CS5617 Software Engineering

Design Specification

Mayank Singla, 111901030, Screen Share Module Team Member

Overview

The screen share module will handle the implementation of the screen share feature in our application. This module will handle the functionality of how the screen will be captured from the client side and will be sent over the network to the server side and displayed there. It also handles efficiently the packet sending process in case of multiple clients and allows the server to pin/unpin the screen of a particular client.

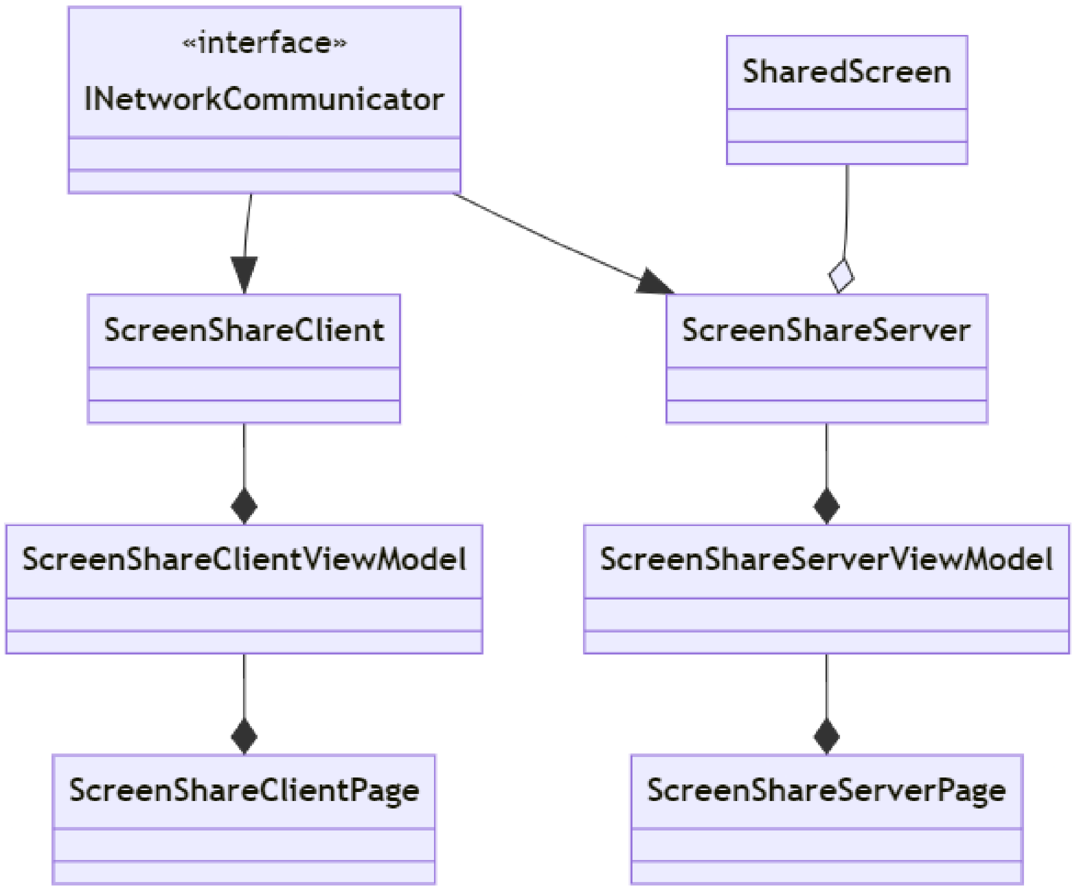
Objective

1. Implement the ViewModel which links the model and the view
2. Handle the logic for requests from the server and responses from the client for screen sharing
3. **On the client side:**
    a. On starting screen sharing, subscribe to the server for screen sharing
    b. The client will receive information about the resolution of the image and whether to send the packets or not from the server and will send the image in the packets as per the request
    c. If the client is sharing the screen, it will start sending the confirmation packets to the networking module to tell that the client is presenting the screen
        i. The networking module will send this packet to the screen share module of the server, and in case of timeout, it will unsubscribe the client
        ii. It will also broadcast this packet to the dashboard module of the server and the client, and in case of timeout, it will remove the screen share icon alongside the name in the list of the clients
    d. On stopping screen sharing, unsubscribe from the server
4. **On the server side:**
    a. The server on receiving every subscription request will add that client as the subscriber

