WebRTC Cookbook

Using WebRTC, it is easy to develop in-browser applications and web services with extended multimedia features such as audio/video calls, VoIP, screen casting, peer-to-peer file transferring and more, without installing any third-party components/plugins on the client. In this book, you’ll explore practical recipes covering the core aspects of application and service development with WebRTC. After a recap of the basics of developing rich media web applications and services, you will delve into core functionalities such as security and SSL integration with other technologies such as VoIP and SIP, debugging clients and servers, working with filters, native applications for mobile platforms, and using third-party components.

By the end of this book, you will be able set up advanced, real-time, communicating networks with a full understanding of core WebRTC features.

What this book will do for you...

- Understand the concepts of WebRTC while setting up rich web applications
- Create a signaling server using Java or Erlang
- Secure your WebRTC application by configuring your own STUN/TURN server
- Integrate with VoIP platforms such as Asterisk and FreeSWITCH and abstract the complexities of communications protocols
- Get acquainted with advanced debugging and troubleshooting techniques
- Process videos using HTML5 filters
- Develop native mobile applications using WebRTC and integrate them with web services
- Discover advanced WebRTC techniques and solutions

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

WebRTC Cookbook

Get to grips with advanced real-time communication applications and services on WebRTC with practical, hands-on recipes

Andrii Sergiienko

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Peer Connections'
- A synopsis of the book’s content
- More information on WebRTC Cookbook

About the Author

Andrii Sergiienko is a computer software developer from Ukraine and is passionate about information technologies. From his early childhood, he was interested in computer programming and hardware. He took his first step into these fields more than 20 years ago. He has experience of a wide set of languages and technologies including C, C++, Java, assembly language, Erlang, JavaScript, PHP, Riak, shell scripting, computer networks, and security.

During his career he worked for both small, local companies such as domestic ISP and large, worldwide corporations such as Hewlett Packard. He also started his own projects—some of them were relatively successful.

Today, he is the owner and inspirer of OSLIKAS OÜ, a computer software company with headquarters in Estonia. The company (http://www.oslikas.com) focuses on modern IT technologies and solutions.

Working on this book was a really great and interesting experience for me. All this would be impossible without the help of certain people. And now is the time for me to say thank you to them.

First of all, I would like to thank my parents Olga and Alexander for my happy childhood that established the foundation for my life and career.

I would like to say thank you to my wife Inna for her patience, encouragement, and support during this process.

I would like to thank the Packt Publishing team as well. These guys are doing really great work and making the world a better place. We contacted some of them directly during the work, and others stayed behind the scenes. However, I know that a lot of people spent part of their lives to make this book possible. Thank you all.
WebRTC Cookbook

WebRTC is a relatively new and revolutionary technology that opens new horizons in the area of interactive applications and services. Most of the popular web browsers support it natively (such as Chrome and Firefox) or via extensions (such as Safari). Mobile platforms such as Android and iOS allow you to develop native WebRTC applications.

This book covers a wide set of topics on how to develop software using a WebRTC stack. Using practical recipes, it considers basic concepts, security, debugging, integration with other technologies, and other important themes of the development process in a friendly manner.

You will not only learn about WebRTC-specific features, but also attendant technologies (CSS3, HTML5, and WebSockets), and how to use them along with WebRTC.

What This Book Covers

Chapter 1, Peer Connections, introduces you to the very basic concepts of WebRTC. This includes practical recipes on peer connections. You will also find simple demo applications in this chapter.

Chapter 2, Supporting Security, leads you through various security-related topics and covers how to secure a typical WebRTC application's infrastructure components: SSL/TLS certificates, WebSockets, web servers, STUN/TURN, data channels, and more.

Chapter 3, Integrating WebRTC, considers integrating a WebRTC application with other technologies and third-party software. This chapter describes practical cases and solutions on integration.

Chapter 4, Debugging a WebRTC Application, is dedicated to application debugging—an important topic of the software development process. In this chapter, you will learn about the topics relating to debugging in the scope of WebRTC.

Chapter 5, Working with Filters, teaches you how to use CSS3 filters with WebRTC applications. This chapter also covers custom image processing.

Chapter 6, Native Applications, contains practical, step-by-step recipes dedicated to developing native WebRTC applications on mobile platforms.

