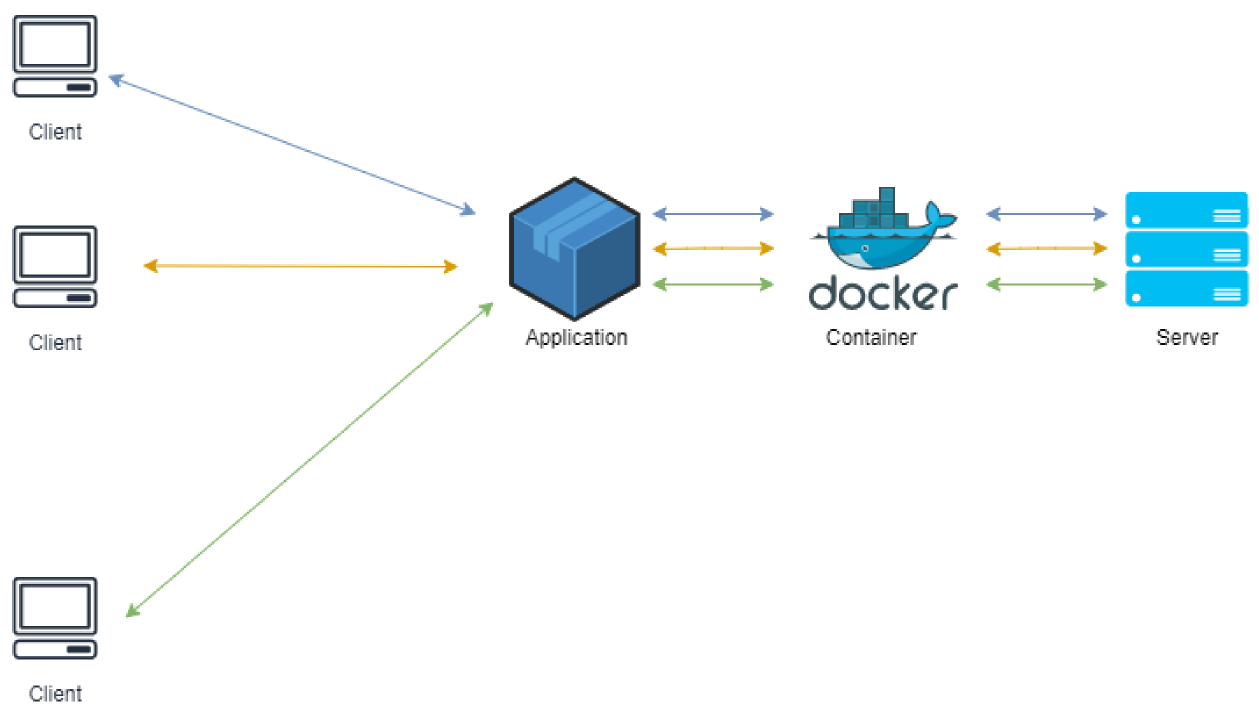
Cloud based Collaborative Whiteboard

Distributed System Design-



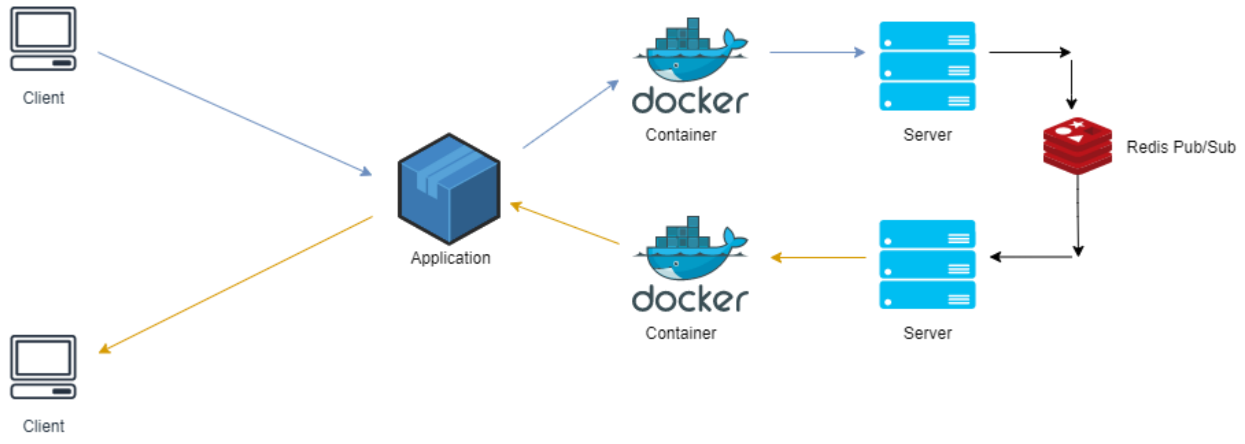
The architecture shows that there can be many clients on their own independent systems connecting to our application which is containerized using docker, and those clients can establish connections with the server.

