Open Peer – A Proposed Peer-to-Peer Signaling Protocol for WebRTC

By Robin Raymond

Chief Architect, Hookflash Inc.

What is Open Peer?

Open Peer (OP) is a Peer-to-Peer protocol designed to offer peer-to-peer signaling (on the wire) and P2P services for Real-Time Communications (RTC) applications. Open Peer works as a signaling layer on top of WebRTC in browsers. OP can also act as an independent P2P signaling stack for RTC standalone applications. An example of this is the Hookflash iPad app.

WebRTC Overview

> "WebRTC" as referenced in this document refers to the newly proposed open standard and not the open source project created and maintained by Google unless otherwise specified.

WebRTC is a standard proposed by joint effort of the IETF and the W3C for browser-to-browser RTC via simple JavaScript APIs, whose efforts are being supported by the major browser vendors.

While WebRTC allows for browsers to perform tasks around real-time communications, WebRTC on it's own does not offer a comprehensive approach to how two browsers should initiate communication to browsers on other machines. In other words, each website must devise its own approach to initiating communication so that a browser on machine "A" can communicate with browser on machine "B". This in essence is the problem that Hookflash is proposing Open Peer would resolve.

WebRTC allows JavaScript to obtain information from the browser on how to create an audio, video or data session, known as Session Description Protocol (SDP), and JavaScript can request access to a microphone, camera, or speaker and allow video to be rendered inside a browser window.

Simply obtaining SDP information from the browser is insufficient to initiate communication between two browsers. Each browser wanting to communicate must mutually exchange their SDP obtained from JavaScript through an intermediate server before the browsers are able to communication with each other.

In the simplest approach, to start real-time communications from JavaScript within a browser, an SDP can be obtained from the browser on machine "A", posted to a webserver and in turn fetched by the browser on machine "B" from that same webserver. The reciprocal SDP can be obtained from the browser on machine "B" and posted to the webserver and fetched by the original browser "A". With the two SDPs exchanged, the browser on machine "A" can quite literally talk to browser on machine "B".

Simple enough? Maybe not.

That approach ignores many things that need to happen in a typical communication scenario, which signaling protocols are designed to perform.

In a signaling scenario, when one user initiates communication to another user, the initiator has to signal to the remote party their desire to communicate and the receiving party has to signal its willingness to accept the communication. Without signaling, a service would have to presume that two independent parties immediately want to communicate regardless if they actually do. Making that presumption might work fine for some specific application scenarios but certainly not most or all scenarios. After all, the contacted party might be indisposed or unwilling to communicate when the initiator attempts communication. Once two parties have finished communicating, one party must signal to the other that it no longer wishes to communicate and terminate the communication.

Even if the presumption is that both parties wish to communicate immediately, signaling is still required. A party that wishes to communicate must literally sit in waiting to be contacted or must somehow be pushed notification that communication is desired.

None of that signaling process is defined by WebRTC and the scenario outlined assumes only the most basic of signaling scenarios of initiating, ringing, answering and hanging up (let alone advanced scenarios like call forwarding, or conference calling). The technical mechanisms of exactly how browsers sit in waiting for SDPs to be exchanged and how to browsers respond to requests is missing from WebRTC.

While this might seem like a huge oversight, WebRTC lacks this signaling by design. The IETF and W3C were wise to know that fixating on a single solution for signaling now would hamper advances in new or no signaling technologies. Technologies like Open Peer would never have been developed if WebRTC had chosen to use an existing protocol. Plus, the various signaling vendors behind SIP, XMPP and Skype are all vying to be position themselves to leverage WebRTC and promote their own signaling protocol in combination with WebRTC. Thus adopting a signaling protocol standard by the IETF and W3C would have caused a war between signaling providers that would have set the efforts to get WebRTC adopted by browser venders pushed back for years to come.

As WebRTC lacks signaling, each website wishing to use WebRTC is left without a critical piece of technology to make the peer-to-peer real-time communication capabilities functional. As a result, each website will either implement their own custom solution (which will likely not openly interoperate with any other website) or chose to use a JavaScript/webserver signaling bridge mechanism to interface to an existing signaling protocol like SIP, XMPP or Skype. Now websites have another alternative, Open Peer, a stack being directly written in JavaScript (and is available in other languages as well).

SIP, XMPP and Skype may not be the Best Answer

The natural solution for the lack of WebRTC signaling would be to take an existing open standards technology like SIP and write up some kind of bridging from JavaScript via a webserver to talk with SIP servers, thus allowing WebRTC to function. SIP is highly successful, but it's success is also it's greatest failing.

