Michael Han
CSC 450
Dr. Mike Konemann
Spring 2015

Project Proposal

For this project I will be producing a cross-platform remote desktop application. This application will provide a video stream of the target machine's screen, allow for remote user input, and will also provide powerful macro tools and task scheduling capability. After this project class is finished I intend to make it open source. I may try to implement the target computer client as an applet but there might be Access Control / Permission Issues when trying to use the Robot remotely. With this project I will need to research the network/server side of java, of which I have little experience, as well as video encoding libraries / methods.

Deliverables

This project will consist of a Java.swing application to allow for remote access to computers running a second application or applet used for encoding and sending the current screen as well as executing remote input.

Mandatory Specifications include:

- Connecting the remote client to the host client using the TCP/IP protocol.
- Creating and sending a low-latency preview of a computer screen remotely.
- Allowing remote users to execute mouse and keyboard inputs remotely.
- Basic security features such as access passwords.

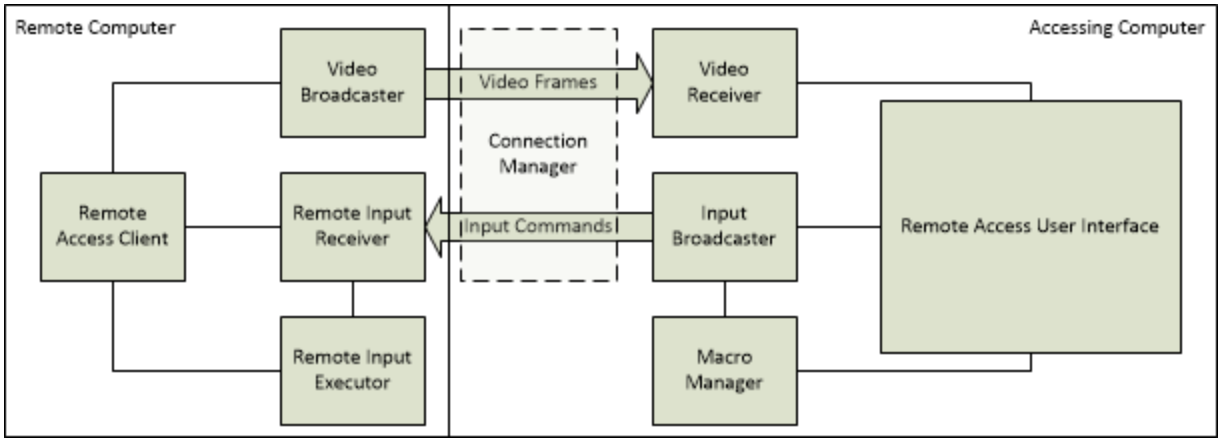
Stretch Specifications (Aka feature-creep stuff) includes:

- Allow for macro creating with UI or custom script (and possibly java API / Plugins)
- Task scheduling of macros / commands.
- Enhanced security options such as SSH communication.

JAccess -Initial Specifications and Requirements

Notes about modifications made for final release in red

Class Overview



Remote Computer Class Descriptions:

Remote Access Client:

This class will serve as the main process in the Remote Computer, it will contain, as members, all the objects needed to provide data and functionality to the Accessing Computer, and manage basic connection tasks and security. It will also provide a basic user interface, possibly in the form of a tray icon or other cross-platform equivalent (Possibly just a command-line interface).

Video Broadcaster:

This class will be responsible for broadcasting the Remote Computer's current state to the Accessing Computer. It will use AWT.Robot to capture the current state of the screen, encode the image into a ByteBuffer using JCodec (jcodec.org) and Send it to the Accessing Computer. I tried using JCodec initially but I found the decoding performance to be extremely lacking taking over a second per frame. I found other solutions such as Java's ImageIO library to have similar issues. In the end I simply converted the images to smaller data format (specifically BufferedImage.TYPE_BYTE_INDEXED) and constructed the packets manually.

Remote Input Receiver:

This class will receive input commands from the Accessing Computer, process them, and send them to the Remote Input Executor for execution.

Remote Input Executor:

This class will use AWT.Robot to execute commands received by the Remote Input Receiver. Executable commands will include mouse clicks, keyboard strokes, and possibly also macro objects.

Accessing Computer Class Descriptions:

Connection Manager:

This class will be responsible for establishing and maintaining a connection to the Remote Access Client, sending and receiving network traffic on the Accessing Computer Side. TODO: Finish this description, I'll know more about this class after I've researched the Java Network API more thoroughly. As I learned more about the java networking framework, this class seemed less and less necessary, so I eventually dropped it, although you could say that some of it's functionality was moved to the Remote Access Client.

Video Receiver:

This class will decode screen captures sent from the video broadcaster and received by the Connection Manager. After decoding, it will send them to the UI for display. Decoding will be done with the same library as the encoding, JCodec. As mentioned before, JCodec did not provide the performance required for a responsive UI. All encoding and decoding was done within the java framework.

Remote Access User Interface:

This will be the main User Interface Class, which will be composed of Swing objects and graphics. I'm sure there will be many trivial member classes but they will all pertain to user interactions. The main function of this class will be to display the screen capture of the Remote Computer and allow for remote user input to be captured by clicking on said display and using the keyboard. There will also be a menu bar or button bar with various actions provided such as initiate a connection or create a macro.

Input Broadcaster:

This class will take the captured inputs from the Remote Access User Interface and format them to be sent to the Remote Input Receiver via the Connection Manager.

Macro Manager:

This class will record and store macros created by the user. It will also use the input broadcaster to executed recorded macros. I will have to decide if these macros will be executed by sending single commands or whole macro objects to the Remote Computer. I might add a class to handle said macro objects to the Remote Access Client. As the project progressed, the "Macros" feature seemed more and more out-of-place and unrelated to the core aspects of the project, so I dropped it in favor of spending more time optimizing other aspects of the project.

Milestone Time Table

Milestone	Target Completion Date	Actual Completion Date
Network related functionality implemented.	2/20	2/21
Video Encoding Implemented	3/06	4/6
Remote Input Implemented	3/20	4/12
Overall User Interface Implemented	4/03	4/16
Stretch Goals Implemented	4/17	N/A
Final Deliverables Completed	4/30	4/29

Michael Han

Spring 2015

CSC 450 Capstone Project Deliverables

Project: JAccess

An application for accessing computer desktops remotely

Part 1: Project Background and General Information

OBJECTIVE

The main objective of this project is to allow a user to connect to a remote computer over a local area network, retrieve a live feed of that computer's screen, and allow the user to execute inputs as if they were sitting in front of it. To accomplish this, there are many factors to consider:

- Latency of the live video feed
- Bandwidth usage / limitations
- Ensuring all inputs are received
- Ensuring inputs are executed in the order requested
- Ensuring there are no concurrency issues between the many threads.

MOTIVATION

I've never been the type of programmer that enjoys working on the front-end, and up until now I've had little to no experience working with the Java Networking Framework. As such, I wanted a project that would challenge me in a way that no record keeping application could. While creating JAccess I've learned a lot about managing the flow of data between threads and computers.

USE CASE: REMOTE ACCESS

This application has but one use case, allowing the user to access a computer remotely:
Actions:

- User insures the Remote Client is running on target PC
- User runs the Remote Access program
- User enters the IP and password (if necessary)
- User has access to the remote computer!

Part 2: Logical Design

OVERVIEW

Since the implementation of the JAccess involves many processes being completed simultaneously, the logical design of the application played a crucial role in ensuring the application was robust and responsive. Seeing as the requirements for this application are straightforward but non-trivial implementation wise, I will go into detail with how I accomplished them. When in use the application runs 7 threads, 4 on the remote PC and 3 on the accessing PC (excluding threads run by the swing framework). They are as follows:

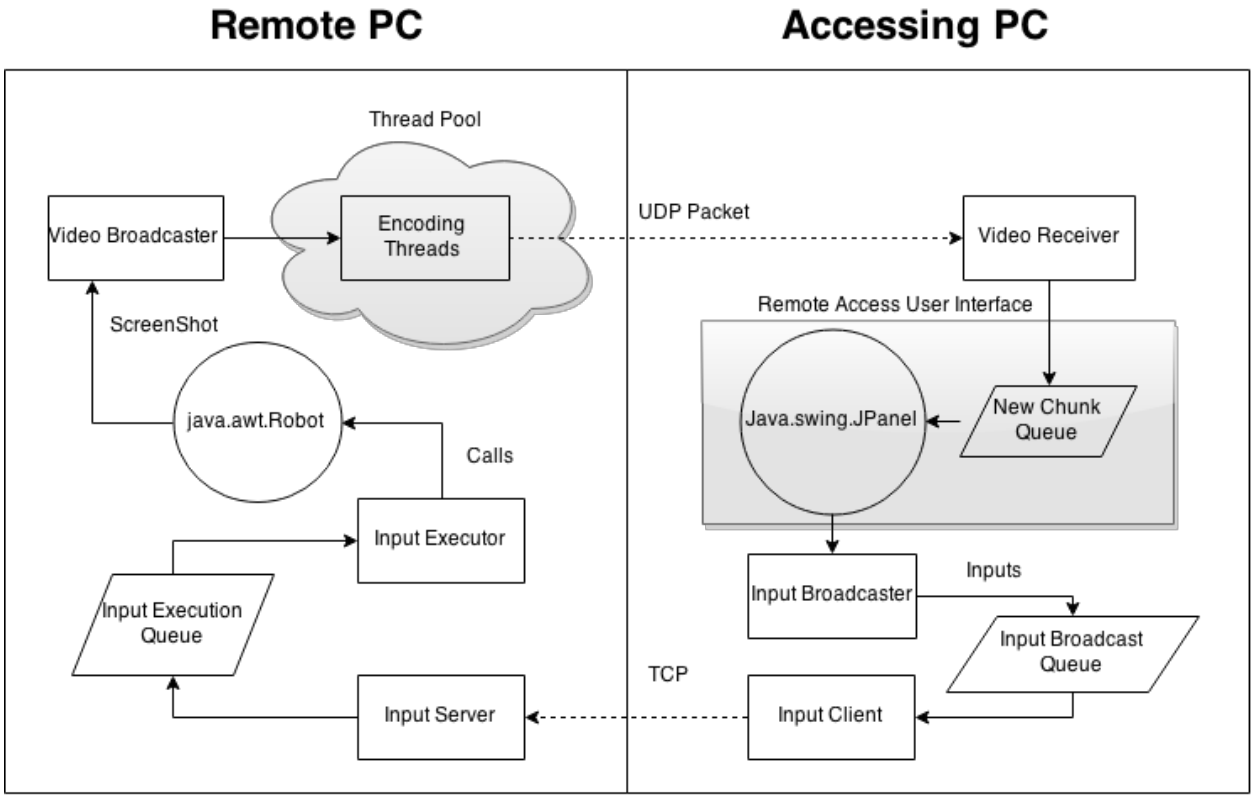
On the remote PC:

- Video Broadcaster
- Frame Sender (many instances in a thread pool)
- Input Server
- Input Executor

On the Accessing PC:

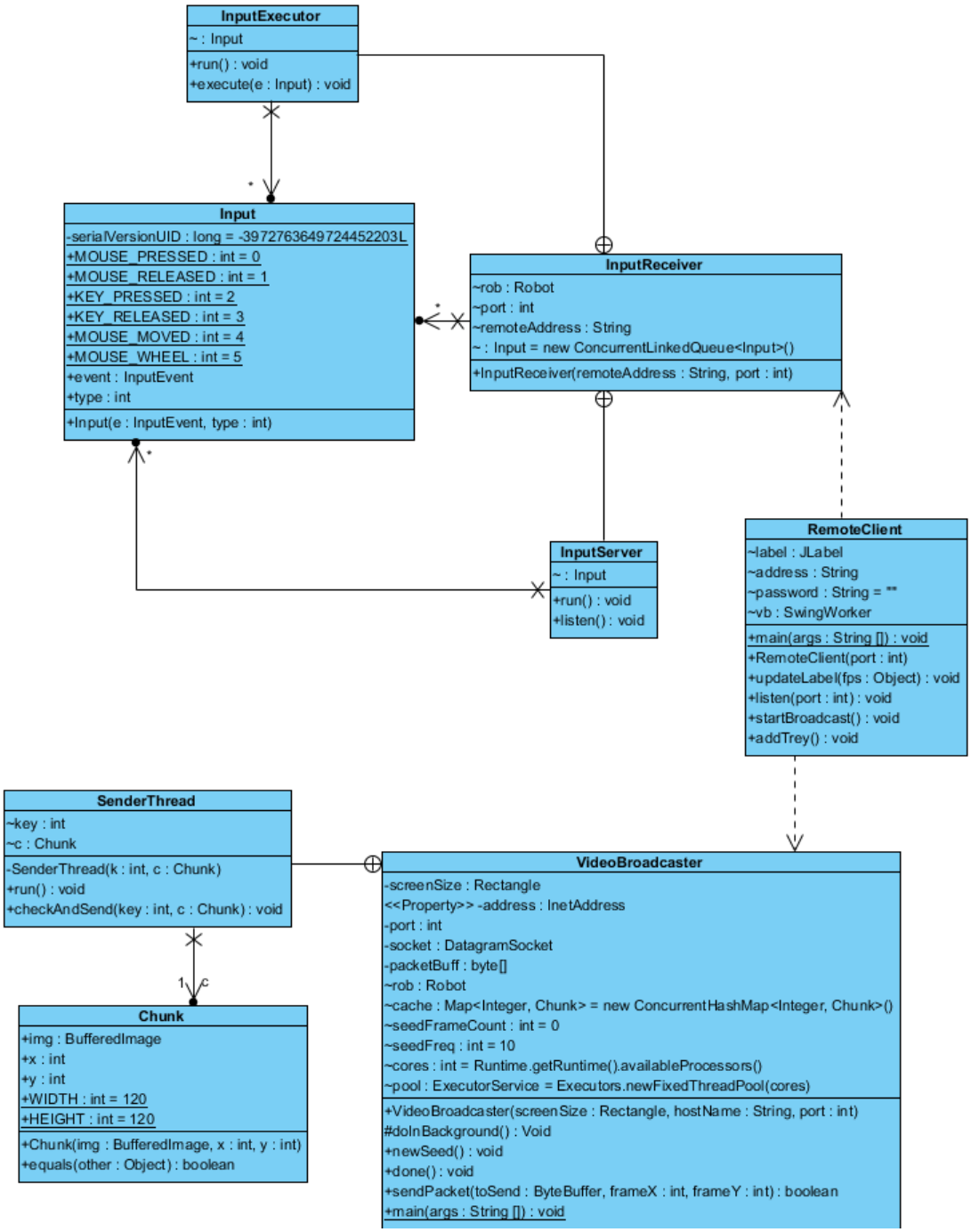
- Video Receiver
- Input Broadcaster
- Input Client

Here's a diagram showing how the threads interact with each other:



REMOTE PC - Remote Client

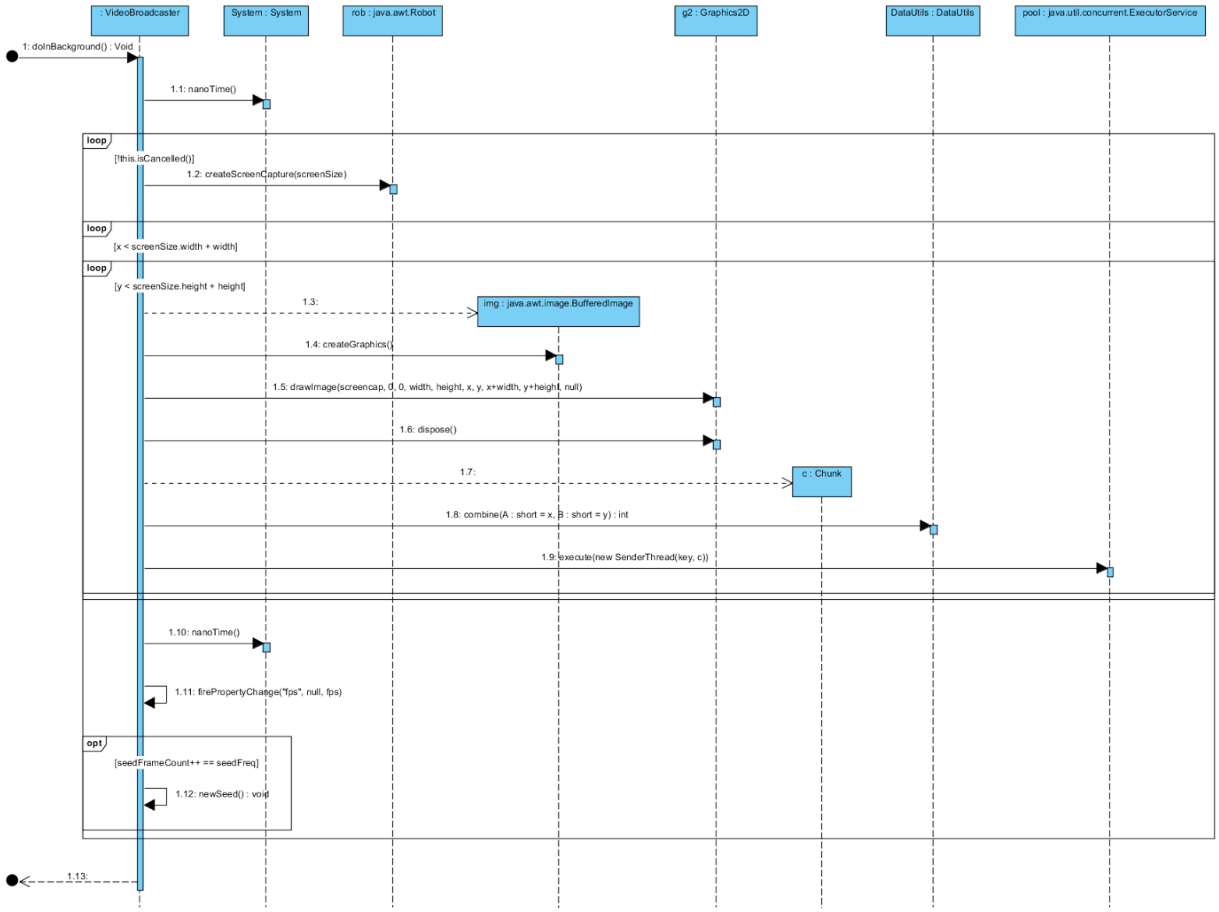
The remote PC has two main aspects, the sending of the live feed and the receiving and executing of remote input. Each of these tasks runs two threads in order to maintain a smooth flow of data. The Video Broadcaster has two thread types, the main thread, and an encoding thread pool. A class diagram of the Remote Client:



REMOTE PC - Video Broadcaster

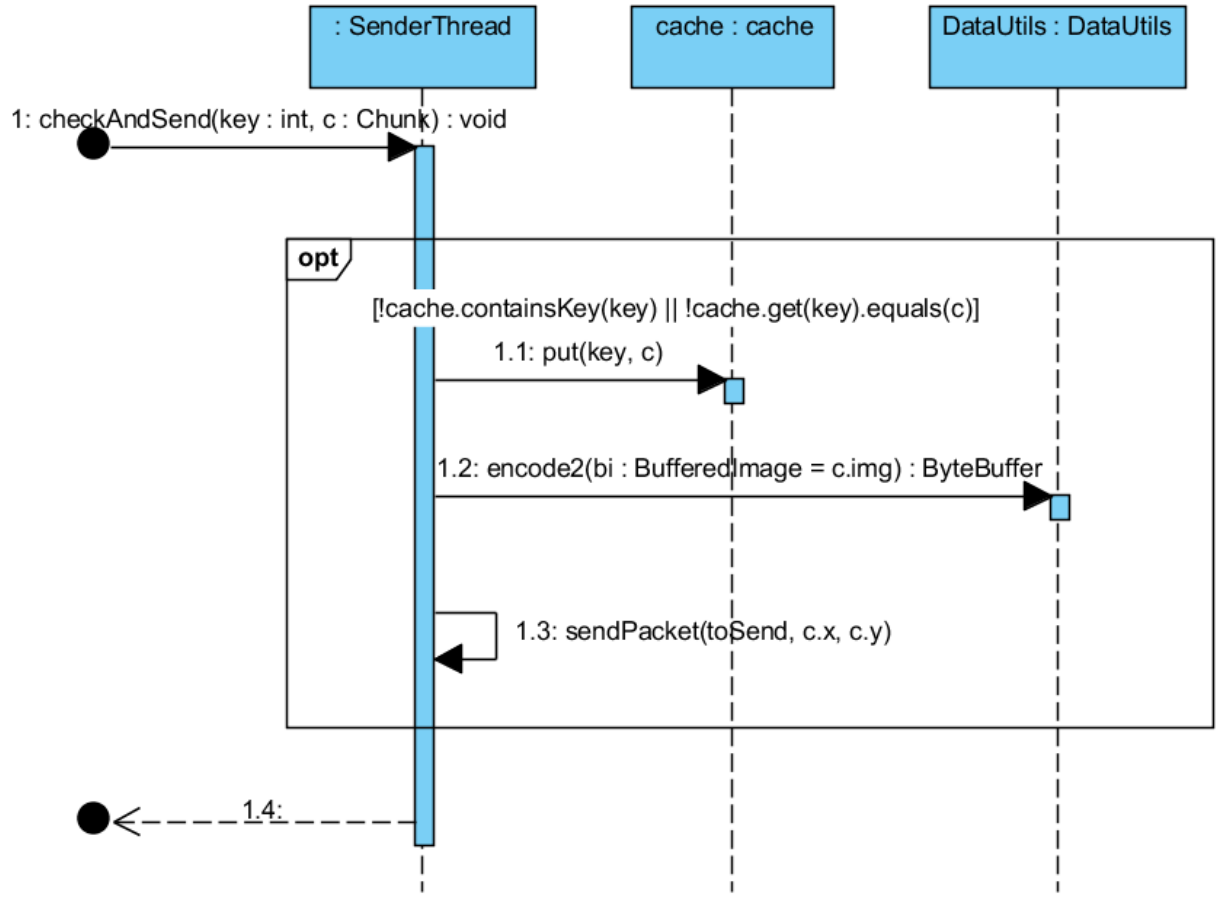
The video broadcaster is a crucial element of how JAccess works. Since getting the data to the accessing PC must be fast and efficient this class went through *many* implementations/iterations before I settled on this one.

The main video thread works like this:



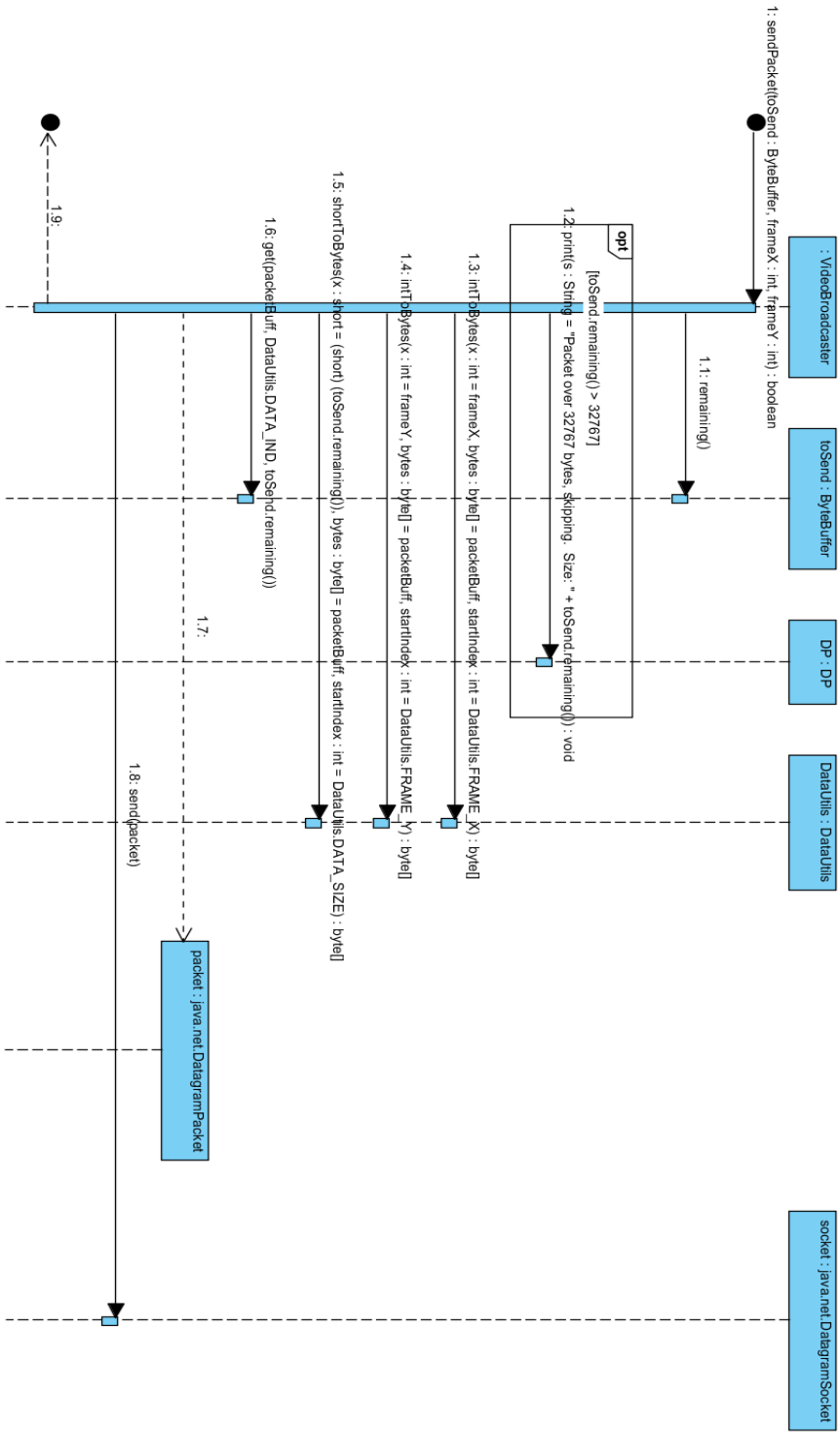
Here we see that the main Video Broadcaster thread uses awt.Robot to create a screen capture. It then splits the image up into “Chunks” which consist of a low-size formatted image, and a screen position. It uses a thread pool to spawn “Sender Threads,” explained below. Finally It does an FPS calculation and fires the result to the Remote Client.

The sender threads have two main jobs, to check the cache for non-duplicate chunks and convert them into a byte array which can be sent over the UDP socket. The cache check works as follows:



I save chunks that are sent to the accessing PC into a thread-safe hash map. Then each chunk with the same position as the cached chunk is checked to see if it has changed. Only chunks with new data are transmitted to the accessing PC. This system is not perfect however, so every so often the cache is cleared to for the broadcaster to send the whole screen over again.