Planning to implement this using Node.js for the server-side because it is event-based and hence is widely used in real-time applications. Node.js is a popular choice for any application which requires constantly updated data to be broadcasted - for example, games. Even though Node.js is a single-threaded application, it is good at handling multiple concurrent requests. To establish communication in the client/server model, planning to use socket.io, a library for Node.js which uses sockets to communicate, and also event-based.

Real time applications are generally applications which have strong consistency and hence I am planning to have strong consistency over eventual consistency. To achieve strong consistency, planning to emit the events to the server as soon as they occur on the client-side, which inturn will emit those events to all the other clients connected to that server.

To achieve scalability and fault tolerance, I am planning to implement a Publish-Subscribe model. By implementing a Pub/sub model, we can have multiple servers and also achieve strong consistency - because whenever a data is written by a client to the server which pushes that data to the database, other servers are informed that there is new data. By having multiple servers, it also makes it fault-tolerant than just having one server - because, whenever a server fails, all the clients which have been connected to that server can easily switchover to another server and still have access to the collaborative whiteboard.

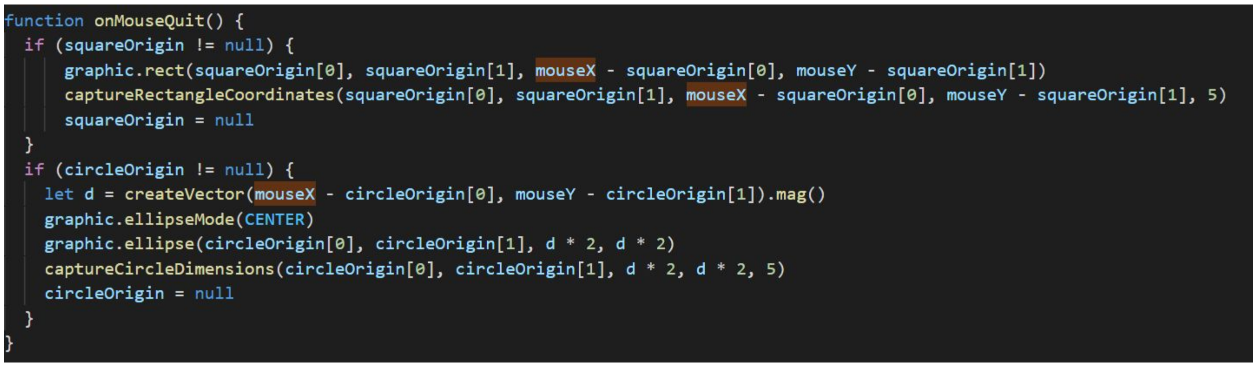
Architecture with publish/subscribe model



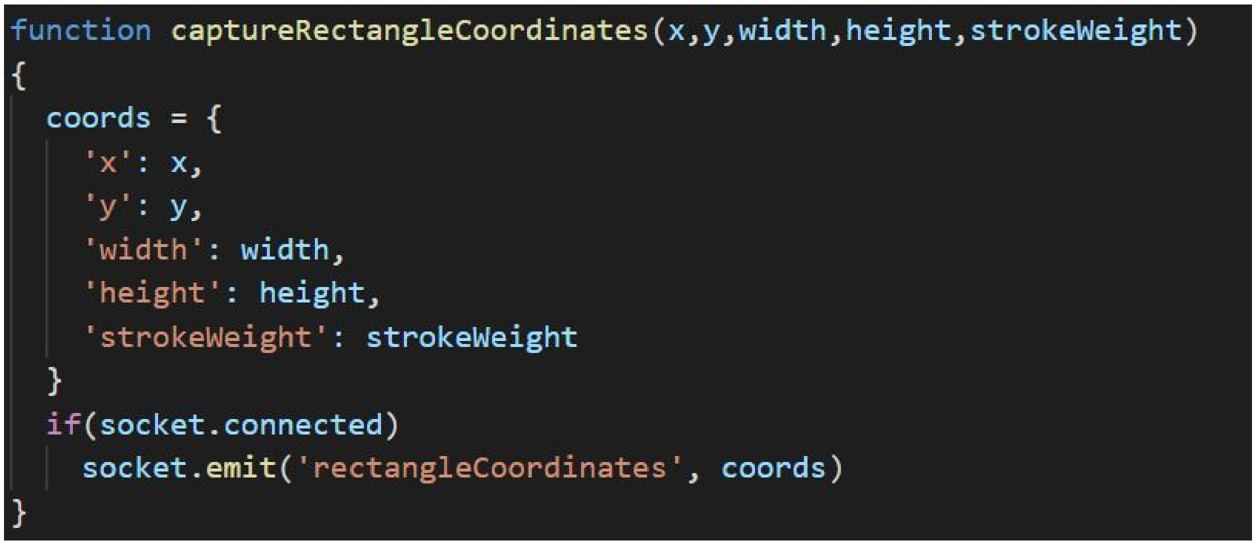
With respect to performance, to avoid creating bottlenecks at the server, I am planning to restrict the user to basic drawing shapes such as a line, rectangle (or square) and a circle. This is to avoid the user from having a free drawing pen where it would require to send lots of messages containing (x,y) coordinates to the server. Whereas, if the user was restricted to the drawing shapes such as rectangle, it would only require sending the x-point,y-point, the width and the height to the server to recreate the rectangle at other clients.