The SIP protocol is now old and long expanded from its early days as RFC 3261 into a tangled complex web of RFCs and technologies to address many of the non-addressed issues in the original SIP design and specification. As such, no two SIP server vendors can now firmly agree what fundamentally is "SIP". Worse, no two SIP endpoint devices from different vendors can fully agree what is "SIP". To say that a device or server is "SIP" compatible isn't truly giving the full picture. Sure, the devices might be RFC 3261 compliant but what other set of related technologies do they support or not support? If companies using various SIP technologies want to inter-communicate, they have to go through an extensive series of inter-op testing to ensure basic compatibility that their definition of SIP can talk to some other party's definition of SIP. That's simply unacceptable to expect each website to go through an extensive testing process to ensure it can talk with every other website on the planet, one-by-one. Users will expect to talk to other users on other websites without problems.

Both SIP and XMPP suffer from scalability challenges. SIP allows for stateless routing mechanisms that in theory should help (but suffer inter-op problems between vendors). XMPP is challenging to scale as messages from any users can be directed to any other user at random and thus complex messaging systems have to be built into the servers to achieve any form of real scalability. To help solve this issue standard proposals like XMPP relay nodes are used. However, like SIP, the simple XMPP standard isn't XMPP anymore. XMPP is XMPP plus a bunch of other standards to make it useable. Unless the vendors agree on the standards, issues start to emerge with XMPP. One advantage to XMPP over SIP is that it at least allows for easier discovery capability than SIP, at least during negotiation as to understand what two XMPP clients support.

Typically, SIP and XMPP installations are done for small companies with limited users and with complex setup processes. Websites wanting to use WebRTC can have hundreds, thousands and millions (and more) of users coming and going constantly. Using a technology like SIP and XMPP for WebRTC will absolutely require that scaling is addressed properly and inter-op across sites is handled. That's not to say it's impossible, but scaling can be a formidable effort.

There are services, such as Google Talk and Skype, that address the scalability issues but they have a major drawback for websites. They require a Google or Skype accounts. That's simply not going to be acceptable to ask users from every website everywhere wanting to use WebRTC to go setup a Gmail or Skype account. Skype is also a closed protocol owed by Microsoft and website owners can never be sure what is really going on behind the scenes.

Lastly, as the messaging is continuously relayed through servers for SIP and XMPP, if a server goes down then all the connected peers go down as well.

Open Peer is a Proposed Solution

Open Peer was designed and architected specifically in response to the challenges that will exist for WebRTC when it becomes ready for prime time. Based on years of

experience working with mostly SIP and XMPP, Open Peer became a proposed solution to address these challenges.

Federated Websites and Identities

WebRTC is being introduced in an environment where many websites exist that have thousands of users already signed up with existing accounts. Each user on each website likely has a profile that represents themselves on each website and may contain a directory users that are somehow connected or related.

For example, Alice might have a Facebook, LinkedIn, Twitter and a "foo.com" account and profile. Each one of those is a profiles is an identity of Alice, but all point to the same Alice behind the scenes. Bob, a friend and college of Alice, might be friends with Alice on Facebook and colleague on LinkedIn. Bob should be able to contact Alice regardless which website Alice is logged into, be it Facebook, LinkedIn, Twitter, or "foo.com". In fact, Alice should be omnipresent on all of her accounts without having to leave a window per website open all the time.

WebRTC, on its own, offers no mechanism for one website to contact users logged into another website, or for a website to contact an independent mobile real-time communication application, web or native. Without a protocol like Open Peer, the website will only be capable of getting users to talk to other users on the same website, as no agreed signaling protocol exists across websites. Websites using WebRTC without adopting a protocol like Open Peer will become communication islands where their users can only talk amongst themselves. As a predictable result, users are likely to login to the one website they are most likely to find their friends or colleagues to the exclusion of other websites, if forced into a situation where they must monitor multiple websites independently.

Open Peer is the glue that allows users on various independent websites to talk together. Open Peer was designed with the concept that users have identities on various websites and these users should be able to login to one website or mobile application and have an online presence across all the various websites that they belong.

For Open Peer to allow intra-website & intra-app communication, certain principles must be introduced. Open Peer assumes that each user on each website has their own identity (i.e. an account and optional profile) and it assumes a user might have accounts on multiple independent websites.