Chapter 7, Third-party Libraries, describes general use cases and practical solutions based on third-party WebRTC frameworks and services.

Chapter 8, Advanced Functions, covers how to use advanced WebRTC features. It contains practical recipes on file transferring, streaming, audio/video controlling, and more.
Peer Connections

In this chapter, we will cover the following topics:

- Building a signaling server in Erlang
- Building a signaling server in Java
- Detecting WebRTC functions supported by a browser
- Making and answering calls
- Implementing a chat using data channels
- Implementing a chat using a signaling server
- Configuring and using STUN
- Configuring and using TURN

Introduction

This chapter covers the basic concepts of how to use WebRTC when developing rich media web applications and services.

With simple and short recipes, you will learn how to create your own signaling server. The key data that needs to be exchanged by peers before they establish a direct connection is called the session description—it specifies the peers' configuration. Signaling server is a component in an application's infrastructure that is accessible by all peers and serves to exchange multimedia's session description. The way peers should exchange data is not described by WebRTC standards, so you should make the decision on your own regarding the protocol and mechanism you will use for this task.
You can build a signaling server using any programming language and technology you like. In general, the signaling protocol can be non-technical and is possible to implement in away where the peers would use just a sheet of paper to exchange necessary data between each other. In this chapter, we use WebSocket to implement signaling, although you can use any other protocol.

The signaling stage is represented in the schema that is shown in the following diagram:

In this chapter, you will find two recipes that are dedicated to signaling server development: Building a signaling server in Erlang and Building a signaling server in Java. Java is probably the most popular and known technology, and it would be easy to get into this topic using Java, even if you don't have programming experience with this technology. Erlang is not widely known yet. Nonetheless, this is a very mature technology, very suitable for writing lightweight and extremely fast server applications with perfect scalability. So, by learning signaling server, you will find simple solutions in Erlang as well.

This chapter also covers the basic use case of how to use WebRTC data channels: file transferring and peer-to-peer chat.

You will also learn how to configure and use Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) services, and of course, this chapter covers making peer-to-peer calls using WebRTC.

Note that in this chapter, we will cover the process of making computer-to-computer calls. If you want to know more about how to use WebRTC with VoIP and SIP, and how to make phone calls from a web page, refer to the Chapter 3, Integrating WebRTC.
Building a signaling server in Erlang

The following recipe shows how to build signaling server using Erlang programming language and WebSockets for transport protocol. We will not introduce Erlang programming in this recipe, so you should have at least basic knowledge of this programming language and its relevant technologies.

Getting ready

To use this solution, you should have Erlang installed on your system to start with. You can download the Erlang distribution relevant to your system from its home page http://www.erlang.org/download.html. The installation process might need specific actions relevant to specific platforms/O Ses, so follow the official installation instructions at http://www.erlang.org/doc/installation_guide/INSTALL.html.

For this example, I've used Erlang 17. You might need to add some minor changes to the code to make it work under future versions of Erlang.

We will also use the Git versioning system to download additional packets and components necessary for our solution, so you should download and install Git distribution relevant to your system. You can download this from http://www.git-scm.com. As a build tool for the project, we will use Rebar; you should also download and install it from https://github.com/basho/rebar.

How to do it...

The following steps will lead you through the process of building a signaling server using Erlang:

1. Create a new folder for the signaling server application and navigate to it.
2. Using the Rebar tool, create a basic Erlang application:
   ```
   rebar create-app appid=sigserver
   ```
   This command will create an src folder and relevant application files in it.
3. Create the rebar.config file, and put the following Rebar configuration in it:
   ```erlang
   {erl_opts, [warnings_as_errors]}.
   {deps, [
   {'gproc', ".*", {git, "git://github.com/esl/gproc.git", {tag, "0.2.16"} }},
   ]}.
   ```
Peer Connections

{
'jsonerl', ".*", {
  git, "git://github.com/fycth/jsonerl.git", "master"
}},
{'cowboy', ".*", {
  git,"https://github.com/extend/cowboy.git","0.9.0"
}}
].

4. Open the src/sigserver.app.src file and add the following components to the application's section list: cowlib, cowboy, compiler, and gproc.

