIENT-SERVER MODEL

As previously stated, the implementation of the communication layer for the ARP follows the client-server model. The client is expected to retrieve information from any number of servers and process that information. By standard definition, the server, or the ARP, provides information requested from the client.

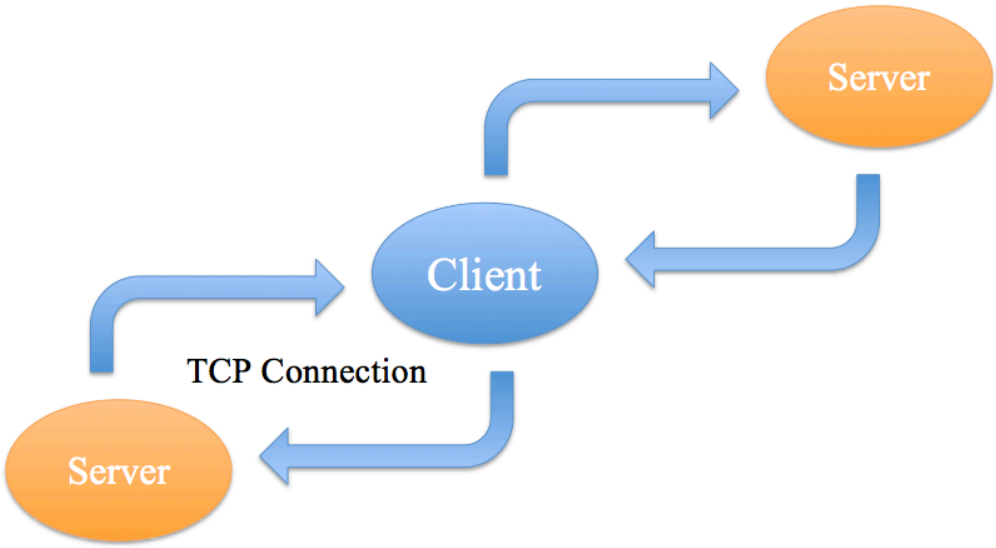

Figure 2: Client-Multi-Server Model

4. IMPLEMENTATION

Both the server and client employ multithreaded implementations to maintain a network connection with each other while processing the video stream. There is no particular reason for choosing threads over other concurrency models (e.g. event-driven or multi-process), but it should be noted that some type of parallel programming must be instituted to allow a server to both collect images from the video capture and stream it simultaneously. The ability of both applications to do tasks in parallel eliminates the issue of an element blocking the entire application.

4.1 SERVER ARCHITECTURE

As previously stated, the server is a multithreaded application. Threads are necessary for the server to be able to simultaneously maintain an OpenCV window that displays the web camera input and send a gray-scaled stream to its client. Therefore, the server employs two threads, each performing one of the aforementioned functions.

The server will only allow a single client to connect and receive its video stream. Should the client exit or drop its connection with the server, the server would then accept the next connection in queue. In order for the server to accept any new connections, the same thread that is responsible for sending the video stream will reset the client socket and continue to listen until a new connection is formed has started, at which point the video stream would be sent to this new client.

Lastly, the following figure demonstrates the program flow of the server, including some steps for initialization and clean up that were not mentioned.

4.2 CLIENT ARCHITECTURE

For the client to be able connect to multiple servers, process the video streams, and display each of the streams, concurrent programming is necessary. Similar to the server, each of the client's tasks requires its own thread in order to prevent the entire application from blocking.

The command line interface (CLI) allows the user add new servers to control or remove a server that it is currently connected to. When the user wants to add a new server stream to monitor, the IP address, operating port, width and height of the frame must be specified. Although the IP and operating ports are the only parameters that are essential to establishing a connection, specifying the width and height of the expected stream reduces complexity, so that the client knows exactly how much data to expect and the correlating IplImage used to display the video stream can be allocated accordingly. For flexibility, future implementations should also provide the ability to automatically detect the server's frame size automatically.

The display thread manages all of the display windows that show the different video streams from all the servers it is connected to. The original design was to have a pair of threads that

would be created for each server the client would connect to, but due the limitations in the OpenCV, a single thread must initialization and update all display windows.

For each connection to a server, the client creates a new networking thread, which manages and maintains a connection to a specified server. This thread is responsible for receiving the video stream from the server and updating the current frame for its corresponding server. If the connection to the server is dropped for whatever reason, the thread will reset the state of connection and attempt to reconnect to the server every three seconds.