Open Peer allows a user to login to all their identities on all their websites while requiring only one website (or mobile) application be used for communication. For example, Alice could be on her mobile Open Peer compatible application and logged into all her identities on every website (without even having those websites windows open). Should Bob contact Alice, Alice will be told her friend and coworker Bob from Facebook and LinkedIn is trying to contact her. Alice can then choose to communicate with Bob if she wishes.

By mapping identities to users behind the scenes, Open Peer allows websites to leverage the online identities and relationships between users that already exist and

those relationships can work across independent websites boundaries. Any website can drop in Open Peer technology along with WebRTC and leverage those same advantages for their real-time communication applications.

Legacy Identities

Open Peer not only can map identities on websites to users, but Open Peer can map identities legacy systems like PSTN phone numbers to users on the Open Peer network. For example, Alice might register her phone number as a legacy identity for herself. Bob, having Alice's phone number in his address book, could find and contact Alice on the Open Peer network rather than having to resort to placing a traditional phone call.

Other legacy identities like email addresses can also be used for users to find one another. As an example, someone knowing Bob's email address can find Bob on the Open Peer network.

Allowing legacy identities, like phone number or email addresses, is an important concept to a universal communication system so that users having existing legacy contact address books can find their contacts on the Open Peer network without requiring updating their address books.

Mobile Web is Trending

Mobile web access is big and getting bigger, with speculation that the majority of web browsing could be performed on mobile devices in a few short years.

Users are not going to want to keep their tiny mobile browser windows open to a bunch of different websites to be able to monitor communications to their contacts, one website at a time. That won't fly.

As Open Peer is a protocol, it allows independent mobile client applications to be created that can directly act as the communication application for websites using WebRTC, without performing any special integration to each individual website. This allows users to user their favorite Open Peer compatible mobile real-time communication application with their favorite website.

Security

Open Peer was designed with a few basic principles: push security to the end point, do not send sensitive data to a server, and take advantage of existing user identities which can be verified.

Open Peer requires each peer generate its own public and private key pair and allows peers to exchange these public keys between peers to establish secure peer-to-peer communications.

Instead of having the key pairs signed by a trusted authority service, like VeriSign, users can obtain signed proof of their identities from websites that can be linked to their public keys. For example, instead of Alice sending a copy of her passport into an authority like VeriSign, paying a substantial fee, and waiting the turn around time, Alice can simply obtain proof of who she is by her already established online social identities, like her account on Facebook, LinkedIn or Twitter. Alice can use

those accounts as proof of who she is when she contacts other users. In turn, those other users can verify that this is the same Alice in their contact lists and not some Alice imposter.

Great care was taken when architecting Open Peer to ensure the servers required for the entire Open Peer protocol would not contain any sensitive data, and this sensitive data would not flow though them. Servers can't be completely avoided as they represent the source of identities, and act as the introductory point between peers but the peers can relay sensitive information directly to other peers and ensure the servers do not contain data that would make them an attractive target for hackers.

Even in the best designs and architectures, flaws and human errors in setup/configuration are often exploitable. Thus servers can and do become compromised all the time by hackers even with the best secure measures available.

To be clear, peers can become compromised just like any server too but a compromised sever might result in many millions of users' data getting compromised rather than a limited select few. As sensitive data in Open Peer goes peer-to-peer, hacking the Open Peer servers results in information of little value. Even the keys to encrypt the sensitive data sent between peers are not known by the Open Peer servers. This makes Open Peer servers low value targets by hackers.

Scalability

Open Peer clients applications use servers only to introduce themselves to other peers they desire to communicate and to connect users with their website identities. Thus, the reliance on servers to maintain communication does not exist with Open Peer. Open Peer clients can continue to communicate even with the Open Peer servers down.

Keeping signaling out of the servers not only has an advantage of reliability, it ensures allows the servers to be scaled easier. The amount of traffic a server needs to handle is lower, contain little state, and the servers themselves can use a highly scalable ring database system such as Cassandra to achieve scalability levels with only simply key/value pair information.

Open Peer was architected specifically to be able to achieve high scalability requirements that can be expected when high traffic websites start offering WebRTC to their user base.

Website Integration and Hosting

Open Peer was architected in such a way to minimize the amount of work required for a website to integrate Open Peer services into their website offering. For any protocol to be successful it must be reasonable to implement. While Open Peer is sophisticated, hosting Open Peer in the cloud and integrating Open Peer into a website couldn't be easier.

Open Peer allows websites to leverage their existing identities for their users. With minimum changes, the website can enable their users to communication with users

on the same website as well as users on other websites using an Open Peer hosted service (like those offered by Hookflash.com).

