Deny and Conquer

Architecture

For this project, we decided to use Java as the main programming language, along with JavaFX for creating the GUI and UDP for the core networking. We used a traditional Client-Server model instead of a peer to peer model. We have 4 different Clients running on 4 different computers and 1 Server (on 1 of the machines) that can be automatically replicated on another machine if the main server goes down after the start of the game.

This program runs using multiple threads as well as queues to successfully implement all the requirements. In general, the code is divided into 3 layers:

- UI: this includes a number of classes that are responsible for displaying and updating the game graphics, and taking inputs from user.
- Logic: the underlying component that actually holds the game states and calculations.
- Networking: with 2 threads (send and receive), it acts pretty much the same as a small application of the transport layer using only UDP.

This module-oriented design is used to make it easier for components to interact as they are all separated and it makes it easier to add more functionality by decoupling. We will further go into the details of our project below.

Why Java and UDP were used

We used Java to develop our program because we were really familiar with the language, and it had more online resources available which were comparatively easy to follow. At the same time, Java allowed us to work easily with queues, threads and synchronization.

Throughout the game, the messages are exchanged in UDP as opposed to TCP. Unlike TCP, there is no guarantee of delivery in UDP. This however, makes the communication almost immediate, which was an essential element in our game. Another important reason behind using UDP was that it makes switching servers when the main server goes down easier as compared to TCP. In order to switch server, we simply had to point to the new server rather than having a handshaking dialogue in the beginning. Lastly, TCP provides a wide range of services but we only need reliable data transfer so using the whole pack just to use 1 service was not an optimal solution.

Even though we wanted the game to be faster, we still needed reliability for some of the key messages (mentioned in the next section). Technically, UDP can be TCP if we implement all the services on the application layer (in TCP all services are built in).

Reliable Data Transfer

In order to achieve this, we keep track of the messages that are supposed to be sent reliably. For every new message of this type, we first record the current timestamp, use it as an ID (so everything will have a unique ID) and append to the end of the message, then put in a dedicated queue to keep track of it. At the receiver's side, once the message arrives, the receiver will send an ack together with the ID back so the sender can pop it from the queue. That means, if the sender does not receive an ack (in our case, within 20ms), it will resend the message with the

same ID. We also keep track of the last time we send and number of retries to make sure we only send once every 20ms, and it is considered as a send failure (and message is dropped) if we do not receive an ack after 5 times.

In the case of duplicates, receiver sends an ack for whatever it receives and push the message upwards to the Logic layer to handle duplicates, where the message will be checked and dropped accordingly. At the sender's side, a duplicate ack is automatically dropped since we cannot find its original message in the reliable queue anyway.

Everything about reliable transfer and acknowledgements is done in the networking component so from Logic we only need to call send and it will be taken care of.

General Message Formats in UDP

The core communications for this game we done through several custom messages send through UDP. Many of these messages were sent unreliability as it was not important but few key messages were sent using a custom reliability message queue we implemented. If it is an unknown message type, it is dropped as we know it is not part of our program. Also, duplicate messages are taken care of by the client/server threads.

We used "#" to split up the parts of every message. The general format of every message is as follow:
[For Client/Server]#[Type of message]#[Payload]#[ID, only for reliable messages]

[For Client/Server]: either 0 (for server to receive) or 1 (for client). The reason for this is because a host machine has 2 both client and server threads running at the same time. So even if it's the same machine, a message from the client still has to go to the networking component to be sent and travel back to the server thread. This architecture is to create a separation between server and client threads (to achieve single responsibility rule) and also it makes it less complex without checking if the same machine is a server or client every time a message comes.

[Type of message]: numbered differently for each type. Refer to the below section for more information.

[Payload]: The number of parts varies depending on the type of message.

[ID, only for reliable messages]: The ID of message, used by the receiver for sending acknowledgements.

Custom Message Formats in UDP

Below are all the custom messages we used and what they did. Incoming messages also indicate what type of message it is and is dealt with differently. Since these are in Logic and UI layers, we will omit [For Client/Server] and [ID, only for reliable messages] parts as they are considered as header and trailer in the networking layer.

Request for Connection:
Format: 1#[Name of machine]

This is the first message sent to the server, if successful connection to the server is established the clients GUI will prompt the user to wait for the server to start the game. If the server is a backup server that spawns, this ensures that we are connecting to the right machine. This uses our custom reliable transfer.

Accept Connection:
Format: 2#[Serialized game board]

This is the answer from server to notify the client that it is connecting to the right server. This is automatically got dropped if the server thread in the machine does not run, meaning that the other user thinks this is the server while it is not. This prevents the inconsistency of the game. This uses our custom reliable transfer.

