CS 5617 Software Engineering

# Specification Document

Team Member : Screen Share Module
Amish Ashish Saxena ( 111901008 )

## Overview

The PlexShare application is being developed for the purpose of monitoring the multiple client sessions from a host machine. It provides a variety of features like shareable whiteboard, chat, file sharing and submission. But the essence of the application (which is monitoring sessions) comes under the ScreenSharing module. It features capturing screens of different users connected to the particular host and sharing them with the host. The host can then look at multiple screens at one time or pin some screens to have a detailed (zoomed) view of the particular user's screen.

### Motivation :

In case of an application that is handling quite an amount of continuous data flow, it becomes an utmost necessity to make sure that each of the modules work as efficiently as possible. So, as a part of the processing sub-team of the ScreenSharing module team, the task at hand is to develop a fast, reliable and efficient mechanism for screen share processing at the client as well as the server side.

## Objectives

The major objectives of the processing component on Client Side include :

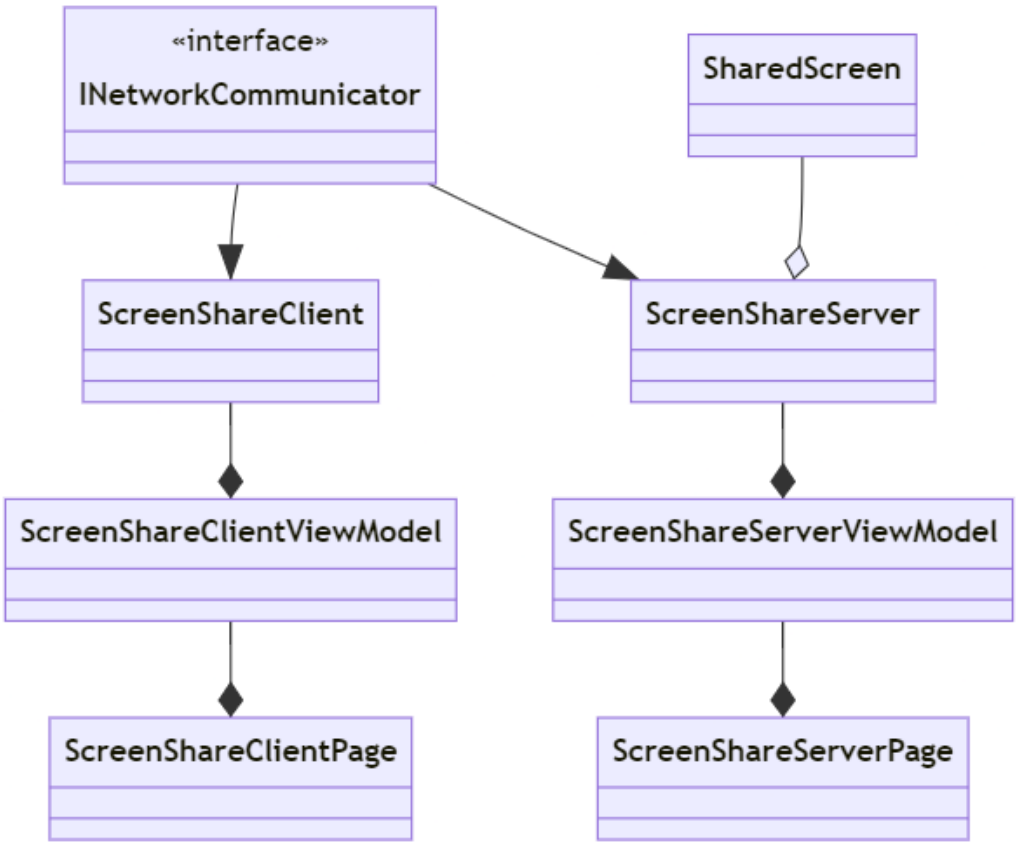
- Create a  thread to Capture the Client's Screen and convert it to a desired lower resolution.
- Create a thread to Process the captured image :
    - Check if the screen has changed, and calculate the pixels to update at the server side
    - Adjust resolutions based on the request from the server (high resolution for pinned screens and lower resolution for group view)
    - Use compression algorithms to achieve a desirable smaller size, since we want to transmit over the network
- Transmit the patches of the screen that have changed with respect to the previous frame. Make it available to the ViewModel for sending to the server via the Networking module.
- Enable interaction between the two threads (Capture and Process) to ensure concurrency ensures speedy execution while preserving the order of captured frames.

