Technical Document

Craig Duff
10325807

0. Table of Contents

1. Introduction

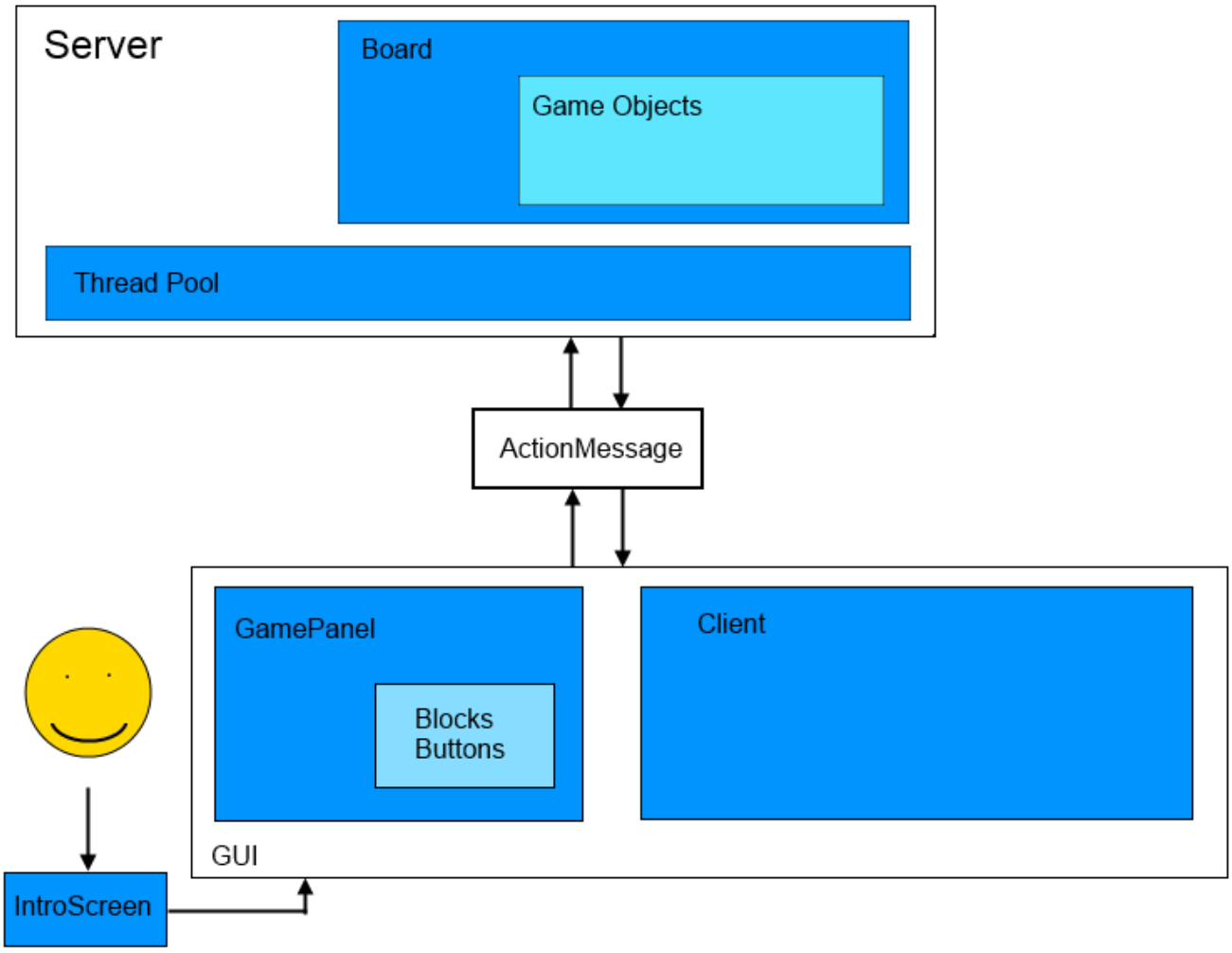
The goal of the system described in this document was to create a multi-player network-based computer game, inspired by the popular board game *"Pandemic"*. The system uses socket-programming and a server/client model to allow users to communicate over the network. Messages from the clients are all sent to the server, which processes the messages and re-broadcasts them to all the clients.

