# Overview:

The primary objective of our TelePresence Robot System is to aid the doctors to remotely monitor in-patients through telepresence robots located in the hospital. The first 30 days of the development was focussed on grasping the technologies to be used, identifying their pros and cons. It comprised of getting an understanding about how to use different APIs and tools for developing applications and develop a basic prototype for our project.

In this month we started with the real development utilizing the technologies we identified in the previous month. Application built currently has the functionality that could stream audio/video and exchange data between two peers (which in our case are doctor and robot). We are using a website in doctor's side and an android app in robot's side. In the following report we describe about the application we built, illustrating its usability and functionality. We mention the technical hurdles faced during the process of development and how we resolved them and technologies used in building the final product (software). We further specify about various technical details related to the features in the application and the procedure describing how doctors can utilize it to communicate with the robot and later with the patient.

**Technical Challenges Faced and Goals Accomplished during this month:**

Our initial plan was to develop an Android application for the doctor and the robot side. Through this application the robot's view could be remotely streamed to the application on the doctor's side. Through the stream received at the doctor side the doctor could further steer the robot through the application at its end. Also the doctor's view would be remotely streamed to the robot side which can be viewed by the nurse or some patient.

For developing our planned application prototype we thought to use the **Web View** support available in the Android APIs. Web View is a technology that lets you render HTML pages inside an Android application. However WebView APIs currently are in their initial stage and are in development mode. It currently is not capable of handling complex JavaScript snippets and referring external CSS files too.

Also for the streaming of the view from camera at one peer to the peer at the other end, we need to gain access to the camera on the device. We tried to access the device camera by embedding some simple JavaScript code in the HTML page to be rendered in Web View. However Web-View being in its novice stage, currently does not support getting access to the hardware modules in the device. Hence due to this one cannot get control over the microphone and the camera through Web View, which are required to stream audio and video, respectively. We build a sample Android application with the functionalities to fetch audio and video, however due to the limitations in the Web-View presently we were unsuccessful in accomplishing our goals.

In order to move ahead, we finalized to develop a Web Application for the doctor's side and an Android application on robot's side. Application would be hosted on a server and can be accessed by the users (Doctor and the Robot) through HTTP URL in the browser and a HTTP request in Android. Doctor's side of the application can run on any device with any form factor (Laptop, Desktop, Tablets, Smartphones, and also embedded devices) just with a basic web browser installed in them.

In this month we have built a full-fledged system where in any two peers could communicate with the other through **audio/video** as well as through **data (normal chat/text data)** transfer. In our context the peers are robots and doctors. Also we have developed a steering for the application at the doctor side and we are successful in transmitting the steering actions from the doctor to the robot. Steering actions are received at the browser on the robot side which can further used to instruct the hardware modules to make required movements.

Server side is built using **Node.js** which is based on JavaScript. It is highly scalable in comparison to the traditional technologies such as PHP/JSP and is capable of handling large number of multiple concurrent requests/connections. We tested the doctor' part of the application on laptop and the robot part of the application installed on a android phone. It worked well across this platform.

## Technologies Used

In the following section we describe about the major technologies, libraries and APIs used in the development of the project. They are listed as below:

**1. WebRTC:**

WebRTC (RTC stands for Real Time Communication) is an API (Application Programming Interface) that provides facilities for building applications with real time communication capabilities. It is a technology which enables audio/video streaming and data sharing in real time between any two peers or clients. RTC applications could be built for **desktop / mobile / IoT** platforms that would allow two browsers to communicate with each other in real time.

One potential advantage of WebRTC is that the users of the application need not have to install external or internal plugins and extensions for audio and video communication. WebRTC is currently supported for all of the modern browsers:

1. Google Chrome
2. Apple Safari
3. Opera Mini
4. Mozilla Firefox for both desktop and mobile platforms.

WebRTC has three major components:

**a.getUserMedia** which allows browser to access webcam and microphone
**b.RTCPeerConnection**, which sets up audio/video calls
**c.RTCDataChannel**, which allows peer to peer data exchange

- PubNub API:PubNub WebRTC API will perform signaling between your users to allow them to connect with a RTCPeerConnection. From there you can use the PubNub API to enhance your peer application with features such as presence and history

**2. Node.js:**

Node.js is an open source API for developing server side applications using JavaScript. JavaScript has being used for a long period of time, primarily at the client side for making Web Pages interactive. It was used for performing background tasks (AJAX requests), which involved communication with a server handling the incoming response and updating only a segment of webpage instead of entire page reload.

Node.js is a revolutionary approach in which JavaScript is used at the server side for building network applications. It is built on Chrome's JavaScript runtime for easily building fast and scalable applications.

