Mint 709 Capstone Project - Final Report: Implementation of a Video-Conferencing Web-Application using the WebRTC and WebSockets APIs within a Local Area Network

ADEGBAMIGBE, ADEDUROTIMI
UNIVERSITY OF ALBERTA
DEPARTMENT OF COMPUTING SCIENCE,
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
MASTER OF SCIENCE IN INTERNETWORKING

Introduction

This is a report on the implementation of a video-conferencing web application (**LAN-Communicator**) that enables real-time video communication between a minimum of two and a maximum of four clients / peers (with each client / peer being a web-browser with access to a camera and microphone) within the same Local Area Network (LAN). The LAN-Communicator application leverages two relatively new web technologies, **Web Real-Time Communications** (WebRTC) and **WebSocket**.

WebRTC natively supports video communication sessions between two peers (see Fig. 1 below). However, extra logic must be implemented in a WebRTC enabled application in order to allow real-time video communication between more than two peers (see Fig. 2 below). This extra logic is what was implemented in the LAN-Communicator application described in this report. The complete source code with installation instructions for the LAN-Communicator application is located at the following bitbucket repository: https://bitbucket.org/rotexdegba/mint-709-webrtc-project.

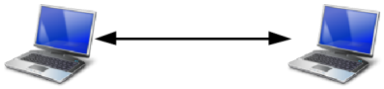


Figure 1: Basic RTCPeerConnection Topology for a WebRTC audio/video conference session between two peers.

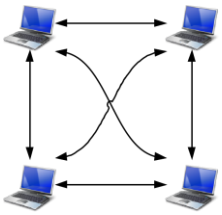


Figure 2: Mesh Topology for a WebRTC audio/video conference session between four peers. Each peer maintains an RTCPeerConnection to each of the other three peers.

Motivation and Problem Definition

The web-application described in this report aims to provide a cost-effective video-conferencing solution to individuals or organizations that are concerned about the privacy and confidentiality of their communication data and have the infrastructure (e.g. adequate network bandwidth, server hardware, etc.) to host and support the application within their home or corporate LAN.

Video-conferencing traffic generated while using this application stays within the network where the application is hosted and all traffic is encrypted:

1. all web-pages and **JavaScript** code are served over a secure **Hypertext Transfer Protocol** (HTTPS) connection,
2. the WebSocket connection between each client and the application's Signaling Server is also secured via the use of the **WebSocket Secure** (WSS) protocol
3. finally, the **RTCPeerConnection** links used to transmit audio/video streams between peers are each automatically secured by the WebRTC engine via the use of the **Secure Real-time Transport Protocol** (SRTP). This is a feature built into WebRTC.

Though video-conferencing solutions like **Skype**, **Google Hangouts**, **GoToMeeting**, **WebEx Meetings**, etc. already exist, most of these solutions involve video-conferencing traffic traveling through external networks. Even if all traffic generated by these existing solutions is encrypted there is no guarantee that any of the external networks through which the traffic travels will never be compromised and these solutions are more susceptible to man-in-the-middle attacks. Hosting such an application within one's own network is far better and helps in accomplishing the goal of having all communication data private and confidential as it greatly reduces the risk of malicious third party entities eavesdropping on or recording communication data via man-in-the-middle attacks.

WebRTC is an open standard that is supported by many modern browsers (Google Chrome 23+, Google Chrome for Android 29+, Mozilla Firefox 22+, Mozilla Firefox for Android 24+, Opera 18+ and Opera Mobile 13+ and a host of others to come in the nearest future). The WebSocket technology is also an open standard supported in the earlier mentioned browsers and more. From a software developer's or vendor's perspective, there is no cost associated with using WebRTC and WebSockets to build a video-conferencing application, both technologies are implemented in supporting browsers by the browser makers, are free to use (making them very cost-effective options) and are constantly being improved upon by browser makers. They both support encryption of traffic which also helps in accomplishing the goal of having all communication data private and confidential.

As at the time of writing this report, the other alternative solutions that exist at the browser level for building web-based video-conferencing applications are technologies like **Adobe Flash**, **Microsoft Silverlight**, **Java Applets**, etc. which all require the end user to have the appropriate plugin installed and enabled in their web-browser. Adobe Flash (which is not supported on most modern mobile devices like the iPhone, etc. because of its high CPU usage and battery draining tendencies) requires the **Adobe Flash Player** plugin to be installed in a user's browser. The client portion of an Adobe Flash powered video-conferencing web application must be developed using the **Adobe Flex SDK**, with the code written in the **ActionScript** programming language. **Adobe Flash Builder** is Adobe's official development environment that is recommended for building such applications and it is not free, it is a paid tool. A complementing server also has to be developed and by the time all these efforts are added up, it becomes apparent that developing such an application using an alternative technology like Adobe Flash is costlier both timewise and financially and clients using mobile devices would not be able to use such an application since Adobe Flash is not available for mobile web-browsers. Browser plugins (Adobe Flash Player particularly) have also historically been a source of security vulnerabilities for web-browsers in which they are installed. In the nearest future, when the **HyperText Markup Language** (HTML5), **Cascading Style Sheets** (CSS3) and JavaScript browser APIs have matured well enough to provide most of the functionality that was previously only available via plugins, there will be little or no need for these plugins. Some browsers like Mozilla Firefox *will soon start requiring click-to-activate approval from users before a website activates the Flash plugin for any content*. [1]

A WebRTC powered application requires less development and deployment (no browser-plugin required) effort in comparison to an Adobe Flash based one and is also cheaper to develop (since the main cost incurred would be the software-developer's fee for building the application or no cost if the organization already has a software developer on their payroll. An SSL certificate could optionally be purchased for a token fee, but a free self-signed SSL certificate would equally be sufficient to support

secure HTTP and WebSocket connections). The WebRTC based solution also has the advantage of being supported in some mobile web-browsers. There is also a growing potential client base as other mainstream web-browsers like Safari and Microsoft Edge plan to support WebRTC in the future.

Commercial video conferencing application solutions that allow organizations to host within their own corporate LAN, like the **Cisco WebEx Meetings Server** and the **Skype for Business Server 2015**, would still involve paying an ongoing monthly / annual usage / license fee (technical support fee may be separate) to the application vendor and may even require the purchase of custom hardware for hosting such applications. Such solutions are usually based on proprietary technologies (rather than on an open standard like WebRTC) and may be difficult to integrate with other software systems a client has already invested in and such integration would most likely lead to the client having to pay the vendor of the solution in question for custom development. This a far more expensive solution for running a video-conferencing application within a LAN, compared to developing and hosting such an application with technologies like Adobe Flash or even better, WebRTC.

The costs (i.e. payment of licensing fees in some cases, the need to purchase development tools and greater software development effort) associated with building a video-conferencing web-application with non-WebRTC based technologies (like Adobe Flash) and the higher recurring licensing and support fees for commercial on-premise hosted video conferencing application solutions (like Skype for Business Server 2015), makes WebRTC the most cost-effective and future-proof option for the task at hand.

Background

WebRTC is a collection of standards, protocols, and JavaScript APIs, the combination of which enables peer-to-peer audio, video, and data sharing between browsers (peers). Instead of relying on third-party plug-ins or proprietary software, WebRTC turns real-time communication into a standard feature that any web application can leverage via a simple JavaScript API. [2]

WebRTC enabled browsers (peers) provide three major JavaScript APIs that can be used to develop web-applications with peer-to-peer audio, video and / or data sharing capabilities:

- **getUserMedia:** for getting access to the audio / video stream(s) from the camera(s) and microphone(s) attached to a client's device.
- **RTCPeerConnection:** *for establishing, maintaining, monitoring and closing connections between peers.* [3] Each established connection is used to transfer audio and video data between peers.
- **RTCDataChannel:** *represents a network channel which can be used for bidirectional peer-to-peer transfers of arbitrary data. Every data channel is associated with an RTCPeerConnection, and each peer connection can have up to a theoretical maximum of 65,534 data channels (the actual limit may vary from browser to browser)*. [4]

Fig. 3 is a diagram of how WebRTC is fits inside a web-browser.