[Serialized game board]: only used when the backup server goes online to resynchronize the game board with the clients since all clients may not receive the last update messages from the main server before losing the connection. This is simply a series of numbers from 0 to 4 representing the owner of the boxes in the grid (left to right, top to bottom), with 0 being no owner and 1-4 being player ID.

Start Game:
Format: 3#[Brush Size]#[Grid Size]#[Fill Percentage]#[Server time]#[Player list]

This is the initial start message sent from the server to all the connected clients. This message includes the server game settings such as brush size, grid size, and fill percentage for the game. The server also sends its own server time so each player can take it as an initial point to calculate delay time. This uses our custom reliable transfer.

[Player list]: the list of players so everyone will maintain the same order of the player list. This helps in the stage of fault tolerance and replication (mentioned below). Each player has a formart: [IP1]#[IP2]#[IP3]#[IP4]#[Player name]#[Player ID] where IPs are fields of an IP (x.x.x.x) and Player ID is just a number assigned to each player by the server.

Update on Box Drawing:
Format: 4#[Mouse Position X]#[Mouse Position Y]#[Player ID]

This is the general message used for sending pixels we are drawing. Since everyone has the same board, we only need to send the mouse position and the receiver will use the coordinates together with brush size to generate the drawings on their own canvas. This saves a lot of bandwidth since we do not have to send every single pixel, only the mouse movements. Player ID is used to indicate who draws. This does not need to be reliable.

Capture Success:
Format: 5#[Box Position X]#[Box Position Y]#[Player ID]

This message is sent when a player successfully captures a square. We send the box coordinates (based on game grid) together with who captures it so the receiver can just fill the whole box. This uses our custom reliable transfer.

Capture Failure / Box Released:
Format: 6#[Box Position X]#[Box Position Y]#[Player ID]

This is pretty much the same as Capture Success. Both are sent when the mouse is released. The client calculates if it is a valid capture based on fill percentage and decides which action to do (success or failure) and send.

Request to Lock Box:
Format: 7#[Box Position X]#[Box Position Y]#[Player ID]

This message is sent when a player initially locks the box, it locks the box with their ID if it is available until the user receives the confirmation from the server. In the case of disallowance, the user will not be able to keep drawing on the box anymore, otherwise it shows like no changes has happened. This uses our custom reliable transfer.

Game Over:
Format: 8#[If Player 1 Wins]#[If Player 2 Wins]#[If Player 3 Wins]#[If Player 4 Wins]

This message is sent by the server when all the boxes in the game are filled, This message will be accompanied by a message indicating the color (or colors if there are multiple winners) of the winning players. This uses our custom reliable transfer.

[If Player x Wins]: if the player is a winner, it is 1, otherwise it is 0.

Ping:
Format: 9#
This message is sent periodically (every half a second) and the server replies, this is to ensure the game will not assume the server is down just because all players stop drawing for 2 seconds. This also is used to synchronize the clock (refers to Coordination).

Ack:
Format: 99#[Message ID]

This is simply a general acknowledgement message for a certain message.

Lost Connection to Server:
Format: 80

This is not a message that is sent through the network, but only an internal message between Logic and UI to indicate that the connection is lost and it is establishing a connection to the backup server. The user will be indicated with a pop up telling about this problem so it is not transparency, which is a good thing in this case.

Threads and Queues

The game consists of threads and message queues. We have 5 main threads on a client (6 on a server). Threads consists of Main Application thread (UI), its listener thread to update the canvas (UI), Client thread (Logic) dealing with client game logic and maintaining the clients GUI, Server thread (Logic) processing the messages on the server and replying to a certain client or broadcasting them, Networking send thread (Networking) taking care of outgoing messages, Networking receive thread (Networking) dealing with incoming messages. All threads

communicate via several queues using Consumer-Producer pattern, implemented with blocking queues.

Main Application thread:
- Takes inputs from user (mouse movements) and generates messages to put in UIsendQueue.

UI listener thread:
- Takes messages in UIrecvQueue and updates UI accordingly.

Client thread:
- Takes messages from UIsendQueue, processes them and pushes to sendQueue (messages with destinations).
- Takes messages from clientRecvQueue, extracts the meanings and puts them in UIrecvQueue so the UI can be updated.

Server thread:
- Takes messages from UIsendQueue, processes them and pushes to sendQueue (messages with destinations).
- Takes messages from clientRecvQueue, extracts the meanings and puts them in UIrecvQueue so the UI can be updated.
- For Request to Lock Box messages, the server will store the initial message in lockRequests and wait for 20ms before processing the message (refers to Concurrency for more information).

Networking send thread:
- Takes messages from sendQueue, adds trailer for reliable messages, record them in reliableQueue (for retransmission if needed) then sends them to the network.