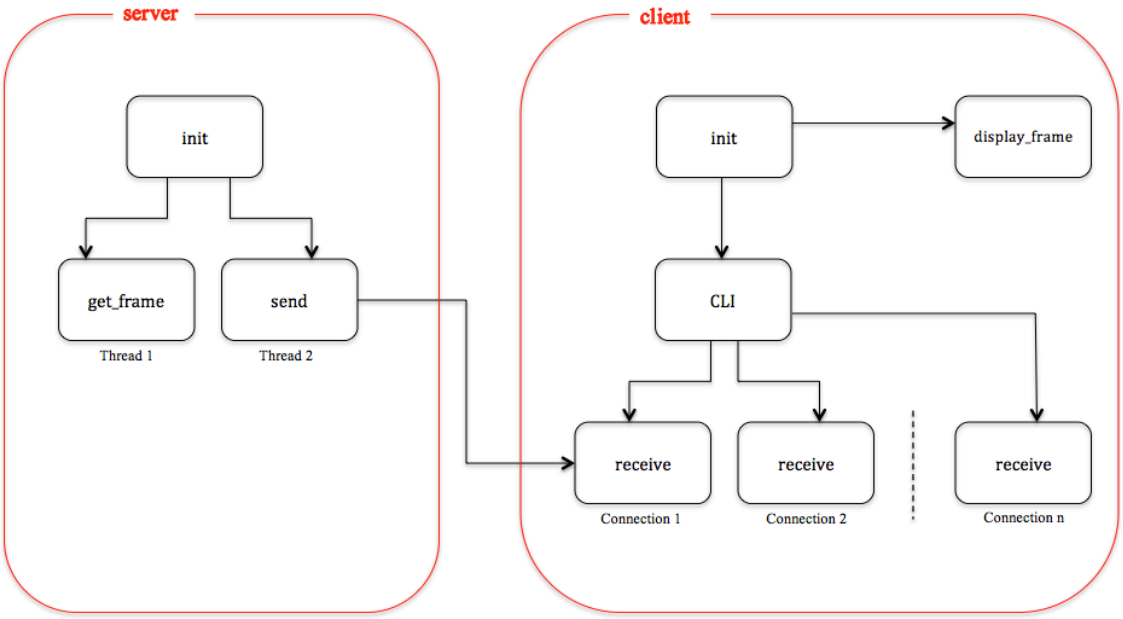

Figure 3: Server and Client Architectural Model

4.3 PERSISTENT CONNECTION

Regardless of the application, whether it is a distributed system or even a web server, it is important to maintain high availability. In both the client and server, maintaining a persistent connection enables the application to not only reduce overhead with less handshaking, but also remain highly available. For each client and server pair, a TCP connection is shared. This connection is kept persistent, which not only allows streaming video, but also message passing over the same connection. Furthermore, this also allows the client to automatically reconnect to any servers that it dropped or lost connection with.

Similarly, when a client disconnects from a server, the server will return to a listening state. In this state, the server accepts the next client connection on queue.

5. RESULTS

The current implementation of both the client and server supports the transmission the video-stream over a persistent connection. The client is also capable maintaining multiple server connections. The figures below show a single client that is connected to multiple servers running on both Ubuntu and Mac OS X.