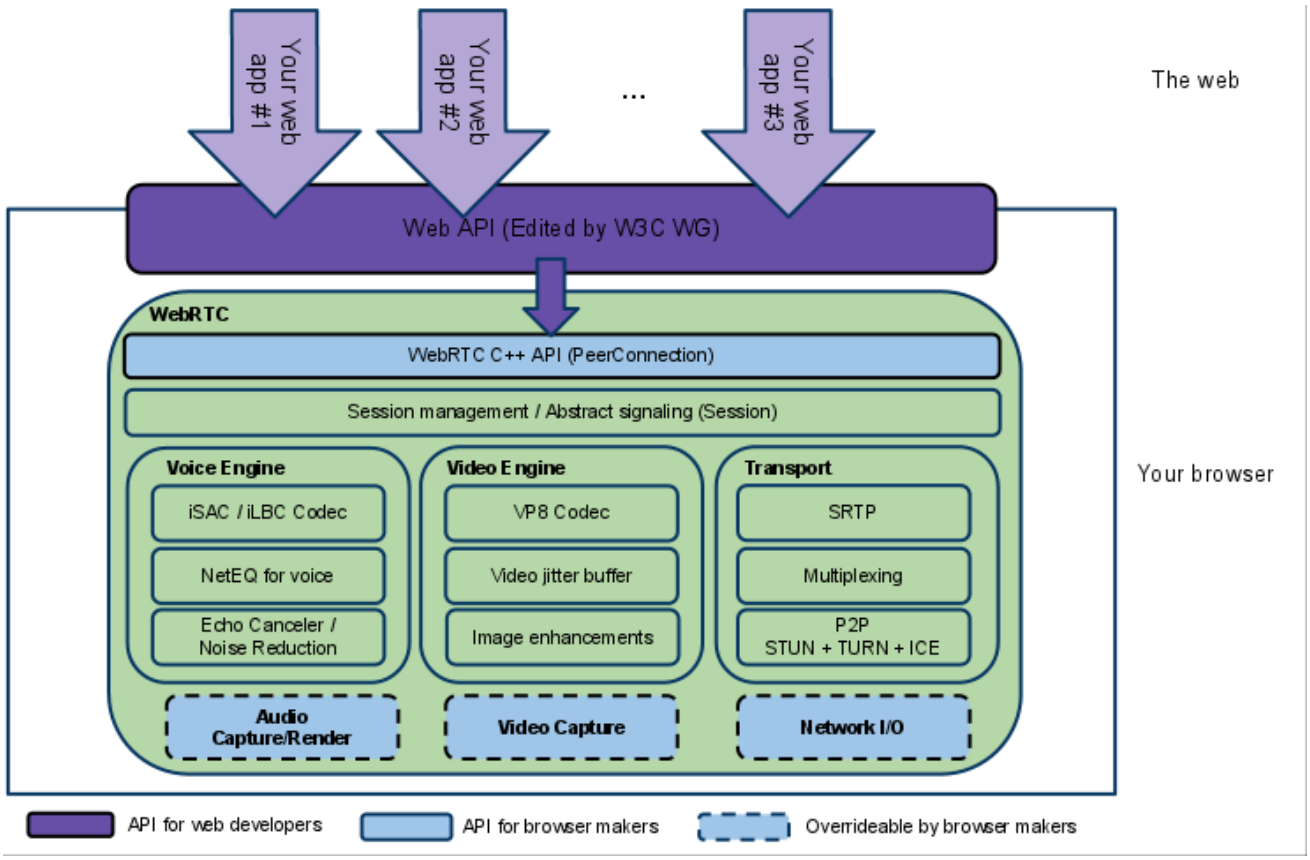


Figure 3: WebRTC inside a browser. [5]

WebRTC was chosen for use in the LAN-Communicator application because of some of its benefits which are listed below:

- *It is an open standard currently backed by the Internet Engineering Task Force - Real-Time Communication in Web-browsers (IETF-RTCWEB) Active Working Group, the World Wide Web Consortium (W3C), Google, the Mozilla Foundation, Microsoft, and hopefully Apple in the nearest future* [6]
- *It is free for developers to use (no need to pay royalties)* [6]
- *It has a simple and easy to understand JavaScript API which enables quick application development in comparison to other APIs for other platforms / frameworks used to build Real-Time-Communications (RTC) enabled applications* [6]
- *All that is needed on the end-user's side is a web-browser that supports the WebRTC standard (no plugins or extra software needs to be installed by the end-users). The Operating System the browser is running on is not relevant.* [6]
- *Connection between peers using the RTCPeerConnection API is encrypted; consequently, resulting in secure audio / video streams* [6]
- Audio / Video / Data transmission between peers is direct no intermediate server needed most of the time (intermediate relay servers may be needed if Network Address Translation (NAT) and firewalls make it impossible to establish direct connection between peers; however, this

problem will not affect the LAN-Communicator application since LAN-Communicator is meant for use by peers within the same subnet in the same LAN)

- The technology is constantly being improved upon (the audio and video quality keeps improving as supporting browsers are updated)
- *It automatically adapts the rate at which audio and video stream data is sent over the network based on the network conditions at each point in time during a WebRTC session* [6]
- *It can also interoperate with existing audio and video communication systems like the Public Switched Telephone Network (PSTN), Extensible Messaging and Presence Protocol (XMPP) based applications, Session Initiation Protocol (SIP) based applications via the use of solutions like FreeSwitch (https://freeswitch.org) and the likes* [6]

With WebRTC being the technology of choice for transporting audio / video streams between peers in the LAN-Communicator application, there still remains the issue of how to get peers to discover each other. *In order for two devices on the same or different network(s) to locate one another, some form of discovery and media format negotiation must take place. This process, called **signaling**, involves both devices connecting to a third, mutually agreed-upon server through which the two devices can locate one another and exchange the needed negotiation messages.* [7] Signaling was deliberately left out of the WebRTC standard in order to give developers, leveraging WebRTC in their applications, more flexibility / freedom in how to implement signaling in their applications. **WebSocket** was chosen as the transport mechanism for signaling in the LAN-Communicator application since it allows bidirectional communication between a web-browser and a WebSocket enabled server (which is the Signaling Server in the LAN-Communicator application's case).

WebSocket enables bidirectional, message-oriented streaming of text and binary data between client and server. It is the closest API to a raw network socket in the browser. A WebSocket connection is also much more than a network socket, as the browser abstracts all the complexity behind a simple API and provides a number of additional services: [8]

- *Connection negotiation and same-origin policy enforcement* [8]
- *Interoperability with existing HTTP infrastructure* [8]
- *Message-oriented communication and efficient message framing* [8]
- *Subprotocol negotiation and extensibility* [8]

Implementation Description

The LAN-Communicator application is based on the concept of meeting rooms and participants. Each meeting room can have a maximum of four participants. A user creates a meeting room which can contain a minimum of one or a maximum of three other participant(s). The creator of the room is automatically the first participant in the room. Each user that joins a meeting room automatically sends RTCPeerConnection (peer connection) offers to each of the other participants in the meeting room in order to establish individual peer connections with each of the members of the meeting room. The joining participant will send his/her own video stream over each peer connection and will also simultaneously receive the video streams from each of the other meeting room members over each peer connection. When a participant leaves a meeting room, the peer connection(s) between the

departing participant and other participants (still in the meeting room) is / are closed and the video for the departing participant is removed from each of the other participants' screen(s).

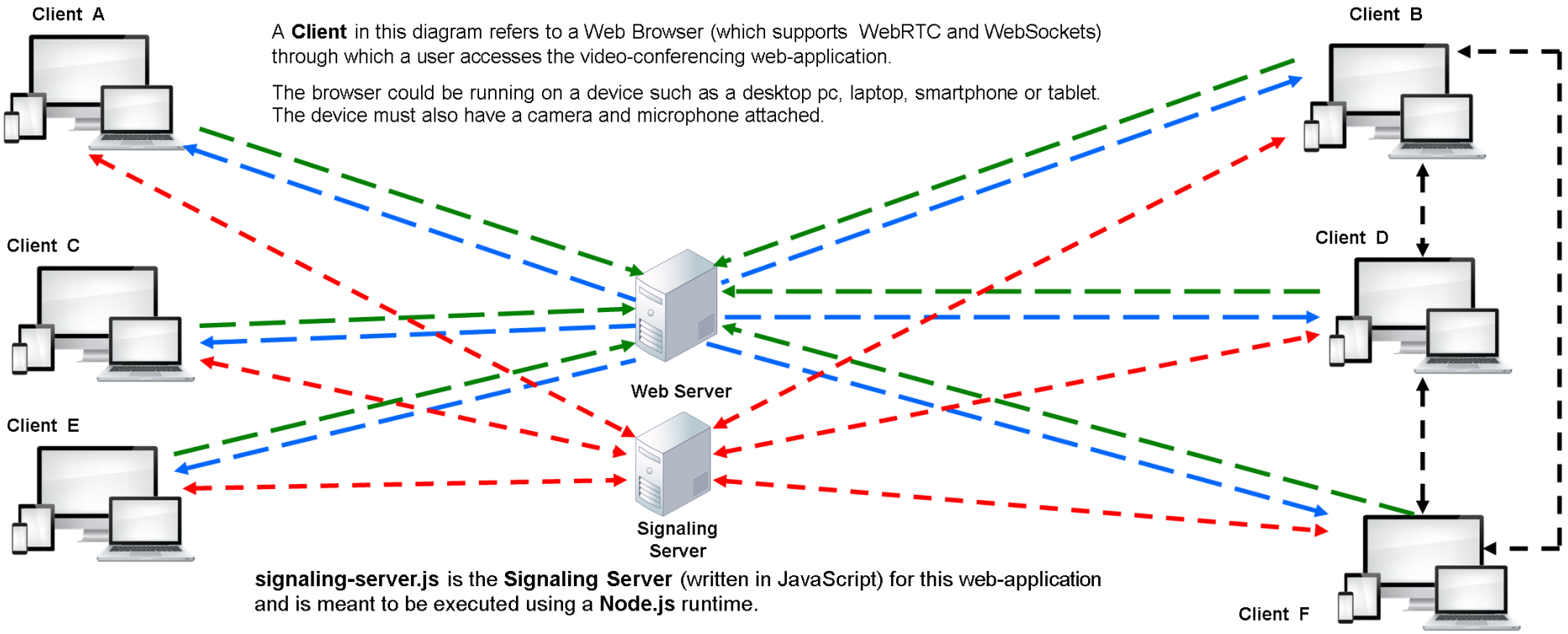
Architecture

The LAN-Communicator application is made up of the following key components:

- a web page (**index.html**) containing the user interface (UI) elements to be displayed on the screen of the user running the application via a web-browser,
- two client-side JavaScript files associated with index.html:
    - **adapter.js:** a library developed by the official WebRTC organization that fixes inconsistencies in different browsers' WebRTC implementations, and
    - **browser-client.js:** all the client-side logic developed for this application.
        - It controls what gets displayed on screen based on user interactions with index.html's UI elements and messages received from the Signaling Server (e.g. when a user clicks on the **Create New Meeting Room** button, this script causes the **Create New Meeting Room** form to be displayed. See Figs. 5 and 6).
        - It also sends messages to the Signaling Server based on user input or user interactions with index.html's UI elements. User input is also validated when necessary (e.g. when a user tries to join a meeting room, the script ensures that the supplied participant name contains at least a character value and is not already in use by another participant in the room)
- a Signaling Server (**signaling-server.js**) written in JavaScript meant to be run via **Node.js** (*a JavaScript runtime built on Chrome's V8 JavaScript engine.* [10]). This script contains all the server-side logic developed for this application.
    - this server exists to setup and keep track of meeting rooms and associated participants. It manages the creation of the meeting rooms and stores the data for each created meeting room. It also manages the addition and removal of participants to and from created meeting rooms. Peer connection setup messages (i.e. Offers, Answers and Candidates) are also forwarded between participants in a meeting room via this server. These peer connections, upon successful setup, are directly between the participants as indicated in Fig. 4.
    - **browser-client.js** connects to this server via the WebSocket Secure (WSS) protocol
    - the **ws** (https://github.com/websockets/ws) WebSocket library (a Node.js library that implements the WebSocket Protocol: RFC-6455) was used to implement all WebSocket related functionality in this server. No WebSocket library is needed on the client side since the WebSocket API is implemented in most browsers.

- a web-server to serve the *index.html* and accompanying client-side JavaScript files to each client's web-browser (any web-server that supports https can be used).

Fig. 4 below is a high level overview of the LAN-Communicator application's architecture.



Client ⟷ Web Server

1.) **Client** sends an http GET request for **index.html** and 2 extra http GET requests for **adapter.js** & **browser-client.js** which are both linked to from within **index.html**.

Client ⟷ Web Server

2.) Web Server returns **index.html**, **adapter.js** & **browser-client.js** back to the **client**

Client ⟷ Signaling Server

3.) A **WebSocket** connection between a **client** and the **Signaling Server**. This is a full duplex connection that can be used to send data back and forth between the **client** and the **Signaling Server**. The establishment of this connection is initiated by the **client** when it executes the **browser-client.js** script upon receiving it from the **Web Server**. This connection is used to exchange meeting room data between each **client** and the **Signaling Server.** The **Signaling Server** also acts as an intermediary that forwards WebRTC video-session setup data between **clients**.

Client X ⟷ Client Y

4) An **RTCPeerConnection** between **Client X** and **Client Y**. This is a full-duplex connection that can be used to send audio / video data back and forth between **Client X** and **Client Y**. The establishment of this connection is initiated by the **client** joining an existing meeting room. The joining **client** sends a WebRTC Offer through the **Signaling Server** to each existing member of the meeting room it's trying to join. When each non-joining client receives the offer, they each send a WebRTC Answer back to the joining **client** via the **Signaling Server**. Finally, the joining client proceeds to exchange WebRTC candidate messages with each of the other non-joining client(s) via the **Signaling Server** which eventually ends with the joining client establishing a peer to peer **RTCPeerConnection** with each of the other non-joining client(s) in the desired meeting room.

Figure 4: Overall architecture of the **LAN-Communicator** application.

In Fig. 4 above, ***Clients B, D*** and ***F*** are in video-conference session. Each client maintains one peer connection with each of the other clients in the video-conference session. For example, *Client B* maintains a peer connection with *Client D* and another peer connection with *Client F*. In a video-conference session containing **N** participants, each participant maintains *N-1* peer connections to the other *N-1* participants. From a programming perspective, each client will create *N-1* instances of the **RTCPeerConnection** class (each instance represents a peer connection to another client).

On the client side of this application, the **Single-Page Application (SPA)** architecture is employed. A *single-page application is a web-application that fits on a single web page with the goal of providing a more fluid user experience similar to a desktop application*. *In a SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions.* [9] In the case of this application the client only loads one html file (**index.html**) and two JavaScript files (i.e. **adapter.js** and **browser-client.js**) referenced in **index.html**. Once these files are loaded, a WebSocket connection is established with the Signaling Server via JavaScript code inside **browser-client.js** and from then on messages are exchanged between the browser and the Signaling Server which lead to portions of the web- page displayed on the screen being updated. The WebSocket API makes it possible for a web-browser to connect to a WebSocket server and then send and receive messages to and from the WebSocket server over the established connection. The WebSocket connection gets closed when a user navigates to another site or closes the browser tab or window used to access the application.

The maximum number of participants allowed in a meeting room can be easily increased to a higher value by increasing the value of the **MAX_NUM_PARTICIPANTS_IN_ROOM** constant in the Signaling Server script (signaling-server.js) and the same constant in the client side script (browser-client.js). This value can be decremented or incremented to allow for optimal video-conferencing performance relative to the capacity of the LAN the LAN Communicator web-application is being deployed in.

All messages exchanged between the clients and the Signaling Server are in JSON (JavaScript Object Notation) format. The **_sendToClient** (a function in signaling-server.js for sending messages via a WebSocket to a connected client) and **_sendToServer** (a function in browser-client.js for sending messages via a WebSocket to the Signaling Server) functions are used for sending messages at the server and client side respectively. These functions convert each message into JSON format before sending. Upon reception at either the client or the server side, the JSON encoded message is converted to a JavaScript object (via the **_parseRecievedMsg** present in both browser-client.js and signaling-server.js) that can be easily inspected to retrieve the actual data embedded inside the received message.

Each message is encoded in the JSON format with the following **three fields** when either the _sendToClient or _sendToServer function is called:

1. **content**: this field is where the data associated with the message is stored.

2. **time_sent**: it represents the exact time (in **YYYY-Mmm-dd hh:mm:ss** format) a message was sent by the client or server. It is set by _sendToClient or _sendToServer function.
3. **type**: this field indicates the type of message being sent. It dictates how a message will be processed by the recipient (i.e. client or server). Below are the message types that were defined for the LAN-Communicator application in the **LAN_COMMUNICATOR_MSG_TYPES** object present in both signaling-server.js and browser-client.js:

    - **INITIALIZE_M_ROOMS**: this type of message is sent only once by the server to the client immediately after a WebSocket connection has been successfully established. Its corresponding content is an object representing all existing meeting rooms. When a client receives this type of message, it updates the screen with the meeting rooms (if any) or displays a **No Meeting Rooms Exist** message on the screen.
    - **TAG_CLIENT_WEBSOCKCONN_WITH_ID_FROM_SVR:** this type of message is also sent only once by the server to the client immediately after the establishment of a successful WebSocket connection. Its corresponding content is a unique integer value (which will be referred to as a **sig_serv_conn_id** in the rest of this report) assigned by the server to each connected client. When a client receives this type of message, it updates the WebSocket connection object on the client-side with the value.
    - **CREATE_M_ROOM:** this type of message is sent by a client to the server when the client wants the server to create a new room. Its corresponding content is an object containing the name of the room to be created and the name the client will be identified by in the room to be created. When the server receives this message, it creates a new room and adds the creator as the first participant.
    - **ADD_NEW_ROOM_TO_CLIENT_UI:** this type of message is sent by the server to all clients connected to it when a new meeting room has been successfully created by the server. Its corresponding content is an object containing information about the new room. Each client uses the content of this message to add the new room to its screen.
    - **LEAVE_ROOM:** this type of message is sent by a client to the server when the client wants to leave an existing meeting room. Its corresponding content is an object containing the identifier for the room to be departed from and the name the client that is departing.
    - **UPDATE_LEFT_ROOM_ON_CLIENT_UI:** this type of message is sent by the server to all clients connected to it when the server has successfully removed a participant from an existing meeting room in response to a previous LEAVE_ROOM message or disconnection of a client (that was a participant in a meeting room) from the server. Its corresponding content is an object containing information about the room a participant just departed from. Each client uses the content of this message to update the list of participants in the room that a participant just departed from on its screen.
    - **DELETE_ROOM_FROM_UI:** this type of message is sent by the server to all clients connected to it when the server has successfully removed the last participant from an existing meeting room in response to a previous LEAVE_ROOM message or disconnection of a client (that was the last participant in a meeting room) from the

server. When the last participant departs from a room, the server also deletes the room. Its corresponding content is an object containing information about the deleted room. Each client uses this message's content to remove the deleted room from its screen.