One significant benefit of Node.js over traditional server side technologies (viz. JSP, PHP) is its event driven architecture and the non-blocking I/O model. Different call-backs are associated with different events, and call-backs are executed when the concerned event takes place. This makes it lightweight, efficient, highly perfect and scalable for real time data intensive applications. Server handles every request, every database connection and every I/O disk request asynchronously. Being asynchronous means server would not be dealing with a single request, or would not be waiting for the requested DB(database) operation (insert/update/delete) to finish or would not wait for the completion of input output operation on the disk. Instead it follows an event driven model and performs any task at the occurrence of that event through call-back. Thus when a DB operation or an I/O operation is complete, an event is generated, meanwhile the server can accept incoming HTTP request from new clients.

**3. Express:**

A web framework designed to build applications using Node.js. One of the components of the express framework is Express generator. Express generator eases the process of creating a node application. It creates a basic skeleton (folder structure) for any node application comprising of the following files and directories:

- **Package.json** file:

  It comprises of basic information related to any node application. Name of the application, their developers and version number. Along with that it consists of the names of the modules/libraries on which the application is dependent along with the API version of those libraries.

- **app.js:** This is the JavaScript file that contains the code for Node server.
- **/data** directory: It comprises of the Mongo database files.
- **/node_modules** directory:
  Node_modules folder comprises of various directories each of which corresponds to different modules on which the node project is dependent and using it internally. Examples of these modules are Express, Mongo DB, Jade or EJS (HTML rendering engines), Morgan (Logger for logging events occurring at the node server in real time), cookie-parser etc.
- **/routes** directory: It comprises of JavaScript files that handles incoming GET and POST requests coming at the node server. A web application can have different URLs relative to the host application URL. For e.g.: www.trh.com could be the home URL, www.trh.com/login, www.trh.com/signup could be some relative URL to the home URL. JavaScript files in the /routes folder handle how to process the incoming requests coming from different application URLs, session and cookie management.

- **/views** directory: It contains different HTML pages associated with the application.
  However the extension of these files are not .html. There are different rendering engines available for rendering HTML pages in Node.js. Some of them are JADE, EJS etc. and therefore the HTML pages in this folder have file names with extension .jade, .ejs, etc.
- /**resources** directory: This folder comprises different elements associated with HTML pages in the /views directory. It comprises of CSS files, images, and JavaScript code associated with the HTML pages.
- **/bin** directory: It comprises of binary files associated with the libraries of the project and its other elements.

**4. Mongo DB:**

Mongo DB is a document database. It is an open source project and a NoSQL database. Information in the Mongo Database is stored in the form of JSON (JavaScript Object Notation) objects. It supports JSON like documents with dynamic schemas, called BSON. Unlike relational databases, Mongo DB stores the business subject in the minimal number of documents. At times it is more intuitive and easier to work with. In analogy with the MySQL database, tables here corresponds to collections and a row in a table here corresponds to a document in a collection. Thus, collections refer to a group of documents in Mongo DB just the way tables refer to a group of rows in MySQL database.

**5. Socket.IO:**
Socket.IO is a JavaScript library which enables real-time, bi-directional event-based communication between web clients and server. It has two parts: a client-side library that runs in the browser, and a server-side library for node.js. Socket.IO uses the web Sockets protocol and like Node.js, it is event-driven and asynchronous.

**6. JavaScript:**
JavaScript is a programming language and is used extensively on the web

**7. JSON:**
JSON stands for JavaScript Object Notation which is a data-interchange format. All client-server communication in our application uses JSON as the medium of data exchange

8. Android API's:
- Android HTTP client:
- PUBNUB WebRTC Signalling API

# Project Description: Implementation Aspects

**Doctor App**
Each doctor who would be using the application has to perform registration to access the application by providing their personal details. In the current implementation we verify the email address of a doctor during the registration **(signup)** phase. However to add a further layer of security we would be integrating the process to verify the doctors phone number by sending **OTP (One Time Password)** to the phone number given by the doctor during registration. Also the credentials so specified by the

doctor during registration would be then cross checked with the database maintained by the respective hospital.

Once registered and verified a doctor can perform the login operation. After the login the doctor would be redirected to the userHome page of the application. UserHome page is the basic crux of the application. Here a doctor could see all the other peers currently active (using the application and connected to the server.) In our current implementation each doctor could see all other peers currently active, where in these peers could be doctors as well as robots. However we could filter out the list of peers a doctor sees such that the peer list for a doctor comprises only active robots currently available.

