Addressing Challenges in Browser Based P2P Content Sharing Framework Using WebRTC

Shikhar Vashishth[1] Yash Sinha[2], K Hari Babu[3]
Department of Computer Science and Information Systems
Birla Institute of Technology and Science
Pilani, India.
{f2012436[1], f2012365[2], khari[3]}@pilani.bits-pilani.ac.in

***Abstract*— Most of the content sharing applications use the client/server model in which all of group managements are done by the server and this sometimes becomes a communication bottleneck. Installing specialized software for different purposes such as file sharing, video conferencing etc., becomes a barrier for the user. Recent technologies like NodeJs and Socket.io have fostered new ideas the ways web browsers can be used. Moreover, the emerging standards of WebRTC open up new paradigm of direct communication channel between web browsers without relaying the data through a web server. But there are certain issues such as lack of full-fledged threading/concurrency support in the JavaScript language, reliance on synchronous loading etc. that restricts modern day browsers to take full advantage of current multiprocessing capabilities. Although, on one hand there are advantages of using web browsers, such as no requirement of specialized software, benefits of emerging technologies etc.; the aforementioned issues pose challenges in implementation in certain areas.**

In this paper, we have tried to couple the benefits of peer-to-peer (P2P) architecture (elimination of centralized dependency, better scalability, shareability etc.) along with the advantages of recent web technologies (NodeJs, WebRTC etc.) by designing and implementing a browser based P2P content sharing framework. We have addressed the aforementioned challenges of a browser based P2P architecture by providing a mechanism to exchange messages asynchronously and facilitating new peer joins via existing peers in the network, thus reducing the dependency on bootstrap server. Our prototypical implementation demonstrates the feasibility, efficiency and scalability of this lightweight framework, on the top of which a variety of applications can be added as a layer of functionality.

Keywords— peer-to-peer, signalling, WebRTC, DHT, NodeJs

I. INTRODUCTION

With rapidly growing need and dependence of society on data sharing and communication, the number of users in networks are growing exponentially. The client-server architecture doesn't seem to be a feasible solution as much as peer-to-peer (P2P) architecture because often too many requests to the server leads to congestion and if it fails, whole network goes down. Further cost and maintenance issues are also there.

On the other hand, P2P architecture is more reliable as it allows direct communication between users thus eliminates the need of any centralized instance. Moreover, P2P is scalable because with increasing number of participants the storage capacity, computation power and bandwidth of the network also increases as the resources are shared among users.

Upcoming web technologies like NodeJs and Socket.io have fostered new innovations in usability of web browsers. The new paradigm of WebRTC has enabled a direct communication channel between web browsers without relaying data through a server. WebRTC is supported by most of the popular browsers. It gives increased security, and higher cross platform compatibility of application on multiple devices. WebRTC has been anticipated to be supported by 4.7 billion devices by 2018 [1]. It allows to build varieties of real time application for web browsers without relying on third party plugins which introduce security, compatibility and performance issues. Therefore, instead of bugging the user to download a new specialised software/plugins for each of his different needs, we have attempted to assimilate the benefits of P2P architecture implemented using recent web technologies on top of web browsers; thus helping him use an already installed software for more diverse purposes. Much work has been done that emphasises that a variety of activities and purposes which had previously required specialized software such as file sharing between peers without a server to relay the files, video and audio chat without the use of proprietary 3rd-party plugins, and multimedia conferencing without the need for proprietary, platform-dependent 3rd-party applications; can be now built with ease using WebRTC.

In this paper we have leveraged the Data channel component of WebRTC to design and implement a web browser compatible framework which will simplify the development of various applications in sharing content among users without knowing about the underlying architecture. The framework forms a structured p2p network which allows any user to search for any resource efficiently. Our framework is a modified version of Chord [2], a distributed lookup protocol for p2p network. We chose Chord because of its simplicity, provable correctness and proven performance. It has proved to be working in large scale implementations [3]. Chord operations runs in predictable time and always result in success or definitive failure. Moreover, it balances the load over entire network making it more decentralized and scalable. Chord in its original form is incompatible with browser environment which lacks in threading/concurrency support. Chord operation relies on synchronous loading which hinders browser interactivity because of availability of a single main thread. Therefore, we have redesigned Chord operation for making them compatible with the browser based environment. We have divided Chord operations into several procedures and associated a call back function with each of the procedure and also tweaked the bootstrap server for decreased dependency. We have delineated

the challenges faced during the process of implementation and proposed solutions for it.