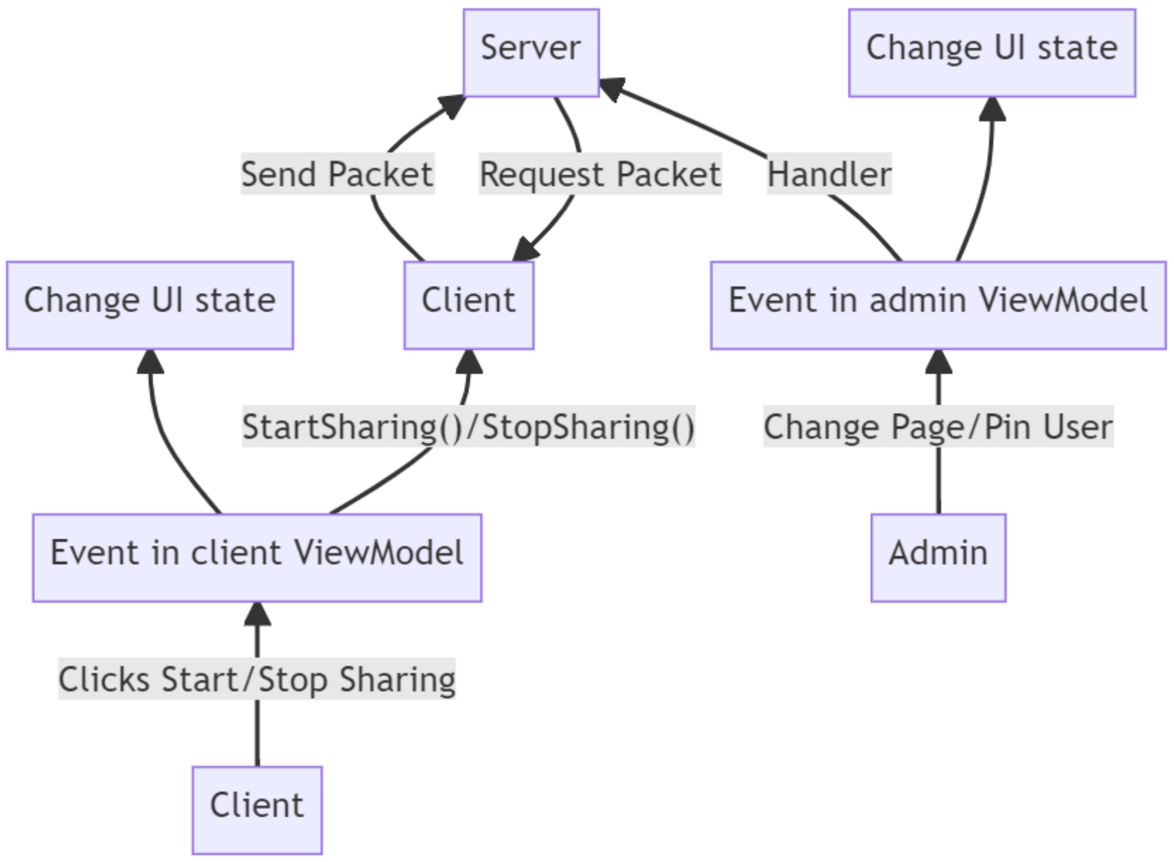
b. It will publish to all the subscribers which are on the current page of the server, the resolution of the image and whether that subscriber should send packets or not i.e. on changing the page, it will ask the previous clients to stop sending the packets and it will ask the new clients to start sending the packets and it will notify the subscribers again on every page change

c. On marking a client as pinned, we should enlarge the view of that client to the full page and ask for a higher-resolution image from that client by notifying it