Once a peer is picked from the peer list available at the userHome page, a call is initiated to that peer. Doctor has to grant permission to allow the application to use the device camera and the microphone. Similarly the peer at the other end has to grant permission to let the application use device microphone and camera. Node.js server in between acts as an intermediary and performs the task of signalling and session set up between two peers. Once both the peers have granted the permission for microphone and camera and signalling phase is completed by Node.js server, the call establishment procedure is completed. The two peers can communicate with each other through audio and video exchange. Each of the peer receives the remote audio and video stream as well as can see their own local stream. Along with these a chat window is also activated once a communication channel is set between the two peers. Using this chat window two peers can exchange textual data in the form of chat messages.

Also when the call is established a steering is enabled in the application User Interface.

Through these a peer can send steering action to a remote peer. This steering actions are communicated through the data channel of Web RTC API through which the regular chat messages are also exchanged.

Along with these userHome page comprises of Call Logs and Patient Logs tab. In the Call Logs tab, a doctor can view logs of all the calls made by the doctor. In the Patient logs tab, a doctor can view all the patient associated records. This would comprise of all the feedbacks, prescription, and diagnosis given by the doctor to the patient though chat messages. Thus a doctor could view all the reports and treatment given to the patient at some later point of time.

**Robot App**

Robot application is currently similar to the doctor application. This was intended so as to determine that any two random peers can communicate with each other. However we would identify a robot during the login procedure. Accordingly the interface at the robot side would not comprise of the steering wheel. Also it would not consist of the Patient Logs and the Call Logs. It would comprise of merely the views for displaying remote and local audio and video stream and a chat window in case a nurse or a patient wants to exchange chat messages with the doctor.

**Implementation**

The application uses WebRTC technology for establishing a connection with a peer and exchanging information. WebRTC is newly developed HTML standard with APIs for creating peer to peer connections and creating channels for exchanging streams as well as data.

WebRTC uses UDP for Real-time communication and TCP for data communication. WebRTC allows two peers to connect via a signalling server. Once connected, the peers exchange information without the need of a server thereby reducing delay. Also absence of a server ensures that two peers can continue to communicate as long as they remain connected to each other, irrespective of the status of the server.

**Signalling:**

WebRTC does require a server for helping the peers connect. The server, known as signalling server, knows the number and availability of each peer. This ensures the server can facilitate the connection of two peers by forwarding their requests. Signalling is defined as the process of coordinating communication. The peers exchange following information:-

- Session control messages to open and close communication

- Error messages
- Metadata such as media type, bandwidth
- Key data, to establish secure connections
- Network Data

The signalling messages are very small and are relatively undemanding in terms of bandwidth. Also these messages do not need much processing or memory as they mostly relay data back and forth and only retain session data.

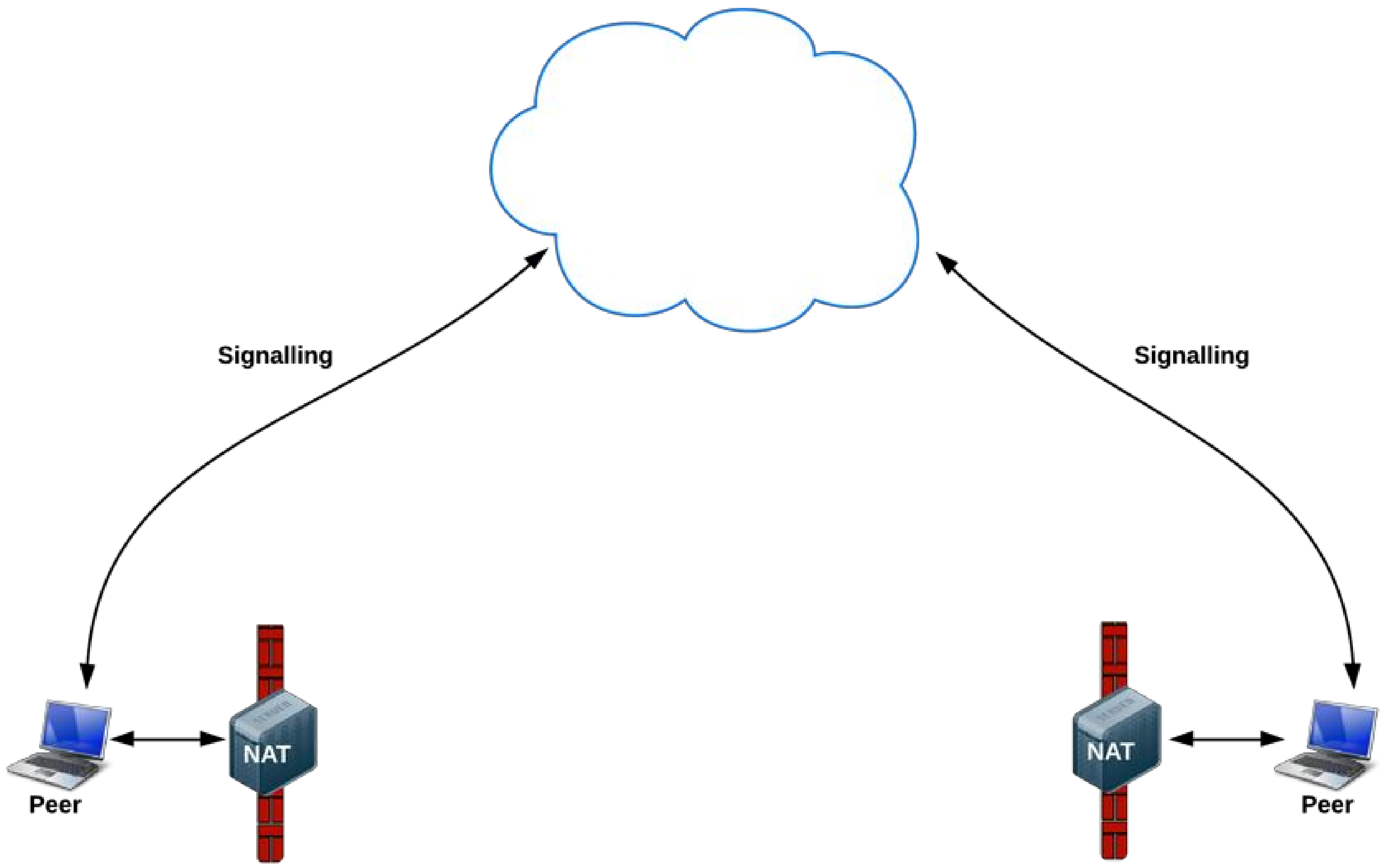
**ICE:**
Once two peers are connected and a session is established, RTCPeerConnection tries to connect the peers directly, peer to peer for media and data streaming. In simple world every endpoint would have a unique address that it could exchange with peers in order to communicate directly.