Distributed State Management

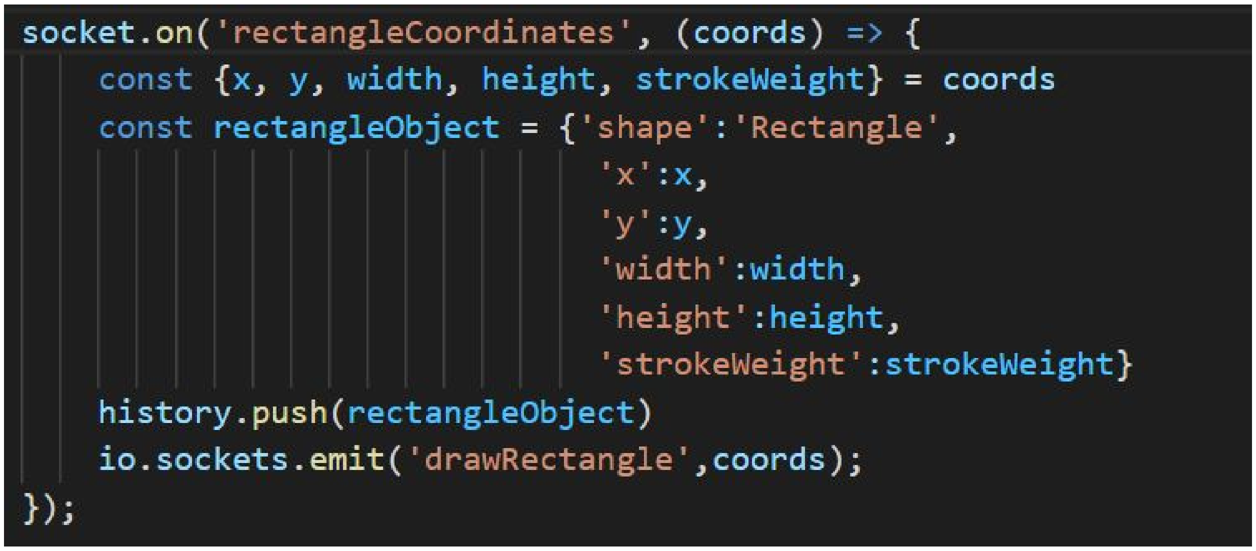
I have achieved state management by storing all the clients drawing history on the server's memory. Whenever a user draws a shape, the coordinates and its shape is sent to the server which stores that information in its memory and whenever a new user joins the server, this information is accessed and drawn on the client's side.