The system's GUI simulates the game board and all of it's components including decks of cards and player pawns. All of the game logic and rules enforcement is implemented server-side
The client provides each player with a GUI interface.

Epidemic is a board game which simulates diseases spreading across the world, the goal of the players is to prevent the diseases from taking over the world. After every turn in the game, cards are drawn from a deck named the *infection deck* to determine which cities shall be infected, these cities then receive a variable amount of *cubes*. If a city accumulates more than 4 cubes, an *outbreak* occurs and the infection spreads to nearby cities.
If 8 outbreaks occur, the players lose the game.

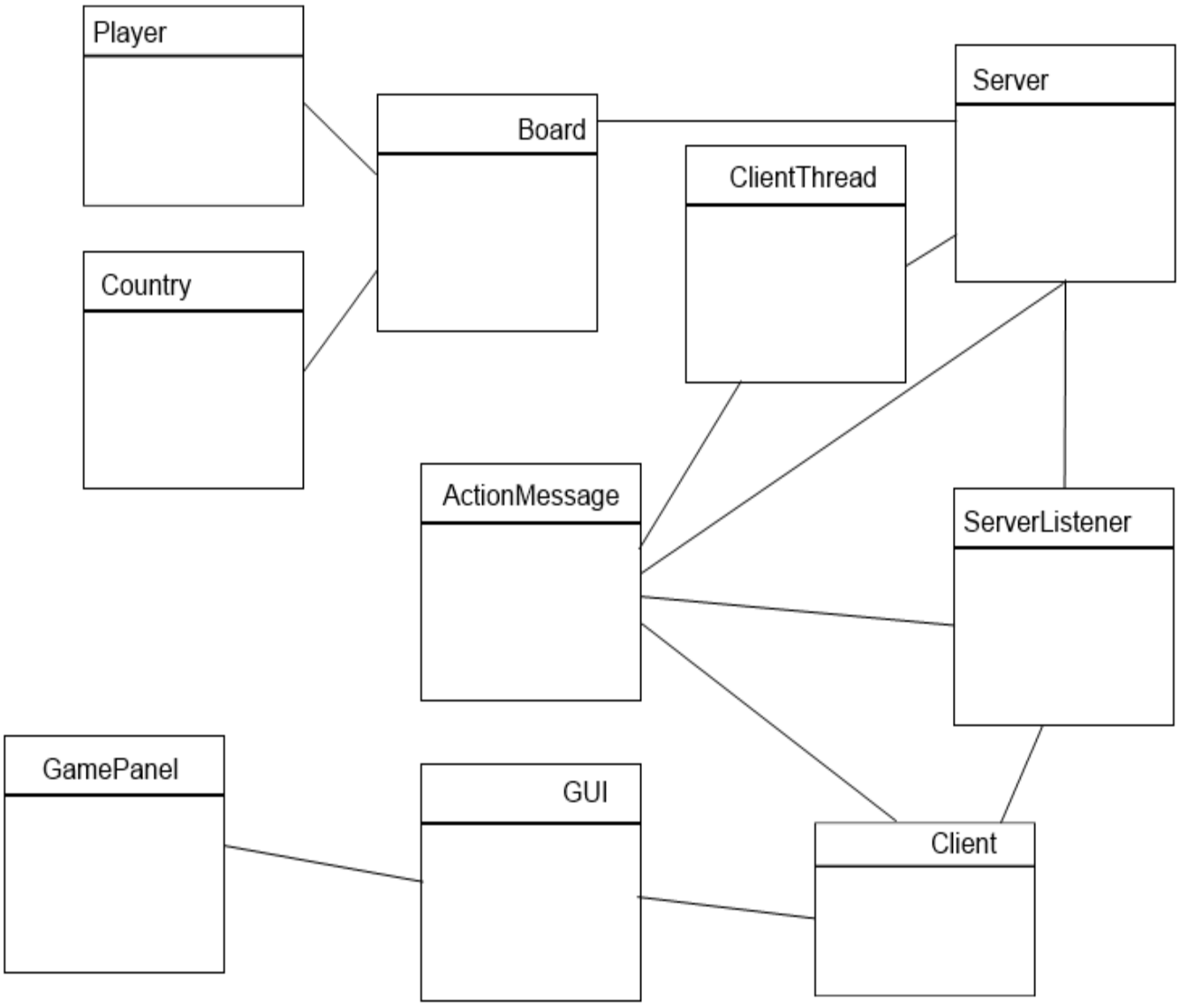
A player may move around the board and *heal* countries by removing cubes. At the end of each players turn they draw two card and add them to their hand, if a player holds 4 card of the same colour, they may discard them to *cure* that colour. If the players collectively *cure* three diseases, they win the game.