However most devices are behind NAT, some are behind proxies and firewalls with several protocols and ports blocked. WebRTC can use the ICE framework to overcome the complexities of real-world networking. ICE tries to find the best path to connect peers. It tries all possibilities in parallel and chooses the most efficient option that works. 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.

In other words:

- A STUN server is used to get an external network address.

- TURN servers are used to relay traffic if direct (peer to peer) connection fails.



**STUN Server**

NATs provide a device with an IP address for use within a private local network, but this address can't be used externally. Without a public address, there's no way for WebRTC peers to communicate. To get around this problem WebRTC uses STUN.

STUN servers live on the public internet and have one simple task: check the IP: port address of an incoming request (from an application running behind a NAT) and send that address back as a response. In other words, the application uses a STUN server to discover its IP: port from a public perspective. This process enables a WebRTC peer to get a publicly accessible address for itself, and then pass that on to another peer via a signalling mechanism, in order to set up a direct link. (In practice, different NATs work in different ways, and there may be multiple NAT layers, but the principle is still the same.)

**TURN Server**

RTCPeerConnection tries to set up direct communication between peers over UDP. If that fails, RTCPeerConnection resorts to TCP. If that fails, TURN servers can be used as a fall back, relaying data between endpoints.

**Signalling Server - Node and Express**

Signalling protocols and mechanisms are not defined by WebRTC standards. So we implemented a signalling server in Node.JS. The signalling server is notified when a new server is up and running and also when a peer wants to connect to another peer.

The signalling server also offers an authentication service to our application to provide security. The server is built on the express framework and hence it takes care of the routing and other requirements.

Once a peer is logged in, it sends a message to the server telling it that it’s up. The server keeps track of all the live peers. Once if a new peer connects to the server, the server sends a push message to all the other peers telling it that a new peer has joined the network. The server identifies every peer by a unique name.

If a peer wants to connect to new peer, it sends a message to the server. The message contains the unique string of the peer who sent the message and also of the peer to which the connection must be forwarded. The entire session initiation is coordinated by the server. The server relays the messages to the corresponding peer.

**Storage API - Mongo DB and JSON**

The personal details of each peer such as name and password are stored in a NoSQL database. This information is necessary for authentication as well as providing personalized information. The signalling server doubles up as a mongo client thereby providing a smooth interface for accessing the database when

required. The Mongo database is a NoSQL database and hence not relational. The data is stored in JSON format.
The doctor can also view old patient records. The database stores the feedback/diagnosis provided by the doctor. The call history between two peers is also stored.