The major objectives of the processing component on Client Side include :

- Receiving updated frames
    - The Host server being subscribed to each of the clients, get notified whenever it receives the data containing the patches to be added.
- Patching the images
    - Implement the algorithm for patching of the images on the server-side.
    - Maintain a map of *SharedScreen* objects pertaining to each of the connected users. Each object contains a queue of the frames (after patching has been done) that need to be displayed on the server-side to the host.

## Class Diagram

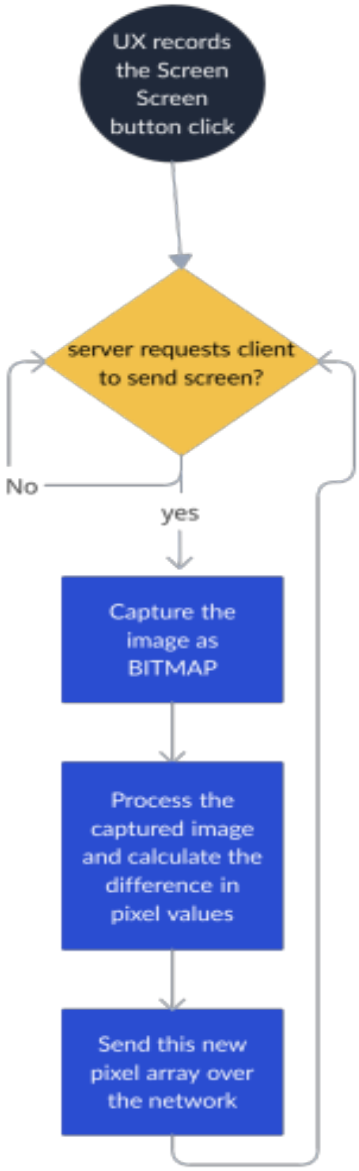
The class diagram for the Screen Share module would look as :




THe processing would be done in the ScreenShareClient Class. Details of which are mentioned below in the Class Explanation section.

## Activity Diagram

The activity diagram representing the flow of the image capture processing is shown below:



## Class Explanation

We have the *ScreenShareClient* class at the client side. The class has the following members and methods :

- `Communicator Object` // networking class object
- `bool req_screen` // variable marks if the server is demanding for screen
- `Thread capture_thread , process_thread` // processing threads
- `Thread sendImage_thread, sendConfirmation_thread` // threads for ViewModel
- `Queue Capture_frame` // maintains the frames that have been captured
- `Queue Processed_frame` // maintains the processed frames to be sent

- `StartSharing()` // triggered for starting the screen share
- `StopSharing()` // triggered for stopping the screen share
- `OnDataReceived()` // for handling packets received from Network Module
- `SendImagePacket()` // send image patching data to the network module
- `SendSharingConfirmationPacket()` // for checking if connection is still alive
- `CapNProc()` // the main processing function
- `Capture()` // runs on the capture_thread and pushes to captured_frame
- `Process()` // runs on process_thread and pushes to processed_frame
- `Compress()` // run the compression algorithm and returns list of changes in pixels

The processing is done using the following functions of the above mentioned class :

CapNProc() , Capture(), Process() and Compress()

Their algorithms are explained as below :

## Algorithms

The CapNProc() function is called from the ViewModel of the ScreenShare module. It has the functions Capture() and Process() inside it. The algorithms for those functions are written below with some implementation extraction as :

```
Capture()
      while(True)
         if(!req_screen)
            continue
          captured_image = screenshot()
          Frame fr = change_res(captured_image)
          captured_frame.push(fr)
```

```
Process()
      while(True)

          if(!req_screen)
                continue

            prev_frame = curr_frame
            curr_frame = captured_frame.pop()

            if( old_res != new_res )
                  send_frame = Compress(req_sz, curr_frame)
                  upate_res(new_res, old_res)
            else :
                send_frame = compare(prev_frame, curr_frame)
            processed_frame.push(send_frame)
```

```
Compress(req_sz, curr_frame)

      Frame new_frame = CompressionAlgorithm(curr_frame, req_sz)
      return convert_to_list(new_frame)
```

## Analysis

### Capture Screen BenchMarking :

Now that we have the algorithm for the Capturing and Processing, we try to measure our different approaches and select the best (most efficient) of them.