5. Open the src/sigserver_app.erl file and add the following code:

```erlang
-module(sigserver_app).
-behaviour(application).
-export([start/2, stop/1, start/0]).
start() ->
    ok = application:start(ranch),
    ok = application:start(crypto),
    ok = application:start(cowlib),
    ok = application:start(cowboy),
    ok = application:start(gproc),
    ok = application:start(compiler),
    ok = application:start(sigserver).
start(_StartType, _StartArgs) ->
    Dispatch = cowboy_router:compile([{',
        {',
          "/ " , handler_websocket, []
        }
      }]),
    {ok, _,} = cowboy:start_http(websocket, 100, [{ip, [127,0,0,1]}, {port, 30001}], [{env, [{dispatch, Dispatch}]}, {max_keepalive, 50},
      {timeout, 500}]),
    sigserver_sup:start_link().
stop(_State) -> ok.
```

6. Create the src/handler_websocket.erl file and put the following code in it:

```erlang
-module(handler_websocket).
-behaviour(cowboy_websocket_handler).
-export([init/3]).
-export([websocket_init/3, websocket_handle/3,
        websocket_info/3, websocket_terminate/3]).
```
-record(state, {
    client = undefined :: undefined | binary(),
    state = undefined :: undefined | connected | running,
    room = undefined :: undefined | integer()
}).

init(_Any, _Req, _Opt) ->
  {upgrade, protocol, cowboy_websocket}.

websocket_init(_TransportName, Req, _Opt) ->
  {Client, Req1} = cowboy_req:header(<-x-forwarded-for>>, Req),
  State = #state{client = Client, state = connected},
  {ok, Req1, State, hibernate}.

websocket_handle({text,Data}, Req, State) ->
  StateNew = case (State#state.state) of
    started ->
        State#state{state = running};
    _ ->
        State
  end,
  JSON = jsonerl:decode(Data),
  {M,Type} = element(1,JSON),
  case M of
    <<"type">> ->
        case Type of
          <<"GETROOM">> ->
            Room = generate_room(),
            R =
                iolist_to_binary(jsonerl:encode({
                    {type, <<"GETROOM">>},
                    {value, Room}})),
                gproc:reg({p,l, Room}),
                S = (StateNew#state{room = Room}),
                {reply, {text, <<R/binary>>}, Req, S, hibernate};
          <<"ENTERROOM">> ->
            {<<"value">>,Room} = element(2,JSON),
            Participants =
                gproc:lookup_pids({p,l,Room}),
            case length(Participants) of
              1 ->
                Room =
                  iolist_to_binary(jsonerl:encode({
                      {type, <<"ENTERROOM">>},
                      {value, Room}})),
                gproc:reg({p,l, Room}),
                S = (StateNew#state{room = Room}),
                {reply, {text, <<R/binary>>}, Req, S, hibernate};
              _ ->
                  R =
                      iolist_to_binary(jsonerl:encode({
                          {type, <<"ENTERROOM">>},
                          {value, Room}})),
                  gproc:reg({p,l, Room}),
                  S = (StateNew#state{room = Room}),
                  {reply, {text, <<R/binary>>}, Req, S, hibernate};
            end;
Peer Connections

gproc:reg({p,l, Room}),
S = (StateNew#state{room = Room}),
{ok, Req, S, hibernate};
_ ->
  R =
  iolist_to_binary(jsoner1:
    encode({{type, <<"WRONGROOM">>}}),
    {reply, {text, <<R/binary>>},
    Req, StateNew, hibernate}
  end;
_ ->
  reply2peer(Data, StateNew#state.room),
    {ok, Req, StateNew, hibernate}
  end;
_ ->
  reply2peer(Data, State#state.room),
    {ok, Req, StateNew, hibernate}
  end;
websocket_handle(_Any, Req, State) -> {ok, Req, State, hibernate}.
websocket_info(_Info, Req, State) -> {reply, {text,_Info}, Req, State, hibernate}.
websocket_terminate(_Reason, _Req, _State) -> ok.
reply2peer(R, Room) ->
  [P ! <<R/binary>> || P <- gproc:lookup_pids({p,l,Room})
    -- [self()]].
generate_room() ->
  random:seed(now()),
  random:uniform(999999).

7. Now we can compile the solution using the Rebar tool:
   rebar get-deps
   rebar compile

If everything was successful, you should not see any errors (warnings are not critical).