Alternatively, since the Open Peer specification has been made available to the world for free, any company can decide to deploy their own Open Peer implementations should they desire.

Hookflash will release reference implementations of Open Peer client applications in Objective-C and in JavaScript to short cut the development of mobile and website applications using Open Peer and to ensure easier interoperability between users since they can share the same internal implementation.

Finding Peers and Bootstrapping

Pure peer-to-peer solutions cannot work on the Internet without some additional help.

While Local Area Networks (LAN) allow peers to discover other peers on the same network via LAN broadcast discovery messages, broadcasting does not work on the Internet as it is specifically prohibited (unless relayed through an intermediate server that unicasts to each peer node). Thus self-organizing peer-to-peer network solutions possible within a corporate/home LAN aren't available on the open Internet.

Firewalls also play havoc with peer-to-peer networks. Firewalls do exactly what they are designed to do, i.e. block incoming traffic to which there is no reciprocal outgoing traffic. In other words, you cannot have one peer send a request to another peer behind the firewall unless that peer is also sending data back first. With two peers behind firewalls, neither peer can initiate contact with the other.

Resolving the firewall issue comes down to two solutions: Requiring users open up ports to allow data traffic to flow (an unreasonable requirement for average home users), or using an introductory intermediate server that isn't blocked behind a firewall.

Open Peer uses an introductory service for one peer to initiate contact to another called a "Finder". With Open Peer, the Finder's job is to initiate contact between two peers then drop out of the picture. Each peer maintains a presence on a Finder so that peers can introduce themselves to other peers without the firewall hindering the connection.

Given that the Finders solve the introduction problem caused by firewalls, the next problem is how peers find the Finder servers. This is done through a Bootstrapper service, which acts as the entry point into the peer-to-peer network and introduces peers to available Finder servers.

Reliable Data Transmission

TCP is currently the most prominent way for one machine to reliably send data to another. Machines frequently need to send data that must be delivered in order, reliably and fairly. For example, a user would never tolerate an application that

downloaded file missing some critical data or if the file was corrupted because the data arrived out of order.

Likewise, many users download data simultaneously, and data has to be delivered fairly amongst users. Users would not accept a network where every time one user downloaded something large, all other users on the same network were unable to download anything because the network pipe was entirely filled with that one user's data.

TCP, which offers fair, reliable, in-order data transmission, is widely popular for control and data transfer between computers and TCP is the standard protocol used for webservers.

UDP is an alternative to TCP, but it does not offer fair, reliable, in-order data. UDP works analogous to postal mail. Envelopes containing data arrive out-of-order from one machine to another (if the data arrives at all). UDP is popular for real-time communications, such as Internet phone applications, where losing 20 milliseconds of digital audio data isn't considered disastrous.

Firewalls have nasty side effect: they block incoming TCP traffic (unless users specifically open ports). In other words, peers cannot open TCP connections to other peers to send data. While efforts to allow TCP traffic across firewalls have been underway for years, no standard has yet been widely adopted. Basically, firewalls allow a peer to initiate TCP connections to a server, but a server cannot initiate TCP back to a peer, and a peer cannot initiate TCP to another peer.

Firewalls frequently do allow incoming and outgoing data over UDP, but typically a peer must transmit data from behind the firewall to an external peer before the firewall allows data back in from the external peer. The IETF has devised techniques such as STUN, ICE and TURN to take advantage of firewall behavior to assist peers in being able to communicate through firewalls with UDP, once the peers discover with whom they wish to communicate through some unspecified process.

The initial version of Open Peer implements a protocol called RUDP (not to be confused with the 1999 RUDP IETF proposal). This new RUDP protocol layers many of the capabilities of TCP on top of UDP, allowing peers to have a fair, reliable, in-order data transmission between peers, suitable for control protocols and data transfer. Open Peer's RUDP is an extension of STUN/ICE protocol and allows peers to open reliable data channels to other peers that easily works across firewalls. RUDP is easy to implement in an application layer, as it is layered on top of UDP rather than requiring a new operating system level communication stack being implemented at the same level as TCP/UDP (thus RUDP can work on any operating system that already supports UDP).

Open Peer as a protocol is designed to be agnostic to which reliable data protocol is used between peers and simply mandates that such a protocol exists. Thus once reference implementations exist, such as those proposed for WebRTC, implementers of Open Peer can replace RUDP with those alternatives.