2. System Architecture



The above diagram shows the infrastructure of the system. The Server and Client packages are kept separate from each other and only communicate by sending and receiving ActionMessages.
The user accesses the Client system by invoking an instance of IntroScreen.java. This class creates an instance of GUI and Client when the user clicks the *start* button.
When the GUI receives the signal, it will create the GamePanel class which contains the Game Object classes.

3. High-Level Design



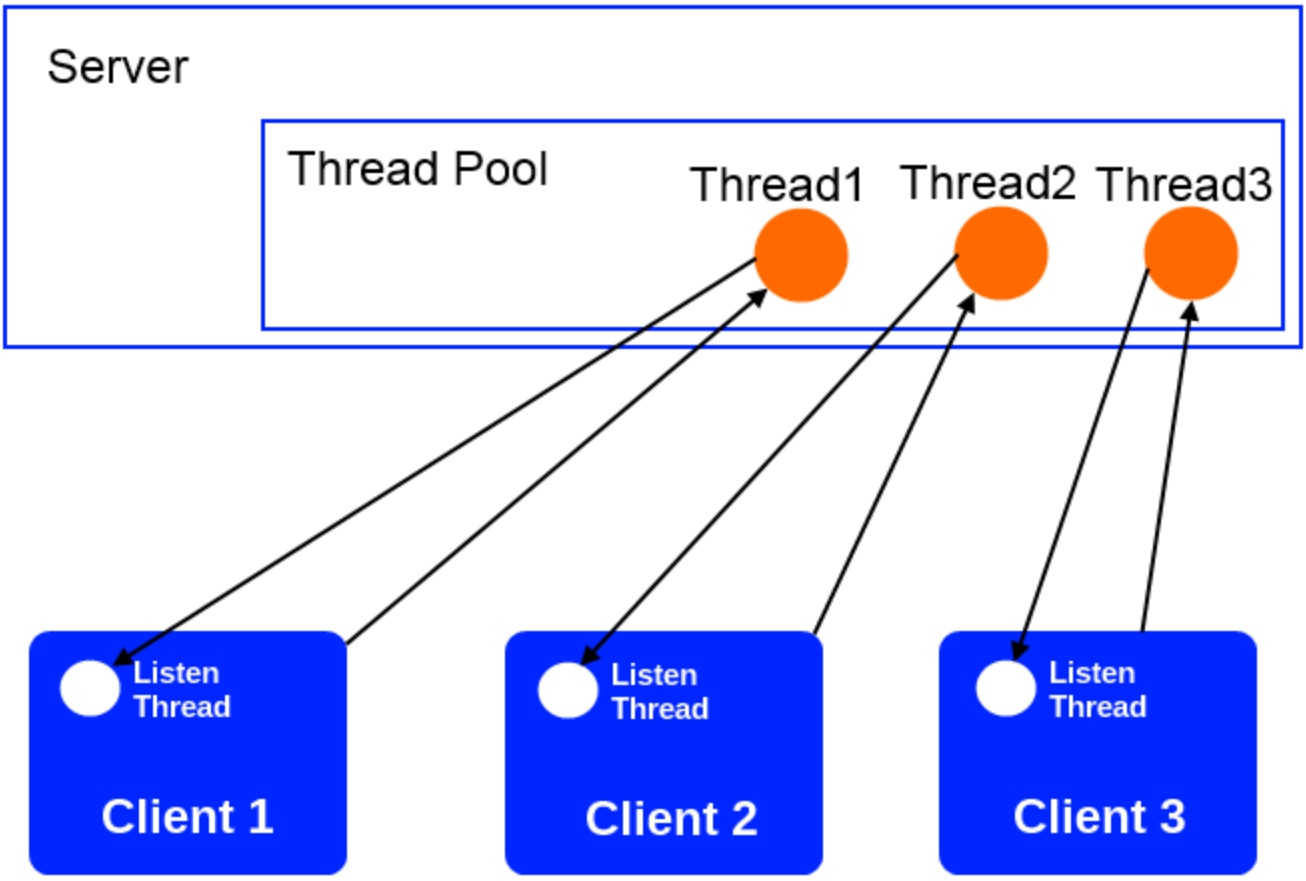
The above diagram explains the interaction of all the major class in the system.
The GUI creates the Client and GamePanel and acts as a middle-man between the two classes. The Client creates the Server Listener and writes ActionMessages to the socket. The ServerListener Receives ActionMessages from the Server class and parses them into actions. The ServerListener may tell the Client to invoke methods.