8. After we build our signaling server, we can start it using the following command:
   erl -pa deps/*/ebin ebin -sasl errlog_type error -s sigserver_app
Windows-based systems can't use a star symbol in such constructions, so if you're working under Windows, you should use the full path name as shown in the following command:

```
erl -pa deps/cowboy/ebin deps/cowlib/ebin deps/gproc/ebin
deps/jsonerl/ebin deps/ranch/ebin ebin -sasl
errlog_type error -s sigserver_app
```

Now your signaling server should be running, and you need to listen for incoming WebSocket connections on port 30001.

Note that full source codes are supplied with this book.

### How it works...

In this recipe, we implemented the WebRTC signaling server in Erlang. The application listens on port 30001 for incoming WebSocket connections from the browser clients.

The first peer will be registered by the server in a virtual room and will get the room number. The second peer after that can use the room number in order to connect to the first peer. The signaling server will check whether the virtual room exists and if so, it will route call/answer requests and answers between the peers in order to make them establish a direct peer-to-peer WebRTC connection.

### There's more...

Basically, this is a very simple signaling server. It doesn't have any advanced features, and the main goal of it is to help peers establish direct connection between each other. Nevertheless, a signaling server can serve additional tasks. For example, it can serve for web chats, file transfers, service data exchanges, and other features specific for certain situations. There are no certain requirements for a signaling server; you can implement it using your favorite programming language and technology.
See also

- For tips on implementing a signaling server in Java, refer to the Building a signaling server in Java recipe
- You can also refer to the Making and answering calls recipe on how to use a signaling server from a browser application using JavaScript

Building a signaling server in Java

In this recipe, we will cover the implementation of a signaling server in Java.

Getting ready

This recipe uses Java, so you should have Java Developer Kit (JDK) installed on your machine. You can download the appropriate version of JDK for your platform from its web page at http://www.java.com.

Java 7 has its own API to implement a WebSocket application. Previous versions of Java don’t have the native support of WebSockets. In this recipe, we will cover the universal solution that works in different Java versions and is based on the third-party component, which you can find on its home page at http://java-websocket.org. This project is also present on GitHub at https://github.com/TooTallNate/Java-WebSocket.

You need to download and install the Java-WebSocket library; it should then be linked to your project.

In this recipe, we pack signaling messages into the JSON format before sending, so we need a Java library to work with JSON structures. For this purpose, we will use Java classes from JSON's home page, http://www.json.org/java/.

Download these classes and link them to your project, or you can just put these classes into your project's folder structure and compile it all together.

It is assumed that you have experience of programming in Java, so we will not cover the basic questions like how to start a Java application and so on.

How to do it...

Create a new project in your Java IDE and link the JSON libraries along with the Java-WebSocket library.
The following code represents a simple signaling server. Compile it and start a Java console application as usual:

```java
package com.webrtcexample.signaler;

import org.java_websocket.WebSocket;
import org.java_websocket.handshake.ClientHandshake;
import org.java_websocket.server.WebSocketServer;
import org.json.JSONException;
import org.json.JSONObject;
import java.net.InetSocketAddress;
import java.util.*;

public class Main extends WebSocketServer {

    private static Map<Integer, Set<WebSocket>> Rooms = new HashMap<>();
    private int myroom;

    public Main() {
        super(new InetSocketAddress(30001));
    }

    @Override
    public void onOpen(WebSocket conn, ClientHandshake handshake) {
        System.out.println("New client connected: " +
                          conn.getRemoteSocketAddress() + " hash " +
                          conn.getRemoteSocketAddress().hashCode());
    }

    @Override
    public void onMessage(WebSocket conn, String message) {
        Set<WebSocket> s;
        try {
            JSONObject obj = new JSONObject(message);
            String msgtype = obj.getString("type");
            switch (msgtype) {
                case "GETROOM":
                    myroom = generateRoomNumber();
                    s = new HashSet<>();
                    s.add(conn);
                    Rooms.put(myroom, s);
                    break;
            }
        } catch (JSONException e) {
            // Handle JSON exception
        }
    }
}
```
System.out.println("Generated new room: " + myroom);
conn.send("{"type":"GETROOM","value":"" + myroom + "}");
break;
case "ENTERROOM":
    myroom = obj.getInt("value");
    System.out.println("New client entered room "+ myroom);
    s = Rooms.get(myroom);
    s.add(conn);
    Rooms.put(myroom, s);
    break;
default:
    sendToAll(conn, message);
break;
} catch (JSONException e) {
    sendToAll(conn, message);
}
System.out.println();

