STREAMROOT-webRTCTest

Peer to Peer Application:

1. Why is it efficient:

* When a user wishes to download a file from website, we submit a HTTP GET request.
* This request for file uses a single TCP socket, and communicates with a single server which transfers entire file.
* P2P protocol creates TCP connections with multiple hosts and makes many small data requests to each.
* The P2P client then combines chunks to recreate file.
* A single file host will usually have limited upload capacity.
* But connecting to many servers simultaneously allows for higher file transfers, and disperses costs associated with data transfer amongst many peers.
* Moreover, a client mid-way through downloading file also acts as a server, hosting bits to others which they have already downloaded.
* These differences from traditional HTTP GET requests allow for lower costs and higher redundancy since many people are sharing the files.

1. Fault Tolerance:

* Peer to Peer technologies can be utilized for fault tolerant decentralized logical networks.
* These logical networks can and could support any type of data, such as Internet DNS information, scientific information, or large distributed and redundant information databases.
* Peer to Peer networks represent a valuable technological tool for preservation of human knowledge.
* P2P networks have been receiving increasing demand from users and are now accepted as standard way of distributing information, because its architecture enables scalability, efficiency and performance as key concepts.
* Peer to Peer network is decentralized, self-organized and dynamic in its purse sense and offers an alternative to traditional client-server model of computing.
* P2P networks are almost unlimited in their scalability.
* In pure P2P systems, every node acts as a server and client and they share resources without any centralized control.
* Most of P2P applications have some degree of centralization.
* These are called “hybrid” P2P networks and they centralize at least list of users.
* This is how instant messengers or file sharing programs work- the system keeps a list of user with their IP addresses.
* Different applications of P2P networks enable users to share the:
  + Computation power (Distributed Systems)
  + Data (File sharing)
  + Bandwidth (File sharing)
* P2P uses an individual’s computer power, resources, instead of powerful centralized servers.
* The shared resources guarantee high availability among peers.
* P2P is a really important area to research, because it has huge potential in distributed computing.
* The Key things of P2P network is to achieve reliability, efficiency, scalability and portability.

Questions:

1. Create a Peer to Peer application that allows several peers to send message to each:

* Peer-to-Peer computing or networking is a distributed application architecture that partitions tasks or workloads between peers.
* Peers are equally privileged, equipotent participants in the application.
* They said to form a peer-to-peer network of nodes.

1. package.json:

{

“name”: “Text”,

“version”: “0.0.1”,

“description”: “socket program”

“dependencies”:

{

“express”:” ^4.10.2”,

“socket.io”:” ^1.4.6”

}

}

1. main.js:

var ws = require(‘express’) ();

var http = require(‘http’).Server(ws);

var io = require(‘socket.io’) (http);

ws.get(‘/’, function(request,response)

{

Response.sendfile(‘main.html);

});

io.on(‘connection’ , function(socket)

{ socket.on(‘text message’ , function(msg)

{

io.emit(‘text message’, msg);

});

});

http.listen(8080, function()

{

console.log (‘running on\*:8080);

});

1. main.html:

<!doctype html>

<html>

<head>

<title>Text Message</title>

<style>

\* { margin: 0; padding: 0; box-sizing: border-box; }

body { font: 13px Helvetica, Arial; }

form { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }

form input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }

form button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }

#messages { list-style-type: none; margin: 0; padding: 0; }

#messages li { padding: 5px 10px; }

#messages li:nth-child(odd) { background: #eee; }

</style>

</head>

<body>

<ul id="messages"></ul>

<form action="">

<input id="m" autocomplete="off" /><button>Send</button>

</form>

<script src="/socket.io/socket.io.js"></script>

<script src="http://code.jquery.com/jquery-1.11.1.js"></script>

<script>

var socket = io();

$('form').submit(function(){

socket.emit('text message', $('#m').val());

$('#m').val('');

return false;

});

socket.on('text message', function(msg){

$('#messages').append($('<li>').text(msg));

});

</script>

</body>

</html>

Implantation:

* install node.js
* populating the dependencies:
  + npm install –save [express@4.10.2](mailto:express@4.10.2)
* during development, socket.io serves the client automatically for us:
  + npm install –save socket.io (This installs the module and the dependency to the package.json)
* start the server
  + node main.js
* open a browser and go for <http://localhost:8080/>

Result:

This architecture creates a P2P connection between the clients (Browser tabs) but for each time when never the client needs to transfer the data, first the data transmits to the server and then server transmits the data to every peers (clients) connected to it.