II. Background and Related Work

Chord is a distributed lookup protocol for structured P2P networks. It is one of the most prominent, simple and effective DHT technique; however, it is dependent on synchronous calls for its successful execution therefore cannot be used in asynchronous environment [2]. Vogt et al. [4] proposes a model for building DHT based Content Distribution Network (CDN) using WebRTC Data Channels. It gives an abstract picture of how peers can join a network and can exchange messages among each other without any support from a centralized server. Zhang et al. [5] has also investigated the possibility of building a CDN service for web browsers based on centralized P2P network using flash plugin provided by Adobe. Their framework is centred around a coordinator node that holds mappings between peers and the data stored on these peer.

WebRTC-Chord is an asynchronous implementation of Chord protocol on Node Package Manager [6]. Although made for web browser, WebRTC-Chord is not as efficient because it doesn't keep bootstrapping server lightweight. The server stores all information about the peers and helps them to update their entries as the network changes dynamically over time which is not scalable. Werner et al. [7] presents a design and implementation of a browser-based secure social network application on top of a WebRTC-based p2p framework, which uses modified version of OpenChord to provide all building blocks to create a social network. ShareFest [8] is a web BitTorrent application which has been realized to enable users for peer-to-peer file sharing using WebRTC. It allows other users to download files via a URL directly from the owner's machine.

CHEWBACCA (CHord, Enhanced With Basic Algorithm Corrections and Concurrent Activation) [9] is a P2P network framework in Java using sockets based messaging which uses synchronous calls thus, is not suitable for event driven, non-blocking systems like web-browsers. Ref. [10] lays out core architecture for building browser based framework for P2P networks. Although being functional, it is not scalable as it maintains full mesh connection between all participating peers.

III. Challenges in Implementation

Here we discuss the challenges faced in course of implementation of the framework:

A. *Lack of full-fledged threading/concurrency support in the Javascript language*

JavaScript historically suffers from an important limitation: all its execution process remains inside a unique thread [11]. This JavaScript limitation implies that a long-running process freezes the main window. The user is unable to interact with the application and user experience becomes unpleasant. The user may decide to kill the tab or the browser instance. The join operation to the network requires peers to be discovered and peer to peer connections to be established which involves a number of procedures like contacting the bootstrap server, handshakes between connecting peers and several message forwarding operations depending on the network size. Moreover, sequential execution of the procedure is required because next procedure requires successful completion of the preceding procedure.

If this operation is implemented in traditional synchronous way, it is required that the thread sleeps (or waits) till the previous procedure executes successfully. There is no provision for sleep in JavaScript [12], and busy waiting, in the worst case, will freeze the main window.

Further, because the web workers operate independently of the main thread [11], they cannot access many of its objects. They cannot access the DOM, so they cannot read or modify the HTML document. In addition, they cannot access any global variables or some special objects like the window, parent and the document. Because the communication with web worker is based on messaging, sequential execution cannot be guaranteed.

B. *Dependency on bootstrap server*

In order for a WebRTC application to set up a P2P connection, its clients need to exchange information such as session control messages, error messages, media metadata, key data (for secure connections), and network data [13]. This signalling process needs a way for clients to pass messages back and forth. To avoid redundancy and to maximize compatibility with established technologies, signalling methods and protocols are not specified by WebRTC standards (as outlined by JSEP) [14]. The main challenge here is signalling servers may have to handle a lot of messages, from different locations, with high levels of concurrency. This signalling server is generally also called the bootstrap server.

In current implementations [6], bootstrap server is also actively involved in connection of new peers to the existing network, facilitation of handshakes and network stabilization. This defeats the purpose of using a P2P architecture as the bootstrap server becomes a communication bottleneck as well as a single point of failure.

C. *Using ICE to cope with NATs and firewalls: STUN and TURN servers*

In real life scenarios most devices function behind one or more layers of NAT. Some may have anti-virus software that blocks certain ports and protocols, and others may be behind proxies and corporate firewalls. A firewall and NAT can be implemented by the same device, such as a home Wi-Fi router. So ICE framework is required to overcome the complexities of real-world networking.

ICE first tries to make a connection using the host address obtained from a device's operating system and network card; if that fails (which it will for devices behind NATs) ICE obtains an external address using a STUN server, and if that fails, traffic is routed via a TURN relay server [13]. A STUN server is used to get an external network address whereas TURN servers are used to relay traffic if direct (peer to peer) connection fails. ICE servers may have to handle a lot of messages, so high levels of concurrency is required.

