SPECS DOC

Screen Sharing

Prepared by Satyam Mishra

OVERVIEW

Screen Sharing module is a very important module in the project which will help clients to share their screen to the host. Multiple users can simultaneously share a screen to the host. The host will have an option to pin users which will maximize the tile for that client.

OBJECTIVES

Client Side Objectives

1. Capture the image as a BITMAP.
2. Comparing this image with the previous image.
3. Sending the changed pixel to the server via the network module.

Server Side Objectives

1. Take data from the network module.
2. Get the image from the pixel values obtained from the network module.
3. Update the current frame visible in the hosts screen

CLASS EXPLANATION

screenShareClient

```
/* this is networking class object
    Communicator Obj
/* this variable will tell if the screen is required by the host or not
    bool req_screen
```

```
/* function called when the screen sharing starts, it will call the capture and
process  functions
        StartSharing()


/* this function will stop screen share
        StopSharing()
/* this will handle the packet received from Network Module
        OnDataReceived()
/* send image packet to the network module
        SendImagePacket()
/* this will share a confirmation packet to inform server that connection is still
alive
        SendSharingConfirmationPacket()
/* the main function for sharing image over network
        CapNProc()
/* this will take screenshots and put it in the capture frame queue
        Capture()
/* this function will do the image processing and take out those pixels which are
different in old and new images
        Process()
/* Since working with 1080p might be difficult therefore we need a function to
compress image to 720p
        Compress()
```

SharedScreen:

IP
Name
FrameQueue
FinalImageQueue
CurrentImage
Bool Pinned
Timer timer

This class will contain basic information about the screen that is being shared on the server.

ScreenShareServer:

```
/* this is networking class object
Networking communicator;
/* this map will hold all the shared screen objects and the unique ids
Map<IP, SharedScreen> subscribers;


Constructor()
        // instantiate networking object
        // subscribe to the networking for OnDataReceive()


OnDataReceive(data)
        // Receive packet from the client via networking
        // Based on the packet header, do further processing
                // SUBSCRIBE → SubscribeUser(ip)
                // UNSUBSCRIBE  → UnsubscribeUser(ip)
                // IMAGE → Stitch(ip, converted array)
                // CONFIRMATION → UpdateTimer(ip)

SubscribeUser(ip)
        // add this ip to the map

UnSubscribe(ip)
        // remove this ip from the map

BroadcastClientsInfoOfSS(currentWindowUsers)
        // resolution of the image to send to each client
        // whether to send the packet or not
```

```
Stitch(ip, new_res, list of : {x, y, (R,G,B)})

UpdateTimer(ip)
        // start/reset timer for the ip with the OnTimeOut()

OnTimeOut(ip)
        // callback for timer
```

ALGORITHMS

Stitch

```
Stitch(ip, new_res, list of : {x, y, (R,G,B)}):

        screenShareObj = subscribers.get(ip)

        ImageFrameList = screenShareObj.FrameQueue.pop()

        for (auto {x, y, (R, G, B)} : list) {

                currentImage[x][y] = (R, G, B);

        }
```

Analysis

Here we are sending only those pixels which got changed in the next frame and their coordinates. Using these pixel values at the server end we can iterate over the current image frame and then update the pixel value of the image to get the next frame. The complexity of this algorithm will be of the order of size of the list sent as an argument to the stitch function. This is a naive algorithm but we will later try to optimize this algorithm to make it faster.