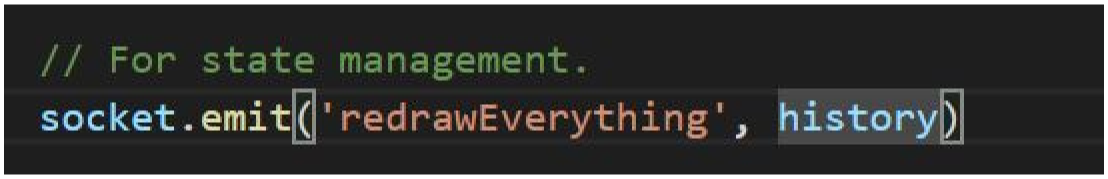


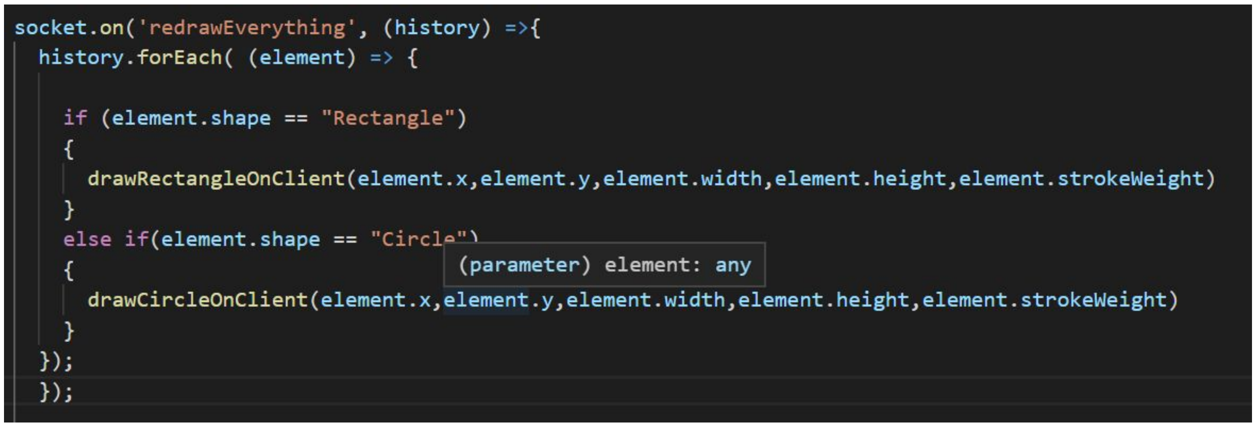
```
function onMouseQuit() {
  if (squareOrigin != null) {
      graphic.rect(squareOrigin[0], squareOrigin[1], mouseX - squareOrigin[0], mouseY - squareOrigin[1])
      captureRectangleCoordinates(squareOrigin[0], squareOrigin[1], mouseX - squareOrigin[0], mouseY - squareOrigin[1], 5)
      squareOrigin = null
  }
  if (circleOrigin != null) {
    let d = createVector(mouseX - circleOrigin[0], mouseY - circleOrigin[1]).mag()
    graphic.ellipseMode(CENTER)
    graphic.ellipse(circleOrigin[0], circleOrigin[1], d * 2, d * 2)
    captureCircleDimensions(circleOrigin[0], circleOrigin[1], d * 2, d * 2, 5)
    circleOrigin = null
  }
}
```


```
function captureRectangleCoordinates(x,y,width,height,strokeWeight)
{
  coords = {
    'x': x,
    'y': y,
    'width': width,
    'height': height,
    'strokeWeight': strokeWeight
  }
  if(socket.connected)
    socket.emit('rectangleCoordinates', coords)
}
```

Code walkthrough - onMouseQuit is where we draw the shapes on the client side. captureRectangleCoordinates is what captures the coordinates and emits those coordinates to the server.


```
socket.on('rectangleCoordinates', (coords) => {
    const {x, y, width, height, strokeWeight} = coords
    const rectangleObject = {'shape':'Rectangle',
                             'x':x,
                             'y':y,
                             'width':width,
                             'height':height,
                             'strokeWeight':strokeWeight}
    history.push(rectangleObject)
    io.sockets.emit('drawRectangle',coords);
});
```


```
// For state management.
socket.emit('redrawEverything', history)
```

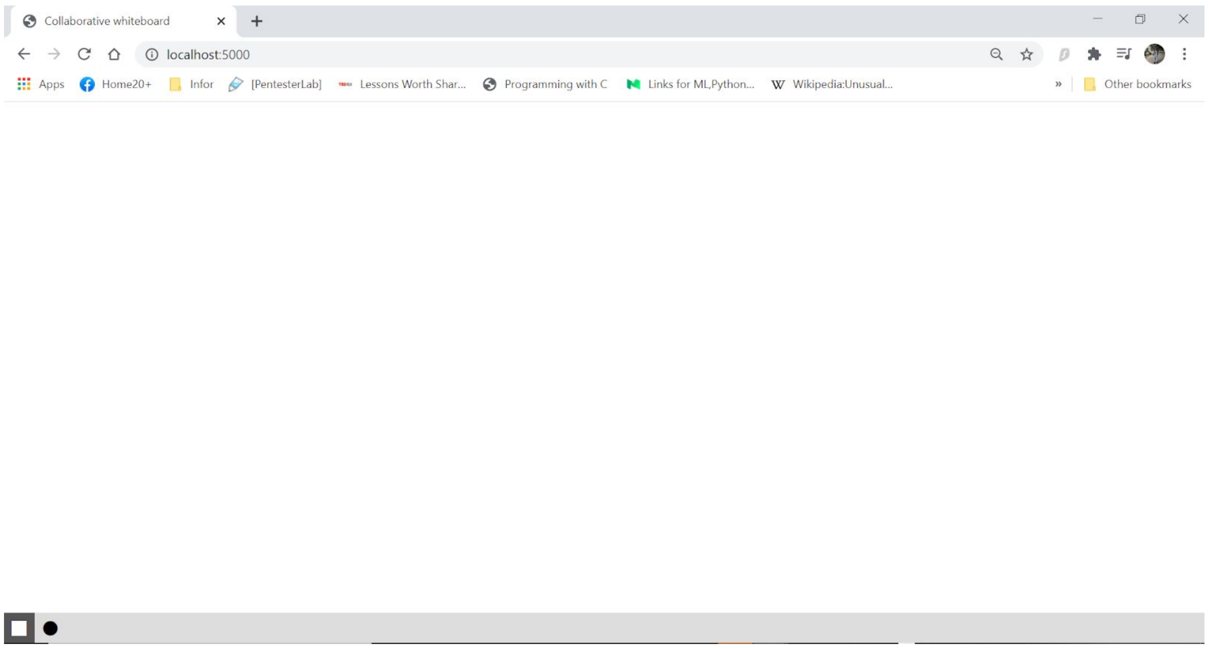


```
socket.on('redrawEverything', (history) =>{
  history.forEach( (element) => {

    if (element.shape == "Rectangle")
    {
      drawRectangleOnClient(element.x,element.y,element.width,element.height,element.strokeWeight)
    }
    else if(element.shape == "Circle")
    {
      drawCircleOnClient(element.x,element.y,element.width,element.height,element.strokeWeight)
    }
  });
  });
```