@Override
public void onClose(WebSocket conn, int code, String reason, boolean remote) {
    System.out.println("Client disconnected: " + reason);
}

@Override
public void onError(WebSocket conn, Exception exc) {
    System.out.println("Error happened: " + exc);
}

private int generateRoomNumber() {
    return new Random(System.currentTimeMillis()).nextInt();
}

private void sendToAll(WebSocket conn, String message) {
    Iterator it = Rooms.get(myroom).iterator();
    while (it.hasNext()) {
        WebSocket c = (WebSocket)it.next();
        if (c != conn) c.send(message);
    }
}
public static void main(String[] args) {
    Main server = new Main();
    server.start();
}

Once the application starts, it will listen on the TCP port 30001 for WebSocket messages from clients. You can write simple client applications to test the signaling server—refer to the Making and answering calls recipe.

Note that you can find a Maven-based project for this example supplied with this book.

**How it works...**

First of all, the client sends a GETROOM message to the signaling server that is listening on TCP port 30001. The server generates a new virtual room number, stores it, and sends it back to the client.

The client constructs a new access URL using the virtual room number received from the server. Then, the second client uses this URL to enter the virtual room and establish a call to the first client.

The second client sends the room number it got from the URL to the signaling server. The server associates the client with the virtual room number. Then, the client makes a call, using signaling server, which forwards its messages to the first client that is present in the room already. The first client answers the call, also using the signaling server as the middle point.

So both clients exchange the necessary data (including network details) and then establish direct peer-to-peer connection. After the connection is established, peers don't use the server anymore.

**There's more...**

The WebSocket signaling server in Java can be implemented using a Java EE stack. For more details, take a look at the home page of JSR 356 at http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html.

You can also find an example at https://github.com/hsilomedus/web-sockets-samples/tree/master/eesockets.

Another solution is to use Spring 4. It has WebSocket support out of the box. For details on this solution, take a look at the example on GitHub at https://github.com/hsilomedus/web-sockets-samples/tree/master/springsockets.
For an alternative solution, you can refer to the *Building a signaling server in Erlang* recipe

---

**Detecting WebRTC functions supported by a browser**

WebRTC is not fully supported by all available web browsers at this time. Moreover, there is a chance that your application will be running under some kind of exotic environment or web browser that does not support WebRTC. So you need to have some mechanism that would enable you to detect whether the environment in which your web application is running supports the necessary WebRTC features the application is going to use. In this recipe, we will cover the basic method of doing that.

**Getting ready**

This task is relevant for the client side only, so all the code will be written in JavaScript. Thus, no specific preparation is needed.

**How to do it...**

You can write a JavaScript library that can be used to detect which WebRTC methods are available under the environment and by what names they are known for your application.

The following code represents a basic but productive example of such a kind of library:

```javascript
var webrtcDetectedVersion = null;
var webrtcDetectedBrowser = null;
window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;

function initWebRTCAdapter() {
  if (navigator.mozGetUserMedia) {
    webrtcDetectedBrowser = "firefox";
    webrtcDetectedVersion = parseInt(navigator.userAgent.match(/Firefox\/[\d.]+/)[1], 10);

    RTCPeerConnection = mozRTCPeerConnection;
    RTCSessionDescription = mozRTCSessionDescription;
    RTCIceCandidate = mozRTCIceCandidate;
    getUserMedia = navigator.mozGetUserMedia.bind(navigator);
  }
}
```
attachMediaStream =
function(element, stream) {
    element.mozSrcObject = stream;
    element.play();
};

reattachMediaStream =
function(to, from) {
    to.mozSrcObject = from.mozSrcObject;
    to.play();
};

MediaStream.prototype.getVideoTracks =
function() {
    return [];
};

MediaStream.prototype.getAudioTracks =
function() {
    return [];
};
return true;
} else if (navigator.webkitGetUserMedia) {
    webrtcDetectedBrowser = "chrome";
    webrtcDetectedVersion =
parseInt(navigator.userAgent.match(/Chrom(e|ium)/\([0-9]+)\./)[2], 10);