- **JOIN_ROOM:** this type of message is sent by a client to the server when the client wants to join an existing meeting room. Its corresponding content is an object containing the identifier for the room to be joined and the name the client has chosen to be identified by within the room.
- **UPDATE_JOINED_ROOM_ON_CLIENT_UI:** this type of message is sent by the server to all clients connected to it when the server has successfully added a participant to an existing meeting room in response to a previous JOIN_ROOM message. Its corresponding content is an object containing information about the room a participant just joined. Each client uses the content of this message to update the list of participants in the room that the participant just joined on its screen.
- **WEBRTC_OFFER:** this type of message is sent by a participant (that just joined a meeting room) to the server. Its content is a peer connection **Offer** and the identifier for another participant in the room that the offer is to be forwarded to. The client sends this message through the server to each of the participants in the room it just joined. It should be noted that a unique offer is sent to each participant. When the server receives this type of message, it locates the WebSocket connection object for the intended recipient and then forwards the message to the recipient via the located connection. This is the first phase of the signaling process that will eventually lead to the establishment of peer-to-peer connections between the participants in the meeting room.
- **WEBRTC_ANSWER:** this type of message is sent by a participant in response to a WEBRTC_OFFER message (forwarded to it via the server) on behalf of another participant. Its content is a peer connection **Answer** and the identifier for another participant in the room that the answer is to be forwarded to. The client sends this message through the server back to the participant that previously sent a WEBRTC_OFFER message (forwarded via the server). When the server receives this type of message, it locates the WebSocket connection object for the intended recipient and then forwards the message to the recipient via the located connection. This is the second phase of the signaling process that will eventually lead to the establishment of peer-to-peer connections between participants in a meeting room.
- **WEBRTC_CANDIDATE:** this type of message is sent by a participant to another participant (in the same room) that it had previously sent a WEBRTC_OFFER or WEBRTC_ANSWER message to. Its content is a peer connection **Candidate** (which contains connectivity information like the sender's IP address, a User Datagram Protocol (UDP) port that the sender is willing to accept incoming connections on, etc.) and the identifier for the other participant in the room that the candidate message is to be forwarded to. When the server receives this type of message, it locates the WebSocket connection object for the intended recipient and then forwards the message to the recipient via the located connection. Participants continue to exchange these

candidate messages with each other (via the server) until peer connections are successfully established between the participants. Once the peer connections are established, the participants begin and continue to send and receive audio / video streams between one another (the video-conference is now in progress at this point and the signaling server is not involved in the movement of media traffic between peers. The WebSocket connection between each participant and the Signaling Server still exists and will be used to send LEAVE_ROOM messages to the server when participants are ready to leave the meeting-room).

- **GENERIC:** this type of message has no meaning or special logic associated with it and could be used for debugging purposes (to test if clients can successfully send a message to the server or vice versa).

More Signaling Server Logic

When a client successfully makes a WebSocket connection to the Signaling Server, the server assigns a unique sig_serv_conn_id value to the connected client. The sig_serv_conn_id is used for retrieving each client's WebSocket connection object on the server-side. The server also sends the sig_serv_conn_id back to the client immediately after it is assigned. The browser-client.js script on the client-side stores the sig_serv_conn_id value in the client-side WebSocket connection object. As a result, both the server and each client are aware of the sig_serv_conn_id value associated with each ongoing WebSocket connection. For example, assuming the Signaling Server assigned Client F a sig_serv_conn_id value of 4, that value will be sent by the server to Client 4 and the browser-client.js running on Client 4's browser will update Client 4's WebSocket connection object with the sig_serv_conn_id value of 4.

Immediately after the successful establishment of a WebSocket connection to the server, the server sends a message containing information about existing meeting rooms to the client. When the client receives this message, it updates the screen with the list of existing meeting rooms and associated participants or displays a message on the screen to indicate that no meeting rooms exist if the message was empty (meaning that the server has not yet received any request from any client to create a meeting room or the previously existing meeting rooms no longer contain participants. The server gets rid of a meeting room the moment the last participant leaves the meeting room). It should be noted that meeting room data is stored in memory on the Signaling Server and once the server is restarted all previously existing rooms are gone and the server starts fresh with no meeting rooms. When a meeting room changes (i.e. a participant joins or leaves), the server broadcasts the updated room's data is to all connected clients and each client updates its screen with the updated data.

Meeting Room Object	
room_id	server assigned unique integer value
room_name	client supplied string value
participants	an array of participant objects

Table 1: The Meeting Room Object Schema (Signaling Server)

Participant Object	
sig_serv_conn_id	server assigned unique integer value associated with the participant's WebSocket connection object
participant_name	client supplied string value
is_creator	a Boolean value that is set to true if the participant is the creator of its associated meeting room or false if not

Table 2: The Participant Object Schema (Signaling Server)

Figs. 5 to 17 show how the application works. There are initially no meeting rooms. A meeting room is then created and gradually filled up with participants until it becomes full and a fifth client is unable to join the full room.

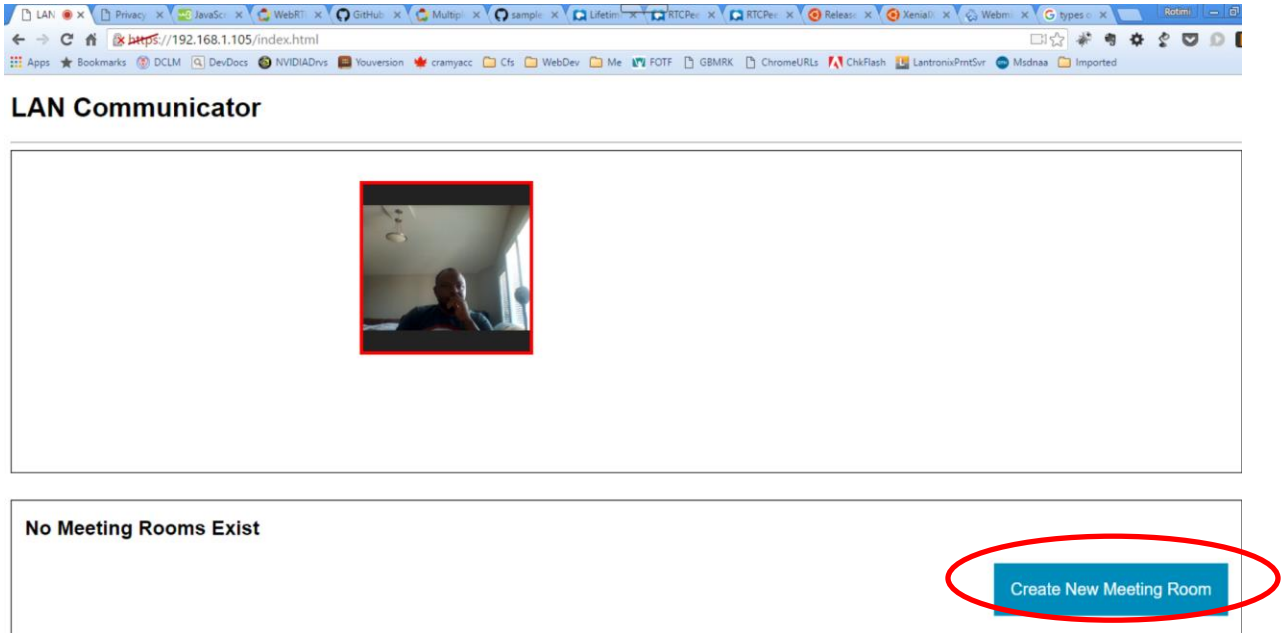


Figure 5: 1st client (with a Chrome Browser on a Windows PC with 2 Webcams) is the first to access the web-application after a fresh start-up of the signaling server (no meeting room exists at this point).

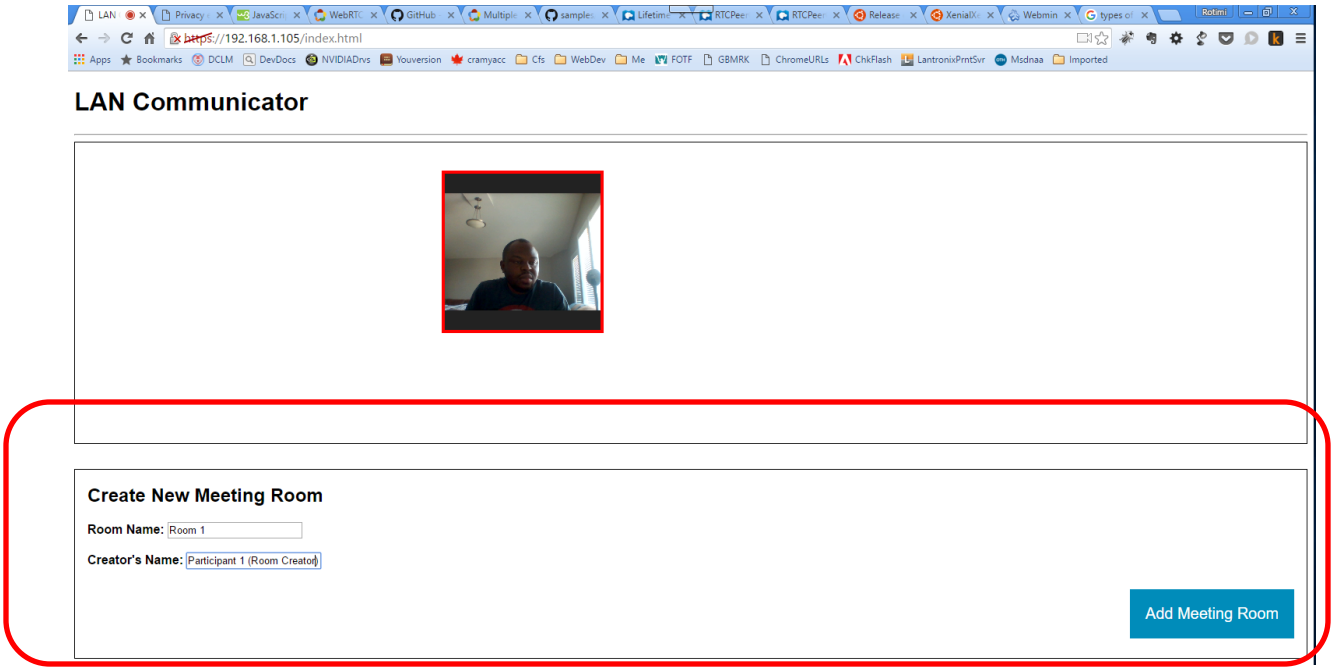


Figure 6: 1st client (with a Chrome Browser on a Windows PC with 2 Webcams) creates a meeting room named ***Room 1*** with a display name of ***Participant 1 (Room Creator)*** in ***Room 1***.