Class Diagram

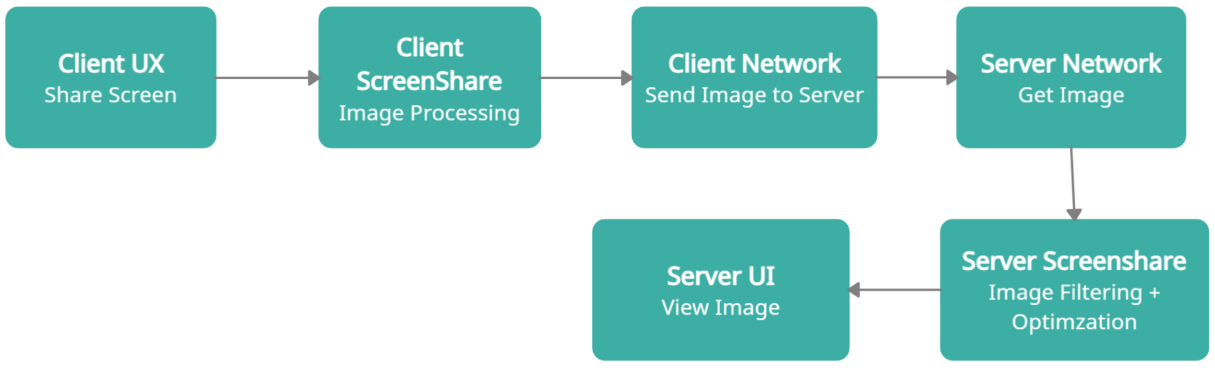


Activity Diagram



Design Analysis

In the current implementation, the student's machines will act as client machines, whereas the administrator's machine will be the server machine. There will be only one server machine. Multiple clients may try to share their screens at the same time.



The above activity diagram shows the basic flow of how the Screenshare component is going to work. Currently, screen share is only supported by the clients and not by the server. Hence the entire flow is single-directional.

- The screen share framework will only have one instance throughout the entire execution of the application. Based on whether it is the client machine or the server machine, it will create only one instance of the class **ScreenShareServer** or **ScreenShareClient**. This pattern is commonly referred to as the Singleton Object pattern.
- To send a packet to any other module or from one client to the server or vice versa, this communication will happen through the networking module. So all the instances of the above classes will subscribe to the networking module in a Publisher-Subscriber pattern. This is done so that they don't have to wait for the packets by continuously listening for the packets, but instead can simply subscribe to the server and whenever a notification is received, they will be able to react to it at that time itself.

Challenges

- The client can send multiple packets containing the screenshots of the displayed screen one after the other. We need to correctly process and display them all so that it will appear like a live video.
  To solve this problem, the server will be maintaining a queue corresponding to each client and on receiving the images from the client, we will enqueue them into that queue and process the packets from that queue. This way we won't lose the processing of any intermediate image and will display all the images received from the client to make it a live video.
- The server will be receiving the packets containing the shared screen image from multiple clients. The processing done on the image received from one client should not block the processing done on the other and this all should be done and reflected on the view parallelly.
  To solve this problem, we will be processing the packets in the queue of each client by creating a separate thread for each client. This way using multi-threading we will be able to process and display the images of each client parallelly without blocking the other.
- There can be latency when the server keeps on changing the page as on every page change, we will ask the previous clients to stop sending the packets and the new clients to start sending the packets. So there can be a lag between page switches.

To solve this problem, the approach that we have thought of for now is we will also allow the clients adjacent to the currently active window to send their packets so that on switching the page we already have the packets of the other clients on the new page and this way we can reduce some lag between a page switch.

- **Implement the pagination logic**:
  Skip `MAX_TILES*(CURR_PAGE - 1)` clients from the beginning of the list
  Send `min(MAX_TILES, remaing_tiles)` from the current list

Interfaces and Classes

We won't be having any interface implemented by us as all the functionalities will be implemented under the defined classes as we are using a singleton object pattern. Our classes will be inheriting the `INetworkCommunicator` interface provided by the networking module to achieve the publisher-subscriber relationship for sending and receiving packets.