Capture and Process functions

```
process(){

        while(True) {

        if(required_screen==false) continue

        /* in the case for some reason we get a signal to stop screen share for this
        client we pass this iteration till we get the signal to again share it. */

        prev_fr = curr_fr
```

```
        /* here we assign the current frame to the previous frame for comparison with
        the new frame*/

        curr_fr = captured_fr.pop()

        /* we will pop a new frame from the queue and will assign it to the current
        frame */

        if( old_res != new_res )

                s_fr = Compress(req_sz, curr_fr)

                upate_res(new_res, old_res)

                /* if the old resolution is not the same as the new resolution which is
                requested by the server, we compress the current frame to the desired
                resolution  */

        else :

                s_fr = compare(prev_fr, curr_fr) processed_fr.push(s_fr)

                /* in case the old resolution is the same as the new then just take out
                the changed part of the image*/

        processed_fr.push(s_fr)

        /* push the processed frame into the queue */

        }

}
```

Now in this process function we will be taking out frames from the queue and after processing we will insert it into the processed frame queue.

```
Compress(req_sz, curr_fr)

        new_fr = compression_algo()

        return conv_to_list(new_fr)

        /* utility function to compress the image */

}
```

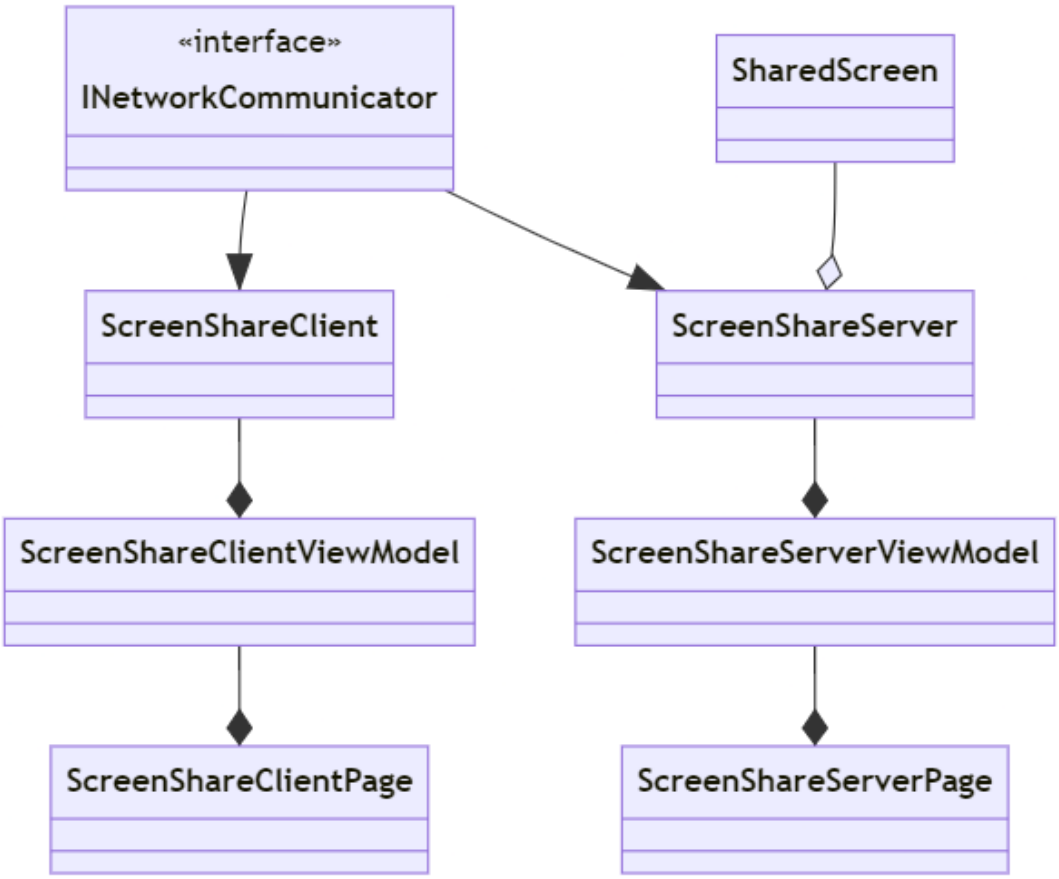
In the above mentioned code we will be continuously taking screenshots and if that screenshot needs to be sent over the network then push it into the captured frame queue.

Performance

We are doing the following things to optimize the performance of our program.