Assuming that most of the screens (laptops / desktops) these days have a 1080p resolution, the captured images for each frame would also have the same resolution.  We want to find the pixels that have been changed between consecutive frames and send them to the server for patching. Therefore, the task is to find the pixels that have changed between the two BitMap images that we have captured. SInce bitmap images store the images' pixel information in the form of a matrix with each value being the (R,G,B) tuple, we can easily compare frames. Below shows code and the time taken for this approach .

We see that this method although sounds okay, but is really inefficient as comparing just 2 frames in the below code takes about 4-5 secs. This is not a realistical way of sharing screen as we obviously want the sharing to be done at atleast 10-15 FPS.

Normal for loop Benchmarking :

```
using System.Drawing;
using System.Diagnostics;
using System.Threading;

Bitmap img = new
Bitmap(@"C:\\Users\\amish\\Pictures\\Screenshots\\ss1.png");
Bitmap img1 = new
Bitmap(@"C:\\Users\\amish\\Pictures\\Screenshots\\ss2.png");
int count = 0;

Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();
for (int i = 0; i < img.Width; i++)  {
    for (int j = 0; j < img.Height; j++)  {
        Color pixel1 = img.GetPixel(i, j);
```

```
        Color pixel2 = img1.GetPixel(i, j);
        if (pixel1 != pixel2)
            count++;
    }
}
stopwatch.Stop();
Console.WriteLine("Difference b/w 2 images is {0} pixels", count);
Console.WriteLine("Elapsed Time is {0} ms", stopwatch.ElapsedMilliseconds);
```

**Output** :

Difference b/w 2 images is 128677 pixels

Elapsed Time is 5372 ms

D:\a\SE2022\bm1\BenchMark1\BenchMark1\bin\Debug\net6.0\BenchMark1.exe (process 11424) exited with code 0.

Therefore, we try another approach to get a better speed for comparing frames. The newer method uses the inbuilt BitMapdata datatype in C# , manipulate it and then compare the pixels. The result is clearly better and we are able to get a comparison of 2 images of 1080p in about 70 - 95ms.

