U-P2P in the browser Architecture

# Overview

This document is largely written to be an informal guide to the current architecture and is included here along with each version of the application with updates along the journey (mainly to make our lives easier when the final project report is due). This document describes the implementation details of the redesigned U-P2P application to accomplish the goal of migrating the it to a browser based application. This typically would involve redesigning or reimagining certain components of the application. The network interface is one of the main components that is expected to change a significant deal. The change will mainly involve the way connections are established and the connection type.

## Task organization

No task organization software is currently being used.

## Integration

No integration software is currently being used.

## Development Language

The language used to develop U-P2P in the browser is JavaScript. This language can only be used as a single event-driven thread which means that the application will only be able to perform one action at any given time as opposed to a multithreading capable language which can run several actions at once. This will mostly likely be the bottleneck of the browser implementation. The effort will most likely have to be made ensuring the application is as efficient and as concise as possible to enable high reliability and avoid the possible race conditions associated with an event driven application. Another possible approach is a defensive implementation with several error checks (try catch statements) and loops to ensure all required conditions are met when performing actions. It is likely that a mixture of these two software engineering paradigms will be used throughout the development life of this application.

ECMAScript is the name of the standardized JavaScript language, a mixture of ECMAScript 5 and ECMAScript 6 features will be used while developing this application.

## Development IDE

An IDE is not currently in use for the development of this application. A simple text editor is currently being used.

## Development JavaScript framework

An JavaScript framework is not currently in use for the development of this application. Pure JavaScript is currently being used.

## APIs used

The following APIs are currently used by this application:

* PubNub 4.0 API is used to interface the signaling service currently used. The API is located at <https://cdn.pubnub.com/sdk/javascript/pubnub.4.0.11.js>.
* RTCPeerConnection API is used to establish peer to peer connections between browser clients using the WebRtc protocol (this uses two other protocols ICE and SDP, both are briefly discussed later). The implementation and the exact name of the API differs from one browser to the other. This discussed in more detail in the following section.

## Code location

The code is located in the repository: <https://github.com/michaelfakhri/BasicWebRtcUsingPubNub>

## Browser support

Currently the supported browser(s) is / are:

* Chrome

This limitation comes due to the nature of the RTCPeerConnection API implementation differences embarked during the early stages of development. For example, the chrome API is called webkitRTCPeerConnection while the firefox API is called RTCPeerConnection with the standard specifying the name to be RTCPeerConnection.

There is a future task to add support for the following browser(s) which support the webRtc protocols through an open source API adapter (<https://github.com/webrtc/adapter>):

* Firefox

## Current Progress

The minimum viable product (MVP) is complete. The MVP uses a hardcoded username approach to establish a connection between two browser clients since peer discovery has not been designed or implemented yet.

This consists of the following features:

* subscribe to the PubNub signaling service.
* Publish and receive messages using the signaling service.
* Use the signaling to exchange connection metadata and establish a peer-to-peer connection between two browses.
* Exchange text messages between the peers using the peer connection.
* Lay the foundation for future development by encapsulating into four main objects / prototypes described below:
  + SignalingService – interface used to communicate with the signaling service chosen in order to exchange connection metadata with other peers and also used for peer discovery.
  + PeerConnection – object used to establish the connection, store the connection details and send messages to a peer. This object can directly contact the signaling service.
  + PeerHandler – object responsible for peer creation and storage. Currently it is not fully implemented but is used by the one peer that the application currently supports.
  + MessageRouter – router used to route incoming messages through the signaling service to the right peer by using the peer Handler to get the peer and then forwarding it to the correct function through the use of public scope function.

# Network interface core component

This consists of the four components mentioned earlier: the signaling service, the peer connection object, the peer handler and the message router.

## Signaling service

The signaling service object is simply a generic interface to the one currently being used is called PubNub. PubNub uses a publish / subscribe model to transfer its message between users. The free package (<https://www.pubnub.com/pricing/>) the following:

* A maximum of 100 connections from different IP addresses daily.
* A maximum of 1 million messages across the network monthly.

Upon instantiation, the constructor establishes connection to the signaling service. A listener is also added that directs incoming messages to the message router. The interface also provides the following function(s):

* generateAndSendMessage – this function takes in the parameters that make up the message structure (discussed in the following subsection).

### Structure of messages sent over the signaling service

The message sent is JSON object that is made up of the following parameters:

* origin – The identity of the originating user from which this message was sent.
* destination – The identity of the target user to which this message is intended.
* msgType – The type of this message which in turn dictates how it is going to be processed (for now there are three types only).
* msg – The message contents (in this case, connection metadata).