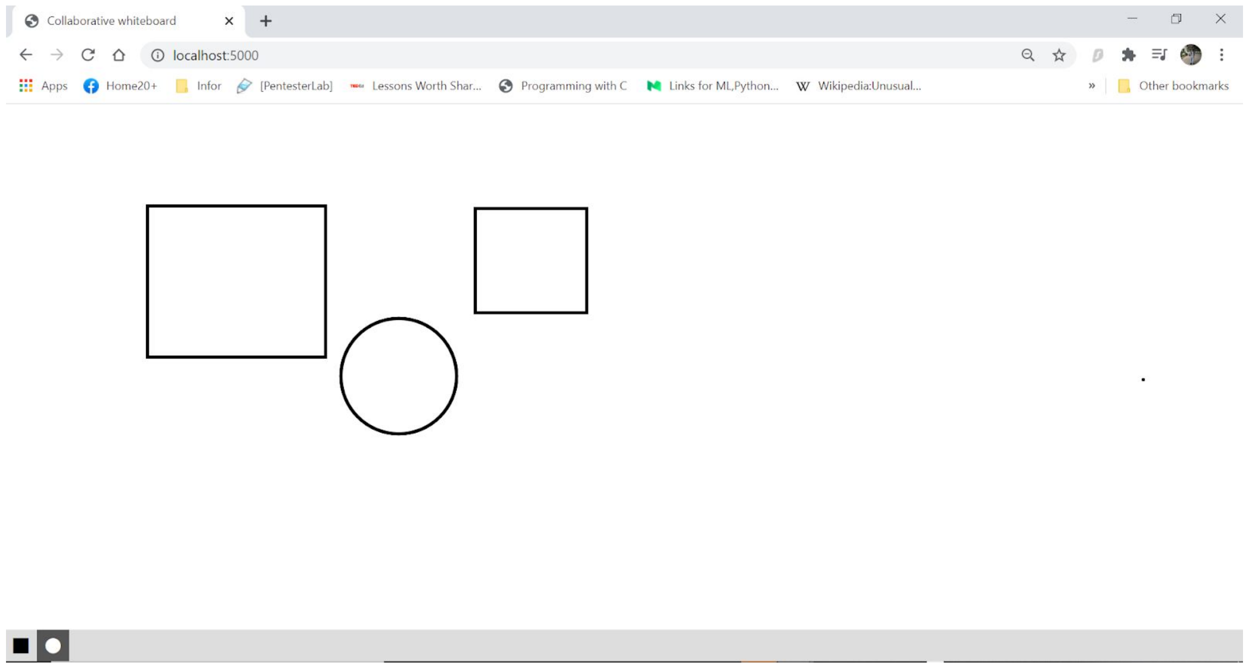
Code walkthrough - On the server, when we get the event 'rectangleCoordinates', we store this information in an array and we emit this array to the newly joining clients. On the client, we loop through each object and draw them correspondingly.

State management -
Step - 1: Open a browser window to establish connection with the server



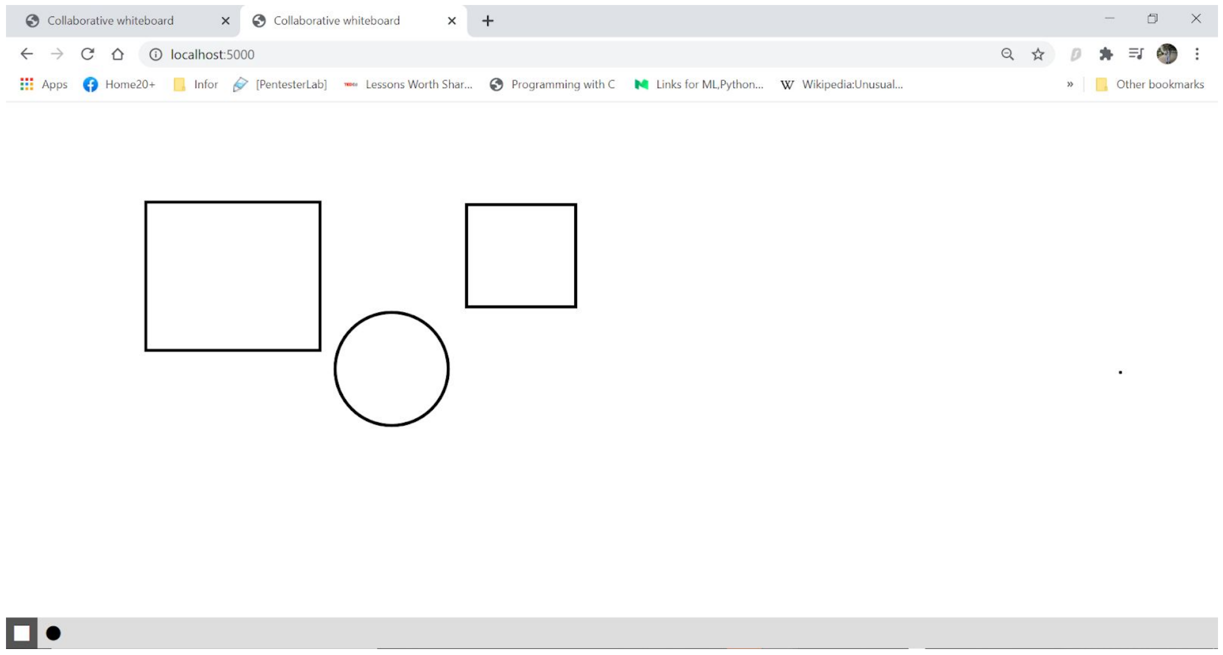
Empty canvas state with only one connection present