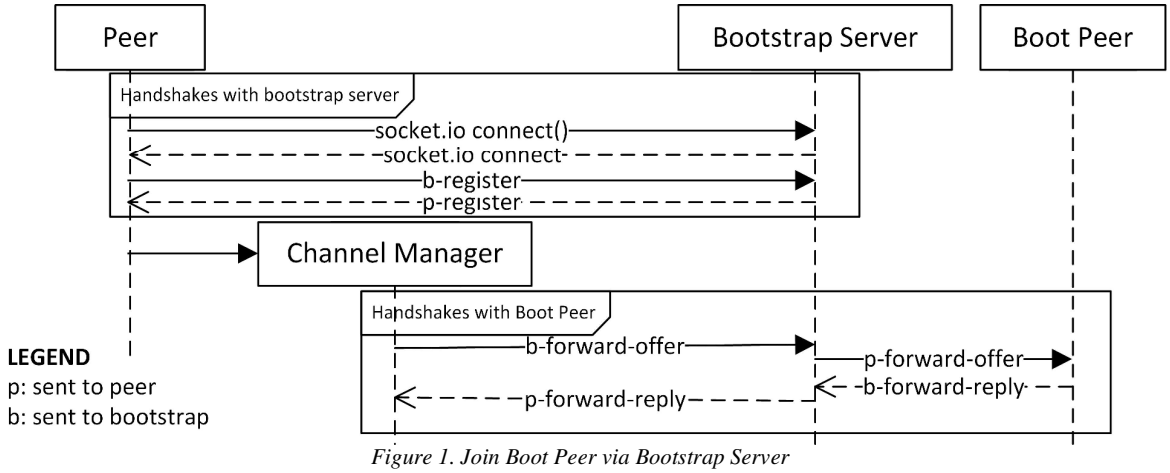

Figure 1. Join Boot Peer via Bootstrap Server

IV. Design and Implementation of Prototype

A. Components

1) Peer

A peer has two primary modules which are explained below:

a) Channel Manager

It is that module which is responsible for handling all WebRTC stack. It creates offers, accepts offers, periodically runs the stabilize operation and keeps track of open connections for a peer. Since all connection establishment in WebRTC is asynchronous, the Channel Manager also stores the state of every connection and acts appropriately on every possible state change. It is also responsible for receiving messages from other peers and processing them according to their type.

b) Node Details

It is the store house for a peer. All information such as successor, predecessor etc. is stored here. Different strategies used for connections in different scenarios access data from this module and hence referential integrity is maintained. It also contains the details about fingers in finger table, sent and received messages in response table and forward table and incomplete connections in channel table.

2) Bootstrap Server

It allows new peer to join network by assigning it a unique identifier and establishing its connection with one of the peers which is already in the network. For a new peer, that peer becomes its boot peer whom it contacts for joining the network and establishing connection with other peers. Bootstrap server has no other role than to help peer in making connection with its boot peer, this helps to keep server lightweight and makes the framework more scalable

3) Message Format

Messages exchanged between peers are encoded in JSON format. It consists of source and destination peer identifiers which are assigned by the bootstrap server. It contains a type identifier which helps to differentiate the messages. Depending on the type of message, it contains other data like signal information, result of query, callback function etc.

B. Modifications to Chord

We made the following modifications to the base Chord protocol:

1) Join Boot Peer via Bootstrap server

As shown in the Figure 1, a new peer, aspiring to connect to the network, contacts the bootstrap server, via a connection through web socket. With the function call b-register, the new peer requests a peer id and configuration information from the bootstrap server, and the server knowing in advance the ids of peers already connected to the network, sends a unique id via the p-register call. Also the information about STUN/TURN servers, size of finger table etc. are sent.

The bootstrap server then facilitates the connection formation with one of the peers (called the Boot peer) in the network.

2) Join network.

As shown in the Figure 2, the join network operation has been divided into four procedures which are asynchronous in nature. In procedure 1, the new peer attempts to form a connection with its successor facilitated by its boot peer. So, it makes a *findSuccessor* call to its boot peer and also sends its connection offer with it. The boot peer queries the network to find its successor, and then forwards the offer to the successor. The successor replies with its accepted offer to the boot peer which is forwarded to the new peer. In procedure 2, the new peer attempts to form a connection with its predecessor in a similar way. In procedures 3 and 4 it notifies its successor and predecessor to update their predecessor and successor respectively.