THE PROBLEM: Low Latency Client-Server and Server-Client Connections for high range of data.

This kind architecture is every good when the data is very less but it’s not suitable for high data transmission and the server should always put on in the background.

Pros and Cons(WebSocket):

Pros:

1. Delivers Full Duplex Communication Model for the Web.
2. Increased Client and Server Communication Efficiency.
3. Avoid Siloed Solutions for the Individual Subsystems.
4. Easy to use API.
5. Open Full Duplex Connection over the Web.

Cons:

Some of the operational issues like Network topology, Firewalls, Kernel configurations, Load testing, Security, Monitoring, Scaling, Redundancy, Load Balancing, Replication.

WEB SOCKETS:

* The Web Socket specification defines an API establishing “socket” connections between a web browser and a server.
* There is a persistent connection between the client and the server and both parties can start sending data at any time.
* The WebSocket constructor accepts one required and one optional parameter:

WebSocket WebSocket (in DOMString url, in optional DOMString[ ] protocols);

* The URL to which to connect; this should be the URL to which the WebSocket server will respond.
* Either a single protocol string or an array of protocol strings These strings are used to indicate sub-protocols, so that a single server can implement multiple WebSocket sub-protocols.
* To Transmit data to the server over the WebSocket connection:

void send (in DOMString data);

void send (in ArrayBuffer data);

void send (in Blob data);

* Data, a text string to send the server.
* To close the WebSocket connections:

void close (in optional unsigned short code, in optional DOMString reason);

Proxy Servers:

* In the case of WebSocket it is the compatibility with proxy servers which mediate HTTP connections in most company networks.
* The WebSocket protocol uses the HTTP upgrade system to “upgrade” an HTTP connection to a WebSocket connection.
* If a given client uses the WebSocket protocol, it may not be possible to establish a connection.

Server:

* Using WebSocket creates a whole new usage pattern for server side applications.
* While traditional server stacks such as LAMP are designed around HTTP request/response cycle they often do not deal well with a large number of open WebSocket connections.
* For the large number of connections open at same time requires an architecture that receives high concurrency at a low performance cost.
* Such architectures are usually designed around either threading or so called non-blocking IO.
* Implementation:
  + Node.js:
    - Socket.IO
    - WebSocket-Node
    - ws

WEBRTC

* RTC has been corporate and complex, requiring expensive audio and video technologies to be lined or developed in house.
* Integrating RTC technology with existing content, data and services has been difficult and time consuming, particularly on the web.
* WebRTC has now implemented open standards for real-time, plugin-free video, audio and data communication.
* The guiding principles of WebRTC project are that its API’s should be open source, free, standardized, built into Web browsers and more efficient than existing technology.
* WebRTC is used in various applications like WhatsApp, Facebook Messenger, and platform such as TokBox.
* There is even an experimental WebRTC enabled IOS Browser named “Brower”.
* WebRTC has also been integrated with “WebKitGTK+” and “Qt” native apps.
* WebRTC implements 3APIs:
  + MediaStream (aka getUserMedia)
  + RTCPeerConnection
  + RTCDataChannel

RTCPeerConnection:

* It is the WebRTC component that handles stable and efficient communication of streaming data between peers.
* The codecs and protocols used by WebRTC do a huge amount of work to make real-time communication possible, even over unreliable networks:
  + Packet loss concealment
  + Echo cancellation
  + Bandwidth adaptivity
  + Dynamic jitter buffering
  + Automatic gain control
  + Noise reduction and suppression
  + Image cleaning.
* RTCPeerConnection without Servers:
  + By considering code which has local and remote RTCPeerConnection on one web page.
  + The caller and callee are on the same page doesn’t constitute anything very useful but it does make the working of RTCPeerConnection API a little clearer, since these objects on the page can exchange data and messages directly without having to use intermediary signalling mechanisms.
* Caller:

Create a new RTCPeerConnection and add stream from getUserMedia()

pc1 = new webkitRTCConnection(servers); //pc1-> localpeer

pc1.addStream(localStream);

* Create an offer and set it as local description for pc1 and as remote description for pc2. Since both Caller and Callee are on same page, this can be done directly in code without using signalling.

pc1.createOffer(getDescription1);

function gotDecription1(desc)

{

pc1.setLocalDescription(desc);

trace(“Offer from pc1 \n”+desc.sdp);

/\*

// sdp-> session description protocol: It is a format for describing streaming media // initialization parameters.