The process of converting the chunk to a packet and sending it looks like this: (Please excuse the orientation, this one is fairly wide)

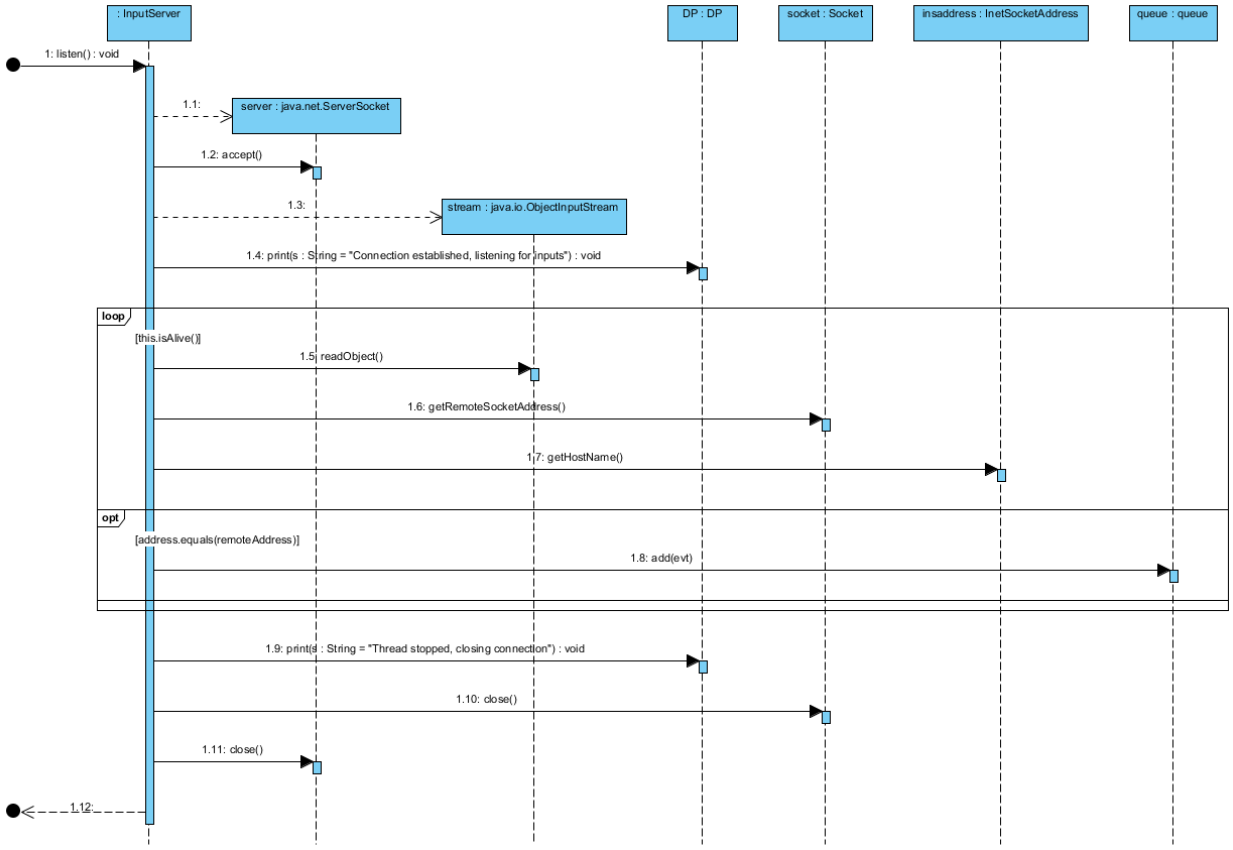


Here I use my Data Utility class to help convert the coordinates and the image data into a packet which can be sent via UDP to the accessing PC.

REMOTE PC - Input Receiver

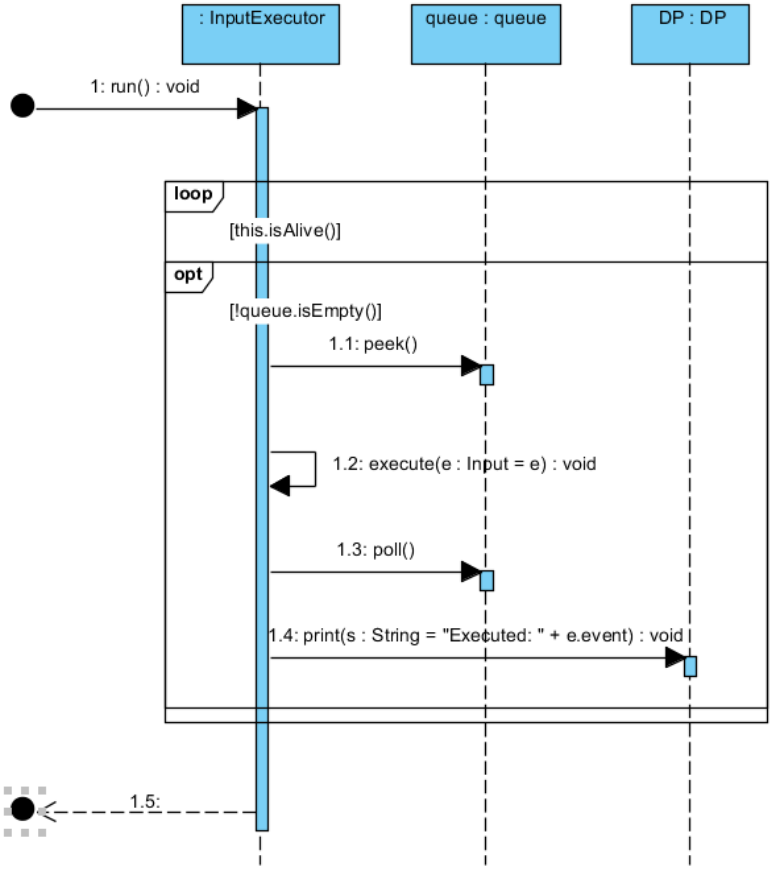
The other aspect of the remote PC is the process of capturing and executing input command sent from the accessing PC. For this, it is crucial that all inputs sent are executed, and that they are executed in the order sent by the accessing PC.

The thread which collects the data from the socket is called the Input Server:



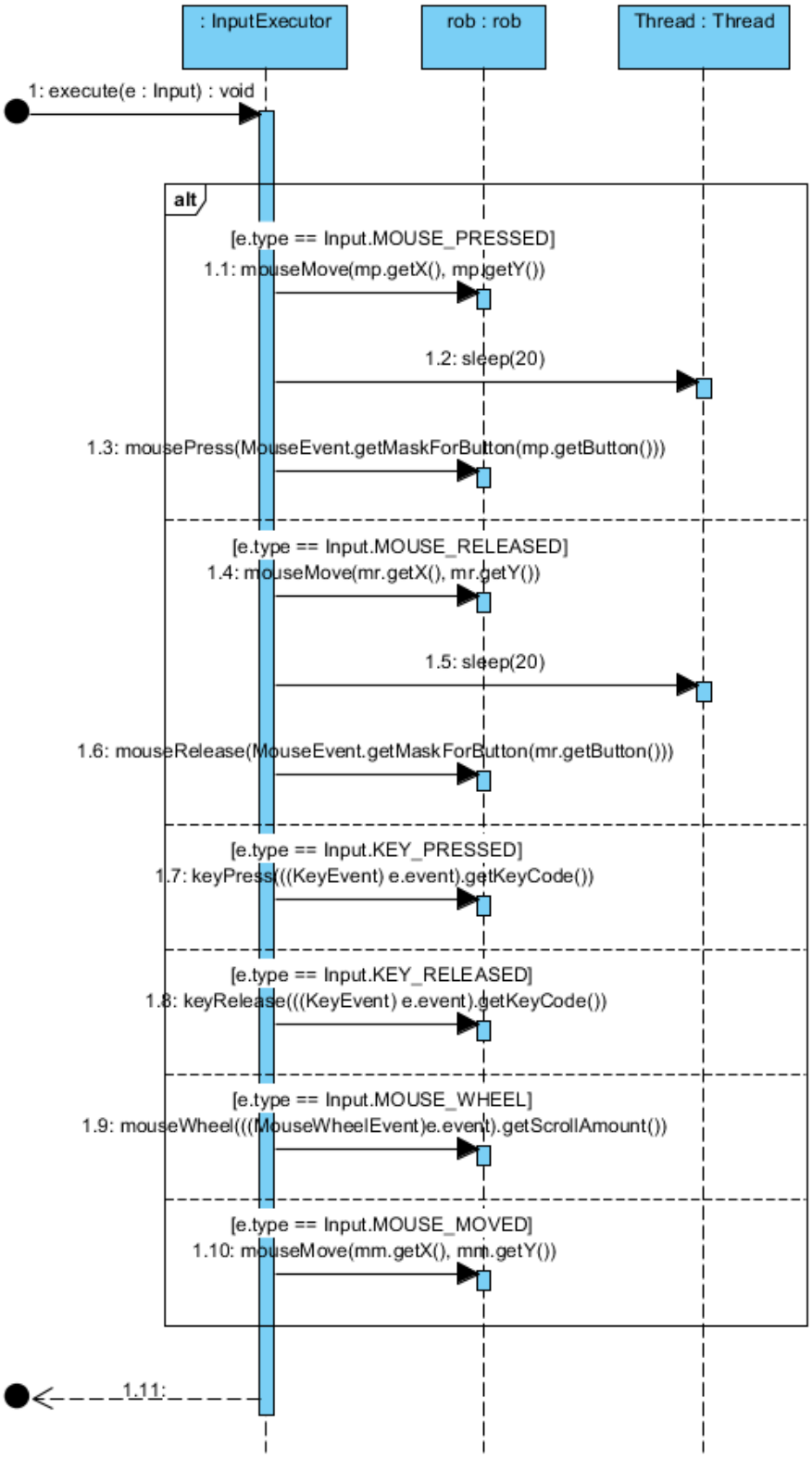
In this class the ObjectInputStream loads serialized data from a TCP socket. The address is checked to confirm it came from the trusted source and then the Input is put in a synchronized queue to be executed by the Input Executor.