    RTCPeerConnection = webkitRTCPeerConnection;
    getUserMedia =
navigator.webkitGetUserMedia.bind(navigator);
    attachMediaStream =
function(element, stream) {
    element.src = webkitURL.createObjectURL(stream);
};

    reattachMediaStream =
function(to, from) {
    to.src = from.src;
};

    if (!webkitMediaStream.prototype.getVideoTracks) {
        webkitMediaStream.prototype.getVideoTracks =
function() {
            return [];
        };
    }
Peer Connections

    return this.videoTracks;
   );
    webkitMediaStream.prototype.getAudioTracks =
    function() {
        return this.audioTracks;
    };"}]

if (!webkitRTCPeerConnection.prototype.getLocalStreams) {
    webkitRTCPeerConnection.prototype.getLocalStreams =
    function() {
        return this.localStreams;
    };
    webkitRTCPeerConnection.prototype.getRemoteStreams =
    function() {
        return this.remoteStreams;
    };"}]
    return true;
    } else return false;
};

How it works...

This solution tests which WebRTC API methods are available in the environment and how they are named. So your application can use certain API function names that will be relevant for any web browser, without using browser-specific function names.

There's more...

There is another way to solve this task. You don't necessary have to write your own adapter. You can take the adapter prepared by Google. It can be found at http://apprtc.webrtc.org/js/adapter.js. You just need to include it in your JavaScript code.

You can also consider using a browser's plugin that enables the use of WebRTC in Safari and Internet Explorer. You can get these at https://temasys.atlassian.net/wiki/display/TWPP/How+to+integrate+the+plugin+with+your+website.

See also

You can find more information on the adapter at the web page http://www.webrtc.org/web-apis/interop.
Making and answering calls

The very basic action of any WebRTC application is making and receiving a call. This recipe shows how to make calls to a remote peer.

Getting ready

At the beginning, peers don't know each other, and they don't know the necessary network information to make direct connection possible. Before establishing a direct connection, peers should exchange necessary data using some middle point—usually, a signaling server. This is a middle point that is known to each peer. So each peer can connect to the signaling server, and then one peer can call another one—by asking the signaling server to exchange specific data with another peer and make peers know each other.

So, you need a signaling server to run.

How to do it...

Before two peers can establish a direct connection, they should exchange specific data (ICE candidates and session descriptions) using a middle point—the signaling server. After that, one peer can call another one, and the direct peer-to-peer connection can be established.

Interactive Connectivity Establishment (ICE) is a technique used in Network Address Translator (NAT), which bypasses the process of establishing peer-to-peer direct communication. Usually, ICE candidates provide information about the IP address and port of the peer. Typically, an ICE candidate message might look like the following:

\[ \text{a=candidate:1 1 UDP 4257021352 192.168.0.10 1211 typ host} \]

Session Description Protocol (SDP) is used by peers in WebRTC to configure exchanging (network configuration, audio/video codecs available, and so on). Every peer sends details regarding its configuration to another peer and gets the same details from it back. The following print depicts a part of an SDP packet representing the audio configuration options of a peer:

\[ \text{m=audio 53275 RTP/SAVPF 121 918 100 1 2 102 90 131 16} \]
\[ \text{c=IN IP4 16.0.0.1} \]
\[ \text{a=rtcp:53275 IN IP4 16.0.0.1} \]
In the schema represented in the following diagram, you can see the generic flow of a call establishing process:

Note that TURN is not showed in the schema. If you used TURN, it would be depicted just after the STUN stage (before the first and second stage).