To decrease the joining time of the new peer to the network, the bootstrap server picks up the (to be) successor itself as the boot peer. Thus connection to successor is made by the bootstrap server itself. Therefore, the set of handshakes required to establish connection with the successor is not needed which is in contrast to the erstwhile approach; where a peer was randomly picked up from the network as the boot peer, which then facilitated the handshakes for connection with the successor. The experimental proof has been discussed in

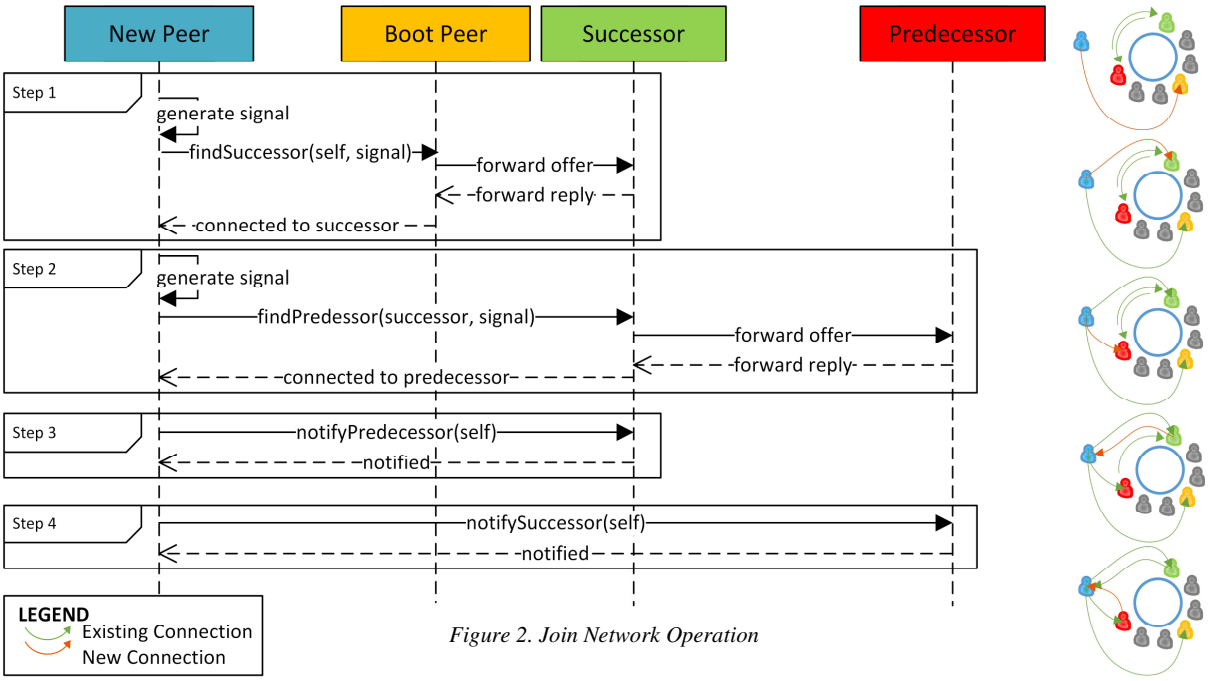

Figure 2. Join Network Operation

Section VI.B.5. It reduces load on the other peers in the network and also the requests to STUN/TURN server for ice candidates.

3) Asynchronous stabilization procedure

Similar to join operation we have modified the stabilize operation to make it asynchronous by dividing it into two procedures. In the procedure 1, the calling node queries for its successor's predecessor and attaches a callback to begin second procedure when it is completion. The second procedure notifies new successor about its new predecessor which is the calling node itself.

4) Find Successor Query Strategies

The time taken for a message to propagate in the network and the time required to generate a new connection offer have different impact on *findSuccessor* query made to the network. The propagation delay is primarily a function of network size as every message needs to be passed through peers in the network whereas the load on STUN server determines the time taken to generate a new offer. Therefore, we have designed two strategies for implementing *findSuccessor* operation which optimize the query response time based on network size and responsiveness of STUN server. Both strategies have their own pros and cons.

The first strategy involves generating connection offer and attaching it with the *findSuccessor* query. This allows to directly offer connection offer to the peer which is the successor of the queried peer id. On the other hand, the second strategy involves sending the query in the network without offer and on receiving the result the peer dynamically decides whether it needs to form connection. If connection does not exists already, another query is sent with the offer.

Strategy 1 is useful in cases when network is too large the query forward time becomes more than signal generation time and thus querying the network twice becomes expensive. It's also efficient in scenarios where many new connections are to be made for example the first call to *fixFinger* operation.