The database consists of following collections.
Users - The user's collection stores the individual information of each peer, which includes the login credentials and personal information in case of doctors. This collection is updated when a new user registers.

- callData - This collection stores all the details of all the calls that happen. It stores the name of peers involved in the call as well as the time and date of the call.
- patientData - It stores the data of specific patients. This includes the feedback provided by doctors etc.
- loggedInUsers - This collection keeps track of all users currently logged in to their accounts. This is essential to the server for creating the peer list for each user. When a user logs in, his/her entry is added to the collection and when he/she logs out, the entry removed from the collection. All the requests by the express server to access the database is via HTTP requests while all the data responses by database server is in JSON.

**Sockets and Socket.IO**
Two peers, before connecting, need to exchange initiation messages. These messages are relayed by the signalling server. The entire communication between signalling server and individual peers occurs via Web Sockets. Web Sockets is an advanced technology that makes it possible to open an interactive communication session between the user's browser and a server. It makes possible to send messages to a server and receive event-driven responses without having to poll the server for a reply.
Socket.io is JavaScript library which facilitates in sending and receiving sockets. When a peer wants to connect to another peer, it sends a socket containing the

message to the signalling server. The signalling server forwards and relays the replies to the peer, again via sockets.

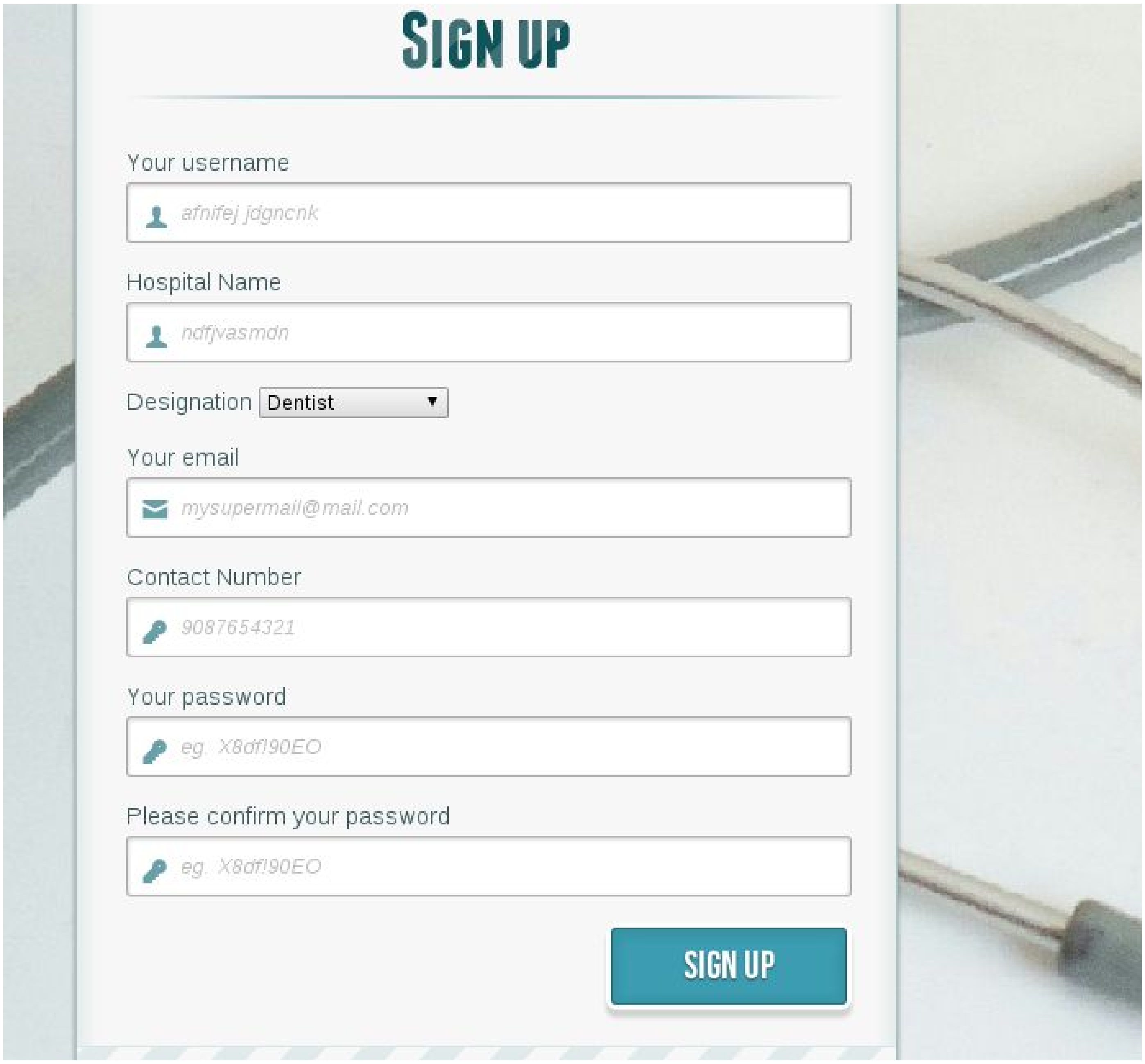
**Steering and Chat**

The WebRTC's Media Stream API is used for communicating between the TRH and the doctor, i.e. to exchange streams. On the other hand, to exchange data WebRTC's RTCDataChannel API is used. The data channel is used in steering module and the chat system.

To steer the robot, the doctor has to click the respective buttons. The data channel sends the corresponding message to the respective peers. The message is processed at the robot end and the corresponding signal is produced.