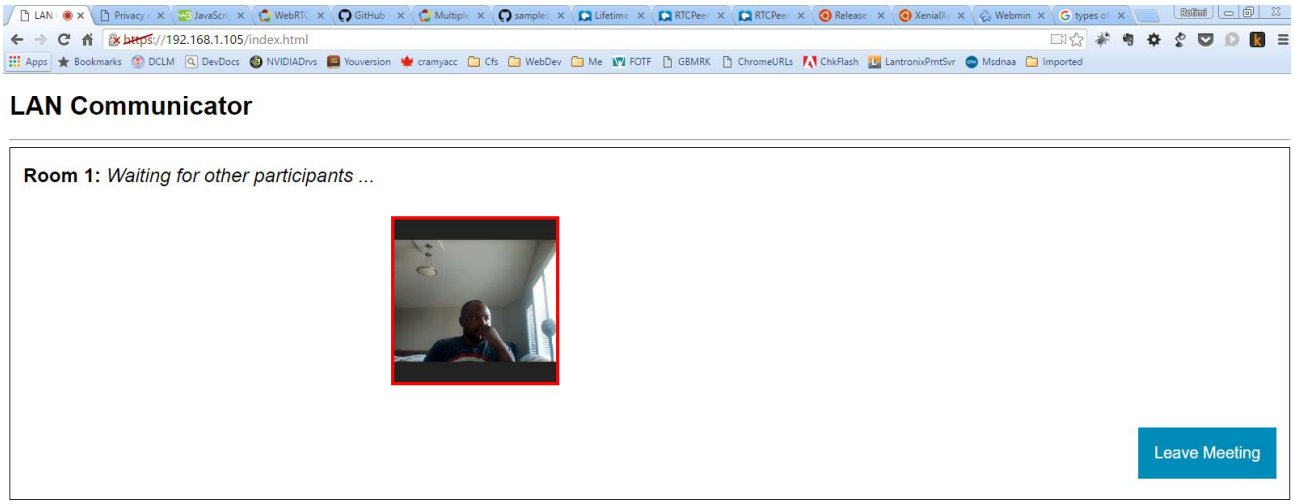


Figure 7: 1st client (with a Chrome Browser on a Windows PC with 2 Webcams) successfully created the meeting room named ***Room 1*** and is waiting for other participant(s) to join the meeting room.

Figure 8: 2nd client (with a Firefox Browser on a Windows PC with 2 Webcams) accesses the web-application after the 1st client has created ***Room 1***.

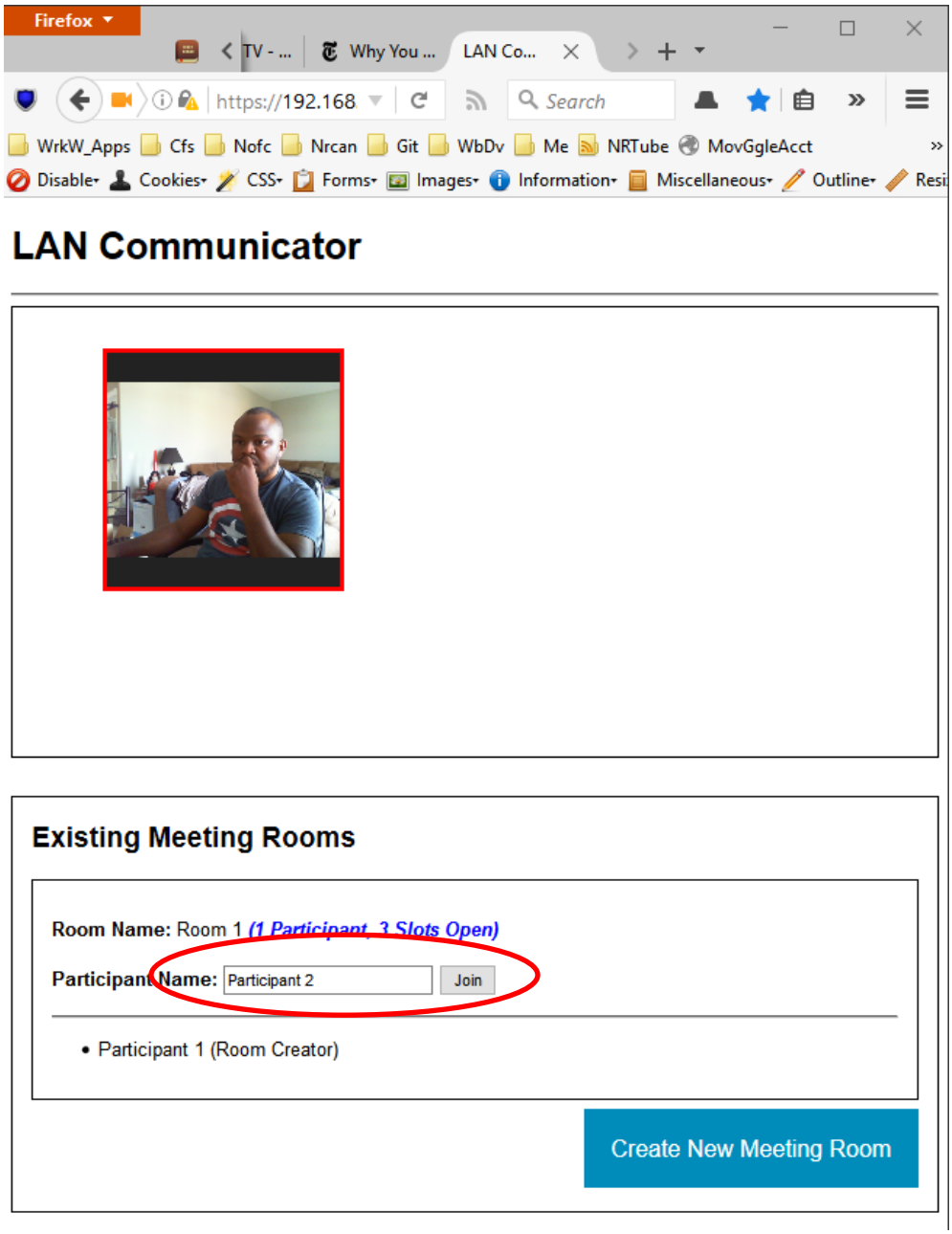


Figure 9: 2nd client (with a Firefox Browser on a Windows PC with 2 Webcams) enters a display name of ***Participant 2*** for ***Room 1*** and will join ***Room 1*** after clicking the join button.

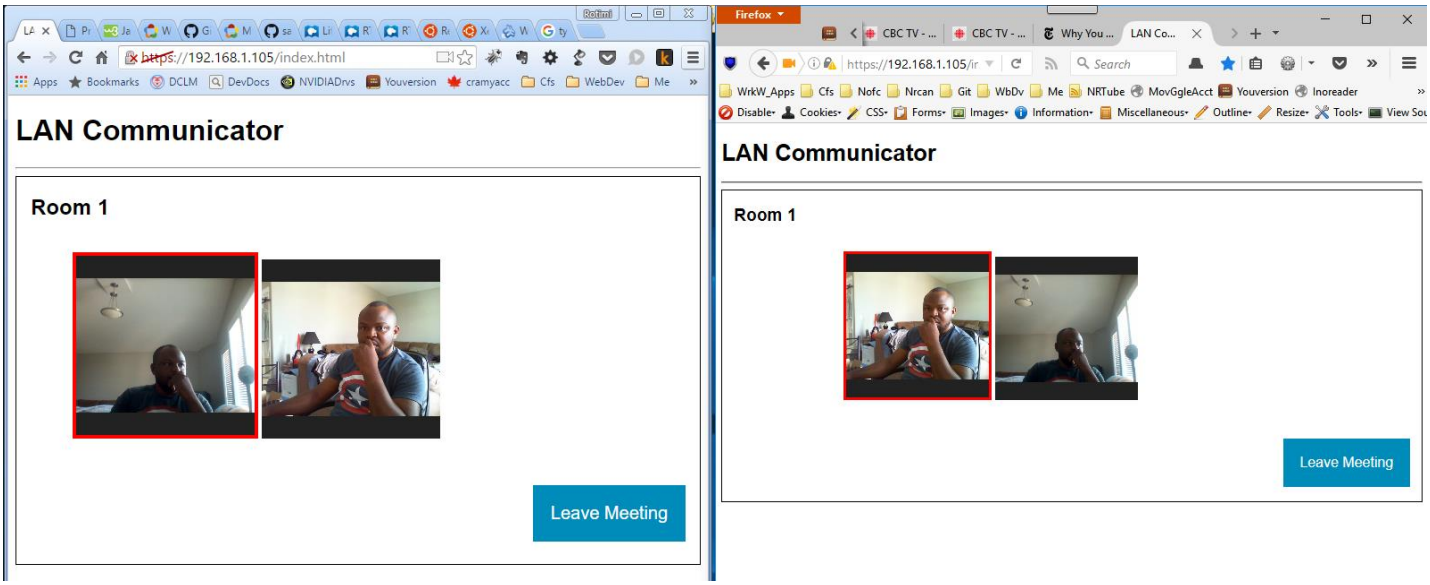


Figure 10: *Clients 1* and *2* are in a video-conference session in ***Room 1*** (from the perspectives of *Clients 1* and *2*).