Although the former strategy requires query to be sent only once in the network; at times when the calling peer already has a connection with the successor of the queried peer id, its offer doesn't hold any significance thus its generation becomes just an overhead for stun server and the query. In this case, the second strategy performs better as it doesn't attach any offer with the query. Moreover, strategy 2 is especially beneficial for periodic fix finger operation because it involves multiple *findSuccesor* queries many of which are meant to validate the existing entries in finger table and ensure the connections are up; as discussed in Section VI.B.3. Only a few new connections are required to be made. We have discussed the experimental results in Section VI.B.4.

V. PROPOSED SOLUTIONS TO THE CHALLENGES

A. *Division of join network operation into asynchronous procedures*

We divided the join network operation into separate procedures that can be called asynchronously. Sequential execution is guaranteed with the help of the response table, which stores the results of preceding procedures based on message ids and *function* parameter of the procedures that indicates which procedure is to be called next once this procedure executes successfully.

B. *Reducing dependency on bootstrap server*

We make the bootstrap server weakly involved by facilitating handshakes between new peer and the network via the boot peer and other peers in the network. The bootstrap server is only involved in assigning a new, unique id to a new peer and connecting it with a peer from the network. If this peer is chosen randomly, it balances the load on peers to facilitate

handshakes but this requires more messages to be exchanged between peers. Therefore, we have chosen the (to be) successor itself as the boot peer.

C. Addressing load on STUN and TURN servers

We use multiple STUN servers to balance load of ICE candidate requests on them. We use TURN server to route traffic via relay server. Further, periodically we run a check for existing connections which are not required, and destroy them.

VI. Evaluation on LAN

In this section, we have tested the framework for functionality, scalability tried to prove the experimental correctness of the claims that we have made above.

A. Experimental Setup

To evaluate our framework, we use a total of 60 hosts. On one of the hosts we run both the bootstrap server and a peer. The other hosts act as peers and connect to the network with the help of bootstrap server. The hosts are interconnected using Gigabit Ethernet (GigE) using multiport 100Mbps Layer 2 switches. We use local STUN servers [15]. For 4) and 5), we assume the STUN server response time to be at least 1000 milliseconds and propagation delay between each pair of peer to be 50 milliseconds, to resemble the emulation to real life scenario.

B. Results

1) Time to join network

Currently, the existing WebRTC implementations are neither feature complete nor do the performance characteristics match the finalized product [16]. Due to the low-delay LAN environment, the measured delays almost solely mirror the actual delay introduced by this framework and the WebRTC stack. As shown in Figure 3, even for large networks which have as many as 60 peers, the time taken to join the network is less than 1000 milliseconds in our prototypical implementation. This includes propagation delays of messages transmitted, time taken to generate connection offer, requests sent to resolve bootstrap server and STUN server and time taken by STUN server to resolve peer. This shows that the framework is quite scalable.

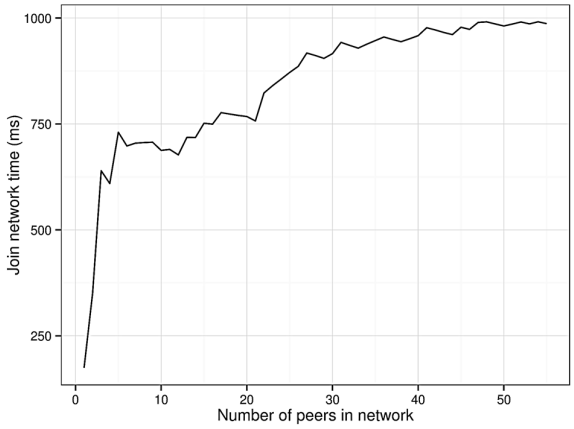

Figure 3. Time to Join Network

2) Performance of join network operation

To offer best user experience, the framework must quickly connect to the network and perform its crucial operations, so that applications can run on the top of it without much delay.

Figure 4 shows the number of messages exchanged between peers in join network operation as function of network size in two different scenarios: with and without fingers. Clearly, with finger table entries, number of messages exchanged reduces logarithmically in contrast to linear fashion in case of without finger table.

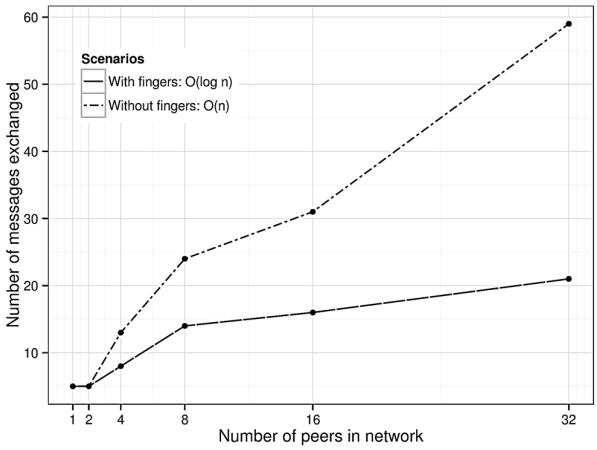

Figure 4. Performance of join network operation

3) Number of message exchanges in fix fingers

Although the network functions better with finger table entries, there is an overhead involved to maintain it. Periodically, every peer calls the *fixFinger* operation to become aware of recent new peer joins and leaves and thus, update connections. Figure 5 shows that as the network grows, more messages are required to be exchanged to fix fingers.