The Server creates the ClientThreads and Board class, the Server also creates and writes ActionMessages to the socket.
The ClientThread works similar to the ServerListener, however there may be multiple ClientThreads while each Client will only have one ServerListener.
The Board class contains the Player and Country class, this creates Countries based on the map.txt input file and creates Players based on the amount of clients.

4. Problems and Resolution

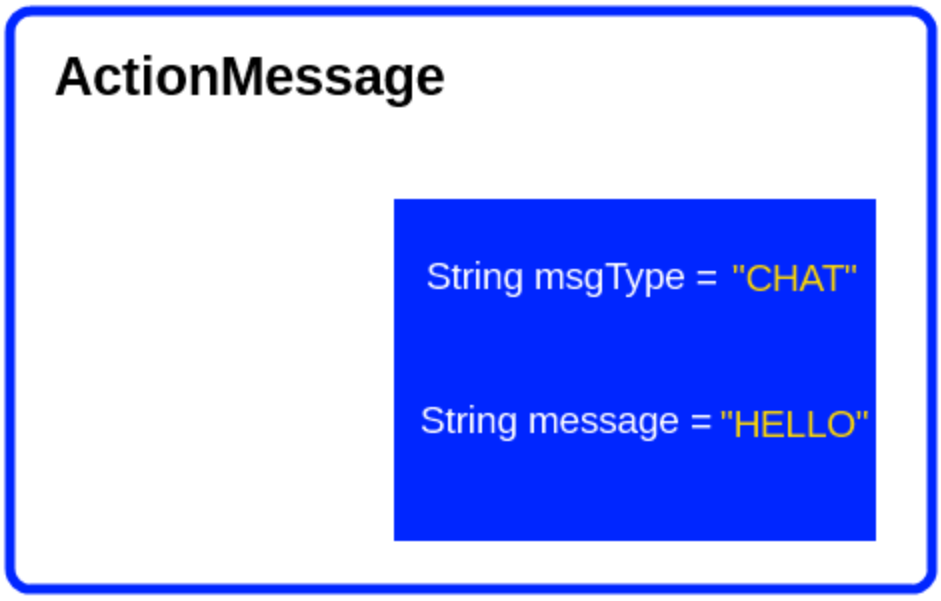
Multi-Threading

The first major problem encountered was dealing with the multiple threads required for this system. The server created a thread pool which created a thread for each client that connects to the server. Each thread listens for a message from the corresponding client and processes it. The problem arose when trying to send messages back from the server to the client. As the client was single-threaded it was difficult to send messages while also listening for messages back from the server. The solution was to create a thread in each client which listens for messages similar to how the server listens.



