Computer Communications 70 (2015) 28–39

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Scalable signaling protocol for Web real-time communication based on a distributed hash table

Jung Ha Paik, Dong Hoon Lee*

Department of Information Security, Korea University, Seoul, Republic of Korea

ARTICLE INFO

Article history:
Received 23 April 2014
Revised 21 May 2015
Accepted 27 May 2015
Available online 3 June 2015

Keywords:
WebRTC
Distributed hash table
Signaling protocol
Peer-to-peer

ABSTRACT

Web real-time communication (WebRTC) provides browser-to-browser communications without installing any plug-in. In WebRTC, peers have to prepare their communication session through a signaling protocol which coordinates peers and exchanges Session Description Protocol (SDP) message between two peers. The problem is that the most well known signaling method cannot provide the scalability because the method relies on only single server. To overcome the problem, this paper presents a scalable WebRTC signaling protocol. The main idea is that each peer forms a peer-to-peer topology by structuring relevant WebRTC connections with each other and then sends signals across those connections. The central server needs to handle only a few connection establishments for newly joining peers. The rest of the signaling process can be performed by peers. We define and justify such a protocol including a bootstrap method, a stabilization scheme, and peer lookup. The procedures are designed to be suitable for WebRTC connections and to be resilient against the churn condition. Furthermore, we implement the proposed protocol in pure JavaScript to show that it is realizable. The performance of the implementation is practical, with signaling latency averaging 0.5 s when the number of peers is 1000. Each peer still correctly locates the other peers even when the network is very congested.

1. Introduction

The Internet has encouraged people to communicate with each other, changing their patterns of communication through Web services. Therefore, modern Web browsers are not limited to browsing the Internet and should support various types of Web applications including peer-to-peer communication. Historically, because Web browsers could only communicate with Web servers, Web applications supporting client interaction only communicated through central servers, which was inefficient. A cost-effective method of interacting using Web browsers is to connect one browser directly to another, enabling real-time communication (RTC). For the past few years, such transparent connection within the browser context has been one of the major challenges for Web applications. Web real-time communication (WebRTC) aims to provide open-access, high-quality RTC technology for Web developers. Before WebRTC, Web applications on which clients collaborate, such as video conferencing, voice chats, and messenger services, have only been available through proprietary plug-ins such as the Adobe Flash Socket.

The presence of WebRTC enables peer-to-peer communication without plug-ins [1]. WebRTC provides the substrate for adding this communication feature to Web applications with minimal effort and reliability. That is, Web application developers can offer features allowing connectivity and data exchange among clients by simply using HyperText Markup Language (HTML) and common JavaScript APIs within their Web applications. The standards for WebRTC APIs are set by the World Wide Web Consortium (W3C) [2], and the corresponding internal protocol suite is defined by the Internet Engineering Task Force (IETF) Working Group RTCWeb [3]. Major browser vendors have already employed the latest standard version of WebRTC in their Web browsers.[1]

To open their connection, WebRTC peers need to initiate `RTCPeerConnection` and construct the connection using `RTCPeerConnection` APIs. However, WebRTC peers face a problem when establishing the first WebRTC connection. Unlike a conventional connection such as the Transmission Control Protocol/Internet Protocol (TCP/IP), in WebRTC, peers cannot locate arbitrary peers by themselves. In TCP/IP communication one assumes that a source client knows a destination client's address (e.g., IP and port address) and that the destination client is waiting for an incoming socket

* Corresponding author. Tel.: +82232904892.
E-mail addresses: jungha.paik@gmail.com (J.H. Paik), donghlee@korea.ac.kr (D.H. Lee).

[1] Currently, WebRTC is stable on Google Chrome Desktop 25+, Chrome Android 29+, Mozilla Firefox 14+, and Opera 18+.

http://dx.doi.org/10.1016/j.comcom.2015.05.013
0140-3664/

connection. Under these conditions, the source can access the destination so that they communicate over a bidirectional socket stream. By contrast, connecting two peers in WebRTC is considerably different. The basic architecture of WebRTC applications requires an additional entity, usually called the signaling server, to deliver the signal between the two parties. A source peer who wants to connect with a destination peer requests to send a message including the source's network address and the type of interaction to the server, and then the server delivers this message to the specific destination. Similarly, the destination generates its message and the signaling server delivers the destination's message to the original source peer. Then, the two peers can negotiate a communication session based on the exchanged messages.

Signaling is a common process used in modern network services such as SIP [4]; however, WebRTC does not mandate the use of any particular signaling method [1], although a central server is usually used to run a signaling protocol. That is, every WebRTC peer connects to the central server and then the server relays the peers signal. Thus far, several examples of WebRTC signaling in practice have followed this approach. Further, very few researches consider the possibility of combining SIP with a WebRTC service [5]. However, the feasibility of these solutions has not yet been proven in a practical environment, which has to include a large number of peers who usually rely on a single signaling server. In a WebRTC application involving a large number of peers, all participating peers must be bound to a unique signaling server thus the signaling server coordinates a large number of peers at any given time. This frequently causes high latency and a single point of failure. To resolve these problems, we believe the best approach is to provide scalability for the WebRTC signaling protocol by applying a Distributed Hash Table (DHT) to the mechanism of peer coordination.