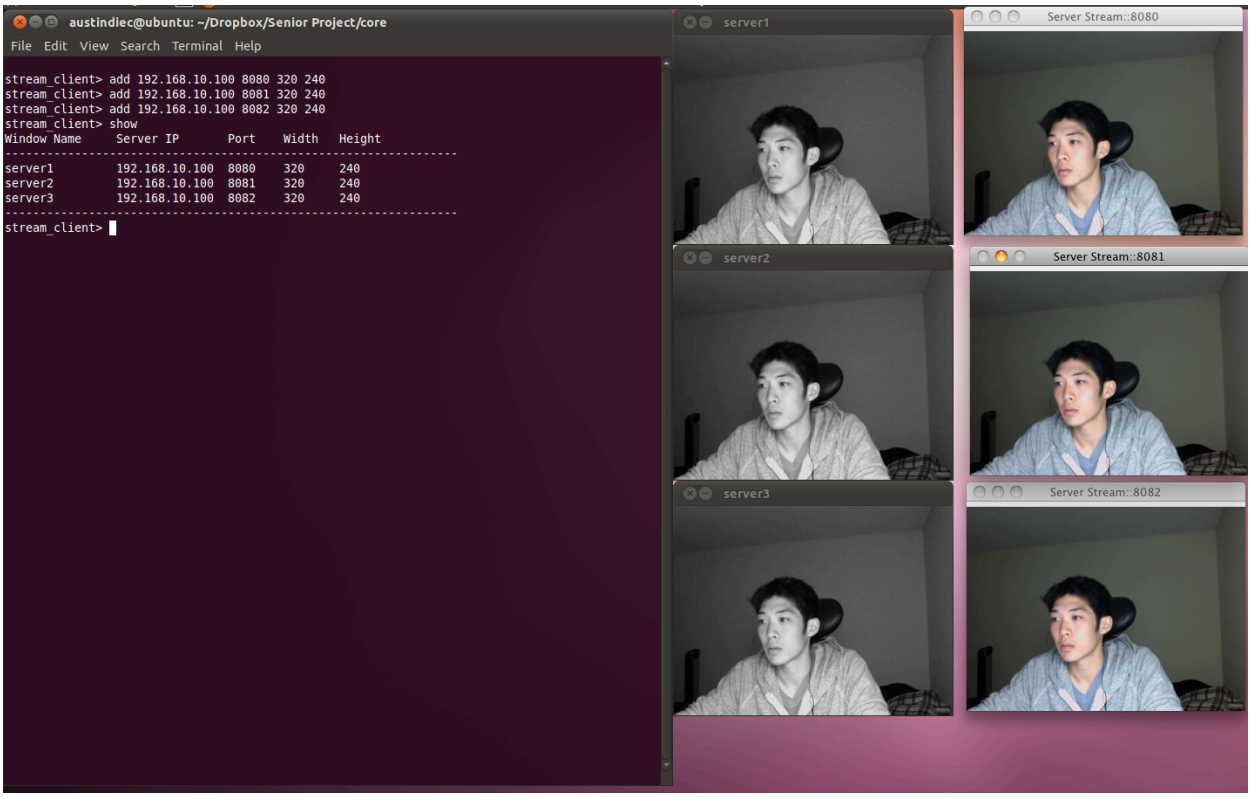

Figure 4: Demo of Server and Client On Test Environment

5.1 TEST ENVIRONMENT

All tests were performed on a single machine. The client compiled on Ubuntu using VMWare Fusion, whereas all three of the servers were compiled and run on Mac OS X 10.6.5.

6. FUTURE WORK

Although the transmission of video stream between both the client and server are complete, a significant amount of work still needs to be done in order for the communication layer of the ARP to be useful and robust. The following sections will describe the most pressing features that still need to be implemented for the communication layer.

6.1 JPEG COMPRESSION

The server is currently collecting video directly from the web camera, meaning that the video stream data is raw and uncompressed. Also, the video-stream is only converted to gray scale before sending it to the client. Although this provides great image quality, the bandwidth usage, as shown in the figure below, is clearly an issue when dealing with scalability. A clear but difficult remedy is to compress the raw image from the web camera before sending it to the client. This would significantly reduce the amount of bandwidth used.

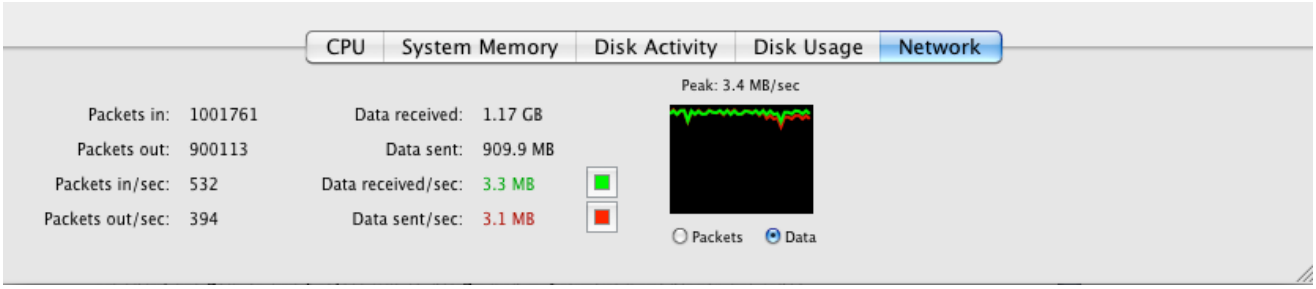

Figure 5: Bandwidth Usage of Three 320x240 Resolution Video Streams

6.2 MESSAGE PASSING

As mentioned in a previous section, both the client and server maintain persistent TCP connections with one another. This allows for message passing, which is particularly useful when the client is capable of sending commands and instructions to the server, and therefore ARP, to perform a task. When the revision of the ARP is finally functional, message passing will become a vital part of having the ARP function completely autonomously.