The main loop for the Input Executor works like this:



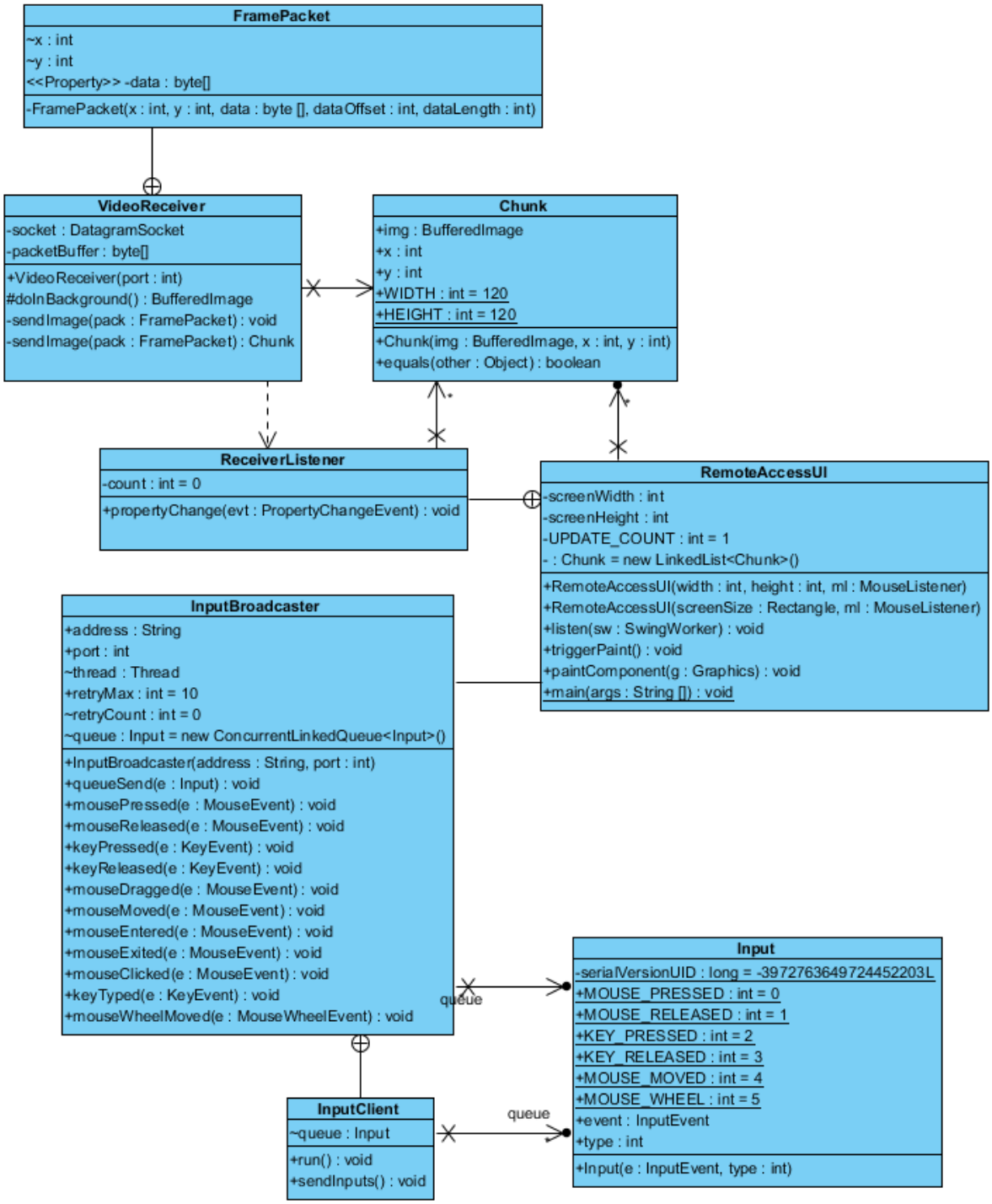
The main reason for using a queue to store the data before execution is to ensure that the socket doesn't overflow when an execution takes too long.

The execution works like this:



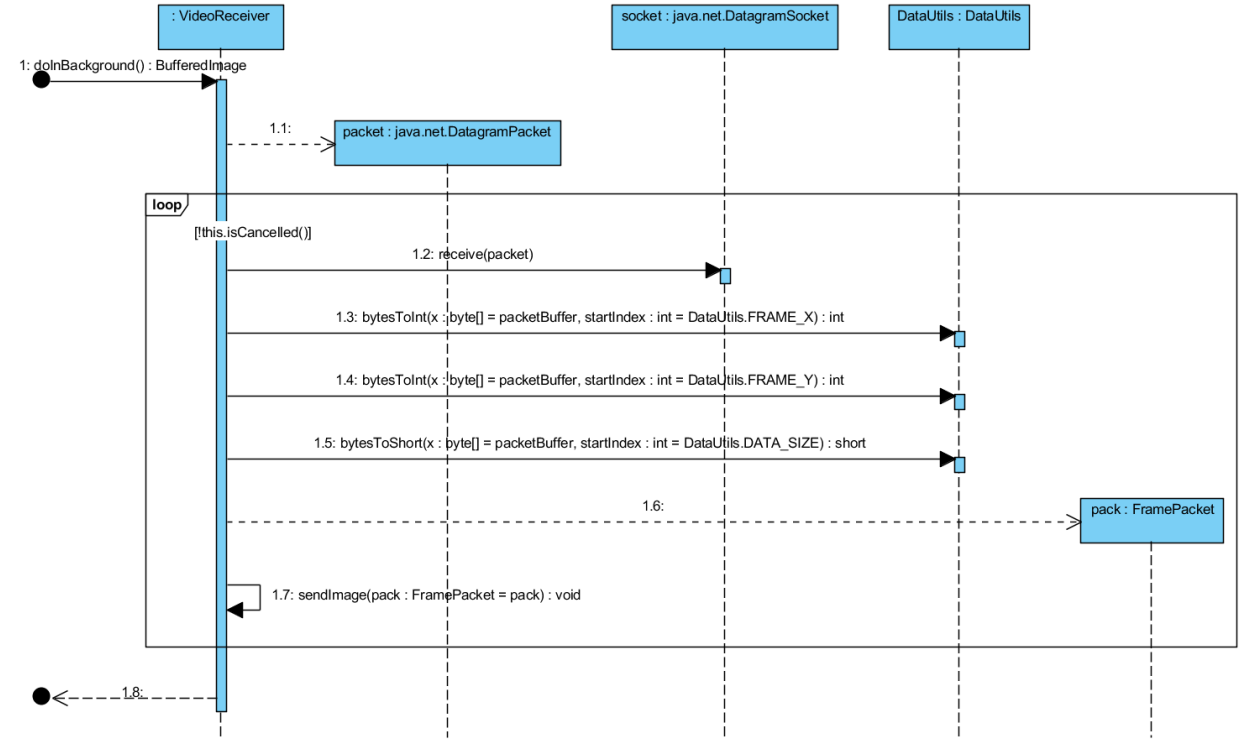
ACCESSING PC - Remote Access Process

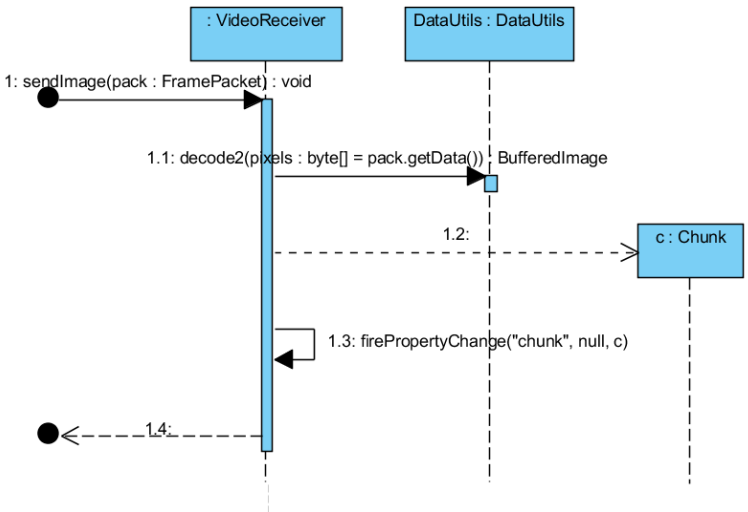
The remote access UI is at the center of the functionality for the accessing PC. It is responsible for displaying the video feed from the remote PC and also for capturing input events for remote execution. Here's a class diagram of the Accessing PC process:



ACCESSING PC - Video Receiver

The video receiver's main job is to capture UDP packets from the remote PC and reconstruct them into images for the UI to display. This takes place in two parts, but they are pretty straightforward.