The `ScreenShareClientViewModel` class below handles the ViewModel logic on the client-side machine.

```
public class ScreenShareClientViewModel {

    // Will trigger the function `StartSharing` in the model class
    // It will be called when the client clicks on the start sharing
    // button
    public void StartSharing() {...}

    // Will trigger the function `StopSharing` in the model
    // It will be called when the client clicks on the stop sharing
    // button
    public void StopSharing() {...}
}
```

The `ScreenShareClient` class below handles the Model logic on the client-side machine.

```
public class ScreenShareClient : INetworkCommunicator {

    // The Networking object to send packets and subscribe to it
    private INetworking communicator;

    // Tells whether the client is sending the packets or not
    private bool req_screen;

    // The old resolution that the client was sending
    // and the new resolution which the server is requesting
    private Resolution old_res, new_res;

    // The queue in which the image will be enqueued after
    // capturing it and processing it
    private Queue<Frame> captured_frame, processed_frame;

    // The threads that were started to capture and process the
screen
    // and to send the packets to the network
    private Thread capture_thread,
                   process_thread,
                   sendImage_thread,
                   sendConfirmation_thread;

    // The constructor for this class. It will instantiate the
    // networking object and subscribe to it for `OnDataReceive()`
    public ScreenShareClient() {...}

    // This method will be called by the ViewModel
    // Sends subscribe requests to the server
    // Call the methods `sendImagePacket` and `capnProc()`
    public void StartSharing() {...}

    // This method will be called by the ViewModel
    // Sends unsubscribe request to the server
    // Stop the currently running threads and clears the queue
    public void StopSharing() {...}

    // This method will be invoked by the networking team
    // This will be the request/response packets from the server
    public void OnDataReceive(DataObject) {...}
    // This method will start the thread in which it will read
    // the processed frames from that queue and start sending
    // those images to the server. It will only send the images
    // if the server has requested for sending the packets
    // i.e. the req_screen variable is true
    public void SendImagePacket() {...}

    // This method will start the thread which it will keep on
    // sending the packet having the header as confirmation to let
    // the server and the dashboard know that the client is not
    // disconnected and is still presenting his screen.
    public void SendConfirmationPacket() {...}
}
```

The `ScreenShareServerViewModel` class below handles the ViewModel logic on the server-side machine.

```
public class ScreenShareServerViewModel {
    // Keeps track of the current page that the server is viewing
    public int currentPage = 1;

    // The maximum number of tiles of the shared screens
    // on a single page that will be shown to the server
    public const int MAX_TILES = 9;

    // For each client in the currently active window, it will start
a
    // separate thread in which the final processed images of clients
    // will be dequeued and sent to the view
    private List<IP, Thread> currWinClients;

    // The constructor will instantiate the `ScreenShareServer`
object
    // It will start a thread for the method `ReadImages()`
    public ScreenShareServerViewModel() {...}

    // It will compute the current window clients from the active
list
    // of the subscribers using the pagination logic, and start the
    // thread for these clients using `SendImageToView()` method
    public void ReadImages() {...}

    // It will read the final processed image of the client and send
    // them to the view. This method will run for an infinite while
    // loop and as soon as it receives an image in the queue, it
    // will dequeue it and send it to the view
    public void SendImageToView() {...}

    // Update the current page number and call the `Broadcast` method
    // in the server class. Ask the previous clients to stop sending
    // packets and stop their threads. Ask the new clients to start
    // sending their packets
    public void OnPageChange(PageNum) {...}

    // Mark the client as pinned. Switch to the page of that client
    // and call the `OnPageChange(newPage)` method
    public void OnPin(IP) {...}
}
```