In this paper, we present PeerConnecion over DataChannel (POD). POD provides a WebRTC signaling protocol via `RTCDataChannel`, requiring minimal effort by the central server. Signaling through peer connections requires a certain coordination mechanism. The heart of our protocol is that the peers participating in the WebRTC application maintain structural `RTCDataChannel` connections based on a DHT so that any peers can connect to any others using underlying connections bypassing the central server. This makes WebRTC signaling more reliable and scalable even when a large number of peers are on the network.

The contribution of this paper is as follows. We define a POD protocol that includes peer lookup (signaling) based on Chord [6,7]. It is easy to conceive an idea such as coordinating peers across `RTCDataChannel`, but to make it work is difficult. Besides, the design principle of POD is based on the fact that an established `RTCDataChannel` between two peers can be utilized mutually as a communication channel whereas Chord and most Chord variants only consider one-way channels. POD contains bootstrap and stabilization methods that accord closely with the characteristics of `RTCDataChannel`. We further justify our methods in a nonadversarial model. Moreover, we implement all the POD functionalities in pure JavaScript to show that POD can be applied in the real world. Developers can easily deploy POD as their WebRTC signaling method by simply importing the POD scripts into their applications. The size of our POD protocol is approximately 30 kb and it needs to be downloaded once. This does not burden clients offering modern network service. Finally, our evaluation results show that the signaling latency and data overhead in each peer are practical. The performance time to establish a connection averages about 0.5 s when 1000 peers are involved on the network. In addition, POD is resilient against the churn condition [8]. Even if each peer continuously joins and leaves the signaling network, the signaling method still reliably locates the specific targets. While half of the peers leave and rejoin the network every minute, two attempts to establish a connection are required on average.

The remainder of the paper is organized as follows. In Section 2, we describe the background on related technology and research. In Section 3, we describe POD protocols and prove the performance of POD procedures. In Section 4, we present the principals of POD implementation. In Section 5, we analyze POD, and in Section 6, we show the results of our POD evaluation. Section 7 concludes the paper.

2. Background

2.1. WebRTC and signaling

To better understand WebRTC architecture, in this section, we briefly describe the manner in which a common WebRTC service scenario could be developed. The scenario is one in which two peers establish an `RTCPeerConnection` along with an `RTCDataChannel` session so that they can transfer the generic data to each other over the underlying session. Basically, the WebRTC application utilizes at least three entities. First, as an application provider, a Web server provides the WebRTC applications (via an HTTP service). The WebRTC application is written in a JavaScript language with the core WebRTC APIs [2]. Second, a signaling server is operated to relay one's WebRTC signal to another. Finally, there are peers who participate in the application service. Peers are divided into two classes: offerers and answerers. The offerer is a peer who initiates the WebRTC connection to another peer. On the opposite side, there is an answerer who is asked for the connections from the offerer. The offerer previously needs to learn about the answerer's information which is used for designating it. A common scenario is that the offerer is assumed to know the answerer's original identifier (e.g. an e-mail address) and then requests a connection through a signaling server. If the answerer is on-line, or equivalently, connected with the signaling server, they can make a WebRTC connection through the server.

After peers download a WebRTC application from the Web server, they can establish a WebRTC connection as follows. The offerer first begins to initialize an `RTCPeerConnection` object. In `RTCPeerConnection`, a `createDataChannel` method is used to create an `RTCDataChannel` object. When an `RTCDataChannel` on the offerer's side is generated, the offerer invokes `createOffer` in member methods of `RTCPeerConnection`, thereby enabling `createOffer` to return an offerer's Session Description Protocol (SDP) message [9,10]. To make a connection, the offerer first generates the SDP-offer message by calling `createOffer` and then needs to send the message to a specific answerer through a certain signaling method for which the signaling server is responsible. Once the SDP-offer message reaches the answerer, the answerer also initiates its `RTCPeerConnection` instance to accept the request. The answerer installs the SDP-offer message into its `RTCPeerConnection` and then creates an SDP-answer message by calling `createAnswer`. The signaling server also hands this message over to the offerer's side. The SDP-offer/answer messages include a set of attributes indicating the client's session information such as the version, bandwidth, etc. [9]. After two peers exchange SDP-offer/answer messages, they can initialize their session. In the meantime, `RTCPeerConnection` of both the offer and answer starts gathering the address of the potential contact point for the recipient, which is called the Interactive Connectivity Establishment (ICE) candidate. The ICE candidate messages from the offerer and answerer sides also need to be exchanged through the signaling server. Based on the exchanged ICE candidates, they can actually contact each other and establish the session. When two peers establish their session, their browsers internally operate the protocol described in the ICE protocol [11,12]. ICE allows two peers in heterogeneous network conditions to accomplish the best transport for communication. ICE uses Session Traversal Utilities for NAT (STUN) [13] and Traversal Using Relays around NAT (TURN) [14] to establish the session. After the session is established, the offerer and answerer acquire `RTCDataChannel` objects for each other. `RTCDataChannel` defines the data-sending API