Step 2 - Draw on the canvas using the rectangle and circle shapes



Canvas with drawings on connection - 1

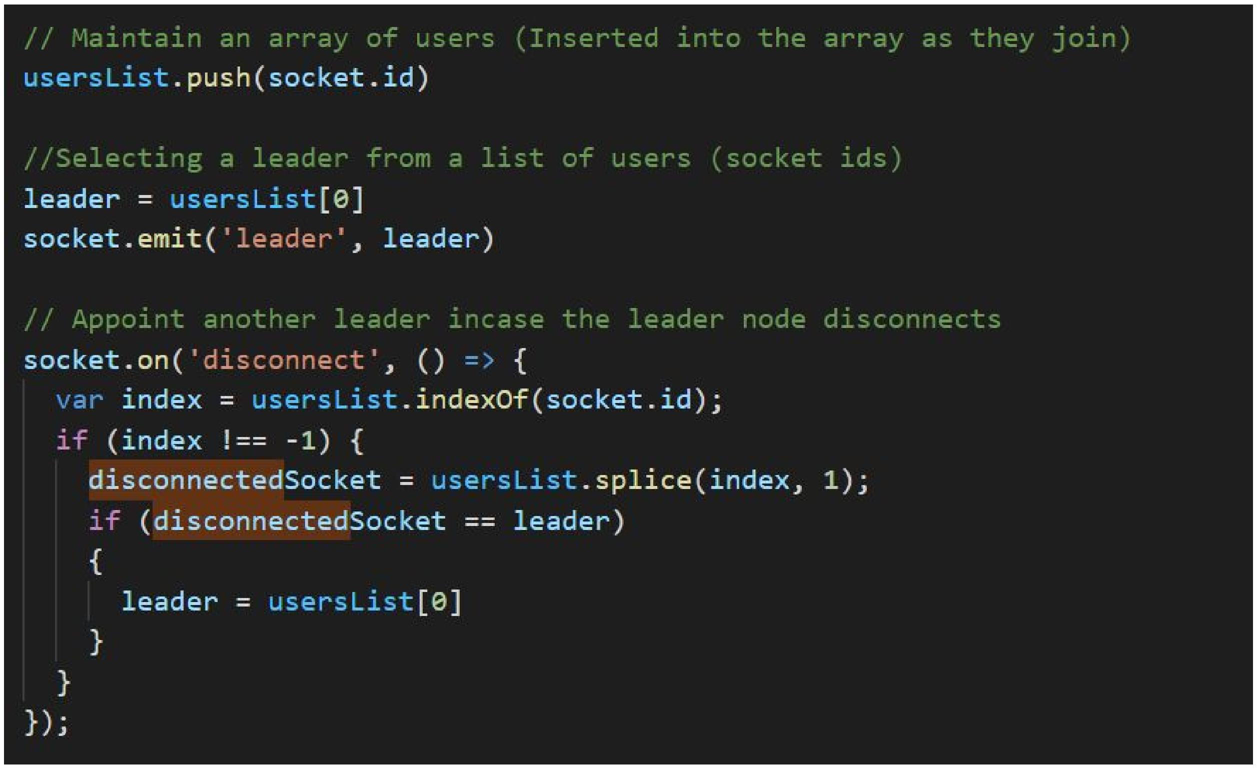
Step 3 - Open a new browser window and observe that the new connection is also able to see the current canvas state



Same canvas state on connection - 2 (new connection)

Consensus and leadership

Consensus can be achieved through a variety of ways (Paxos,Raft,etc…). Consensus is basically an agreement among all the nodes in the system. If consensus is agreed upon on a particular state (or command), then no other node will apply a different command - so that all the nodes in the system will apply those commands agreed upon in series and will reach the same state.
I implemented an algorithm which will choose the leader based who connected to the server first. If that leader node fails, the algorithm will appoint the next leader based on the order of connections.



```
// Maintain an array of users (Inserted into the array as they join)
usersList.push(socket.id)

//Selecting a leader from a list of users (socket ids)
leader = usersList[0]
socket.emit('leader', leader)

// Appoint another leader incase the leader node disconnects
socket.on('disconnect', () => {
  var index = usersList.indexOf(socket.id);
  if (index !== -1) {
    disconnectedSocket = usersList.splice(index, 1);
    if (disconnectedSocket == leader)
    {
      leader = usersList[0]
    }
  }
});
```

Code walkthrough - Whenever a connection is made to the server, the user’s socket ID is stored and the leader is basically choosing the first socket ID from the list, and whenever a connection is lost, we check if the disconnectedUser is the same as our leader, and appoint a new leader if the leader lost connection.