**Usage:**

1. For a new doctor to connect to the application and start using, he/she must first register.



SIGN UP

Your username

afnifej jdgncnk

Hospital Name

ndfjvasmdn

Designation: Dentist

Your email

mysupermail@mail.com

Contact Number

9087654321

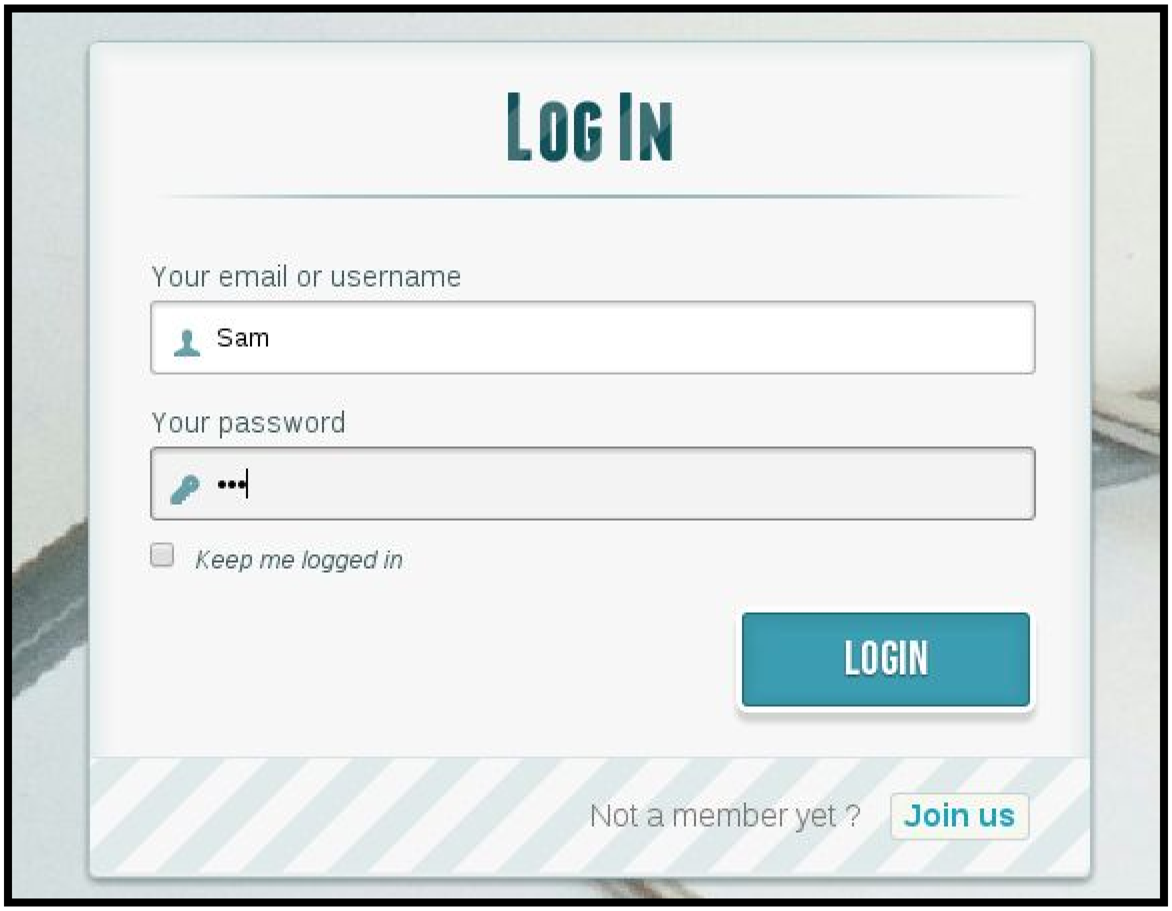
Your password

eg. X8df!90EO

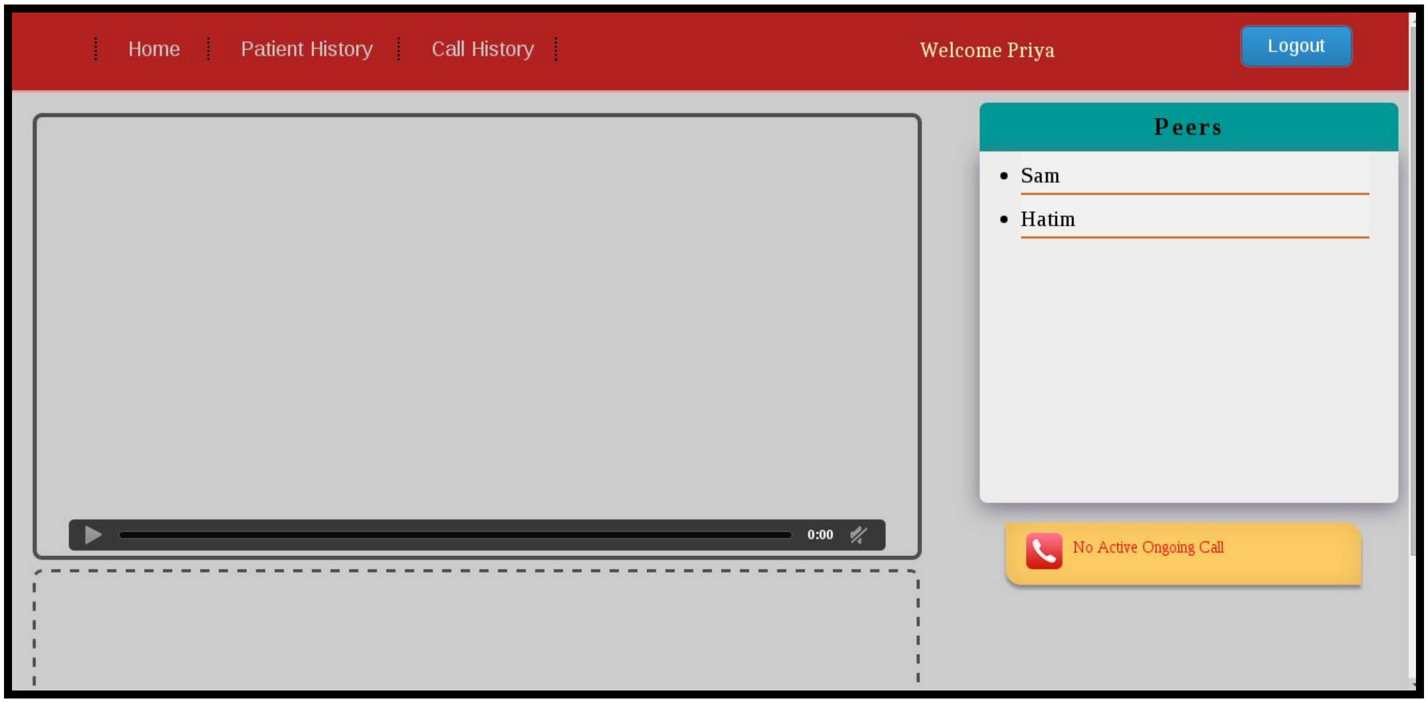
Please confirm your password

eg. X8df!90EO

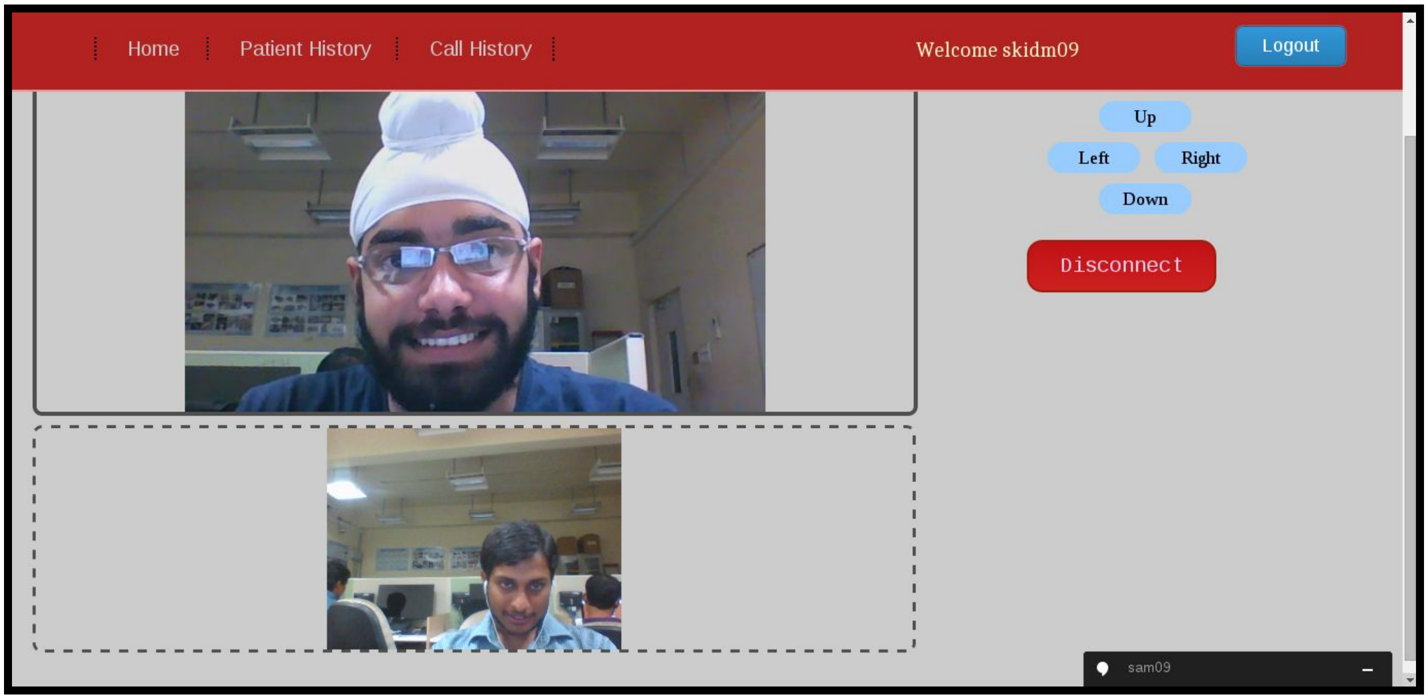
