Hong Kong University of Science & Technology

DisPlay: Mitigating Heavy-duty Networking Latency and Error through Cross-platform Multithreaded Channel Coding and Media Compression

25th Anniversary Project Supervised by Dr. David Rossiter

IWASAKI, Kenta
5-12-2016

1 TABLE OF CONTENTS

2 INTRODUCTION

A substantial percentage of people in the world have a modern smart phone and/or a tablet. Many people have 2 or 3 such devices. However, there is currently a huge wasted opportunity for these devices to work together with each other. For example, why can't a group of students put their devices close to each other in a two by two arrangement and watch a YouTube video together, with a quarter of the video being shown on each device? On a larger scale, an 80 inch monitor may cost in the region of HK$80,000. Yet by using smart devices in conjunction with each other the cost of an equivalent system would be a fraction of the price and can be used in multiple innovative ways which a flat 2D display cannot.

As a result, we would like to introduce the system DisPlay which would provide such a function to its users to simplify the interconnectivity of multiple device's screens for the sake of convenience.

3 BREATH OF DEVELOPMENT

For the length of this project creation and implementation, the prime goal for the project was to complete the one-to-N implementation for DisPlay as denoted below and in **Figure 1**:

"*In this mode the display of 1 individual device is shown on n devices. There are multiple applications where this can be of great benefit. For example, in a classroom environment a teacher may project the display of his device directly to the display of all students' devices, to help with tutoring. This could have direct cost benefits. For example, a school or University would not need to purchase a large display device for each classroom such as a large monitor or projector, as is typically the case now*."

Figure 1: A demonstration of the one-to-N mode where multiple students are capable of seeing the screen of the teacher on their own device.

The one-to-N implementation is to be capable of providing a frame of reference of one device and transmit it to N amount of devices through a series of computer networking, operating system, and graphics techniques.

For the sake of this implementation, Java and the cross-platform OpenGL-based framework LibGDX was used primarily because of its capabilities in accessing device-centric sensors such as the accelerometer and gyroscope [4]. With the incorporation of these sensors, the project is capable of providing a sleeker human experience given that angle & position of the device's viewport is incorporated when it comes towards screen frame projection calculations.

Although the combination of these sensors alongside other sensors may yield greater accuracy in the system's display attunements, it comes at the cost of lack of cross-compatibility which is why we chose the most commonly incorporated sensors within a smart phone/tablet device nowadays being the accelerometer and gyroscope.

The network implementation for the entire project was done with LibGDX's cross-platform TCP sockets as well, with the majority of image streaming/buffer operations done through LibGDX's thread-safe data structure implementations [4] as shown in **Figure 2**.

```
public void sendScreenBuffer(byte[] screenshot) {
    if (NetworkManager.SERVER_MODE) {
        try {
            ByteArray screenBuffer = new ByteArray();
            screenBuffer.addAll(screenshot);

            if (lastScreenBuffer != null) {

            } else {
            }

            if (screenBuffer.size > 0) {
                String encodedString = Base64.encodeToString(
                        screenBuffer.toArray(), Base64.DEFAULT);
                System.out.println(encodedString.substring(0, 100));
                System.out.println("Last Character: "
                        + encodedString.charAt(encodedString.length() - 1));
                List<String> encodedBuffer = splitEqually(encodedString,
                        1000);
                out.writeUTF("TRANSFER_REQ_START,"
                        + String.valueOf(encodedString.length()) + ","
                        + String.valueOf(Gdx.graphics.getWidth()) + ","
                        + String.valueOf(Gdx.graphics.getHeight()));
                for (String data : encodedBuffer) {
                    out.writeUTF(data);
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Figure 2: Implementation for the screen image data buffer receiver and handler through a TCP network socket.

4 THREADING MANAGEMENT

Networking and GUI handling on each client node is done in separate threads as shown in **Figure 3**, and a synchronized thread lock is used to pass data between intermittent threads. As for the image processing server medium, each TCP client is socket is managed under one emulated thread instance.

This is done to ensure that lag or any blocking of the networking thread does not affect the user's performance in any way when it comes towards viewing a projected device screen buffer [2]. If the networking were to be done within the same thread context as the GUI rendering, any latencies due to overloading on the network buffer in real-time would cause the GUI's OpenGL context to potentially lag, or even crash given that OpenGL is native and directly holding access to an operating system's display drivers.