1. Patching :- Sharing the whole image is a very costly job to do. It will lead to a heavy network usage. In order to prevent this we are only sending those bits of the image which are different from the previous image. This will be a lot easier to send as generally there won't be much change happening in the screen between two frames.
2. We even use multithreading in our program to get parallel execution. For example, each client who is sharing a screen is independent of each other. So we are using multithreading here by which all of the clients will share the screen simultaneously.
3. We are using queue data structure to remove the dependency of one module on the other. So if one module is taking screenshots and another module wants those screenshots then instead of waiting for it we can insert it in a queue and let that other module take it from the queue as and when it is free.

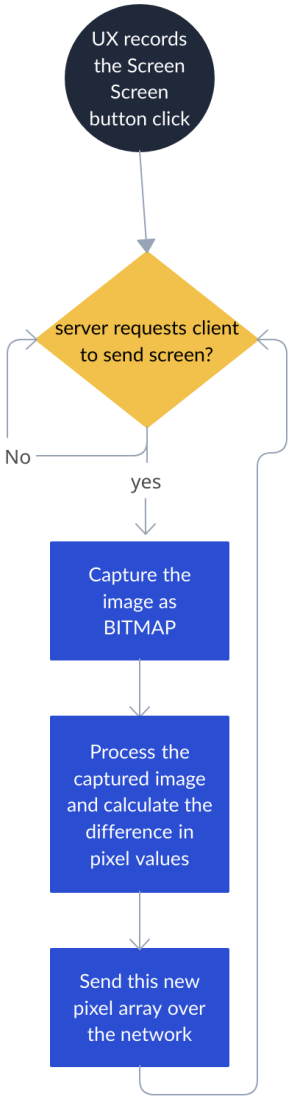
CLASS DIAGRAM

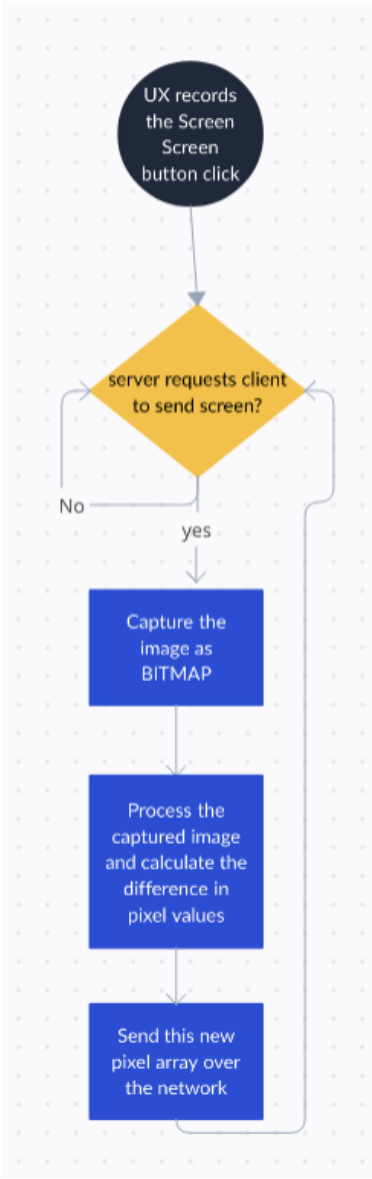


the class diagram for our screen sharing module.

ACTIVITY DIAGRAM

Activity diagram for client side will be as shown.





CONCLUSIONS

This document is an outline of how we are going to implement screen capturing and sharing. We discussed the time and space analysis of screen sharing. We even explained the structure of the code that we will be implementing.

FUTURE WORK

1. We may introduce that the server can also share its screen, right now only clients (students) can share them.
2. We can enhance the patching algorithm, in which instead of traversing all the pixels, if we can store multiple pixels information in the same pixels which is kind of compression will save more time.
3. Rightnow when the server (professor) switches the screen or changes the tab, there may be some latency in showing the screen of a new user on screen. We can improve that if we cache some screens.