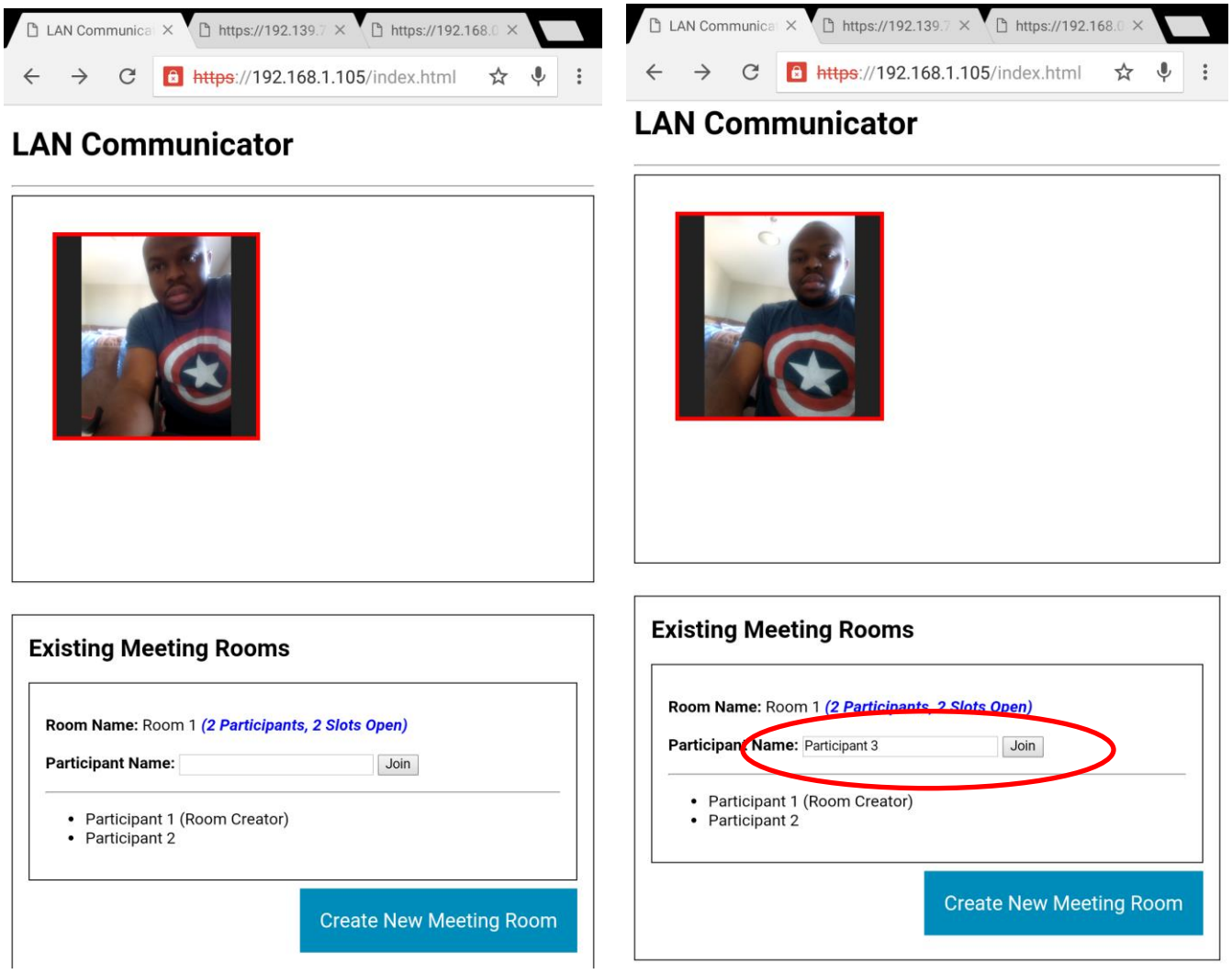


Figure 11: 3rd client (with a Chrome Browser on an Android Tablet - NVidia Shield) in the process of joining ***Room 1*** with a display name of ***Participant 3***.

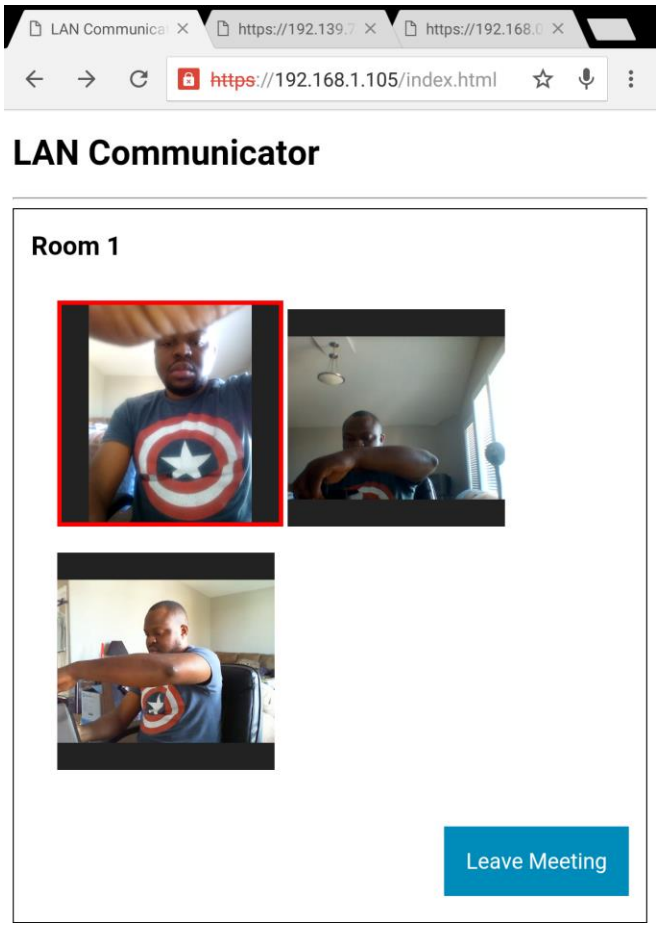


Figure 12: 3rd client (with a Chrome Browser on an Android Tablet - Nvidia Shield) successfully joined ***Room 1*** with a display name of ***Participant 3*** and is now in a video-conference session with Clients 1 and 2.

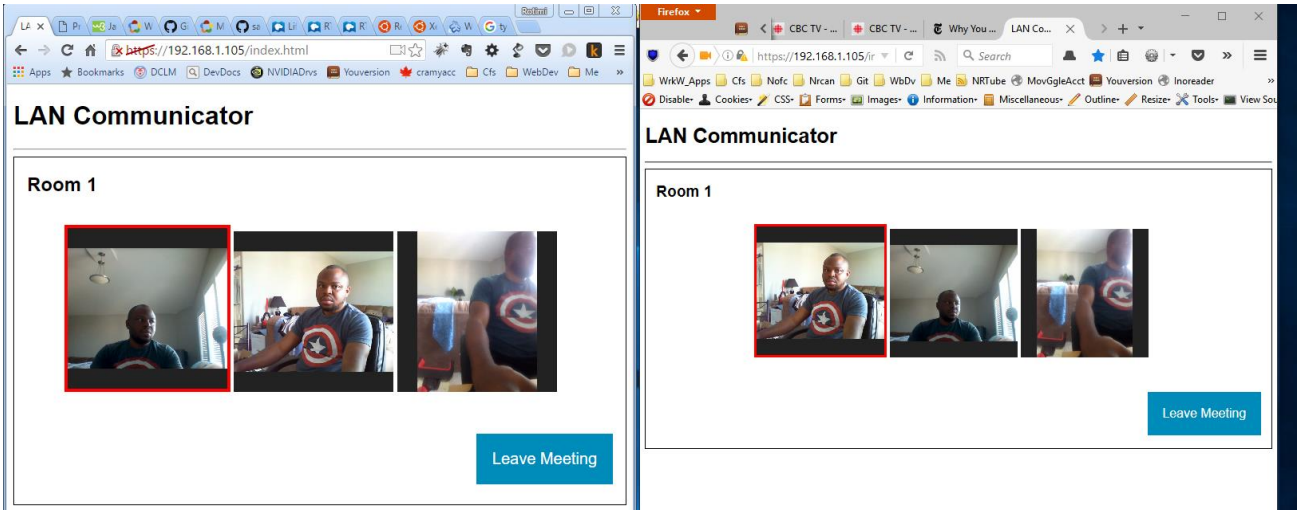


Figure 13: *Clients 1, 2 & 3* are in a video-conference session in ***Room 1*** (from the perspectives of Clients 1 and 2).