```
@Override
public void create() {
    NetworkManager.getInstance().setScreenCapturer(screenCapturer);
    NetworkManager.getInstance().createConnection();

    int w = Gdx.graphics.getWidth();
    int h = Gdx.graphics.getHeight();
    camera = new OrthographicCamera(w, h);
    camera.position.set(w / 2, h / 2, 0);
    camera.update();

    batch = new SpriteBatch();
    stage = new Stage(new FitViewport(w, h));

    texture = new Texture(w, h, Format.RGB888);
    font = new BitmapFont();
}
```

Figure 3: Creation of separate threads for the networking and graphics rendering of DisPlay.

The networking for both server and client node is processed through a buffered input stream which decodes all received bytes through the UTF-8 character set due to its cross-compatibility with many modern-day device operating systems as shown in **Figure 4**. Without the buffering of the network I/O streams, data that has not been processed fast enough would be discarded by the operating system's native network buffer as soon as new data is sent/received. Given the fact that the system is expected to run with no visible latency in real-time, such performance would be unacceptable and hence why multi-threading is used.

```
try {
    out = new DataOutputStream(new BufferedOutputStream(
            socket.getOutputStream()));
    in = new DataInputStream(new BufferedInputStream(
            socket.getInputStream()));
} catch (IOException ex) {
    System.out.println("Failed to open I/O streams for client.");
}
```

Figure 4: Implementation of Buffered I/O Network Streams.

5 COMPUTER NETWORKING

In order to transmit the frame of reference of one device, each frame is compressed as a PNG and is transmitted through a TCP protocol encoded with variable length number of parity bits [1] for the sake of ensured information integrity [3]. The entire network comprises of a series of client nodes, which connects to a server medium hosted on HKUSTs clusters that manages the image processing and encoding of frames from either sides of the nodes. Client nodes which broadcast images/frames accordingly both display and enable other client nodes to receive and display frames of the respective client node's device through the function shown in **Figure 5**.

```
public void sendImageToClients(byte[] imageData) {
    for (ClientThread client : clientThreads) {
        DataOutputStream out = client.getOutputStream();
        try {
            String encodedString = Base64.encodeToString(
                    imageData, Base64.DEFAULT);
            List<String> encodedBuffer = splitEqually(encodedString, 1000);
            out.writeUTF("IMAGE," + String.valueOf(encodedString.length())
                    + "," + String.valueOf(client.getScreenWidth()) + ","
                    + String.valueOf(client.getScreenHeight()));
            for (String data : encodedBuffer) {
                out.writeUTF(data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Figure 5: Implementation of the packet transmitter for multiple client nodes of screen buffer image data.

Hence, all client nodes are capable of being run as either a master or slave when it comes towards reference frames transmission, display, and sharing, though only the first client node shall have priority in terms of screen/device sharing.

The reason for choosing such a network architecture is due to two main necessities of the system:

1. An user receiving network data can at any time choose to be the host of the system which is the person that presents their screens to a number of users, and
2. The loss of connection of one node can be compensated through the receiving of screen buffer data from another node.

Through the implementation of such a system, P2P network integrity can be reinforced and thus users can ensure that they will have a display of the screen with the smallest latency possible [3].

In order to manage a minimal screen buffer size for each client node, a Ping-Pong-based packet infrastructure[3] is implemented which contains information of each client node device's screen size alongside possible inset sizes for the sake of proper rendering of a device's referenced screen buffer frame as shown in **Figure 6**. Having a Ping-Pong-based packet infrastructure also allows the system to identify potentially dead/disconnected nodes within the network system.

```
new Thread(new Runnable() {

    @Override
    public void run() {
        StringBuffer imageData = new StringBuffer();
        int receivedImageSize = -1, receivedScreenWidth = -1, receivedScreenHeight = -1;
        boolean transferring = false;

        while (running) {
            try {
                String command = in.readUTF();
                if (command == null)
                    continue;

                if (transferring) {
                    imageData.append(command);
                    if (imageData.length() >= receivedImageSize) {
                        transferring = false;

                        receivedScreenBuffer = Base64.decode(
                                imageData.toString(), Base64.DEFAULT);
                    }
                }

                String[] data = command.split(",");

                switch (data[0]) {
                case "IMAGE":
                    receivedImageSize = Integer.valueOf(data[1]);
                    receivedScreenWidth = Integer.valueOf(data[2]);
                    receivedScreenHeight = Integer.valueOf(data[3]);
                    transferring = true;

                    imageData = new StringBuffer();
```

Figure 6: Network Packet Handler for client-nodes within the system.

6 Frame Grabbing Techniques

To account for multiple operating systems and devices, the entire program was done in Java as it contains standard libraries for cross-platform device screen frame grabbing in a multitude of image formats. Java's AWT Robot was used for frame grabbing, and is thereafter sent to the image processing server for quality control and compression of the entire image stream for the entire network of clients.