Socket writing/reading

In the early stages of development, the clients communicated with the server by writing and reading String messages. This was sufficient for sending chat messages but posed a problem when trying to send game actions to the server and back. The first option was to create separate listener threads for game actions, however this would create extra work for the server as there would have to be two threads per client. It was decided to send another object instead of a String. The system now sends an *ActionMesssage* object to the server, this object has a *type* attribute to tell the receiver how to parse it. For example, the chat message will have the type “CHAT” while a move action will have the type “MOVE”. Each action message will also have optional parameters, the receiver can parse the parameters depending on the *type* of the message.



Displaying the game board

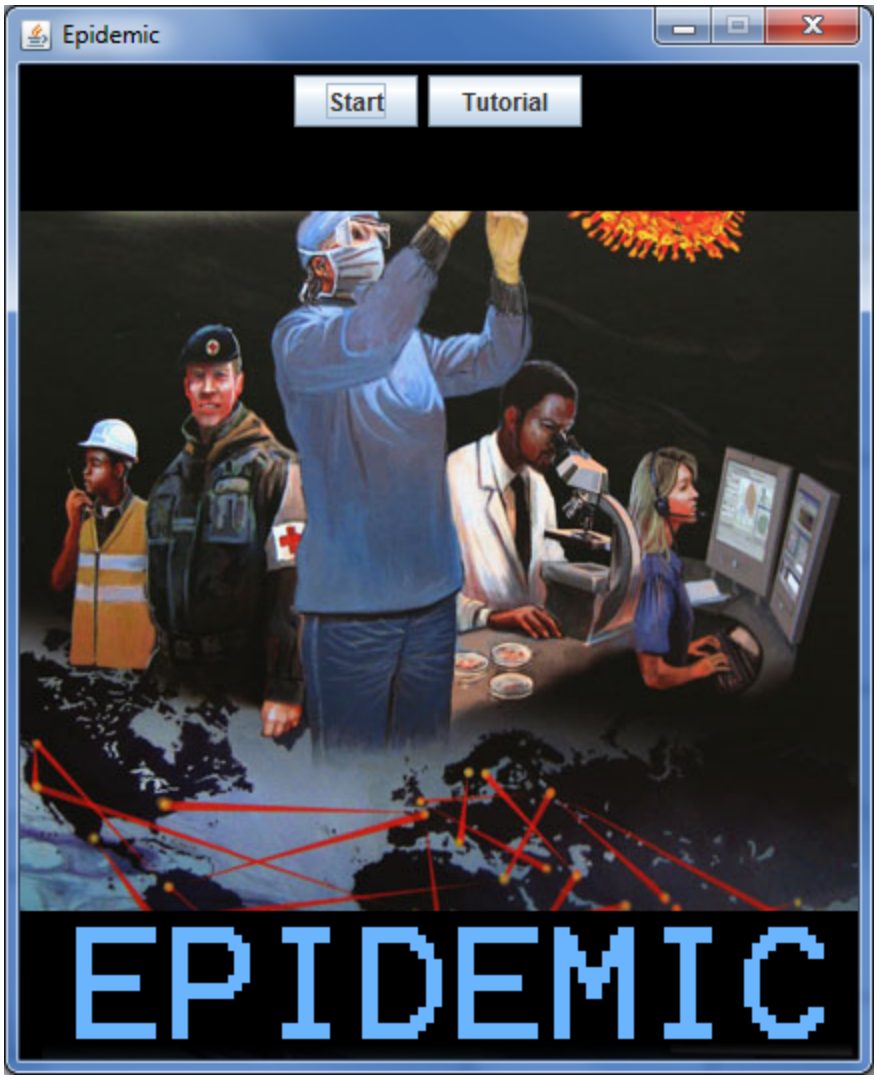
In the original system design, the game board was planned to be a static image displayed in the background that the game piece would be painted on top of. While this method would work for the default settings, it did not offer any room for change.