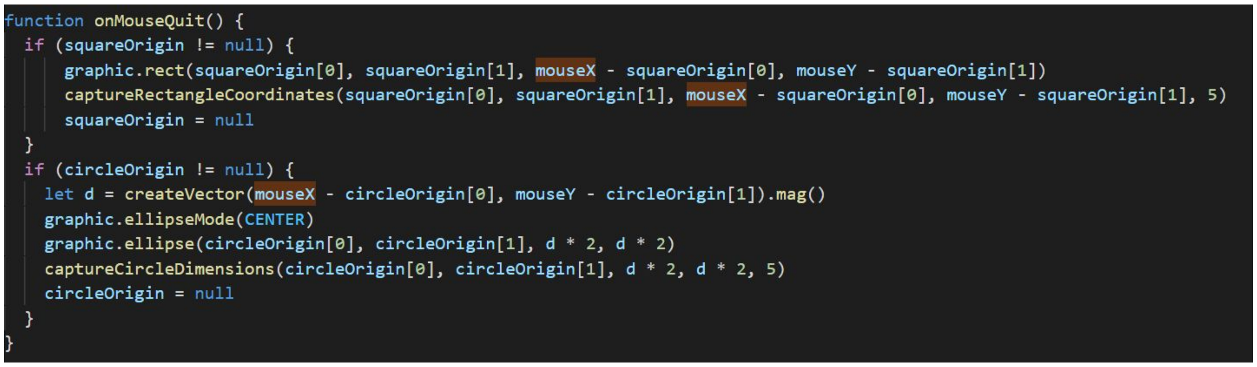
Replication

I have tried to implement the application involving Redis publish-subscribe model which would publish to the Redis channel whenever data has changed on a server and notify other servers which are subscribed to the Redis channel that data has changed on a server allowing them to fetch the latest data to achieve strong consistency.

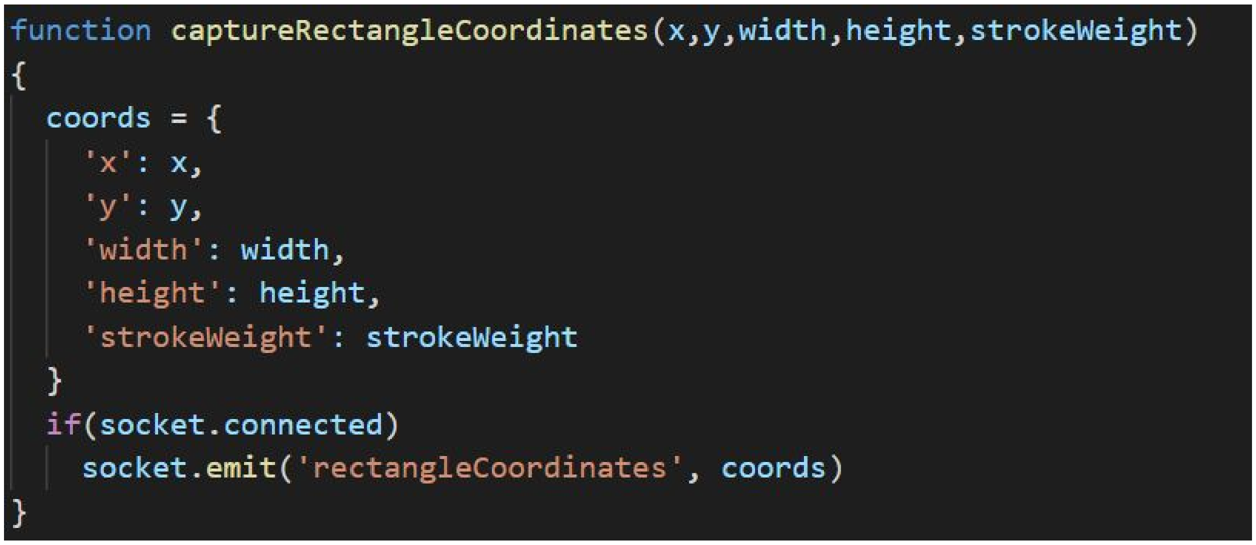
I have chosen strong consistency over eventual consistency because I have considered this application a real-time application. Eventual consistency would have led clients to access stale canvas states and it would have not been real-time. However, with strong consistency, we will usually have high latency problems.

