Peer-To-Playlist: A Friendship Matching App Based on Music Taste

Shahir Ahmed[1], Megan Triplett[2]

Department of Mathematics and Computer Science
Dickinson College Carlisle, PA
[1]`ahmeds@dickinson.edu`
[2]`tripletm@dickinson.edu`

I. INTRODUCTION

In an increasingly online world, it has become more difficult for peers to connect with one another over common interests. With this application, we hope to bring people together over shared musical interests. Despite its name after a play on Peer-to-Peer model, our application follows a client-server model.

The user logs in with a username and password, which we authenticate in our database. There, the user can choose to like or pass on other users' music tastes based on a matching algorithm. If two users like one another, they match and can chat by sharing files, writing texts, or by sharing emojis.

On the server side, chats and accounts are stored in a MongoDB database. A user shares a chat, it moves through our server into the database, and the server sends it to the specified other user.

II. SYSTEM ARCHITECTURE

Our system (fig 1) lies largely in the application layer, and communicates with the transport layer through the socket (fig 2). We treat the network, link, and physical layers as an abstraction, and do not manipulate their normal function in our application. We use a TCP connection, and create our server with the "node:http" module, and the Express and socket.io frameworks. Our client side uses socket.io to create a socket to connect with the server, and sends POST messages through the axios library.

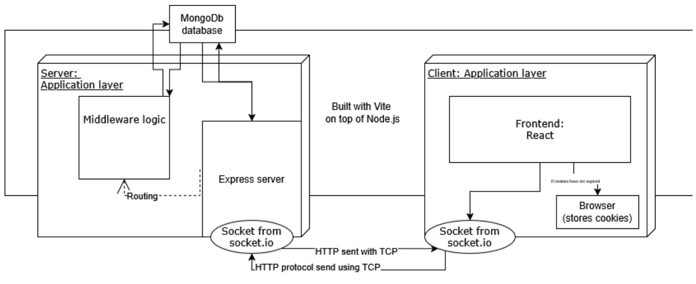


Fig. 1 The entire system architecture

A. Build

For building our application, we used the build tool Vite which consists of two parts: a dev server and a build command that bundles your code with Rollup https://vite.dev/guide/ on top of Node.js which is a free, open-source, cross-platform JavaScript runtime environment [9].

B. Frontend

For the frontend we use React, which is a framework that helps build UI out of components [7]. This benefits us both on the server side and the client side. On the server, React allows you to simultaneously stream HTML as you're fetching data. And on the client side, it keeps the UI responsive, even in the middle of rendering, with its use of standard Web APIs.

C. Server Backend

To ensure reliable transportation of data and because our app is not required to be as fast as possible, we use a TCP connection. This is achieved beginning with the HTTP server on the server side, aptly named httpServer using the "node:http" module. Then, we initialize a socket using Socket.io and, from there, specify a port on which to listen.

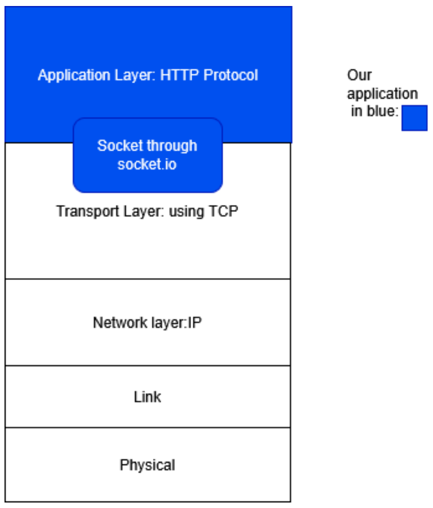

Application Layer: HTTP Protocol
Socket through socket.io
Transport Layer: using TCP
Network layer:IP
Link
Physical
Our application in blue: 🟦

Fig. 2 TCP/IP Diagram

From there, Express, a web application framework for routing and middleware is used to bind our Authorization, User, Message, and Match routes to an instance of the app object using the .use() function [2] Express handles the routing system which defines the routes and the functions that run when the routes are hit: the route handlers. These Authorization, User, Message, and Match routes files route to application-level middleware functions, which do things such as authorize the user, update the user's profile, send and display received and sent messages, and inform users of a match.

For example, in messageRoutes.js, the function getConversation is converted to a http get response with the line:

```
router.get("/conversation/:userId",
getConversation);
```

This function, getConversation, is a route handler, as it is responsible for processing an incoming request – to fetch the relevant messages – and for sending back the response – a "200 Okay" along with the relevant messages if successful, a "404 No messages found" if no messages are found, and a "500 Internal Server Error" if not. Between processing the request and creating the response, the route handle often performs some logic, called middleware: functions that run in the middle of the request-response cycle [8].

Thus, we see that Express is the framework used to handle the core HTTP server functionality and allows us to dictate our responses to different client requests.