This can be further improved by first making the 1080p screen captures into images of 720p resolution and then using the above mentioned algorithm. This gives us a significant boost and the comparison times go down to 15-20 ms per comparison. This along with considering the other processing overheads can help us achieve sufficient FPS, thus making the screen share look smoother.

```
using System.Diagnostics;
using System.Drawing;
using System.Drawing.Imaging;
using System.Runtime.InteropServices;

Bitmap img = new
Bitmap(@"C:\\Users\\amish\\Pictures\\Screenshots\\ss1.png");
Bitmap img1 = new
Bitmap(@"C:\\Users\\amish\\Pictures\\Screenshots\\ss2.png");

int count = 0;

Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();

bool ProcessUsingLockbits(Bitmap processedBitmap, Bitmap processedBitmap1)
{
    BitmapData bitmapData = processedBitmap.LockBits(new Rectangle(0, 0,
processedBitmap.Width, processedBitmap.Height), ImageLockMode.ReadWrite,
processedBitmap.PixelFormat);

    int bytesPerPixel =
Bitmap.GetPixelFormatSize(processedBitmap.PixelFormat) / 8;
    int byteCount = bitmapData.Stride * processedBitmap.Height;
    byte[] pixels = new byte[byteCount];
    IntPtr ptrFirstPixel = bitmapData.Scan0;
    Marshal.Copy(ptrFirstPixel, pixels, 0, pixels.Length);
    int heightInPixels = bitmapData.Height;
    int widthInBytes = bitmapData.Width * bytesPerPixel;
    processedBitmap.UnlockBits(bitmapData);

    BitmapData bitmapData1 = processedBitmap1.LockBits(new Rectangle(0, 0,
processedBitmap1.Width, processedBitmap1.Height), ImageLockMode.ReadWrite,
processedBitmap1.PixelFormat);

    int bytesPerPixel1 =
Bitmap.GetPixelFormatSize(processedBitmap1.PixelFormat) / 8;
    int byteCount1 = bitmapData1.Stride * processedBitmap1.Height;
    byte[] pixels1 = new byte[byteCount1];
    IntPtr ptrFirstPixel1 = bitmapData1.Scan0;
```

```
    Marshal.Copy(ptrFirstPixel1, pixels1, 0, pixels1.Length);
    int heightInPixels1 = bitmapData1.Height;
    int widthInBytes1 = bitmapData1.Width * bytesPerPixel1;

    processedBitmap1.UnlockBits(bitmapData1);


    for (int y = 0; y < heightInPixels; y++)
    {
        int currentLine = y * bitmapData.Stride;
        int currentLine1 = y * bitmapData1.Stride;
        for (int x = 0; x < widthInBytes; x = x + bytesPerPixel)
        {
            int oldBlue = pixels[currentLine + x];
            int oldGreen = pixels[currentLine + x + 1];
            int oldRed = pixels[currentLine + x + 2];

            int newBlue = pixels1[currentLine1 + x];
            int newGreen = pixels1[currentLine1 + x + 1];
            int newRed = pixels1[currentLine1 + x + 2];

            if (oldBlue != newBlue || oldGreen != newGreen || oldRed !=
newRed)
                count++;
        }
    }
    return true;
}

ProcessUsingLockbits(img, img1);
stopwatch.Stop();
Console.WriteLine("Difference b/w 2 images is {0} pixels", count);
Console.WriteLine("Elapsed Time is {0} ms", stopwatch.ElapsedMilliseconds);
```

**Output** :

Difference b/w 2 images is 128677 pixels

Elapsed Time is 90 ms

D:\a\SE2022\bm1\BenchMark1\BenchMark1\bin\Debug\net6.0\BenchMark1.exe (process 20640) exited with code 0.

### Multi-Threading

Another aspect used to improve the performance is utilizing the multiple cores of the processor to perform the capturing and processing tasks concurrently. We have 2 different threads for process and capture that run parallelly and interact with each other through a queue. Capture thread pushes the captured frames to the Captured_frame queue whereas the Process thread pops from it and then processes it. This makes sure that while capture is taking one frame and pushing it to the queue, the process is processing the previous frame. Thus making sure that the capture function does not have to wait for the completion of the process function for one frame. The utilization of the queue also helps to ensure the correct ordering of the frames at the same time.

### Latency Handling

There can be a scenario where a packet containing the updated pixels' information has left the client and is yet to reach the server but the server changes the resolution for that particular user's screen. In this case, we drop this certain packet and notify the client about the change in the required screen resolution and the client sends a whole new image of the resolution for the first time and then subsequently sends the update pixels' for the new resolution.

## Summary & Conclusion

This document discusses in detail about the implementation of the core screen sharing component . We have designed the class for the Client-side and Server-side that constitutes the Model of the MVVM design methodology. Different optimisation techniques are utilized to have a faster, reliable and smoother screen sharing in the application. Combining the View, ViewModel and the Model parts of the screen sharing module will integrate with the other components of the PlexShare application.

## Future Work

- Screen sharing for the host server.
- Further optimization of the compression and patching algorithms.
- Support for multiple host servers.
- Screen sharing is not limited to the whole screen but an option for sharing particular applications or tabs.
- Users as well as the Host can have the option to select the resolution of choice for sharing and viewing.

## References


- https://learn.microsoft.com/en-us/dotnet/api/system.drawing.image?view=dotnet-plat-ext-6.0
- https://csharpexamples.com/fast-image-processing-c/
- https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics?view=net-7.0