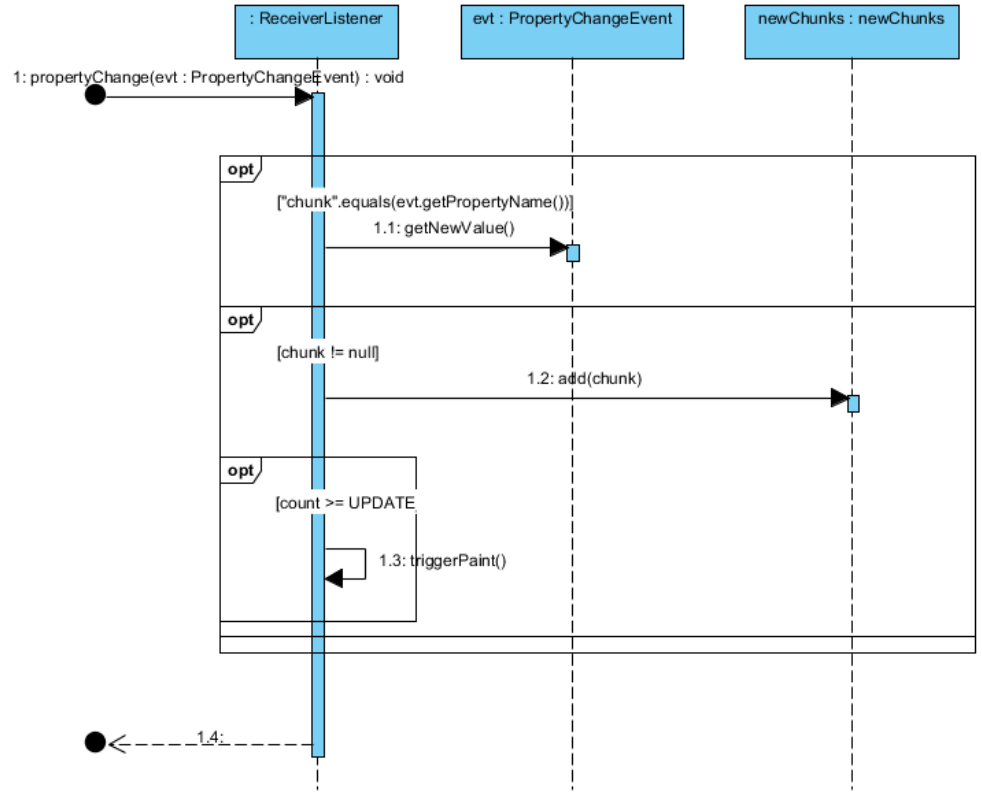




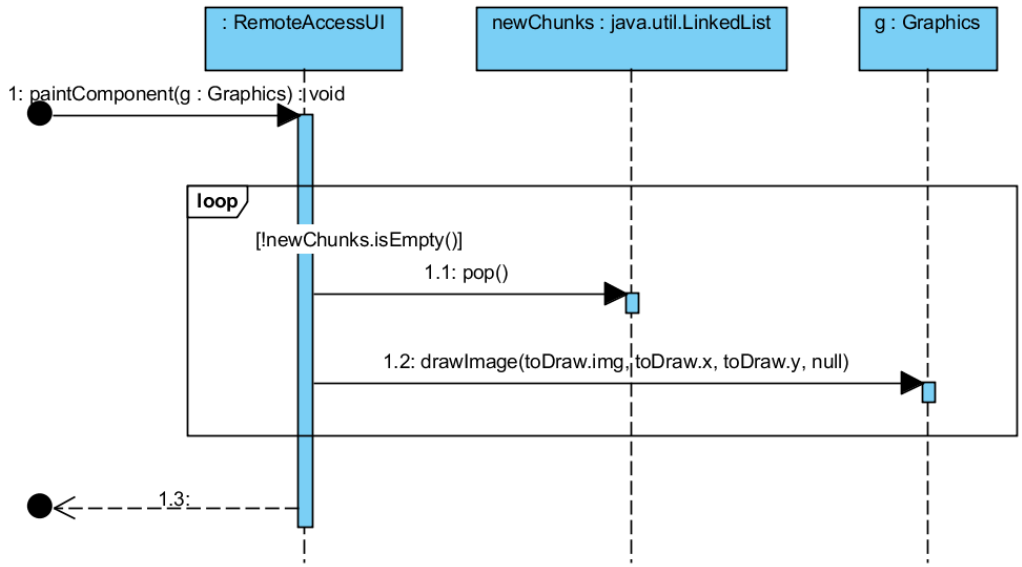
Since UDP is already lossy by nature I don't bother using a queue to ensure all packets are read as soon as they come in, the system is built to absorb data loss.

ACCESSING PC - Remote Access UI

When the Video Receiver is finished collecting a Chunk, it's "fired" to the ReceiverListener, which is an element of the User Interface:

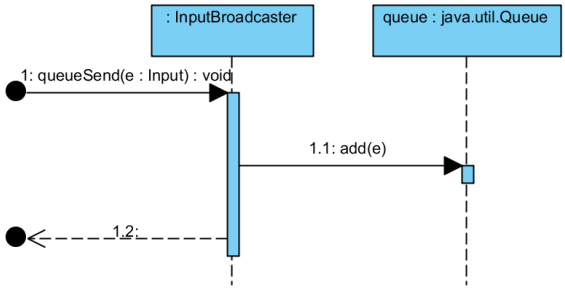


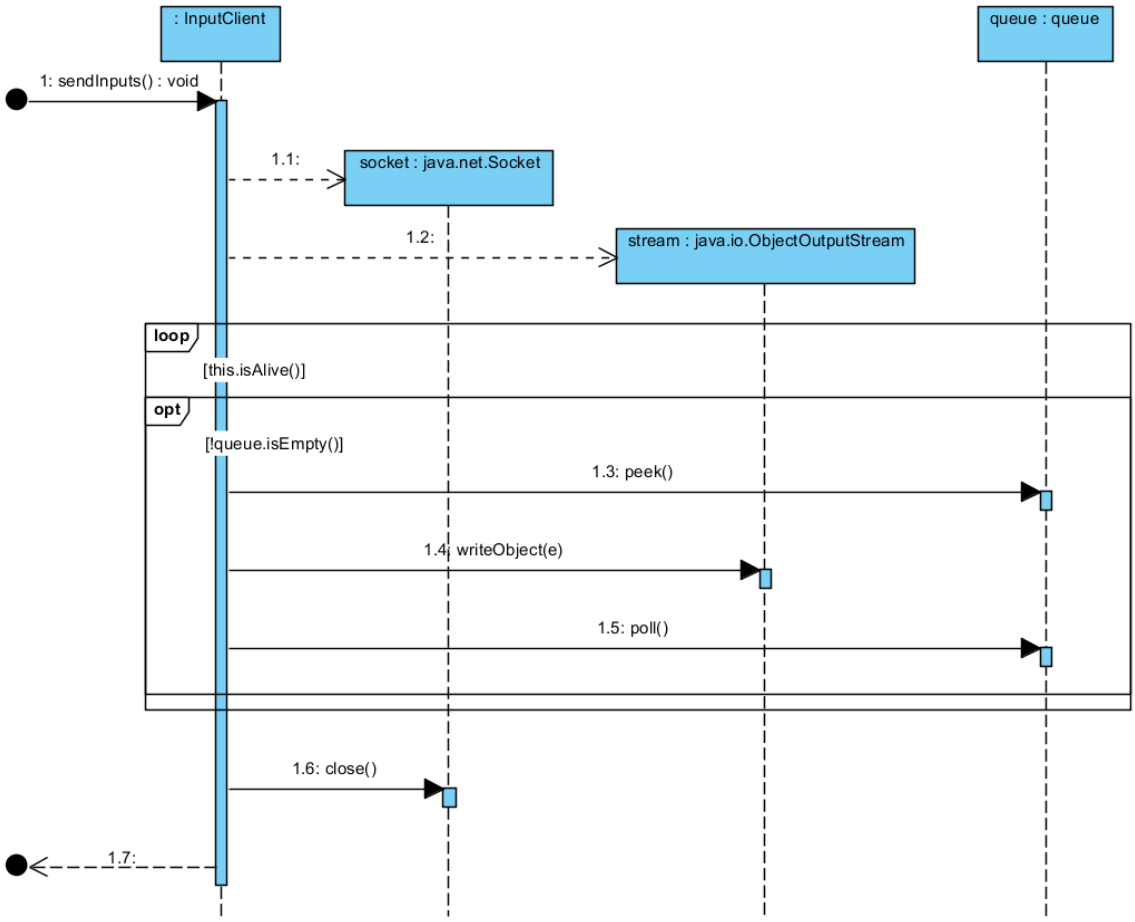
When a chunk is received by the receiver, it adds it to another queue of new chunks to be drawn on screen. It will also trigger a new painting of the chunks received when appropriate. The paint is fairly simple:



ACCESSING PC - Input Broadcaster

The final element of JAccess is the Input Broadcaster. This class implements KeyListener, MouseListener, MouseMotionListener, and MouseWheelListener. Using these interfaces it hooks into the Remote Access UI and capture the inputs it needs to send to the remote PC. This takes place in two parts:





The Broadcaster adds all inputs to a queue immediately upon receiving them, this avoids any delay based on the network and allows the inputs to keep being captured. The Input Client runs a separate thread that sends the inputs over the network to the remote PC

Part 3: Physical Design

There are three main data structures used in the project, and two network protocols.