7. CONCLUSION

The client and server currently model a significant portion of the expected behavior of the eventual ARP communication layer. However, the imaging processing, compression, and message/command passing between the client and servers still remain. Nonetheless, the completion of both the server and client models provided new skills, many of which were unexpected (i.e. rigorously searching forums as a method of debugging due to the lack of documentation for open source libraries). It is clear that the communication layer for a new ARP is far from complete, these implementations for both the client and server models serve as a viable example of how the next iteration of ARPs at Cal Poly can communicate.

I. REFERENCES

[1] Dr. John Seng. Calfornia Polytechnic State University.

http://users.csc.calpoly.edu/~jseng/

[2] OpenCV. http://opencv.willowgarage.com

[3] VideoLAN. http://www.videolan.org/

[4] Nash Ruddin. http://nashruddin.com/tag/opencv

[5] FFmpeg. http://www.ffmpeg.org/

II. APPENDIX

A. ANALYSIS OF SENIOR PROJECT

A.1 SUMMARY OF FUNCTIONAL REQUIREMENTS

This project requires the design and implementation of an application that models the communication layer of the revision of autonomous robot platform project. The server is required to interface with the integrated web camera, display the input from the web camera, and transmit input to any client on the wireless local area network. The client must be able to connect and maintain a persistent connection to multiple servers.

A.2 PRIMARY CONSTRAINTS

The most difficult part of this senior project was designing the behavior of both applications and creating assumptions for the expected behavior of the revision for the ARP. Since the revision of the ARP is nonexistent, it was rather difficult to predict how implement a communication layer for an application that does not exist.

A.3 ECONOMIC

Every aspect of this project is software-based. As such, no other materials were needed to complete this project. A very rough estimate of the time that it would have taken to complete this project would be around 100 hours of development time. After completing this project, it seemed to have taken well over 100 hours of development time.

A.4 ENVIRONMENTAL

Since there is no physical aspect of the project, there is no direct impact of the project onto the environment. However, as mentioned in the sustainability section, robustness is key for reducing power usage of any device that the applications are placed on.

A.5 MANUFACTURABILITY

Since the project is completely software-based, it cannot be manufactured. The revision of the ARP that is supposed to use it however is. Unfortunately, it is nonexistent and cannot be estimated.

A.6 SUSTAINABILITY

The project itself has no direct correlation to any means of measuring sustainability. However, it is important to note that the efficiency of the implementation of both the client and server inadvertently affect the power usage of any device that it is loaded onto. So, in a sense, the more robust and efficient the implementation, the less power is used.

A.7 ETHICAL

There are no ethical implications with this project. Both applications simply communicate with one another over a TCP connection.

A.8 HEALTH AND SAFETY

This project does not impact the health and safety of anyone. Although, when the revision of the ARP is complete, it is essential for the communication layer to be fully functional, otherwise misleading commands issued from the client could potentially cause physical harm to either the robot or anything that it may crash into.

A.9 SOCIAL AND POLITICAL

There are no social or political implications or ties that the project affects.

A.10 DEVELOPMENT

Through the lifespan of this project, many skills were acquired. The direct and apparent one is the ability to combine all of the knowledge and experiences I've gained in past classes, and using it to design and implement both of these applications. Additionally, the completion this project vastly improved my ability to research the topics of interest.

B. Source Code

B.1 CLIENT.C

- Iterative updates all displays maintained in the client:

```
/*
 * Unfortunately, OpenCV's CvShowImage and CvNamedWindow are not thread safe.
 * In order to display the frame of each server correctly, this function has been
 * hacked together to make it work.
 */
void display_frames(void) {
    server_node *iter = nodes_head;

    while (!quit) {
        while (iter) {
            if (SUCCESS == pthread_mutex_trylock(&iter->mutex)) {
                if (!strlen(iter->window_name)) {
                    if (init_display(iter)) {
                        perror("Failed to initialize window");
                    }
                }
                if (iter->new_data) {
                    cvShowImage(iter->window_name, iter->frame);
                    iter->new_data = false;
                }
                pthread_mutex_unlock(&iter->mutex);
            }
            iter = iter->next;
        }
        iter = nodes_head;
        cvWaitKey(30);
    }
}
```