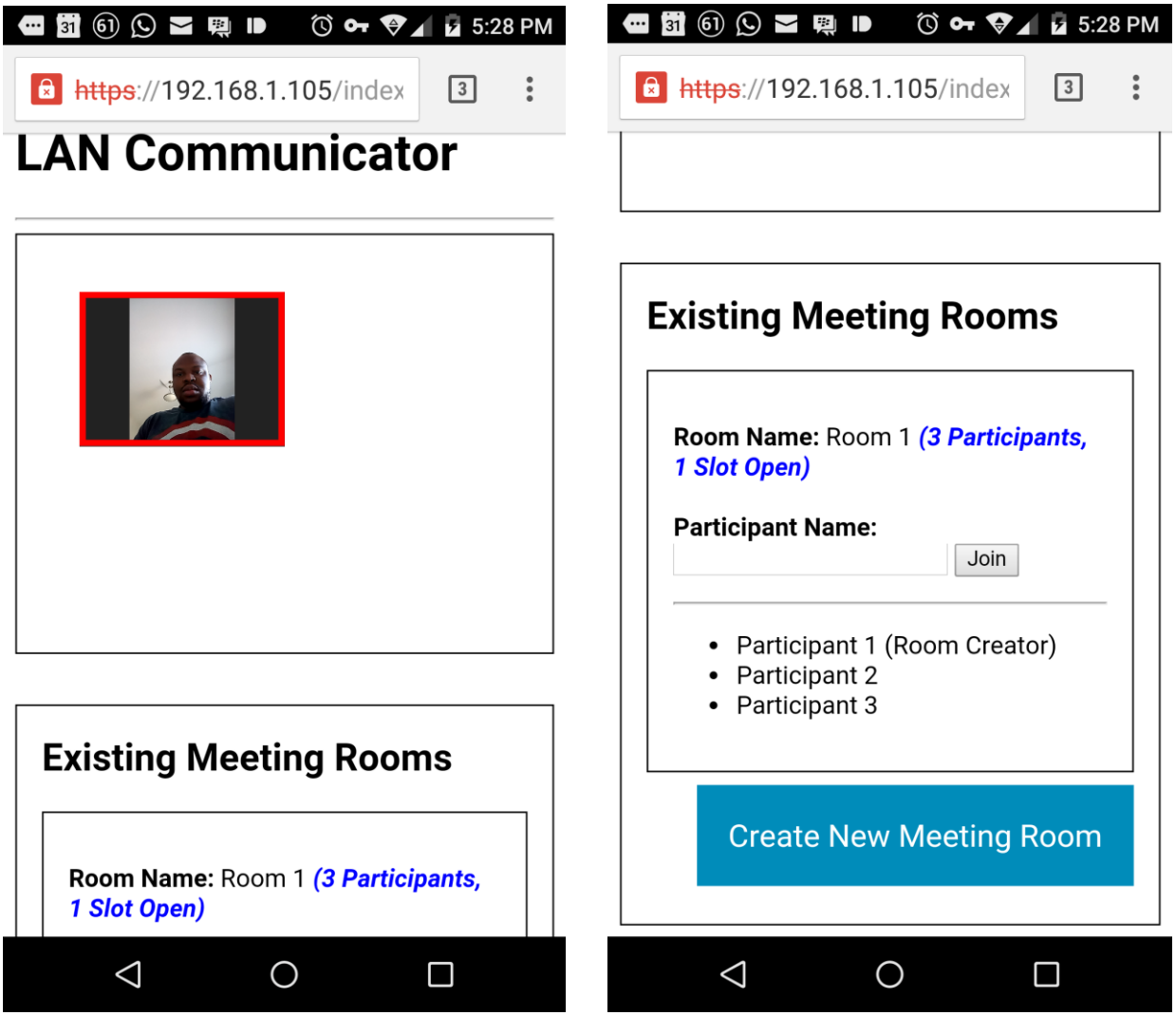


Figure 14: 4th client (with a Chrome Browser on an Android Phone - Blu-Life One X) in the process of joining ***Room 1*** with a display name of ***Participant 4***.

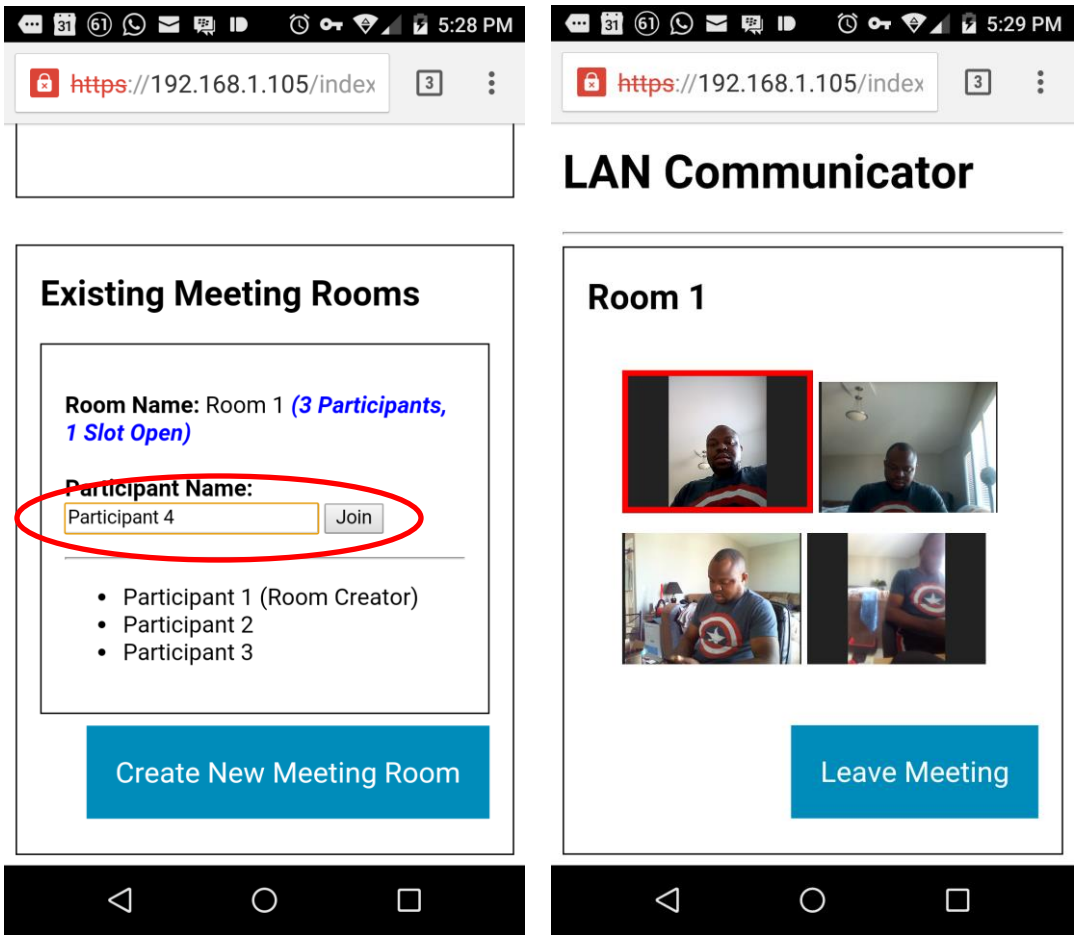


Figure 15: 4th client (with a Chrome Browser on an Android Phone - Blu-Life One X) joined ***Room 1*** with a display name of ***Participant 4***.

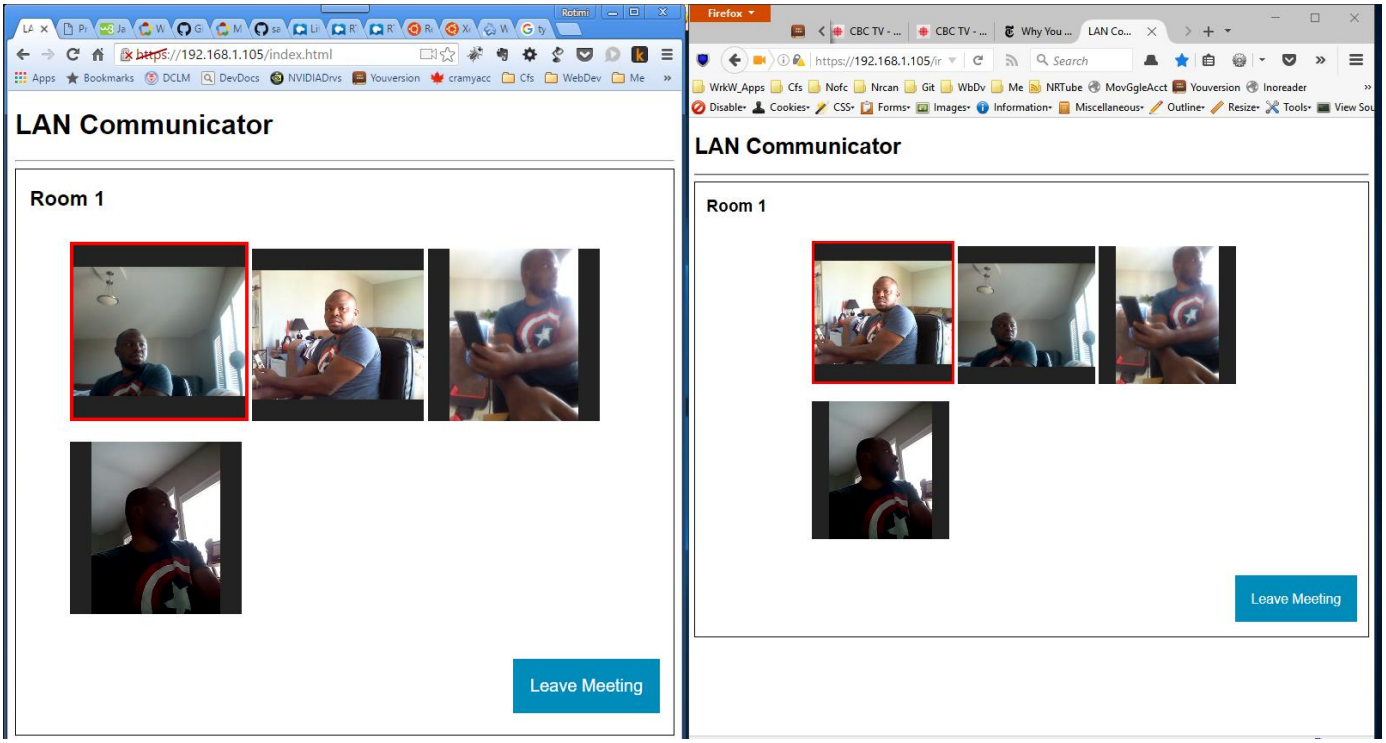


Figure 16: Clients 1, 2, 3 & 4 are in a video-conference session in ***Room 1*** (from the perspectives of Clients 1 and 2).