The game board in the final implementation is dynamically drawn. Each “Country” is drawn on the board with lines connecting adjacent countries drawn beneath them. In early implementations, the player could drag the countries around the window and arrange them to their satisfaction. This shows that the game board could be modified in future versions of the game. Currently the positions of the game pieces are located inside the “Countries.txt” file, the system reads this file and creates the objects based on the text file.

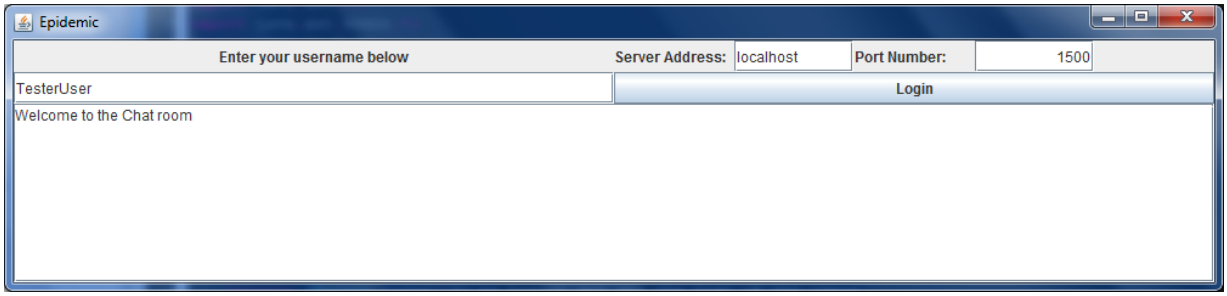
Turn-taking

As the entire system is multi-threaded, it was not feasible to have client’s threads “sleep” while waiting for other clients to take their turn, as the user may still want to perform actions such as sending chat messages and displaying updated information, therefore a safe method of telling each user if they can make a move in the game was required. It was also important to make sure that no two players could move at the same time.
Each client now has an *isTurn* boolean to determine whether it is their turn, only one client will have *isTurn = true* at any one time. If *isTurn* is false the player may not act. After each turn the server will broadcast a message, setting all client’s *isTurn* variable to false, then will send a message to the player whose turn is about to begin and set their *isTurn* variable to *true*.
Sending a blanked *false* message before setting the current player’s turn ensures that no two clients may act simultaneously.

