Decentralized WebRCT P2P network using Kademlia

Ryle Zhou
Stanford Univerisity
rylezhou@stanford.edu

Abstract—**Web Real-Time Communication (WebRTC) is a new standard and industry effort that extends the web browsing model. For the first time, browsers are able to directly exchange real-time media with other browsers in a peer-to-peer fashion. Before WebRTC was introduced, it was cumbersome to build smooth chat and video applications, users often experience unstable connection, blurry videos and unclear sound. WebRTC's peer-to-peer communication paradigm establishes real-time connection between browsers using SIP(Session Initiation Protocol) Trapezoid. A wide set of protocols are bundled in WebRTC API, such as, connection management, encoding/decoding negotiation, media control, selection and control, firewall and NAT element traversal, etc. However, almost all current WebRTC applications are using centralized signaling infrastructure which brings the problems of scalability, stability, fault-tolerance. In this paper, I am presenting a decentralized architecture by introducing Kademlia network into WebRTC to reduce the need of a centralized signalling service for WebRTC.**

Index Terms—**WebRTC, P2P, Kademlia Network, Distributed Systems**

I. INTRODUCTION

In 2020, COVID-19 has changed all of our lives. Video conferencing software are becoming essential to us. Because of the pandemic, cloud video conferencing services have never been more popular. With hundreds of millions of people working from home, the need of having smooth, high-quality voice and video chat systems that can support a large group of people staying online together at the same time has increased significantly. In this paper, I primarily explore the current progress with WebRTC and briefly talk about the combination of WebRTC and Kademlia Network with the purpose of having a decentralized way for WebRTC Peers to establish connections.

II. FUNDAMENTALS

A. WebRTC and its problem

1) WebRTC Infrastructure: WebRTC [2] stands for Web Real-Time Communications. For the first time, browsers are able to directly exchange real-time media with other browsers in a peer-to-peer fashion. WebRTC leverages a set of plugin-free APIs that can be used in both desktop and mobile browsers, and is progressively becoming supported by all major modern browser vendors. Previously, external plugins were required in order to achieve similar functionality as is offered by WebRTC. It includes a stack of protocols combined together(see Fig 1). In order to establish a peer-to-peer communication, web browsers must agree to begin communication, know how to locate one another, bypass security and firewall protections, and transmit all multimedia communications in real-time. In the web application scenario, the server can embed some JavaScript code in the HTML page it sends back to the client. Such code can interact with browsers through standard JavaScript APIs and with users through the user interface.
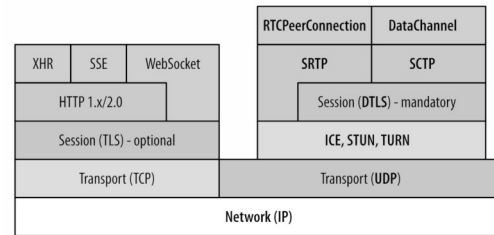

Fig. 1. WebRTC protocol stack

In WebRTC, both browsers are running a web application, which is downloaded from a different web server. Signaling messages are used to set up and terminate communications. They are transported by the HTTP or WebSocket protocol via web servers that can modify, translate, or manage them as needed. However, signalling is not implemented within WebRTC and it is a part that was left for developers to build based on different use cases(see Fig 2).
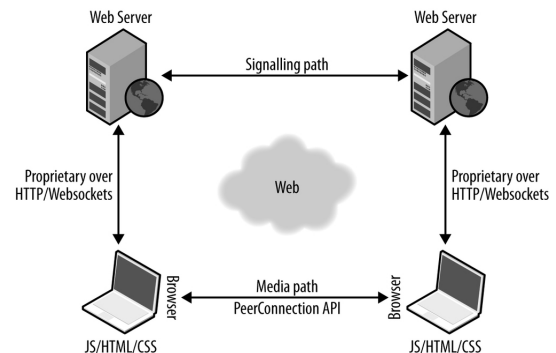

Fig. 2. WebRTC Infrastructure

2) Data Channel: The *DataChannel* API is designed to provide a generic transport service allowing web browsers to exchange generic data in a bidirectional peer-to-peer fashion. The encapsulation of SCTP over DTLS over UDP together

with ICE provides a NAT traversal solution, as well as confidentiality, source authentication, and integrity protected transfers. Moreover, this solution allows the data transport to interwork smoothly with the parallel media transports, and both can potentially also share a single transport-layer port number. SCTP has been chosen since it natively supports multiple streams with either reliable or partially reliable delivery modes. It provides the possibility of opening several independent streams within an SCTP association towards a peering SCTP endpoint. Each stream actually represents a uni-directional logical channel providing the notion of in-sequence delivery. A message sequence can be sent either ordered or unordered. The message delivery order is preserved only for all ordered messages sent on the same stream. However, the *DataChannel* API has been designed to be bidirectional, which means that each *DataChannel* is composed as a bundle of an incoming and an outgoing SCTP stream.

