Cmpt 431

Project Report

Group 17

Korban Torsun

Student #: 301326447

Email: ktorsen@sfu.ca

Rohm Laxton

Student #: 301294828

Email: rlaxton@sfu.ca

Rishabh Thanki

Student #: 301270143

Email: rthanki@sfu.ca

Grant Lee

Student #: 301197073

Email: grantl@sfu.ca

Architecture (Programing Language: Java)

Overall View

The system's communication architecture works as a mix between the request-reply and multicast protocol. Initially, the server waits all clients to join. Once the connection is established, all clients undergo a sync to the server and further communication is in the form of message objects. These messages are passed through two streams generated from the connection: ObjectInputStream and ObjectOutputStream. The messages are responsible for all types of actions, ranging from locking a square to transferring vital information for network initialization. For the game, player has their own local game board. The game board is comprised of individual squares that contain many individual properties such as locks, occupation, and size. Through the use of messages, each action performed by a player is mirrored onto all player game boards. For the graphics, Java's Graphics2D, Canvas, JFrame, and BufferStrategy are used.

Server (Main Classes: **Host.java,HostMenu.java, GameBoard.java, Message.java, ConnectionEndpoint.java, GameState.java)**

Once a player enters the host menu, their application will create a socket on the port defined by GameState. Once the socket is made, the host will block until a player connects to this socket. After a connection has been established, the host adds creates a new player object and adds them to a player list. The player object contains information such as id (the ip address) and the color of the player. The host also creates a ConnectionEndpoint object for the player and adds it to a connection list that can be found in GameState.java. The ConnectionEndpoint is responsible for the sending, reading and sometimes acting of messages. The connection list is important as it holds the reference to each client socket. After processing the first client, the host continues to repeat the same process for the next 2 players. Finally, once all players have joined the host sends a sync message to all players and the game begins. During gameplay, the host receives a message from clients. When this occurs, it may either be a message flagged as check, declaration, or release. When a check occurs, the host checks their game board for the square in question. If the square is in use, the host responds with a denial message which informs the client the request has been denied. But, if the square is vacant, the server will lock the square on their own game board and send a lock message to all players except the original requestor. The host will then send an approve message to the requestor, informing them they are free to use their requested square. A similar process is used for the unlocking of square as well as declaration of occupancy (takeover).

Client (Main classes: Client.java, ClientMenu.java, GameBoard.java, Message.java, ConnectionEndpoint.java, GameState.java**)**

Players are able to join the host through the join menu. A client inputs the host's ip and available port. The application then creates a socket based on the input and creates a new ConnectionEndpoint and passes that to a connection list. A delay, used for the syncing to process, had also been calculated using Java's System.nanoTime(). Once all clients have connected, a sync message is received from the host. The client then performs the sync, utilizing the information received, and the game begins. Once the

game has started, clients communicate with the host, sending messages regarding checking, releasing, or declaring. Clients may also receive messages involving lock, unlock, approve, deny, and update.

Messages (Main classes: ConnectionEndpoint.java, GameState.java, Host.java, Client.java**)**

Is the main source of communication between the clients and the host. There are two types of messages. One type is the synchronization message which is used for synchronization purposes. The second type is a gameplay message, used for gameplay updates. For synchronization, the synchronization carries the following information: A sync flag, the server's current time, the server's player list, the server's address, and the client's player index from the player list. A sync flag indicates to the receiver that the message is used for syncing. The server's time, in addition to the delay calculated, is used by clients to sync their own internal times. The server's player list contains all players participating in the game. The server's address is the ip address of the server. The client's player index, is used so the client may set their own id using the newly received player list. For gameplay, the message holds information regarding: timestamp, playerID, action, squareID, and sourceIPAddress. The timestamp allows clients to determine which order the messages should be performed. The playerID is the requesting player's ID. The action is the action to be performed on the gameboard by hosts or clients and include operations such as "lock", "unlock", "update". The squareID is the id of the square to which the action should be performed on. The sourceIpAddress is the address of the sender of the message.

Receive Buffer(Main classes: ConnectionEndpoint.java**)**

The receive buffer is an ArrayList that is used for receiving messages. It is used in unison with a receiver thread which constantly runs in a loop and checks whether a message had arrived through the ObjectInputStream. Whenever a host or client wants a read a message, the receive buffer will perform a sort using the selection sort algorithm. This sort arranges messages based on their timestamp, placing newer messages to the beginning of the list. Thus, whenever a client or host reads a message, they are reading the most earliest time stamped message found in the receive buffer.

GameBoard (Main classes: GameBoard.java, Square.java**)**

The game board is comprised of numerous square objects. At initialization, the game board creates a list and adds a new square objects to populate the board. Once the board has been populated the game is ready to be played. The game board also listens for the mouse actions. An ArrayList of shapes called penHistory is used for recording the player's current draw attempt. This list is used for the rending and calculating of potential occupancy (territory take over). Each client and host has their own local copy of the game board.

GameState (Main classes: GameState.java**)**

The GameState is where the player and connection list is located. It is also where game properties such as territoryLimit and penThickness may be set. Moreover, the GameState is in charge of the state of the

application. One example is the transitioning from the main menu to the client menu. Additionally, each player contains their own local copy of the GameState.

Crashing (Main classes: ConnectionEndpoint.java**)**

Because replication is present, the game may continue despite the host crashing. When a host crashes, the client in the first index in the player list is appointed as the new host. This new host removes the old host from the player list and creates new sockets on available ports for the other players to join. Once everyone has joined, the new host resends a sync message and the game continues. This process is repeated if the new host also crashes. The system also handles client crashing. When a client crashes, the host will simply drop their connection from the player list and connection list. It is not necessary to update everyones player list because only the server had direct communication with the crashed client.

Design Choices

TCP was used instead of UDP because it is able to detect when a crash occurs quickly. It also ensures messages are not corrupt upon arrival. Whenever a host or client crashes, an IOException is thrown by the protocol and the required method is executed.

A mixture of reply-request and multicasting was used instead of other protocols because of its simplicity and effectiveness in accomplishing our goals. By using these protocols, the number of messages required for communication is dramatically reduced. For instance, if the architecture were solely reply-request, a connection for a client or host must be made with everyone. This would result in (n-1)*n total connections, as opposed to n-1 connections.

Receive buffer as opposed to none. By utilizing a receive buffer, messages won't be dropped on arrival and are instead stored for later processing. The receive buffer also allows for the sorting of messages based on time stamp. An important process that must be performed when governing which actions had occurred before others.

A render and tick method for classes. It was implemented as a more organized and structured way to perform the drawings and logical updates required of the application and project them onto the main canvas.

Initial Design