As we mentioned earlier, socket.io is used to initialize the socket on the server side. Socket.IO manages the WebSocket protocol [8] and the real-time event-driven communication [3]. Because of socket.io, we can send realtime notifications when a user makes a match and we can allow users to send and receive messages in real time.

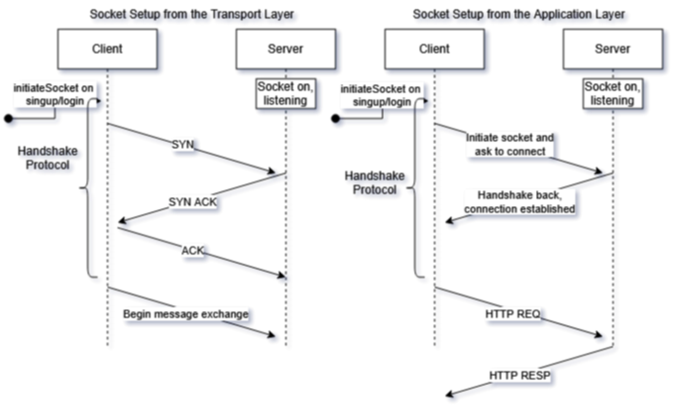
When client sockets connect to the server socket, they are stored in the Map connectedUsers. JavaScript is naturally asynchronous (a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events [5]), so multiple clients are automatically handled concurrently. Thus, we do not need to initiate a new socket on the server side for each client socket [6].

Within initializing listening, we also connect to our database, which is configured through and hosted on MongoDb. Files, both user profiles and file messages between users, are converted to a URL through Cloudinary. Storing information within requests from the client onto the database is handled through the aforementioned middleware, as is fetching necessary information.

D. Socket.io on the Client Side

The client initiates the TCP socket when they login/signup/authenticate and then communicate (see Fig 3). If a socket is already initiated from that client, it is disconnected and a new one is initiated. Once it is initiated, it is made known to the server side, and the socket on the server side handshakes with it. As we have already discusses, JavaScript is naturally asynchronous, so for our purposes, we do not have to create a new socket on the server side for this client socket.

Additionally, we do not have to loop for this TCP socket connection to remain open. This is because Node's callback architecture, by default, loops. Instead, the user side must explicitly end the connection, rather than explicitly keeping it open [6].



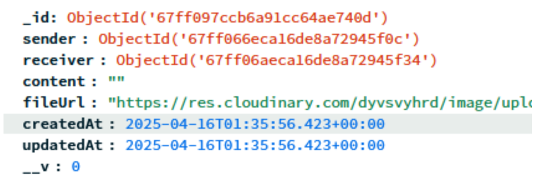
Socket Setup from the Transport Layer
Client
Server
initiateSocket on singup/login
Socket on, listening
Handshake Protocol
SYN
SYN ACK
ACK
Begin message exchange
Socket Setup from the Application Layer
Client
Server
initiateSocket on singup/login
Socket on, listening
Handshake Protocol
Initiate socket and ask to connect
Handshake back, connection established
HTTP REQ
HTTP RESP

Fig. 3 Timing Diagram of Initializing the Client Socket

Client HTTP requests are handled through the axios library on top of Node.js, which constructs a POST message based on the user's interaction with the browser components.

D. Messages between Two Users (Fig 6)

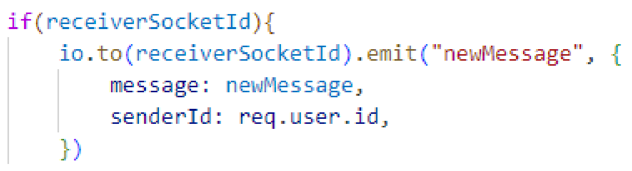
Once a user has matched with another user, they can chat and/or exchange files. Despite the name playing off the Peer-to-Peer model, this message exchange occurs from client to the intermediary server, then to the other client. Once a client has uploaded a file or written a message, axios creates an HTTP POST message, which is sends to the server. This HTTP message is then routed to messageRoutes.js, which calls middleware function getConversation from our messageController.js. File exchange is achieved by creating a Message object defined by Message.js on the server side by creating a Cloudinary link for each file. Within this .create() method, each message, now either a string or a url, is stored in our MongoDb database (see Fig 5).



```
_id: ObjectId('67ff097ccb6a91cc64ae740d')
sender : ObjectId('67ff066eca16de8a72945f0c')
receiver : ObjectId('67ff06aeca16de8a72945f34')
content : ""
fileUrl : "https://res.cloudinary.com/dyvsvyhrd/image/upl
createdAt : 2025-04-16T01:35:56.423+00:00
updatedAt : 2025-04-16T01:35:56.423+00:00
__v : 0
```

Fig. 4 An example of a file in our database

Then, the server emits the Message to the receiver's socket, as in Fig 5.



```
if(receiverSocketId){
    io.to(receiverSocketId).emit("newMessage", {
        message: newMessage,
        senderId: req.user.id,
    })
```

Fig. 5 The server emitting a message to the receiver's socket

And the message is sent along with the HTTP Response 201 to the original sender.