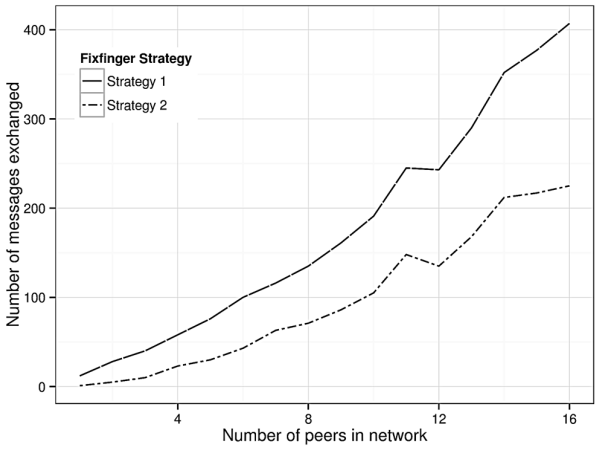

Figure 5. Comparing strategies for periodic fix fingers operation

Periodic calls to *fixFinger* operation performs better with strategy 2 as fewer connections are required to be established. Strategy 1 perform poorly because it generates a connection offer with every query most of which get discarded later on.

4) Comparing findSuccessor strategies: tradeoff between signal generation time and query propagation time

Figure 6 and 7 prove our claims of pros and cons of the two strategies used to implement *findSuccessor* operation in different scenarios. When network is small, performance of both strategy 1 and strategy 2 are almost same because message forwarding time is negligible in comparison to signal generation time. The number of messages to be exchanged is also comparable.

As the network grows in size, the message forwarding time becomes much more considerable than signal generation time. Therefore, strategy 2 that sends queries twice in the network performs poorly.

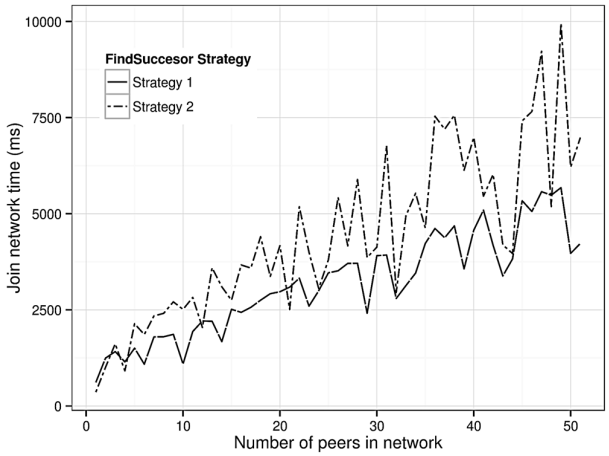

Figure 6. Comparing find successor strategies (Join Time)

The sudden improvements in strategy 2 performance can attributed to the scenarios where the peer already has connection with the result of *findSuccesor* query and thus, the second connection query need not be sent.

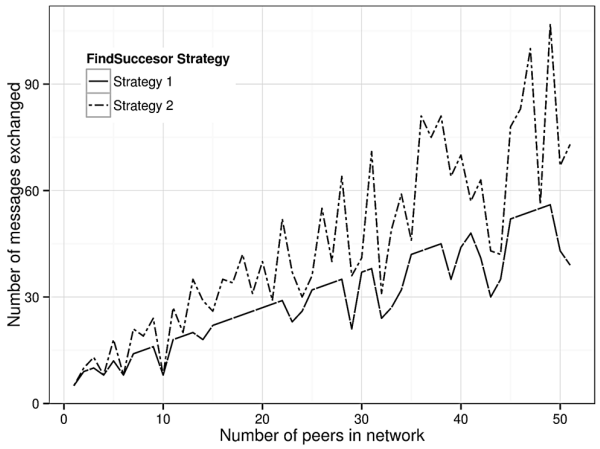

Figure 7. Comparing find successor strategies (Messages exchanged)

Another reason why strategy 1 performs better is because when a new peer joining the network is less likely to have a connection with other peers. Consequently, it is less likely that the connection offer will be discarded. Therefore, it is more efficient to include connection offer in the query itself (strategy 1) rather than sending another query for forming the connection after the first query (strategy 2).