2. After registration he/she must login.



3. The panel on the left shows the list of all available TRH

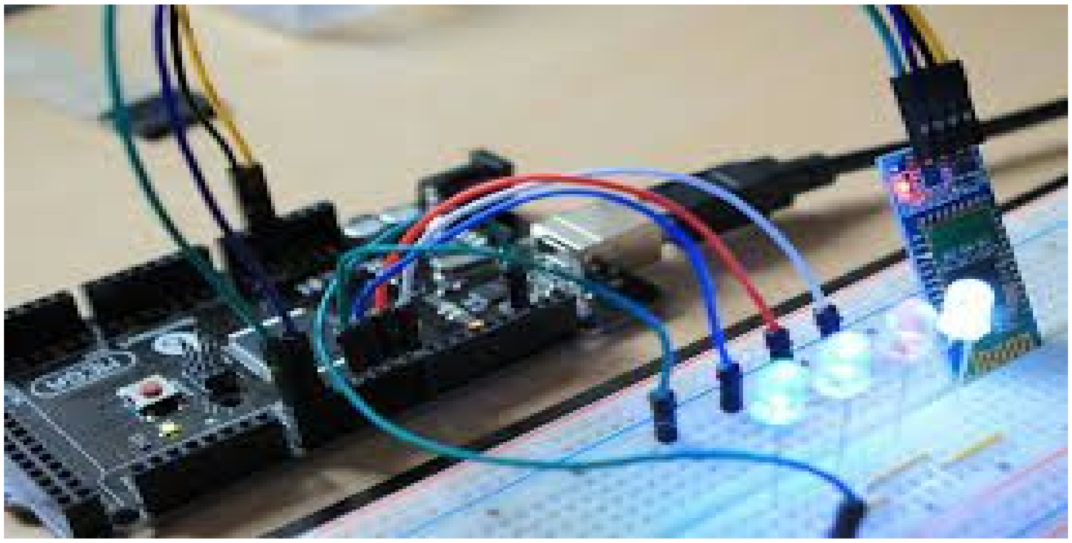


4. To connect to a particular peer he/she must click on the peer and click ok when the prompt appears.
5. The browser asks for permission to use the webcam. Granting it permission will allow the application to use the webcam and microphone, thereby enabling the doctor to call the TRH.



6.The Up ,Down ,Left ,Right Buttons can be clicked to control the robot movements which is located  in remote area far from doctor.

7.These buttons when clicked will go as a chat message to the app and will parsed accordingly and appropriate message will be sent to arduino over bluetooth.



8.We Tested the communication Web->WebRTC->Android->Bluetooth->Arduino using Led Lights on BreadBoard and have successfully achieved it with minimal delay.