The *DataChannel* setup is carried out (i.e., the SCTP association is created) when the CreateDataChannel() function is called for the first time on an instantiated PeerConnection object. Each subsequent call to the *CreateDataChannel()* function just creates a new *DataChannel* within the existing SCTP association.

3) Peer Connection: The peer-to-peer connectivity is handled by the *RTCPeerConnection* interface. A *RTCPeerConnection* allows two users to communicate directly, browser to browser. It then represents an association with a remote peer, which is usually another instance of the same JavaScript application running at the remote end. Communications are coordinated via a signaling channel provided by scripting code in the page via the web server, e.g., using XMLHttpRequest or WebSocket. Once a peer connection is established, media streams can be sent directly to the remote browser. Media capture devices includes video cameras and microphones, but also screen capturing "devices". For cameras and microphones, the *navigator.mediaDevices.getUserMedia()* handles the *MediaStreams*. For example, Peer 1 creates an offer and initiates signalling with STUN/TURN server, and as a result, Peer 1 receives the ICE candidates. Both the offer and the ICE candidates are sent to peer 2 through the signaling channel. As soon as Peer 2 receives the offer, it creates an answer and performs the same process and sends its connectivity information through the same signaling channel back to Peer 1. After the signaling is complete, both peers have all connectivity information: answer, offer and both ICE candidates.

4) STUN AND TURN: The Session Traversal Utilities for NAT (STUN) protocol (RFC5389) allows a host application to discover the presence of a network address translator on the network, and in such a case to obtain the allocated public IP and port tuple for the current connection. To do so, the protocol requires assistance from a configured, third-party STUN server that must reside on the public network.

The Traversal Using Relays around NAT (TURN) protocol (RFC5766) allows a host behind a NAT to obtain a public IP address and port from a relay server residing on the public Internet. Thanks to the relayed transport address, the host can then receive media from any peer that can send packets to the public Internet.

The *RTCPeerConnection* mechanism uses the ICE protocol together with the STUN and TURN servers to let UDP-based media streams traverse NAT boxes and firewalls. ICE allows the browsers to discover enough information about the topology of the network where they are deployed to find the best exploitable communication path. Using ICE also provides a security measure, as it prevents untrusted web pages and applications from sending data to hosts that are not expecting to receive them.

Each signaling message is fed into the receiving *RTCPeerConnection* upon arrival. The APIs send signaling messages that most applications will treat as opaque blobs, but which must be transferred securely and efficiently to the other peer by the web application via the web server.

5) The problem with ICE, STUN AND TURN: STUN servers live on the public internet and have one simple task: check the IP:port address of an incoming request (from an application running behind a NAT) and send that address back as a response. In other words, the application uses a STUN server to discover its IP + port from a public perspective. This process enables a WebRTC peer to get a publicly accessible address for itself, and then pass that on to another peer via a signaling mechanism, in order to set up a direct link(see Fig 3).
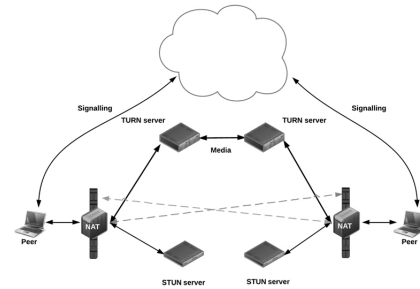

Fig. 3. WebRTC Peer Connection

RTCPeerConnection tries to set up direct communication between peers over UDP. If that fails, RTCPeerConnection resorts to TCP. If that fails, TURN servers can be used as a fallback, relaying data between endpoints. TURN servers have public addresses, so they can be contacted by peers even if the peers are behind firewalls or proxies. WebRTC-based P2P networks always require two components PeerConnction: A signaling mechanism to exchange the offer/answer messages and the ICE candidates, and a STUN/TURN server to establish connections from/to peers behind NATs. As a result, WebRTC is behind centralized distributed systems that might be signal point of failures, downtime, instability and other unreliable problems.

B. Kademlia Network

Kademlia [1] is a distributed hash table for decentralized P2P networks. It specifies the structure of the network and

the exchange of information through node lookups. Kademlia nodes communicate among themselves using UDP. A virtual or overlay network is formed by the participant nodes. Each node is identified by a number or node ID. The node ID serves not only as identification, but the Kademlia algorithm uses the node ID to locate values (usually file hashes or keywords). In fact, the node ID provides a direct map to file hashes and that node stores information on where to obtain the file or resource. In Kademlia, XOR metric is introduced to define a distance between two nodes. The XOR distance is the bitwise exclusive OR on the peers' identifiers interpreted as an integer(see Fig 4).
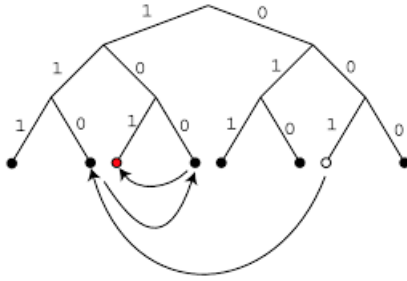

Fig. 4. Kadenlia XOR Metric

When searching for some value, the algorithm needs to know the associated key and explores the network in several steps. Each step will find nodes that are closer to the key until the contacted node returns the value or no more closer nodes are found. This is very efficient: like many other DHTs, Kademlia contacts only $O(\log(n))$ nodes during the search out of a total of n nodes in the system. For example, if there are 10,000,000 Kademlia nodes, only about 20 hops would be necessary at most for communication with any subset of nodes. Kademlia based networks are highly resistant to denial of service attacks and the loss of a group of nodes as the protocol simply routes around the unavailable nodes.