5) Recommending successor as Boot Peer vs random selection

Figure 8 shows how join time of new peers is reduced, thus performance of network is improved, if the bootstrap server recommends the (to be) successor of the new peer as its boot peer. This behaviour has been explained in Section IV.B.2. To resemble the emulation to real life scenario, aforementioned assumptions hold true here also.

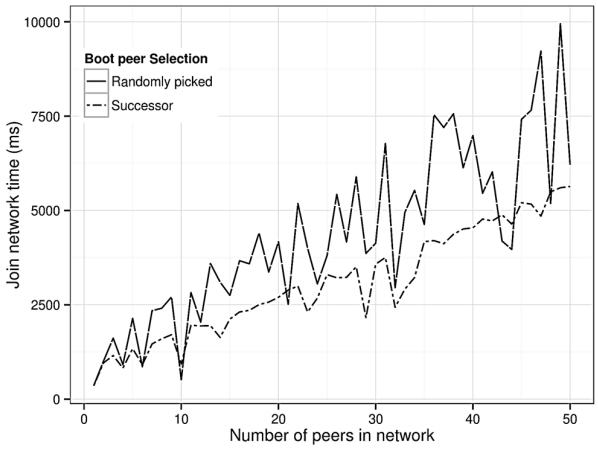

Figure 8. Strategies of recommending Boot peer

VII. Evaluation on Internet

A. Experimental Setup

We evaluate our framework on global scale using Amazon Web Services and Google STUN [17] servers. We deploy a total of thirty peers on eight Amazon EC2 [18] micro instances at different location across the globe (Figure. 9). On one of the instances at Singapore, we run the bootstrap server and the other instances at different locations act as peers and connect to the network randomly with the help of bootstrap server. On each instance multiple we run multiple peers. To check the correctness of framework we also destroyed few peers randomly in between the experiment.

Figure 9. Peer Locations across the globe

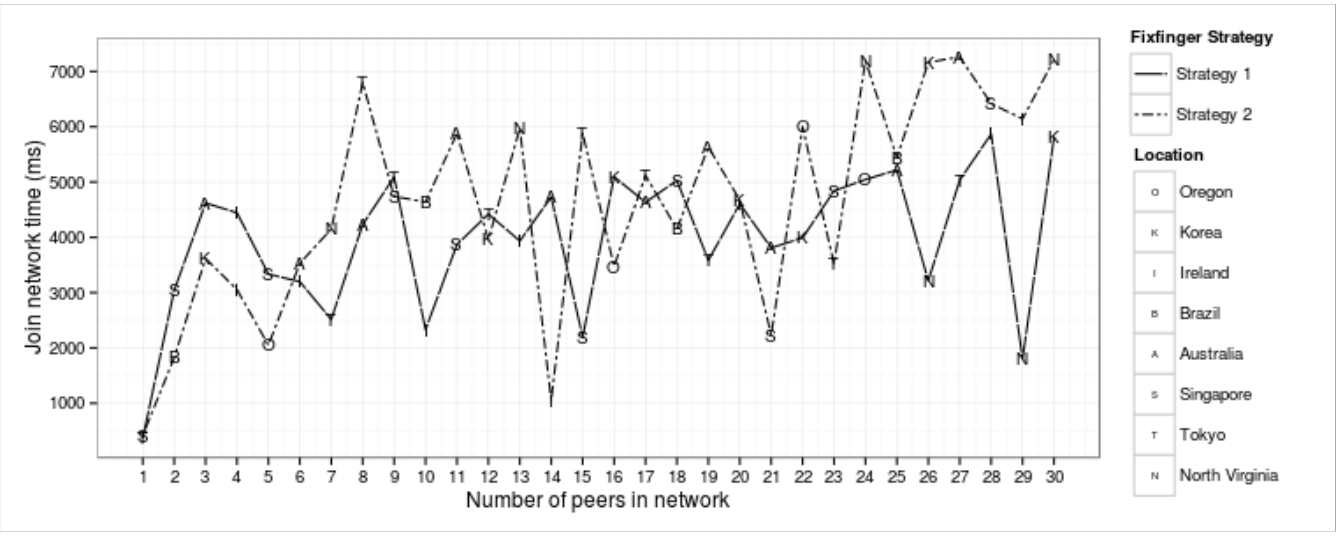

Figure 10. Comparing find successor strategies on Internet

B. Results

1) Comparing find successor strategies on Internet

As shown in Figure 10, the trends between strategies is similar to LAN but the difference is inflated from 1000 milliseconds to 4000 milliseconds approximately.