In order to implement screen grabbing operations on mobile devices, interface proxies were created so that a wide scope of operating systems may have their own screen buffer collector implementations [4]. For the scope of this project, an Android and Desktop proxy implementation was done for demonstration purposes as shown in **Figure 7**. Providing interface proxies also allows for other programmers to incorporate DisPlay as an alternative display medium for their own hardware-based systems running on a variety of firmwares.

All proxy implementations provide a byte buffer consisting of the image bytes encoded as 16-bit RGB integers, which is thereafter sent to the mediating image processing server.

```
public DesktopScreenCapture(int width, int height) {
    try {
        robot = new Robot();
        screenSize = new Rectangle(Toolkit.getDefaultToolkit()
                .getScreenSize());
        this.width = (int) (width);
        this.height = (int) (height);
        screenCapture = new BufferedImage(width, height,
                BufferedImage.TYPE_3BYTE_BGR);
    } catch (AWTException ex) {
        System.out
                .println("Unable to capture screen on this desktop platform. Exiting...");
        ex.printStackTrace();
    }
}
```

Figure 7: Desktop proxy implementation for screen buffer image capture.

7 Image Stream Compression

In order to minimize the amount of bits sent throughout the stream to minimize bandwidth usage, the image processing server medium which is programmed in Java manually handles a screenshot image buffer for each client which keeps track of broadcasted image bytes [2]. A simple yet timely XOR operation is applied per image pixel throughout every server tick encoded as 16-bit RGB integers to determine necessary image sectors to update on every client node's display for the sake of mitigating latency after the receiving of the initial screen buffer image frame.

Inclusion of image compression methods from the Java standard library toolkit is done [3] when a given image buffer's size is deemed far too large after base image protocol compression as shown in **Figure 8**. The compression method works through the nearest-neighbors method and reduces the amount of bits taken by a given image to minimalize the size of each screen buffer image. Reduction of the resolution of the screen image is also done to minimize bandwidth consumage.

```
BufferedImage img = new BufferedImage(screenWidth,
        screenHeight, BufferedImage.TYPE_3BYTE_BGR);
img.getRaster().setDataElements(0, 0, screenWidth,
        screenHeight, decodedImageData);

ByteArrayOutputStream compressedStream = compressImage(
        img, 1.0f);
ImageIO.write(img, "jpeg", compressedStream);
compressedStream.close();

float compressable = 128000.0f / compressedStream.size();
if (compressable < 1f) {
    System.out.println("Compressed by "
            + (compressable * 100) + "%.");
    compressedStream = compressImage(img, compressable);
}

sendImageToClients(compressedStream.toByteArray());
compressedStream.close();
```

Figure 8: JPEG Stream Compression for Received Image Buffer Data

This approach was taken to reduce a typical 1920x1080 laptop/Android smartphone screen buffer data with a pixel density of 32-bits ranging approximately ~1.86mb to ~1.26mb. After the reduction of image resolution from 1920x1080 to 640x480 which is suitable on a number of devices, the data reduced to a size of ~148kb which is suitable for many consumers network bandwidth usage.

8 LIMITATIONS & IMPROVEMENTS

Although the majority of the one-to-N implementation has been completed, several more work needs to be done in terms of reducing the amount of bandwidth consumed per image transmitted. The inclusion of better methods to approximately determine changed pixel regions as done in commercial screen sharing software can be incorporated to improve the project's present bandwidth consumption which spans to approximately 56kb per screenshot as of now.

Better implementations of screen frame buffer grabbing would also speed up performance by a lot and keep it up to optimal commercial performance, which will require more native-centric implementations of screen buffer grabbing straight from the operating system's set of kernel

commands themselves. An alternative approach to the XOR Gate for the determination of screen buffer image regions which need to be updated can be taken in order to improve user experience through mitigating latency.

Cross-platform proxy implementations of the screen buffer grabber need to be implemented on several other operating systems given that the system is only implemented on the Desktop and Android at the moment.

9 REFERENCES

[1] Boutell, Thomas. "PNG (Portable Network Graphics) Specification Version 1.0." (1997).

[2] Kay, Jonathan, and Joseph Pasquale. "The importance of non-data touching processing overheads in TCP/IP." *ACM SIGCOMM Computer Communication Review*. Vol. 23. No. 4. ACM, 1993.

[3] Jolitz, William Frederick, Matthew Todd Lawson, and Lynne Greer Jolitz. "TCP/IP network accelerator system and method which identifies classes of packet traffic for predictable protocols." U.S. Patent No. 6,173,333. 9 Jan. 2001.

[4] Zechner, M. "LibGDX documentation initiative." *Online]. Tillgänglig pa: http://www. badlogicgames. com/wordpress* (2012).