The getConversation function then realizes that the database has updated, and will display the sent message in future loads of this chat.

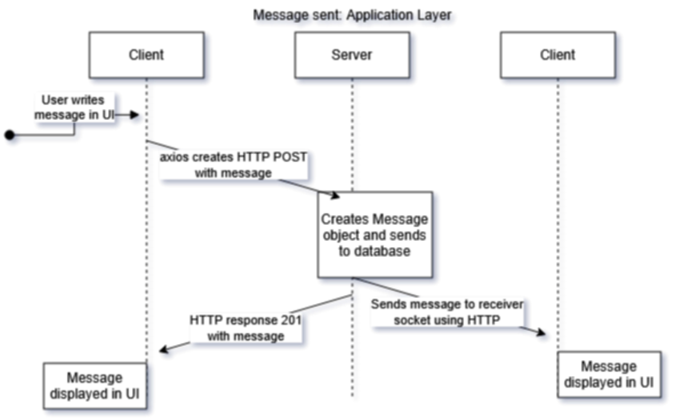

Message sent: Application Layer
Client
Server
Client
User writes message in UI
axios creates HTTP POST with message
Creates Message object and sends to database
HTTP response 201 with message
Sends message to receiver socket using HTTP
Message displayed in UI
Message displayed in UI

Fig. 6 The message diagram of a message exchange between users

III. SYSTEM IMPLEMENTATION

Our application is written mostly in JavaScript, with some HTML and CSS for styling purposes.

For our application to communicate over the Web, we have built our own API, which holds all of our server-side files. This API handles server-side socket creation, all routing and middleware, the two models for Users and Messages, and the configuration for Cloudinary and connecting to MongoDb. It also contains the seeds to randomly generate users when our database is reset. In the future, we hope to connect to Spotify's robust API to match users based on their taste in artists, albums, and genre.

Our application also stores cookies on a user's browser, allowing them to stay logged in upon reload or close. Our cookies are implemented using protectedRoute middleware (in api/middleware/auth.js) to check if a JWT (JSON Web Token) token is in the user's cookies, verify the token, and if valid, attach the user to req.user and return.

Additionally, our application supports all browsers: Firefox, Chrome, Safari, Edge, or Internet Explorer if you're feeling old school.

The application can be executed on the client side by following our deploy link:

https://peer-to-playlist.onrender.com/auth

or it can be executed entirely on one device by running

```
npm install
npm run dev
```

in the ./Peer-to-Playlist directory. And, in another terminal, running:

```
npm install
npm run dev
```

in ./Peer-to-Playlist/client.

IV. CONCLUSIONS

Whether it's the connection of a client socket to our server socket or a user to another user, our application emphasizes the connections that we hope to facilitate. Our use of frontend frameworks Node.js, Vite, and React ensures seamless integration with our backend, built with Express, Socket.io, MongoDb, and Cloudinary. This makes our performance excellent. Pages load quickly and matches are notified in real-time. Chats between users appear to the receiver just after they are sent.

In the future, we would like to expand our application by allowing users to log in to their Spotify to create their profile. Additionally, we could implement multi-person chatrooms based on shared music interests, rather than the current one-on-one chats. Currently, our system allows a user to select a gender and to specify a gender preference, and our gender selections only include the traditional male/female binary. In the future, we would like to expand this choice to allow for diverse gender expressions.

REFERENCES

(1) Basques, K., LePage, P., & Steiner , T. (n.d.). *Read files in JavaScript | Articles*. Web.dev; Chrome. https://web.dev/articles/read-files

(2) Bird, A. (2012, July 4). *NodeJS / Express: what is "app.use"?* Stack Overflow. https://stackoverflow.com/questions/11321635/nodejs-express-what-is-app-use.

(3) Codesistency. (2024, October 9). *Build a Full Stack Tinder Clone - Node, Express, Tailwind - Full Tutorial 2024*. YouTube. https://www.youtube.com/watch?v=o-XOBJRNeqk

(4) Getting Started. (2019). Vitejs. https://vite.dev/guide/

(5) *Introducing asynchronous JavaScript - Learn web development | MDN*. (2024, December 24). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Async_JS/Introducing

(6) JavaScript Socket Programming Examples. (2022). Lmu.edu. https://cs.lmu.edu/~ray/notes/jsnetexamples/

(7) Meta Open Source. (2025). *React*. React.dev. https://react.dev/

(8) Emadamerho-Atori N., Software Engineering Blogger. (2024, November 15). *The Good and the Bad of Express.js Web Framework*. AltexSoft. https://www.altexsoft.com/blog/expressjs-pros-and-cons/

(9) Node.js. (2023). Node.js. Node.js. https://nodejs.org/en