2) Join Network Time in LAN and Internet

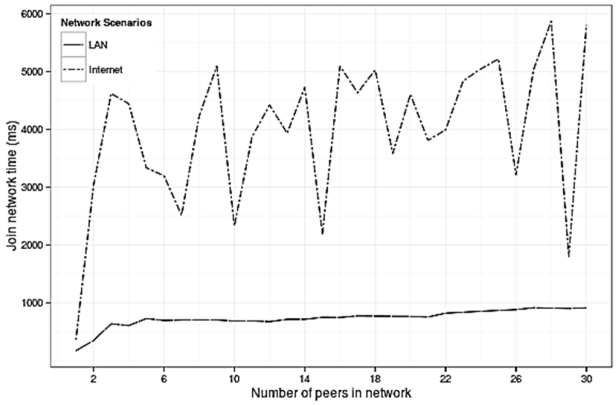

Figure 11. Comparing Join Network Time in Network Scenarios

The difference in join network time on LAN and Internet owes to multiple reasons: time required to contact STUN servers, packet propagation time, signal generation time, other traffic in network etc.

VIII. FUTURE SCOPE

Because JavaScript engines have only a single thread, asynchronous events are forced to be queued for execution. Queueing in JavaScript varies from browser to browser. Since the network is dynamic, messages may be delivered out of order to a peer and this can lead to errors.

Security issues and privacy concerns of users have not been investigated.

IX. CONCLUSION

We have tried to leverage the benefits of recent technologies like NodeJs and WebRTC of modern browsers in P2P based content sharing framework. We implemented the Chord protocol for our test environment but other protocols like Pastry or CAN are certainly possible, too. We are waiting for the WebRTC technology to become stable and mature for further explorations in other protocols.

X. REFERENCES

[1] Ian Clarky, Oskar Sandberg, Brandon Wiley, Theodore W. Hong: Freenet: A Distributed Anonymous Information Storage and Retrieval System 1999.

[2] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.

[3] Jennings, Cullen et al. "Resource location and discovery (reload) base protocol." REsource (2014).

[4] Vogt, Christian, Max Jonas Werner, and Thomas C. Schmidt. "Leveraging WebRTC for P2P content distribution in web browsers." Network Protocols (ICNP), 2013 21st IEEE International Conference on. IEEE, 2013.

[5] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram, "Maygh: Building a CDN from Client Web Browsers," in Proc. of 8th ACM European Conference on Computer Systems (EuroSys'13). New York, NY, USA: ACM, 2013, pp. 281–294.

[6] D. Dias, "webrtc-chord," [Online]. Available: http://www.npmjs.com/package/webrtc-chord.

[7] Werner, Max Jonas, Christian Vogt, and Thomas C. Schmidt. "Let Our Browsers Socialize: Building User-Centric Content Communities on WebRTC. "Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on. IEEE, 2014.

[8] "Peer 5: Sharefest;" [Online]. Available: https://www.sharefest.me/

[9] Baker, Matthew, F. Russ, T. David and W. Adam, "Implementing a Distributed Peer to Peer File Sharing System using CHEWBACCA--CHord, Enhanced With Basic Algorithm Corrections and Concurrent Activation," 2003.
[10] Werner, Max Jonas, and Christian Vogt. "Implementation of a Browser-based P2P Network using WebRTC." Hamburg University of Applied Sciences, Technical Report, January (2014).
[11] "Introduction to HTML5 Web Workers: The JavaScript Multi-threading Approach" [Online]. Available: https://msdn.microsoft.com/en-us/hh549259.aspx
[12] "Javascript Madness: The Javascript Sleep Deficiency" [Online]. Available: http://unixpapa.com/js/sleep.html
[13] "WebRTC in the real world: STUN, TURN and signaling" [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/
[14] "Javascript Session Establishment Protocol draft-ietf-rtcweb-jsep-03" [Online]. Available: http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-1.1
[15] "Stuntman - open source STUN server" [Online]. Available: http://www.stunprotocol.org/
[16] Werner, Max Jonas, and Christian Vogt. "Implementation and Evaluation of a DHT-based content distribution system using WebRTC."
[17] Google STUN Servers [Online]. Available: http://stun.l.google.com:19302
[18] "Elastic Compute Cloud (EC2) Cloud Server & Hosting – AWS" [Online]. Available: https://aws.amazon.com/ec2/