- Receives the frame from a server:

```
/**
 * Initializes the server struct inside the specified node object.
 * After all connections are established, it begins receivng the frames
 * transmitted by the server.
 */
void *receive_frame(void *arg) {
    server_node *node = (server_node *)arg;
    ssize_t img_size = node->frame->imageSize;
    char sockdata[img_size];
    int reset = false;
    int oldtype = 0;
    int i, j, k, bytes;

    /*
     * Starts a connection to the server using the specified ipaddr and port
     */
    if (init_network(node)) {
        perror(node->window_name);
        return;
    }

    while (1) {
        for (i = 0; i < img_size; i += bytes) {
            if (0 >= (bytes = recv(node->socket, sockdata + i, img_size - i, 0)))
{
                /*
```

```
                 * Error occured either in transmission, or the server is down.
                 * Regardless of the error, client will continually wait till
                 * it can reconnect to the server. This behavior only stops
                 * when the user explicitly stops streaming frome this server.
                 */
                perror(node->window_name);
                if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype))
{
                    perror(__FUNCTION__);
                    return;
                }

                close(node->socket);

                while (init_network(node)) {
                    sleep(3);
                }
                /* Client is reset, data must be gathered again */
                reset = true;
                /* Resets default cancelability of thread */
                if (pthread_setcanceltype(oldtype, NULL)) {
                    perror(__FUNCTION__);
                    return;
                }
                break;
            }
        }
        if (!reset){
            pthread_mutex_lock(&node->mutex);

            for (i = 0, k = 0; i < node->frame->height; i++) {
                for (j = 0; j < node->frame->width; j++) {
                    ((uchar*)(node->frame->imageData + i * node->frame-
>widthStep))[j] = sockdata[k++];
                }
            }

            node->new_data = true;

            pthread_mutex_unlock(&node->mutex);

            /* Check for any pending cancels*/
            pthread_testcancel();

            usleep(3000);
        } else {
            reset = false;
        }
    }
}
```

B.2 SERVER.C

- Continually updates the frame, which contains the video stream data:

```
/*
 * Continuously updates the frames coming from the capture.
 */
void *get_frame(void) {

    while (quit != 'q') {
        /* Get a frame from the camera */
        frame = cvQueryFrame(capture);
        if (!frame) {
            quit = 'q';
            return (void *)FAILURE;
            break;
        }
        /*
         * JPEG compression does not work yet. But capture frame should be
compressed
         * either before or after the frame is converted into grayscale.
         */

        /**
         * Converts the frame to grayscale. Since the grayscaled image will be
         * transmitted to the master, it must be thread safe.
         **/
        pthread_mutex_lock(&mutex);

        cvCvtColor(frame, tx_frame, CV_BGR2GRAY);
        new_data = true;
        /* display images in server window as well */
        cvShowImage(window_name, frame);

        pthread_mutex_unlock(&mutex);

        quit = cvWaitKey(30);
    }
    printf("Silent quit...?\n");

    return (void *)SUCCESS;
}
```

- Code responsible for streaming the frame to a client:

```
/*
 * Establashes a connection with a client. If a client drops the connection
 * to the server, the server will block until accept() returns with a new
 * client to connect to
 */
void *stream_frame(void *arg) {
    int frame_size = tx_frame->imageSize;
    int bytes_sent = 0;
    int retv = 0;
    int oldtype = 0;

    if (-1 == (conn->client_sock = accept(conn->server_sock, NULL, NULL))) {
        return (void *)FAILURE;
    }

    while (quit != 'q') {
        /* Sends gray-scaled frame. */
        pthread_mutex_lock(&mutex);
```

```
        if (new_data) {
            bytes_sent = send(conn->client_sock, tx_frame->imageData, frame_size,
0);
            new_data = false;
        }
        pthread_mutex_unlock(&mutex);

        /* Check to see if the data sent is correct */
        if (bytes_sent != frame_size) {
            if (-1 == close(conn->client_sock)) {
                perror(__FUNCTION__);
            }
            fprintf(stderr, "Client connection terminated, accepting new
connection...\n");
            if (-1 == pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,
&oldtype)) {
                perror(__FUNCTION__);
                return (void *)FAILURE;
            }
            if (-1 == (conn->client_sock = accept(conn->server_sock, NULL, NULL)))
{
                perror("reconnect error");
                return (void *)FAILURE;
            }
            if (-1 == pthread_setcanceltype(oldtype, NULL)) {
                perror(__FUNCTION__);
                return (void *)FAILURE;
            }
        }
        usleep(3000);
    }

    return (void *)retv;
}
```