CHUNK

```
public class Chunk
{
	public BufferedImage img;
	public final int x;
	public final int y;
	public static final int WIDTH = 120;
	public static final int HEIGHT = 120;
	public static final int IMAGE_FORMAT = BufferedImage.TYPE_BYTE_INDEXED;

	public Chunk(BufferedImage img, int x, int y) {
		super();
		this.img = img;
		this.x = x;
		this.y = y;
	}

	public boolean equals(Object other)
	{
		Chunk o = (Chunk) other;
		byte[] tb = DataUtils.getBytes(this.img);
		byte[] ob = DataUtils.getBytes(o.img);

		return Arrays.equals(tb, ob);
	}
}
```

Chunk is used by the video broadcaster and receiver to couple image data with it's position on the remote PC. It also contains information on how big the chunks should be and what data format will be used to transfer.

UDP PACKET (no class but it is defined in DataUtils)

```
public class DataUtils {
	public static final int FRAME_X = 0;
	public static final int FRAME_Y = 4;
	public static final int DATA_SIZE = 8;
	public static final int DATA_IND = 10;
	public static final int VIDEO_PORT = 9527;
	public static final int INPUT_PORT = 9528;
	public static final int HANDSHAKE_PORT = 9529;
	...
```

All UDP traffic is in the form of byte arrays using java's UDP framework, so I needed to define a data format that would put all the relevant information into a single packet.

FRAME_X: This is the index of the x coordinate of the chunk (4 bytes)
FRAME_Y: This is the index of the y coordinate of the chunk (4 bytes)
DATA_SIZE: This is the index that says how big the image data is (2 bytes)
DATA_IND: This is the index for where the image data starts. (byte value in DATA_SIZE)

It also holds the port numbers for each type of communication.

INPUT

```
public class Input implements Serializable {
	private static final long serialVersionUID = -3972763649724452203L;
	public static final int MOUSE_PRESSED = 0;
	public static final int MOUSE_RELEASED = 1;
	public static final int KEY_PRESSED = 2;
	public static final int KEY_RELEASED = 3;
	public static final int MOUSE_MOVED = 4;
	public static final int MOUSE_WHEEL = 5;
	public final InputEvent event;
	public final int type;

	public Input(InputEvent e, int type) {
		this.event = e;
		this.type = type;
	}
}
```

Input consists of two elements, an int enumeration that indicates what type of event caused this input, and the actual InputEvent Object created by swing (serializable). With these two

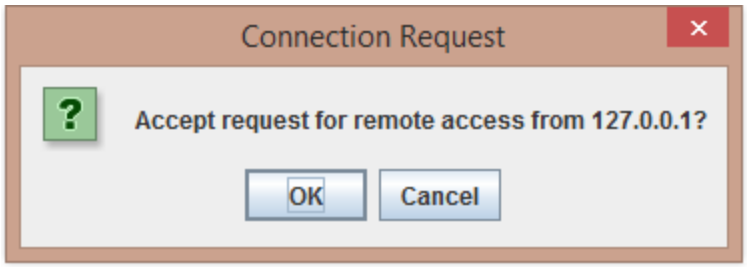
elements the input executor has all the information it needs to execute the input on the remote PC.

Part 4: User Manual

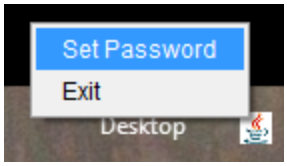
On remote computer:

To allow remote access one must ensure the client is running by running RemoteClient.jar. To verify that this is the case, simple look in the system tray for the Java Icon: 

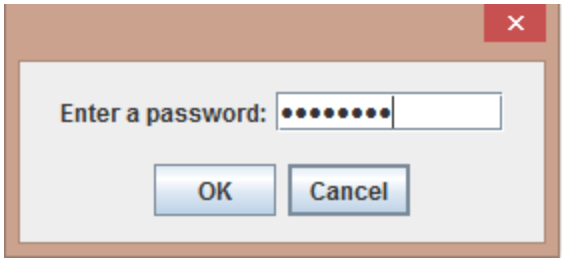
By default, someone who wants remote access will need someone at the remote computer to confirm the access:



To bypass this requirement, the remote access client must set a password by right clicking:



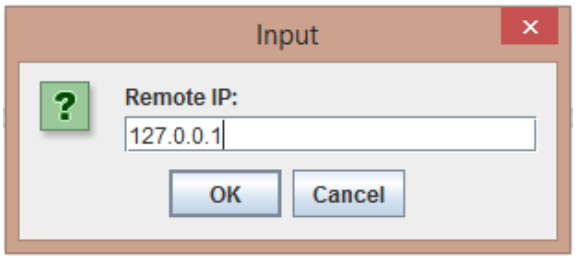
Then entering a new password:



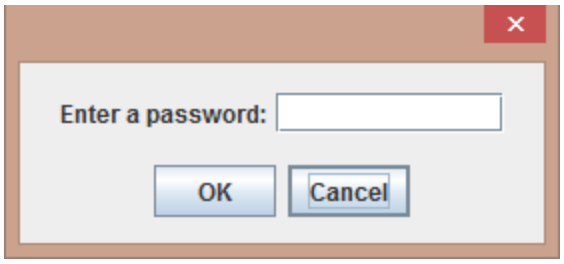
After setting a password, the “Connection Requests” will not be shown, and the broadcast will start immediately assuming the accessing user used the correct password.

On the accessing computer:
To start a remote connection to a computer running the remote client you run the RemoteAccess.jar

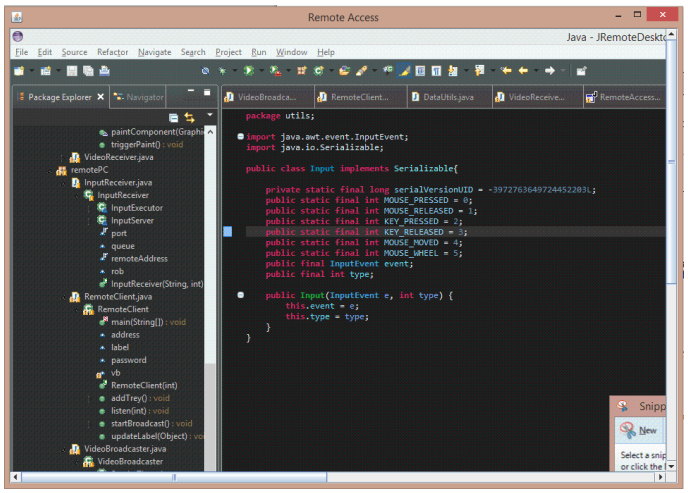
First you will be prompted for the IP address of the remote computer:



Then you will be asked for a password (may be optional):

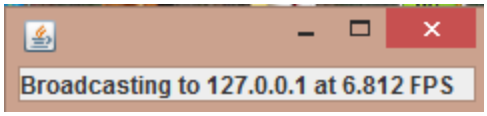


If the IP is correct and the remote computer grants access you will see a screen like this:



While the main “Remote Access” window is up, it will act as a desktop for the other computer, key inputs and mouse inputs done in this window will be sent directly to the remote computer.

And on the remote PC a window will popup showing something like this:



Closing either of these windows will end the remote session.

Part 5: Journal

I didn't really keep a journal of this project unfortunately, but I'd estimate I spent ~200 hours total on this project, 60% of which was spent experimenting with different sorts of optimizations, and probably another 20% finding bugs and fixing them.