Performance and concurrency

Performance is handled by only sending the required content needed to replicate the same drawing on other clients. For example, if a user wanted to draw a rectangle - they would use the rectangle drawing tool to draw the rectangle and only the information needed to recreate it at the client would be passed. However, if the user had used a free drawing pen to draw a rectangle - they would need to manually sketch the whole rectangle and this would require sending lots of (x,y) points to the server which might create a bottleneck at the server.

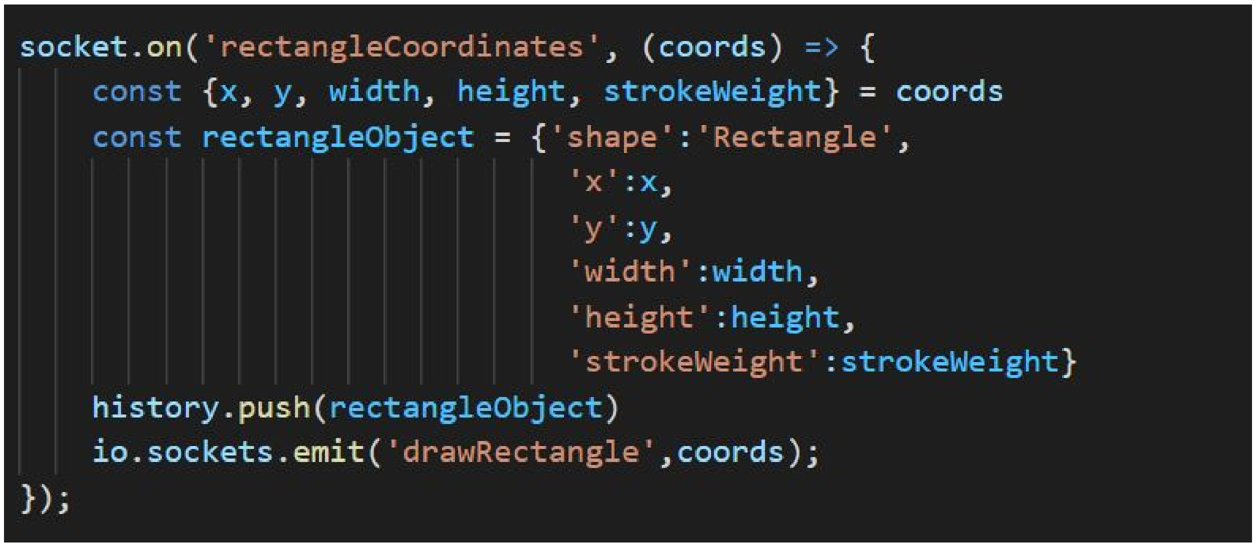


```
function onMouseQuit() {
  if (squareOrigin != null) {
      graphic.rect(squareOrigin[0], squareOrigin[1], mouseX - squareOrigin[0], mouseY - squareOrigin[1])
      captureRectangleCoordinates(squareOrigin[0], squareOrigin[1], mouseX - squareOrigin[0], mouseY - squareOrigin[1], 5)
      squareOrigin = null
  }
  if (circleOrigin != null) {
    let d = createVector(mouseX - circleOrigin[0], mouseY - circleOrigin[1]).mag()
    graphic.ellipseMode(CENTER)
    graphic.ellipse(circleOrigin[0], circleOrigin[1], d * 2, d * 2)
    captureCircleDimensions(circleOrigin[0], circleOrigin[1], d * 2, d * 2, 5)
    circleOrigin = null
  }
}
```



```
function captureRectangleCoordinates(x,y,width,height,strokeWeight)
{
  coords = {
    'x': x,
    'y': y,
    'width': width,
    'height': height,
    'strokeWeight': strokeWeight
  }
  if(socket.connected)
    socket.emit('rectangleCoordinates', coords)
}
```

Code walkthrough - onMouseQuit is where we draw the shape (on that client's system), and we capture the coordinates in captureRectangleCoordinates and emit it to the server. We only send the essential information (x - Point of x in the canvas, y - Point of y in the canvas, width - Width of the rectangle, height - Height of the rectangle, strokeWeight - Strength of the stroke) needed to recreate this rectangle on the other clients.



```
socket.on('rectangleCoordinates', (coords) => {
    const {x, y, width, height, strokeWeight} = coords
    const rectangleObject = {'shape':'Rectangle',
                             'x':x,
                             'y':y,
                             'width':width,
                             'height':height,
                             'strokeWeight':strokeWeight}
    history.push(rectangleObject)
    io.sockets.emit('drawRectangle',coords);
});
```

Code walkthrough - On the server, when this event is triggered, it captures the coordinates and then emits it to all the other clients which will then get drawn on their canvas.

Note - I have taken a little of the front end code from codepen

Docker Service

I have deployed the cloud collaborative drawing application on Docker. To access the application, the following steps have to be followed -

1. Download the code folder (https://github.com/iamkavinarasu/CloudWhiteboardCW2/tree/master)
2. Navigate to the downloaded folder via the command prompt
3. docker-compose build
4. docker-compose up
5. Once the application is up, you can access the application on your web browser by typing - ‘localhost:8081’