Networking receive thread:
- Listens to any incoming messages from the network.
- Sends them to server thread using serverRecvQueue or client thread using clientRecvQueue accordingly.
- If the message needs an ack, creates the ack and put it in the sendQueue, which will be processed by Networking send thread later.
- If the received message is an ack, find the corresponding original message in the reliableQueue and pop it, indicating that the message has been transmitted reliably.

Game Initialization

This game can be played with up to 4 players. One of the players will have to choose to be the server, upon starting they will be greeted by a page where they can set up the game with the brush size, grid size and fill percentage needed to capture a square. Their IP will also be shown during this phase. Any other players wanting to connect will chose the client option from the GUI and enter the server IP. Upon connecting, the players will now be informed they have to wait until the server starts the game.

After the server is ready and all required players have successfully connected to the server, the server presses Start button. This then sends a list of all the players in the game in the same order to everyone connected (will be used for server replication), as well as the server settings and then starts the game for all the clients at once. To summarize, all clients now have the same list of players (in the same order) and the same game settings.

Game ending

Every time the server receives a Capture Success message, it will check the board to see if all boxes are filled. Then it calculates the winner(s) and broadcast a Game Over message. Once the client receives the message, it pops up a notification on who wins and stops all the threads. The gameplay ends here.

Coordination

Coordination is done by using a custom time synchronization algorithm to ensure all the clients use time relative to the server, for example if current client time is 10:30 and current server time is 10:45, the message the client sends will be time stamped 10:45 (as well as accounting for the rtt delay clients have).

The algorithm is fairly simple, it is as follows (assume 10:30 client time, 10:45 server time, 00:01 is RTT/2)

1. Client starts timer, saves current client time and sends server timeSync message
2. Server replies with its current time and client stops timer.
3. Client calculates RTT/2 using timer (we shall call this delay)
4. Client compares client time and server time along with delay added (10:45+00:01-10:30) to get a difference of 16min (meaning server is 16min ahead) we save the difference
5. Next time we send message we add on this difference to the client time so it will be at the same time scheme as server time.

We use this timestamp to determine who gains the lock on a box. Read the below section for more information.

Concurrency

The game achieves concurrency in the grid by sending lock requests to the server for the box that is being drawn in. If there is more than one player attempting to draw in a single box, their packets compete for the lock and the server decides who gets such a lock based on the timestamp.

In particular, the server, upon receiving the first request, will put it in the queue (called lockRequests) together with the time server receives it. The head of the queue is check in every iteration of the server loop to see if 20ms has passed (since every thread is basically a loop of checking if there's anything to process). In other words, the server will wait for 20ms before resolving any conflict. The it will iterate through the list to find the lowest time (sent by the client, including delays) amongst all the requests on the same box then broadcast the winner to everyone.

The lock functionality on the client side is done based on events, in this case, mouse pressed and dragged. Every time the user presses or drags the mouse on UI layer, it will first check the corresponding box on the logic layer to allow or disallow the update. So when the client receives the broadcast from server, it will update its grid in the logic and the UI will stop the user from drawing on the same box if it is currently locked by another user. Once the clients understand

that a box is locked they will not be able to draw in the square unless it gets unlocked due to the other player losing the box.

We also used 2 threads (1 for user inputs and 1 for updates from server) and synchronization to allow concurrent drawings on the gameboard. Doing this allowed us to see what other players were drawing on every clients screen in real time while we draw.

Fault Tolerance by Replication

As mentioned, during game initialization the server creates a chronological list of all the players that join along with their IP, with the server's IP being the head of the list. During the game, the clients keep track of the latest message received from the server. This includes updates and ping replies (reasons for this is already explained above). If the last-time-seen exceeds 2 seconds, the client will assume that connection is lost. Since the server is always in the head of the queue, the server is removed from the player list, and the next player on the list is selected as server. This player will start a server thread with that players current game board. The other players will now point to that player as the server from then on (as all the players have identical player lists in order).

There might be a case when a client loses connection to the server while it is still running. Then that particular client will assume the second one in the list to be the server and will try to request for connection. As mentioned in Custom Message Formats in UDP - Accept Connection, there will be no replies so the client will have to contact another person. This will keep going until the same machine becomes the server so it will never crash. Also, if there are multiple servers trying to send their own update message to the same client, only one is accepted as the client only accepts messages coming from the IP that it believes is the server.

During the restart, users will be notified with a pop up saying connection to the server is lost. This must be transparent to let the users know why their game is not responding and also buy some time for the backup server thread to start and resynchronization of the game. This backup server takes the same settings and game board from its client thread then sends to whoever requests for the connection.