Online Chat

Anthony Tung

Abstract:

This project consists of a practical demonstration of peer-to-peer networking and how it can be applied to create a simple yet powerful web chat platform. My implementation began by studying the existing technology, then choosing a relevant technology (WebRTC), and then finally settling on a framework (PeerJS) and preparing a Javascript code example. While my example currently only supports text chat and image based file sharing, it does so while supporting an arbitrary amount of concurrent users. Additionally, the base work is laid for implementing voice and video streaming in the future.

Introduction:

Online communications platform using peer-to-peer networking removes the need of a server. Clients' communications are then made direct between one another. My goal was to do this exact thing; create a simple online peer-to-peer environment that would allow users to communicate with a secure and reliable connection. Some issues I planned to address from the beginning with our implementation included security with end to end encryption for the web app, I also planned to optimize our code and ensure a reliable connection; nobody wants to be disconnected from their friends while chatting.

Technical section:

Online chat is a broad term for any kind of communication over the internet that offers a real-time transmission of text messages from a sender to a receiver. There are different approaches to designing an online chat, with multiple factors to consider; such as which forms of communication will be supported, which forms of networking will be used, and how the users will interact with the application. My project demonstrates group text messaging and file sharing, through a web-based interface with peer-to-peer networking over Secure Real-time Transfer Protocol (SRTP). WebRTC uses a specific version of SRTP known as DTLS-SRTP to add AES encryption of data packets, as well as additional features such as message authentication/validation and replay attack protection. (Mellen, 2018)

Peer-to-peer is particularly useful way to implement an online chat, since it is serverless, each node acts as its own host, with a connection to all of its peers. Distributed Denial of Service (DDoS) attacks are largely prevented, as a result of having a direct connection between machines. Since peer-to-peer does not rely on a server to host connection, if an attacker targets one machine the others will not be affected. My implementation uses WebRTC as a base, which includes excellent foundations for what I desire.

ICQ was the first major web chat client. ICQ was originally released in November 1996, and was made available to anyone with a computer and internet connection. In 2001, ICQ had over 100 million users registered out of the 361 million internet users in 2001 (Knight, 2018). In this model the server is the heavy lifter because the

server relays all communications between connected machines and allows for an efficient connection between users. The bad thing about this model was that it is susceptible to DDoS attacks which in turn affects all users.

Some modern examples of online chats are Discord and Skype. Discord was built for gamers and allows for communications between users, whether it is one on one or within a large group in a channel. Skype was made with one goal in mind, webcam sharing. This allowed friends, family, or even business partners to talk to one another not only with just audio but with the ability see each other. Being able to do this made communication more enjoyable if, for example, someone has not seen a friend or family member in a long time. Also on the business end it allowed for remote meetings and/or presentations. Both these applications run on a client/server network simply because they host such a large number of users. My idea was to create a different chat system using a peer-to-peer network. I decided to do this using a pre-existing library called PeerJS.

The PeerJS API is built using WebRTC and allows implementation of media calling, media and file sharing, text communications, and it is easy to connect to peers using user identifiers. PeerJS provides a wrapper for the browser's existing WebRTC implementation, providing us the developers with a slightly simplified API. Instead of complicated identifiers, a peer can create a P2P data or media stream connection to a remote peer by simply sending an ID to a STUN server. PeerJS allow us to preconnect to clients faster because PeerJS provides binary data support. Each person that wants to share data just needs to provide an identifier which is randomly generated and give the identifier to people or peers so they can connect to the same server. "The PeerJS service deals with WebRTC handshake and handles NAT traversals for the users. PeerJS brokers connections by connecting to PeerServer. Users can choose to run their own PeerServer or use a free version hosted by PeerJS. This hosted server can be accessed programmatically using an indirect API via a JavaScript SDK. " (Pwtempuser, 2018).

WebRTC, which is what is being used for my project is open web project, currently being standardized by W3C, that allows real time communication on a browser known as web app. WebRTC has the fundamental blocks for high quality communication on the web and provides numerous powerful tools through its API. The framework WebRTC uses include HTML, HTTP, and TCP/IP, and some known ones include Session Traversal Utilities for Nat (STUN) and Traversal Using Relay around NAT (TURN). A key factor in the success of the web is that its core technologies – such as HTML, HTTP, and TCP/IP – are open and freely implementable. There was no standardized, high-quality, complete solution available to enable real time communication in the browser, until WebRTC enabled it. As for STUN, "a host uses Session Traversal Utilities for NAT (STUN) to discover its public IP address when it is located behind a NAT/Firewall. When this host wants to receive an incoming connection from another party, it provides this public IP address as a possible location where it can receive a connection. If the NAT/Firewall still won't allow the two hosts to connect directly, they make a connection to a server using TURN, which will relay media between the two parties" (Twilio Docs, 2019)

Routing mechanism allow WebRTC client to reduce load from the uplink by spreading media across participants, although it requires asymmetric bandwidth from participants. Routing mechanism is able to reduce load to the server because it does not require it to decode, layout and then re encode media. Although the disadvantage was that it is extremely hard to implement to code.