The `SharedScreen` class represents an object that stored the relevant information on the server about the screen being shared by the client.

```
public class SharedScreen {
    // The IP of the client sharing this screen
    public string ip;

    // The name of the client sharing this screen
    public string name;

    // The images received from the clients as packets
    public Queue<Frame> frame_queue;

    // The images which will be received after patching the previous
    // screen image of the client with the new image and
    // ready to be displayed
    public Queue<Image> final_image_queue;

    // The current screen image of the client displayed
    public Image current_image;

    // Whether the client is pinned or not
    public bool pinned;

    // Timer which keeps track of the time the confirmation packet
was
    // received that the client is presenting the screen
    public Timer timer;
}
```

The `ScreenShareServer` class below handles the Model logic on the client-side machine.

```
public class ScreenShareServer : INetworkCommunicator {
    // The Networking object to send packets and subscribe to it
    private INetworking communicator;

    // The map between each IP and the shared screen object for
    // all the active subscribers
    public Map<IP, SharedScreen> subscribers;

    // The constructor for this class. It will instantiate the
    // networking object and subscribe to it for `OnDataReceive()`
    public ScreenShareServer() {...}

    // This method will be invoked by the networking team
    // This will be the response packets from the clients
    // Based on the header in the packet received, do further
    // processing as follows:
    /*
        SUBSCRIBE    --> SubscribeClient(ip)
        UNSUBSCRIBE  --> UnsubscribeClient(ip)
        IMAGE        --> Stitch(ip, converted_image)
        CONFIRMATION --> UpdateTimer(ip)
    */
    public void OnDataReceive(DataObject) {...}

    // Add this client to the map
    public void SubscribeClient(IP) {...}

    // Remove this client from the map
    public void UnsubscribeClient(IP) {...}

    // Tell the clients the information about the resolution
    // of the image to be sent and whether to send the image or not
    public void BroadcastClients(CurrWinClients) {...}

    // Start/Reset the timer for the client with the `OnTimeOut()`
    public void UpdateTimer(IP) {...}

    // Callback method for the timer. It will unsubscribe the client.
    public void OnTimeOut(IP) {...}
}
```

Analysis

Performance

- Let `N` be the number of currently active subscribers in the lab session. As discussed, we will be maintaining a queue for each client in which we will be enqueuing the images received from them as packets. This operation will be done quite often and we will need to update the queue for each client on every packet received. If we would have maintained a `List<SharedScreen>`, then for each packet received, we would have to first find out that object from that list that would have caused `O(N)` operations every time. Now using a `Map<IP, SharedScreen>` reduced that operation cost to `O(logN)` per operation as internally the map is implemented using a BST.
- Instead of accepting the packets from all the clients, the server is only accepting the packets from the clients that are currently displayed on the active page of the server. This way we are reducing the network load by a huge amount.
- Instead of processing the images of the clients one after the other, we will be processing them in parallel using multi-threading. This way we will be able to display the screens to the server in parallel and it will reduce the loading time of the screens of the client on the server screen.

Security

Each thread is either enqueuing in a queue or dequeuing from a queue, and the dequeuing is only done when the queue is not empty. We do not have any critical section code in the API between the threads and there are no shared resources between the threads on which a race condition will arise, hence the API is thread-safe ensuring consistency.

Summary and Conclusions

- We will be displaying the screens of multiple users over multiple pages with a certain maximum number of screens being displayed on one page
- For handling a continuous stream of images, everywhere a queue is used for processing
- The images are processed in separate threads everywhere to allow parallel computation
- Only the clients that are present on the active page of the admin and the clients adjacent to that page will be asked to send the packets and others will be asked to stop sending the packets to reduce network load
- The server is maintaining a map between the IP and its objects, so all the operations for searching/updating are logarithmic
- The client will send a confirmation packet to the server to let it know that it is not disconnected and is still sharing the screen

Future Work

1. Admin should be able to share the screen with all the clients in the teaching mode
2. Allow client to share a particular application (a window, tab, or full screen), allowed by admin if not in test mode
3. Allow sharing screens at different resolution
4. Allow multiple admin support