Further advantages are found particularly in the decentralized structure, which increases the resistance against a denial-of-service attack. Even if a whole set of nodes is flooded, this will have limited effect on network availability, since the network will recover itself by knitting the network around these "holes".

Another advantage of Kademlia is that the protocol naturally prefers long-lived nodes over newer entrants. The process of joining a Kademlia network requires discovery of only one peer, whereby the node then broadcasts its appearance. The initiator then collects the NodeID from each response and adds it to its own peer table.

1) Kademlia NodeID: Kademlia treats each node on a network as a leaf on a binary tree. Generally, each Kademlia node has a 160-bit NodeID (SHA-1), and its position is determined by the shortest unique prefix of its ID. To assign key-value pairs to particular nodes, Kademlia relies on a notion of distance between two identifiers. Given two 160-bit identifiers, x and y, Kademlia defines the distance between

them as the XOR. From a node point of view, the tree is divided into series of successive sub-trees where the 160th subtree contains the individual node. The Kademlia protocol ensures that each node knows of at least one node on each of its sub-trees. With this guarantee, a node can locate any other node by its ID.

2) Routing Table K-buckets: The routing table is a binary tree whose leaves are k-buckets. The structure of the Kademlia routing table is such that nodes maintain detailed knowledge of the address space closest to them, and exponentially decreasing knowledge of more distant address space. The symmetry is useful since it means that each of these closest contacts will be maintaining detailed knowledge of a similar part of the address space, rather than a remote part. K-buckets are a list of routing addresses of other nodes in the network, which are maintained by each node and contain the IP address, port, and NodeID for peer participants in the system. They prefer the longest-lived nodes, which means that one cannot overtake a node's routing state by flooding the system with new nodes. The routing table size is asymptotically bounded by O(log(n/k)) where n is the actual number of nodes in the network and k is the bucket size, so larger bucket implementations slightly reduce the total number of buckets in the routing table.

The Kademlia protocol consists of four Remote Procedure Calls :
PING: probes a node to see if it's online.
STORE: instructs a node to store a key-value pair.
FIND_NODE: returns information about the k nodes closest to the target ID.
FIND_VALUE: similar to the FIND_NODE , but if the recipient has received a STORE for the given key, it just returns the stored value.

III. Decentralized WebRTC with Kademlia

From the above section, I found that the ICE, STUN/TURN part of WebRTC is used to deal with NAT traversal in a centralized way. In this section, I bring up the notion that connecting peers in WebRTC through a Kademlia network. Although, the actual network might still be in a server network owned by some companies. However, using Kademila can create this WebRTC service in a decentralized way. To join the network, local Peer A make a connection through WebSockets to a node(system configuration, this node is defined as a long-lived node) in Kademlia network and sends a FIND_NODE request that returns the k closest node to the remote Peer B. Here, the node nominated in Kademlia network acts as the signalling channel. The implementation of Kademlia in AWS was actually chanllenging to me. I encountered many bugs and a lot of details could go wrong. If implementatd correctly, STUN/TURN server can be replaced by the Kademlia network. And theoretically, through iterative routing, the time to find the target node will be fast. To evaluate this model, I thought about a few metrics to measure:

- RTCConnection time for each call session to establish measured for 100 times

- Failure Rate (here failure means no connection was established)
- Call Session time that had no dropped connection

IV. CONCLUSION AND FUTURE WORK

In this paper, I explored WebRTC work process, Kademlia Network and the combination of the two technology in order to create a decentralized chat service in a distributed system. Because of the centralization nature of the of ICE, STUN/TURN server and signaling channel, I thought it would be a problem for to have a reliable tool for people. The purpose is to reduce the risk of single point of failure and establish a more reliable and available system. There are a lot of technical challenges involved such as, deploying Kademlia network on cloud, signalling configuration and the front end implementation etc. Here, I did not explore how the security of the network, which is a big concern among the public right now.

My future work includes continue figuring out how well Kademlia can work with WebRTC, possibly using a load balancer to further test scalability. The possibility of extending WebRTC to mobile devices is also worth taking into consideration.

REFERENCES

[1] Maymounkov, P., and D. Mazières. "Kademilia: A Peer-to-peer Information System Based on XOR Metric, 1st Intl." Workshop on Peer-to-Peer Systems. 2002.
[2] Real-time communication for the web. https://webrtc.org/