### Message Types

There are 3 types of messages currently supported:

* offer (RTCSessionDescription): The offer is the first step in establishing a connection using the WebRTC API created by a user (the offering user) that wants to establish a connection with another user (the receiving user).
* Answer (RTCSessionDescription): The second step is the receiving user processing the offer and creating an answer (this happens only if the receiving user is willing to accept the connection). The answer is then sent to the offering user.
* ice\_candidate (RTCIceCandidate): The third step occurs after the exchange of the offer and the answer, both users use the ice framework to gather ice candidates which are sent.

RTCSessionDescription and RTCIceCandidate are discussed in more detail in the following sections.

#### Peer Discovery (not implemented yet)

Currently peer discovery is a topic that is still up for discussion. As far as I can tell there seems to be two methods this can be used to do this without server side coding. These two are mentioned briefly below:

* I am connected now: A broadcast telling everyone who is connected and listening that I (a new user) am connected to the network. This method generates the least traffic but you have no control on who decides to connect or not. You could end up waiting forever for a peer to contact you.
* Who is connected now: A broadcast asking to know who are the users connected to the signaling service and are willing to connect right now. This method will generate the most traffic, possibly flooding the network

One of the most critical factors in browser based peer to peer applications is avoiding the flooding of the signaling service. Therefore, the more likely choice is a pulsing “I am connected now” message until a few exit conditions (number of pulses, number of offers received or an estimate of how many people were listening) are met.

Currently 8 messages are required in order to establish a connection without the use of peer discovery implementation, whether it is a pulsing message, single message without response or single message with multiple responses. It is possible to reduce this number to just 2 messages but will have to use a few webRTC and JavaScript hacks in order to achieve this. I believe, there is no simple way this can be done using just the API.

## Peer connection

The peer connection object encapsulates the WebRtc API in order to provide a unified object with simple methods that is responsible for connection establishment. The following instance variables are available through the peer connection object:

* Channel – The peer to peer data channel used to send data or text over to the other peer directly.
* RTCpeerConnection – The accessor to the WebRtc object created using the API.

The following methods are also included with the peer connection object:

* createOffer – Used to create the offering user’s RTCSessionDescription (the offer) and send it to the receiving user.
* processOffer – Used to create the receiving user’s RTCSessionDescription (the answer) from the offering user’s “offer”. The answer is then sent to the offering user.
* processAnswer – Used by the offering user to handle the answer received from the receiving user.
* processIceCandidate – used by both parties (the offering and receiving users) to process the RTCIceCandidate.

### RTCPeerConnection

The RTCPeerConnection is the main API object that is used to handle the connection details such as creating the offer, the answer and the ice candidate. In simple words it is the connection broker to the other side. It uses the following classes.

#### RTCSessionDescription

This is a wrapper to the session description protocol [1] or simply a text based connection metadata (includes stuff like encryption and requested services). It is used to negotiate a connection with the other side by specifying the requested data channels, audio or video services needed.

#### RTCIceCandidate

This is a wrapper to the ICE protocol [2] or simply a NAT traversal technique (STUN - Session Traversal Utilities for NAT) along with a fallback for server relay for any data (TURN - Traversal Using Relay NAT). These two techniques are used to establish a pathway between the two peers. In our case, it tries the local IP address and if that does not work, it uses the external NAT IP address. If both do not work it uses the TURN server if one is provided.

## Message router

The message router is currently used only to forward incoming messages through the signaling service and route them to the correct peer connection function handler. It currently has only one function associated with the object:

* handleSignalingMessage: This function takes in the source of the message, the message type and the message itself and forwards it to the correct peer and function by using public scope functions (one for each message type).

## Peer handler (not implemented fully yet)

The peer handler is used to create and get peer connections from some kind of map implementation. It has two methods shown below:

* createPeer – creates a peer connection object.
* getPeer – gets an already created peer connection object.

# Interesting / Relevant resources

## https://www.ipify.org/

* <http://javascript.info/tutorial/events-and-timing-depth>
* <http://altitudelabs.com/blog/what-is-the-javascript-event-loop/>

## JavaScript’s weird (craziest unimaginable thing I have seen so far) implementation of “this” operator / pointer

* <http://www.quirksmode.org/js/this.html>
* <http://unschooled.org/2012/03/understanding-javascript-this/>

## WEBRTC adapter API for different browsers

* <https://github.com/webrtc/adapter>

## Find out your IP address using JavaScript

* <https://www.ipify.org/>

# References

[1] <https://tools.ietf.org/html/rfc4566>

[2] <https://tools.ietf.org/search/rfc5245>