In lieu of a journal I give you my git log, please note that the commit dates are NOT when the code was actually finished for the most part, just when I decided it was worth committing:

```
$ git log --stat

commit c407c79d70703f305192b9cb816dff966f6b7673
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Wed Apr 29 19:31:33 2015 -0500

    Video Broadcaster synchronization fix

 JRemoteDesktop/src/remotePC/VideoBroadcaster.java | 7 +++++--
 1 file changed, 5 insertions(+), 2 deletions(-)

commit 7407b44266c6d3a9278d6f98d6dcb75ac6b35d6a
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Wed Apr 29 12:04:00 2015 -0500

    Fixed weird input serialization bug

 JRemoteDesktop/src/utils/Input.java | 12 +++---------
 1 file changed, 3 insertions(+), 9 deletions(-)

commit ada444040beb2239f5b2e2cd0cb33fb65d0debbf
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Wed Apr 29 11:47:22 2015 -0500

    Fixed InputReceiver address check.

 JRemoteDesktop/src/remotePC/InputReceiver.java |  8 ++++--
 JRemoteDesktop/src/remotePC/RemoteClient.java  | 38 ++++++++++----------------
 JRemoteDesktop/src/utils/Input.java            |  2 --
 3 files changed, 21 insertions(+), 27 deletions(-)

commit 56b6246a479555bfc5518564534537c75328bd48
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Wed Apr 29 10:46:01 2015 -0500
```

```
    Final Commit before presentation, hopefully everything works.

 JRemoteDesktop/icons/Java16.gif                  | Bin 0 -> 554 bytes
 JRemoteDesktop/icons/java32.gif                  | Bin 0 -> 1154 bytes
 .../src/accessingPC/InputBroadcaster.java        |  23 +-
 JRemoteDesktop/src/accessingPC/RemoteAccessUI.java |  48 ++++-
 JRemoteDesktop/src/accessingPC/VideoReceiver.java  |   2 -
 JRemoteDesktop/src/remotePC/InputReceiver.java     |  12 +-
 JRemoteDesktop/src/remotePC/RemoteClient.java      | 232 ++++++++++++++++++++
 JRemoteDesktop/src/remotePC/VideoBroadcaster.java  |  89 +++++++-
 JRemoteDesktop/src/tests/DP.java                 |  36 ++-
 JRemoteDesktop/src/tests/JCodecTest.java         |  40 ----
 JRemoteDesktop/src/utils/Chunk.java              |  10 +
 JRemoteDesktop/src/utils/DataUtils.java          |  17 +-
 JRemoteDesktop/src/utils/Input.java              |  10 +-
 13 files changed, 452 insertions(+), 67 deletions(-)

commit f387273435028fcc5685b2475a162393537d3ac0
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Sun Apr 12 23:00:46 2015 -0500

    Added command line for addresses

 JRemoteDesktop/src/accessingPC/RemoteAccessUI.java | 5 ++++-
 JRemoteDesktop/src/remotePC/VideoBroadcaster.java  | 6 ++----
 2 files changed, 6 insertions(+), 5 deletions(-)

commit e4f20ad5abd6c3829bb380c690603a13df13b513
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Sun Apr 12 22:39:08 2015 -0500

    Remote inputs added

 JRemoteDesktop/bin/.gitignore                    |   1 +
 .../src/accessingPC/InputBroadcaster.java        | 128 ++++++++++++++++++++
 JRemoteDesktop/src/accessingPC/RemoteAccessUI.java |  19 ++-
 JRemoteDesktop/src/remotePC/InputReceiver.java     | 128 ++++++++++++++++++++
 JRemoteDesktop/src/remotePC/VideoBroadcaster.java  |  12 +-
 JRemoteDesktop/src/tests/DP.java                 |   5 +
 JRemoteDesktop/src/utils/DataUtils.java          |   1 +
 JRemoteDesktop/src/utils/Input.java              |  23 ++++
 8 files changed, 306 insertions(+), 11 deletions(-)
```

```
commit b4526e530f08789dbb3d07430660f99304be1cff
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Sun Apr 12 19:54:15 2015 -0500

    Updated video stream for better quality

 JRemoteDesktop/src/accessingPC/RemoteAccessUI.java |  2 +-
 JRemoteDesktop/src/accessingPC/VideoReceiver.java  |  2 ++
 JRemoteDesktop/src/remotePC/VideoBroadcaster.java  | 24 +++++++----------
 JRemoteDesktop/src/utils/Chunk.java                |  4 ++--
 JRemoteDesktop/src/utils/DataUtils.java            |  2 --
 5 files changed, 12 insertions(+), 22 deletions(-)

commit c1d5b463fde2a15ca34b2e2be320d45097a1d09b
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Sun Apr 12 19:20:14 2015 -0500

    WORKING Video stream :D

 JRemoteDesktop/bin/.gitignore                      |   3 +-
 JRemoteDesktop/src/accessingPC/RemoteAccessUI.java | 108 ++++++++++++
 JRemoteDesktop/src/accessingPC/VideoReceiver.java  | 214 +++++++++++++++++++++
 JRemoteDesktop/src/remotePC/VideoBroadcaster.java  | 132 +++++++++++++++
 JRemoteDesktop/src/utils/Chunk.java                |  19 ++
 JRemoteDesktop/src/utils/DataUtils.java            |  62 +++++++
 6 files changed, 264 insertions(+), 274 deletions(-)

commit 3efb6dfa6697cf038a14e9af433860c97568541e
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Mon Apr 6 18:15:57 2015 -0500

    last attempt at encoding, doesn't work

 JRemoteDesktop/src/remotePC/VideoBroadcaster.java | 45 +++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 14 insertions(+), 31 deletions(-)

commit 51ea4853f115f7a2f9b01dd643490cf464587252
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Sun Feb 22 00:35:38 2015 -0600

    Hit a dead end with raw data, gunna try NAL units
```

```
 JRemoteDesktop/src/accessingPC/VideoReceiver.java | 189 ++++++++++++++++++++-
 JRemoteDesktop/src/remotePC/VideoBroadcaster.java |  31 ++--
 JRemoteDesktop/src/utils/DataUtils.java           |   7 +-
 3 files changed, 202 insertions(+), 25 deletions(-)

commit 49eccab0a687fa836944a22982e0bbb53bfe91b4
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Sat Feb 21 13:35:49 2015 -0600

    Basic UDP client/server setup

 .gitignore                                 | Bin 189 -> 281 bytes
 JRemoteDesktop/bin/.gitignore              |   4 +
 JRemoteDesktop/src/accessingPC/VideoReceiver.java |  67 +++++++
 JRemoteDesktop/src/remotePC/VideoBroadcaster.java | 207 ++++++++++++++++++++++
 JRemoteDesktop/src/tests/DP.java           |  11 ++
 JRemoteDesktop/src/tests/JCodecTest.java   |  40 +++++
 JRemoteDesktop/src/utils/DataUtils.java    |  44 +++++
 7 files changed, 373 insertions(+)

commit c57346bc00c2f2279c635072e425800cca68f11d
Author: Michael Han <mhan.sensei@gmail.com>
Date:   Thu Jan 29 18:02:31 2015 -0600

    Initial commit

 .gitignore | 12 ++++++++++
 1 file changed, 12 insertions(+)
```

Part 6: Self Evaluation

PROJECT APPROPRIATENESS - 5

While this project might not seem very complicated from a requirements and user perspective, it incorporated lots of exploration into technical areas not covered in my education here at Carroll. Java's networking framework and network programming in general were very new to me, and required quite a bit of research to get working, and then a *lot* of experimentation to get working well. Additionally, the sheer number of different threads and objects that needed to work together simultaneously made for a very engaging challenge. Optimizing one part would often show bottlenecks in other areas. I tried many strategies, simple and complex, that either had no impact on the results, degraded performance, or even caused major unforeseen issues in other components. I would estimate that for each line of code in the final result, there were *at least* 10 lines of code that didn't make the final cut. These weren't caused by design failures, the design actually remained fairly rigid. Overall, this project supplied an impressive technical challenge, not one I overcame lightly.

PROJECT PLANNING - 3

My initial design ended up being exactly what I needed, and the requirements were well laid out for how simple they were. However, I will concede that I failed to keep a regular journal. My github repository only has 12 commits, and there was a lot of changes and experimentations not documented there.

SOFTWARE ANALYSIS AND DESIGN - 5

Not much to say here, the design specified in the original specifications was carried through to the final result, and my analysis of the resulting implementation was the most thorough part of this document. With this project I have proven to myself that even without understanding large aspects of the implementation, I can come up with a good design that, when implemented, produces a working result.

CODE QUALITY - 4

In following my design, I ensured that the classes and independent modules were well defined in their responsibilities and, of course, followed the standard naming conventions and formatting of Java source code. I will admit there are places where things aren't written quite as eloquently as they could have been, but when you've changed a block of code 10 times over in the name of optimization this is hard to avoid.

SOFTWARE USABILITY - 5

Due to the nature of this application, I would redefine this evaluation category to include responsiveness of the end product. In this sense, I am very confident in saying there's not much I could do to improve the end result. Trust me, I've tried to. The result, is a product that is perfectly usable for everyday administrative tasks. I wouldn't play any videogames over it but the few remotely-streamed videogame programs out there are developed by large teams at Valve or NVidia (and definitely NOT written in java). Additionally, I took great pains to ensure the experience was seamless, even in the event of networking issues / errors.

PROJECT VERIFICATION - 3

In the end, the application delivered what I promised in the project proposal, excluding the "Macro manager" which I decided early on didn't fit within the scope of a remote access application. There are, however, definitely some things I would do to improve the application. I haven't spent much effort on security features, and would definitely work some sort of encryption of the inputs being sent over the network into a production version of the application.

EFFORT - 4

I'd estimate I spent ~200 hours over the semester on this project. Many hours researching and experimenting with the java network framework. Many hours experimenting with various video-streaming methods, (h264, JPEG/MP4, JCodec, Java ImageIO, raw data) and trying to get the most performance out of them. And many hours on ensuring that the many moving parts of the application played together nicely without errors or bottlenecks.

BINDER - 4

I provided all the documents required. With the exception of the "Journal" (sorry), it is all very well organized and easy to read.

PRESENTATION - 4

This is really just a guess, I haven't done it yet… I did attend all scheduled meetings with Dr. Konemann, so there's that.