Open Peer versus P2PSIP/Chord for Peer-to-peer Networking

Chord (and P2PSIP which is based on Chord) is a peer-to-peer approach allowing peers to self organize, typically into a highly scalable ring formation. In a ring scenario, each peer has a neighbor peer and that peer has a neighbor, and so on. As peers come and go, they get introduced into the ring and join and leave the ring.

The Chord ring is used in the same manner as the Finder's are used in Open Peer. Each peer in the ring acts as an introductory service to other peers. The difference is that Finders are typically hosted in Open Peer, whereas Chord assumes peers organize themselves into a ring of servers. Typically, peers in a ring have shared and maintained tables of their neighbors at various intervals along the ring allowing one peer to quickly be able to be introduced to a far away neighbor, and allow a peer to eventually home-in on a desired peer.

Chord is a wonderful experimental protocol but has serious practical issues that need solutions to be a viable alternative. Entire articles are written explaining (in detail) the problems Chord presents in real world deployments but can be summarized to a few key issues.

Firewalls play havoc with Chord. Chord assumes any peer can easily communicate to any other peer along the ring. If any peer joined into a chord ring is behind a firewall, it must be introduced to new peers via a previous peer that it already has established communication with.

Firewalls close inactive communication channels automatically; peers in a chord ring must remain constantly "chatty" to keep connection to all their neighbors alive. Further, firewalls only allow limited number of peer channels before all the available firewalls channels become consumed. A Chord ring that allows firewalled peers as part of the ring can require a tremendous amount of Internet traffic, and consume many firewall ports just to maintain a basic ring formation.

Alternatively, Chord rings can limit only those peers that are not behind a firewall to be 'super nodes' in a ring and all other peers behind firewalls become leafs of peers off a 'super node'. This often leads to eventual collapse (proven in practice). Most users don't like paying for Internet traffic used by random strangers with no reciprocal benefit to themselves. Thus users whose machines start out as 'super nodes' discover their bandwidth costs and convert their machines into leafs which leech off some other 'super node' instead. This process continues until there are too few 'super nodes' left to sufficiently power the Chord ring. Eventually the system collapses under it's own weight of success. It has been reported that Skype had a similar problem and decided to host super nodes rather than relying solely on peer based super nodes when its own network collapsed.

Chord has another problem. Chord requires that each peer in the ring have no evil intentions. The Chord ring is only viable if each and every peer in the ring acts in the best interest of maintaining the integrity of the ring. If peers get introduced into the ring that intentionally break the ring, the entire Chord network can be brought down. Thus Chord implementers often discuss "reputation systems" to solve evil peer scenarios. Basically, algorithms randomly spot check peers to ensure they

behave correctly and reputations are formed over time for each peer. This might work well to spot individual rouge peers in a ring but reputation systems can easily be subverted by coordinated peer attacks or by way of clever subversion of the reputation systems. In the days of bot nets and hackers, entire rings could have their entire reputation systems rendered completely useless and their rings brought down by sets of rouge misbehaving peers.

In a nutshell, Chord relies on users running altruistic peers for the sole benefit of other users where bot nets and hackers are successfully kept out.

Open Peer remains agnostic. If the solutions to Chord's problems can be found, Open Peer Finders can be layered on Chord peers in a ring, instead of hosting Finders. In fact, the Hookflash Finder implementation uses a distributed database that is self-organized not unlike a Chord-like ring for high scalability (on a private network of course). This allows peers to have a solution that works now until viable solutions to the issues with Chord can be found be.

Summary

Open Peer appears to be the only open P2P protocol openly available today that works with WebRTC and on its own and in hybrid RTC models. My hope is to see Open Peer become as ubiquitous as WebRTC itself and maybe even more so with regards to stand-alone P2P deployments.

About the Author

Robin Raymond is an expert software architect, technical leader and developer, whose specialty is for highly scalable network asynchronous software architectures, typically in the field of peer-to-peer telecommunications. Robin has been producing software since the dawn the computer age and has worked in everything from coding and software architecture to managing entire software departments and performing the duties of CTO. He was the original author one of the most widely downloaded and installed SIP softphone clients, X-Lite/X-PRO. Robin currently serves as Chief Architect @ Hookflash Inc.

Reference Material

Open Peer Specification: http://openpeer.org

Open Peer Github Project: http://github.com/openpeer/op

W3C WebRTC: http://www.w3.org/TR/webrtc/

IETF RTCWEB: http://tools.ietf.org/wg/rtcweb/