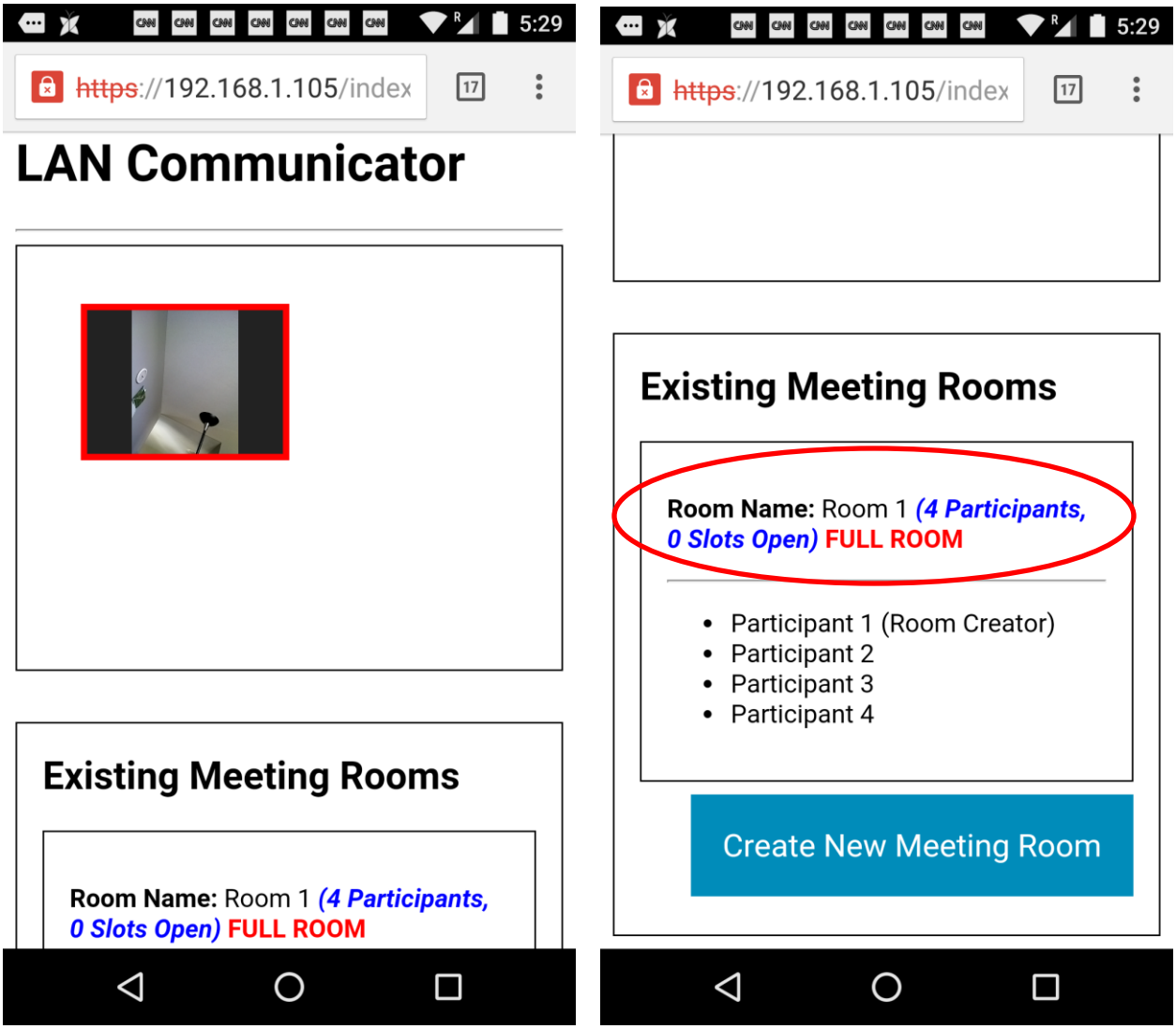


Figure 17: 5th client (with a Chrome Browser on an Android Phone - Nexus 5) accesses the web-application while Clients 1, 2, 3 & 4 are in a video-conference session and sees a ***FULL ROOM*** message in the ***Room 1*** section of the screen.

Concluding Remarks

The LAN-Communicator application described in this report can be further enhanced via the addition of features like instant-messaging (which could be implemented via the use of the RTCDataChannel API), password protectable rooms, etc. The application can also be re-architected to work across two or more LANs across the internet (e.g. across multiple locations of an organization). To reduce and optimize bandwidth consumption a Multi-Point Conferencing Unit (MCU) like **Licode** [11] could be introduced into the application. This MCU will mix the audio/video streams and will lead to each client in a video-chat session maintain only one peer connection instead of **N-1** connection(s) with all the other clients in a chat session (where **N** is the number of clients in a session). Fig. 18 below illustrates how an MCU will fit into the system:

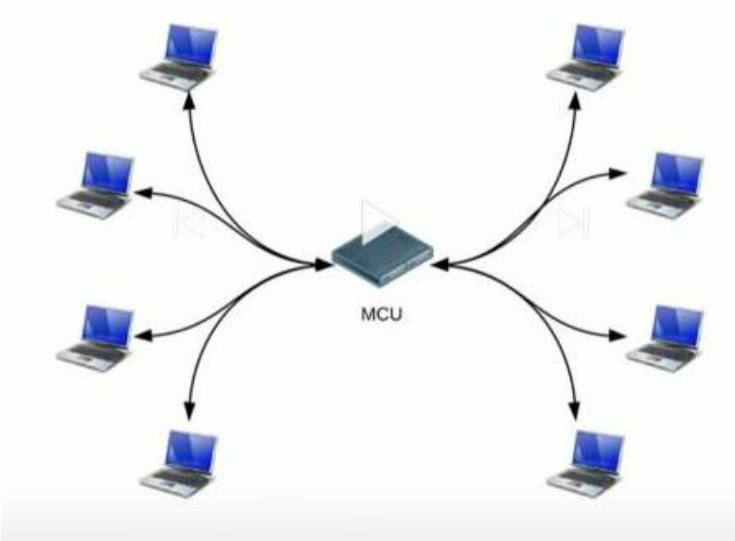


Figure 18: Video-conference session managed by an MCU

WebRTC is definitely a promising technology that has the potential of revolutionizing Internet-based communications. It has made it easier for developers to be able to build RTC enabled web-application via three simple JavaScript APIs, mentioned earlier in this report. As the technology matures and all the major browser vendors implement it in their various browsers, individuals and companies that were previously skeptical about its viability will hopefully be more confident to adopt it in their web-applications.

References

[1] Smedberg, B. Reducing Adobe Flash Usage in Firefox, July 20 2016. Retrieved July 26 2016 from Mozilla Future Releases Blog: https://blog.mozilla.org/futurereleases/2016/07/20/reducing-adobe-flash-usage-in-firefox/

[2] Grigorik,I. WebRTC Browser APIs and Protocols, Chapter 18. Retrieved July 26 2016, from High Performance Browser Networking (Copyright © 2013): https://hpbn.co/webrtc/

[3] Mozilla Developer Network (MDN): RTCPeerConnection, last updated Jul 11, 2016. Retrieved July 27 2016, from Mozilla Developer Network (MDN): https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection

[4] Mozilla Developer Network (MDN): RTCDataChannel, last updated Jul 11, 2016. Retrieved July 27 2016, from Mozilla Developer Network (MDN): https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel

[5] Official WebRTC Website: Architecture. Retrieved July 27 2016, from the Official WebRTC Website: https://webrtc.org/architecture/

[6] Audin, G. 9 Advantages Of WebRTC, last updated January 3 2014. Retrieved July 27 2016, from InformationWeek Network Computing: http://www.networkcomputing.com/unified-communications/9-advantages-webrtc/1953259845

[7] Mozilla Developer Network (MDN): Signaling and video calling, last updated Jul 5, 2016. Retrieved July 27 2016, from Mozilla Developer Network (MDN): https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling

[8] Grigorik,I. WebSocket Browser APIs and Protocols, Chapter 17. Retrieved July 27 2016, from High Performance Browser Networking (Copyright © 2013): https://hpbn.co/websocket/

[9] Wikipedia: Single-page application, last updated Jul 4, 2016. Retrieved July 28 2016, from Wikipedia https://en.wikipedia.org/wiki/Single-page_application

[10] Official Node.js Website. Retrieved July 27 2016, from the Official Node.js Website: https://nodejs.org/

[11] Official Licode Website. Retrieved July 30 2016, from the Official Licode Website: http://lynckia.com/licode/