For my demonstration we used the aforementioned utilities to create our own peer-to-peer online chat. PeerJS was the underlying foundation for our code as we used much of what is already in that library. While PeerJS provides a lot of the heavy lifting, implementing their functions was not a walk in the park. One challenge I found was how to give each user a customizable name. By default each user is provide 16 character long identifier- for example: sitcv04y4in00000 and while these identifiers are unique they are not convenient. Eventually I was able to implement the ability to create a custom username to display whenever that user would send a message. While PeerJS allows for transmission of many different types of communications- audio and video calling, text, picture and video sharing, file sharing there was simply not enough time to fully implement every possibility.

Text was the first thing I was able to get functional, which did not present too many problems. PeerJS is primarily built around connections between two users, so if we only have client1 and client2, there are no issues. However, when client3 joins, there needs to be a 1-3 connection and a 2-3 connection. As more users join, this number of total connections grows exponentially. As a result, our implementation makes each user store a collection of all their connections, which can then be accessed and manipulated as users join and leave. This collection serves as a pool used when sending messages, and also as a member directory used to inform new members of which peers to connect to. In this case, each connection is a data connection allowing for arbitrary data transmission. For audio/video streaming, media connections would be required as well, increasing the total number of connections required by two.

Next I implemented picture/gif sharing. This created problems that I did not envision. One problem we encountered here was that, when pictures would send they would always say they were sent by the viewer and not the sender. So if client1 was viewing a picture client3 sent, it would appear to client1 as if it was sent by him/herself. This created a big problem cause there was no way for the user to know where the picture came from. After looking back at my code I was able to realize where I went wrong and were eventually able to figure this problem out.

Another obvious difficulty I found was designing the interface for user appeal; whilst I mainly designed it for functionality, I had a rough time moving certain things around to make more sense of the UI. Also once I included picture sharing I could not find a good way to resize the picture so they would display nicely to every user. Given more time I could have made the UI more appealing, but since my goal was a functional proof of concept, I decided not to worry much about how the UI looked.

Once the deadline approached for my presentation I decided to stop where I was and refine what I had rather than implementing more things. Had I kept going I could have ran into problems that would not allow me to demonstrate my work to my classmates. That being said, the groundwork for implementing audio/and video calling is there, as well as file sharing. I was close to getting file sharing to work, it is just that I could not find a clean way to let users download files that were sent in chat with our limited UI.

Conclusions:

My implementation with PeerJS allowed me to build a secure online chat solution, that allows users to do media/file sharing and text communications, and makes it easy to connect to peers using user identifiers. This was a success, with multiple successful demos using many simultaneous users. Any transmission delay is imperceptible, and in my experience all communications were reliable and without issue. I learned a lot about the inner workings of online chat systems and how they work all the way from the beginning. I learned about routing and how different online chat system reduce loads to allow more simultaneous users, the prime example of which would be Discord.

There were also a lot of hardships along the way when building the web app, which included even getting multiple user chat to work. Accurate records of all connections need to be maintained so that no users are left out of the conversation. My original design broke easily because of many unfamiliar quirks and errors with the framework. Even handling user disconnects or displaying images in the message window was difficult. Another hardship was creating the user interface, because I was not familiar with how to present dynamic text/image messages in a page that works across mobile and desktop, especially when each message has its own timestamp and sender name. Some features took longer than expected or were hard to implement, just because this was new technology with a new API.

In the end I had a fun time learning how some common modern applications work on the backend. Working through the hardships and learning how to work through the PeerJS library was a quite a task that allowed me to appreciate the intricacies involved with modern peer to peer implementations.

References:

"Architecture." WebRTC, webrtc.org/architecture/.
"Routing." WebRTC Glossary, 6 Jan. 2017, webrtcglossary.com/routing/.

Bidgoli, Hossein (2004). The Internet Encyclopedia. John Wiley & Sons. pp. 665–. ISBN 9780471222040. Retrieved 25 April 2019.

"Frequent Questions." WebRTC, webrtc.org/faq/#what-is-webrtc.

Hogg, Scott. "What About Stream Control Transmission Protocol (SCTP)?" Network World, Network World, 30 Apr. 2012, www.networkworld.com/article/2222277/what-about-stream-control-transmission-protocol--sctp--.html.

Jmcker. "Jmcker/Peer-to-Peer-Cue-System." GitHub, 21 Jan. 2019, github.com/jmcker/Peer-to-Peer-Cue-System.

Knight, Shawn. "What Ever Happened to ICQ?" TechSpot, TechSpot, 23 Dec. 2018, www.techspot.com/article/1771-icq/.

Mellen, Allie. "Explaining the Secure Real-Time Transport Protocol (SRTP)." Callstats, www.callstats.io/blog/2018/05/16/a-explaining-the-secure-real-time-transport-protocol-srtp.

"PeerJS Docs." PeerJS Documentation, peerjs.com/docs.html#start.

Pwtempuser. "PeerJS." ProgrammableWeb, 8 Aug. 2018, www.programmableweb.com/api/peerjs.

"Taming WebRTC with PeerJS: Making a Simple P2P Web Game." Toptal Engineering Blog, www.toptal.com/webrtc/taming-webrtc-with-peerjs.

"Twilio Docs: API Reference, Tutorials, and Integration." Twilio. 3 Feb. 2019, www.twilio.com/docs/stun-turn/faq.

Vass, Jozsef. "How Discord Handles Two and Half Million Concurrent Voice Users Using WebRTC." Discord Blog, Discord Blog, 10 Sept. 2018, blog.discordapp.com/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc-ce01c3187429.