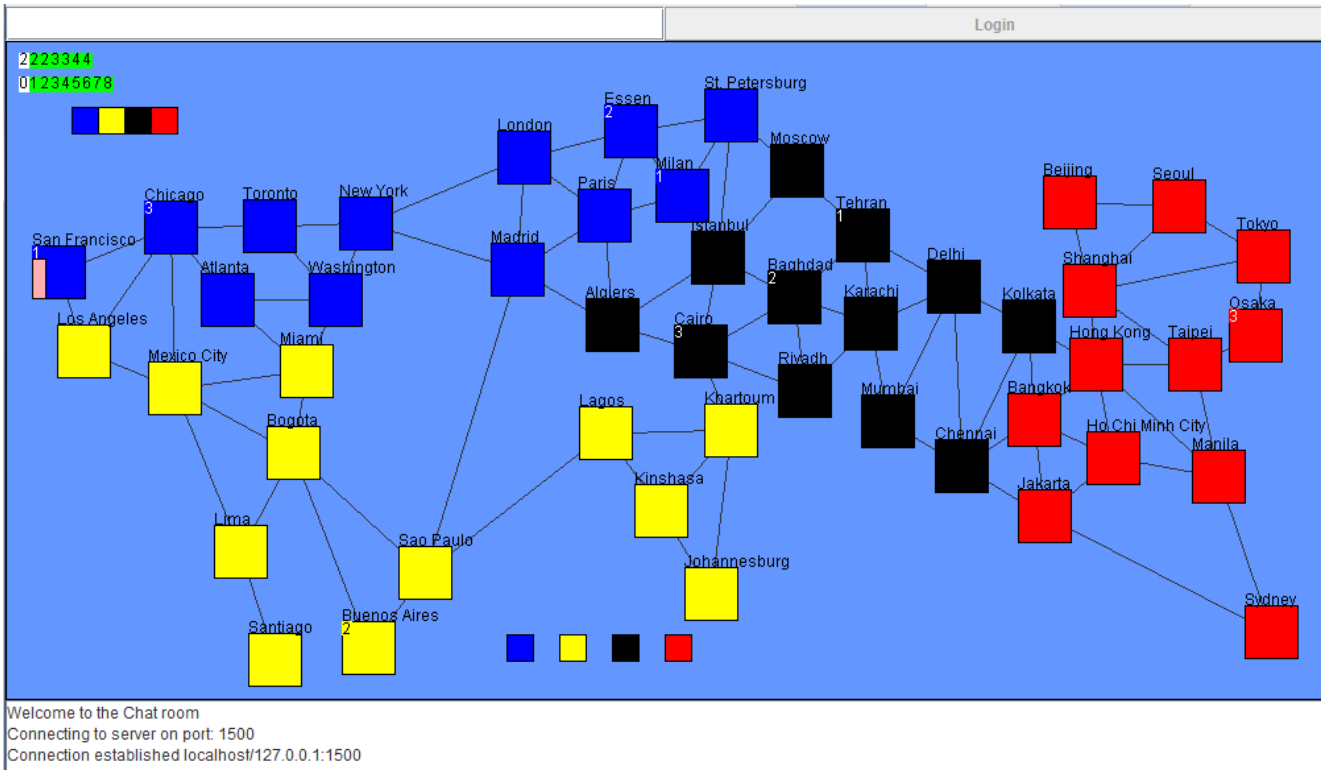
5. User manual



At the intro screen, click Tutorial to read the rules to the official game, *Pandemic*, or click Start to open the Client.



Enter your username in the text field and click *Login*.
You may now chat with other client by entering your message in the text field, then hitting the enter button.
When all clients are ready to play, type *!start*.



The meter in the top-left of the screen represents the infection rate, currently the value is 2, which means two cities will be infected at the end of the turn. The meter below this is the number of outbreaks, currently 0, if this reaches 8 the players lose.
The number in the top-left corner of each city represents the number of infection cubes in the city. A player has five action per turn and may spend one action to remove a cube from their current city by clicking the appropriately colour block in the top left corner of the screen.
The four boxes at the bottom of the screen represents cures, if a player holds 4 or more cards of the same colour they make click that coloured box to *cure* that colour.
The player may move to an adjacent country by dragging their player piece to the appropriate square.
The players may still chat freely using the chat box, even if it is not their turn.

6. Installation Guide

There are two packages required to operate the project, “Pandemic” and “Pandemic-Client”.

Contents of *Pandemic*:

- Server.java
- Player.java
- Country.java
- Board.java
- ActionMessage.java
- Countries.txt

Contents of *Pandemic-Client*:

- GUI.java
- IntroScreen.java
- Client.java
- Card.java
- Block.java
- ButtonBlock.java
- CountryBlock.java
- PlayerBlock.java
- ActionMessage.java
- Countries.txt
- Intro2.png

Additionally, the user will also require Java Developer Kit 6.0 or later.
If the user only wishes to connect to another server, they will only require *Pandemic-Client*.
To run the server, compile the .java files with javac, then run the server class using “java Server.java“.
To run the client, compile the .java files with javac, then run the server class using “java IntroScreen.java“ or “java GUI.java”.