\*/

pc2.setRemoteDescription(desc);

pc2.createAnswer(gotDescription);

}

* Callee:

Create pc2 and when stream from pc1 is added, display it in video element:

pc2 = new webkitRTCPeerConnection(servers);

pc2.onaddstream = gotRemoteStream;

function gotRemoteStream(e)

{

Vid2.src = URL.createObjectURL(e.stram);

}

* RTCPeerConnection plus Servers:
* In real-world, WebRTC need server, however simple:

1.Users discover each other and exchange ‘real-world’ details such as names.

2.WebRTC Client applications(peer) exchange network information.

3.Peers exchange data about media such as video format and resolution.

4.WebRTC client applications traverse NAT gateways and firewalls.

* WebRTC needs 4 types of server-side functionality:

1.User discovery and communication

2.Signaling

3.NAT/Firewall traversal

4.Relay servers in case peer to per communication fails.

* NAT traversal, peer to peer networking and requirements for building a server app for user discovery and signalling are beyond the scope of this article.
* The STUN (Session Traversal Utilities for NAT) protocol and its extension TURN (Traversal Using Relays around NAT) are used by ICE (Internet Communication Engine) framework to enable RTCPeerConnection to cope with NAT traversal and other network vagaries.
* ICE is a framework for connecting peers, such a two video chat clients.
* Initially, ICE tries to connect peers directly, with lowest possible latency, via UDP.
* In this process, STUN servers have a single task: to enable a peer behind a NAT to find out its public address and port.
* If UDP fails, ICE tries TCP: first HTTP, then HTTPS.
* If direct connection fails because of enterprise NAT traversal and firewalls, ICE uses an intermediary(relay) TURN servers.
* ICE first uses STUN with UDP to directly connect peers and if that fails, will fall back to a TURN relay server.

RTCDataChannel:

* WebRTC supports real-time communication for other types of data.
* The RTCDataChannel API enables peer to peer exchange of arbitrary data, with low latency and high throughput.
* Many potential use cases for API, including;
  + Gaming
  + Remote desktop applications
  + Real-time text chat
  + File transfer
  + Decentralized networks.
* The API has several features to make most of RTCPeerConnection and enable powerful and flexible peer to peer communication:
  + Leveraging of RTCPeerConnection session setup.
  + Multiple simultaneous channels with prioritization.
  + Reliable and unreliable delivery semantics.
  + Built-in security(DTLS) and congestion control.
  + Ability to use with or without audio or video.
* The syntax is deliberately similar to WebSocket, with a send () method and a message event:

var pc = new webkitRTCPeerConnection(servers, {optional:[{RtpDataChannels:true}]);

pc.ondatachannel = function(event)

{

Document.querySelector(“div#receive”).innerHTML=event.date;

};

};

sendChannel = pc.createDataChannel(“sendDataChannel”,{reliable:false});

document.querySelector(“button#send”).onClick=function()

{

var data = document.querySelector(“textarea#send”).value;

sendChannel.send(data);

};

* Communication occurs directly between browsers, so RTCDataChannel can be much faster than WebSocket even if a relay(TURN) server is required when ‘hole punching’ to cope with firewalls and NATs fails.
* Hole Punching:
  + It is a technique in networking for establishing a direct connection between two parties in which one or both behind firewall or behind routers that use NAT.

SECURITY:

* There are several ways a RTC application or plugin might compromise security, for example:
  + Unencrypted media or data might be intercepted on route between browsers or between a browser and server.
  + An application might record and distribute video or audio without user knowing.
  + Malware or viruses might be installed alongside an apparently innocuous plugin or application.
* WebRTC has several features to avoid these problems:
  + WebRTC implementations use secure protocols such as DTLS (Datagram Transport Layer Security) and SRTP (Secure Real Time Protocol).
  + Encryption is mandatory for all WebRTC components, including signalling mechanism.
  + WebRTC is not a plugin: its components run in browsers sandbox and not in separate process, components do not require separate installation and are updated whenever browser is updated.
  + Camera and microphone access must be granted explicitly and when camera or microphone are running, this is clearly shown by user interface.

Datagram Transport Layer Security (DTLS):

* + DTLS communication protocol provides communications security for datagram protocols. (Basic transfer unit associated with a packet switched network)
  + DTLS allows datagram-based applications to communicate in way that is designed to prevent eavesdropping, tampering and message forgery.
  + DTLS protocol is based on the stream-oriented Transport Layer Security (TLS) protocol and is intended to provide similar security guarantees.

Secure Real Time Protocol(SRTP):

* + It defines a profile of RTP, intended to provide encryption, message authentication and integrity and replay protection to RTP data in both unicast and multicast applications.