**Making a call**
To make a call, we need to take some steps to prepare (such as getting access to the browser's media):

1. Get access to the user's media:
   ```javascript
   function doGetUserMedia() {
     var constraints = {"audio": true, "video":
       {"mandatory": {}, "optional": []}};
     try {
       getUserMedia(constraints, onUserMediaSuccess,
       function(e) {
         console.log("getUserMedia error " +
         e.toString());
       });
     } catch (e) {
       console.log(e.toString());
     }
   }
   ```
2. If you succeed, create a peer connection object and make a call:

```javascript
function onUserMediaSuccess(stream) {
    attachMediaStream(localVideo, stream);
    localStream = stream;
    createPeerConnection();
    pc.addStream(localStream);
    if (initiator) doCall();
}

function createPeerConnection() {
    var pc_constraints = {"optional":
        ["DtlsSrtpKeyAgreement": true]};
    try {
        pc = new RTCPeerConnection(pc_config,
        pc_constraints);
        pc.onicecandidate = onIceCandidate;
    } catch (e) {
        console.log(e.toString());
        pc = null;
        return;
    }
    pc.onaddstream = onRemoteStreamAdded;
};

function onIceCandidate(event) {
    if (event.candidate)
        sendMessage({type: 'candidate', label:
            event.candidate.sdpMLineIndex, id:
            event.candidate.sdpMid,
            candidate: event.candidate.candidate});
};

function onRemoteStreamAdded(event) {
    attachMediaStream(remoteVideo, event.stream);
    remoteStream = event.stream;
};

function doCall() {
    var constraints = {"optional": [], "mandatory":
        {"MozDontOfferDataChannel": true}};
    if (webrtcDetectedBrowser === "chrome")
        for (var prop in constraints.mandatory) if
            (prop.indexOf("Moz") !== -1) delete
        constraints.mandatory[prop];
```
Peer Connections

```javascript
constraints = mergeConstraints(constraints, sdpConstraints);
pc.createOffer(setLocalAndSendMessage, errorCallBack, constraints);
};
```

**Answering a call**

Assuming that we will use WebSockets as a transport protocol for exchanging data with signaling server, every client application should have a function to process messages coming from the server. In general, it looks as follows:

```javascript
function processSignalingMessage(message) {
    var msg = JSON.parse(message);
    if (msg.type === 'offer') {
        pc.setRemoteDescription(new RTCSessionDescription(msg));
        doAnswer();
    } else if (msg.type === 'answer') {
        pc.setRemoteDescription(new RTCSessionDescription(msg));
    } else if (msg.type === 'candidate') {
        var candidate = new RTCIceCandidate({sdpMLineIndex:msg.label, candidate:msg.candidate});
        pc.addIceCandidate(candidate);
    } else if (msg.type === 'GETROOM') {
        room = msg.value;
        onRoomReceived(room);
    } else if (msg.type === 'WRONGROOM') {
        window.location.href = "/";
    }
}
```

This function receives messages from the signaling server using the WebSockets layer and acts appropriately. For this recipe, we are interested in the `offer` type of message and `doAnswer` function.

The `doAnswer` function is presented in the following listing:

```javascript
function doAnswer() {
    pc.createAnswer(setLocalAndSendMessage, errorCallBack, sdpConstraints);
};
```
The `sdpConstraints` object describes the WebRTC connection options to be used. In general, it looks as follows:

```javascript
var sdpConstraints = {'mandatory': {'OfferToReceiveAudio':true, 'OfferToReceiveVideo':true }};
```

Here we can say that we would like to use both audio and video while establishing WebRTC peer-to-peer connection.

The `errorCallback` method is a callback function that is called in case of an error during the calling of the `createAnswer` function. In this callback function, you can print a message to the console that might help to debug the application.

The `setLocalAndSendMessage` function sets the local session description and sends it back to the signaling server. This data will be sent as an answer type of message, and then the signaling server will route this message to the caller:

```javascript
function setLocalAndSendMessage(sessionDescription) {
    pc.setLocalDescription(sessionDescription);
    sendMessage(sessionDescription);
}
```

Note that you can find the full source code for this example supplied with this book.

**How it works…**

Firstly, we will ask the web browser to gain access to the user media (audio and video). The web browser will ask the user for these access rights. If we get the access, we can create a connection peer entity and send the call message to the signaling server, which will route this message to the remote peer.

The workflow of the code is very simple. The `processSignalingMessage` function should be called every time we get a message from the signaling server. Usually, you should set it as an `onmessage` event handler of the WebSocket JavaScript object.

After the message is received, this function detects the message type and acts appropriately. To answer an incoming call, it calls the `doAnswer` function that will do the rest of the magic—prepare the session description and send it back to the server.

The signaling server will get this reply as an answer message and will route it to the remote peer. After that, peers will have all the necessary data on each other to start establishing a direct connection.
There's more...

This is the basic functionality of WebRTC. Most of your applications will probably have the same code for this task. The only big difference might be communication with the signaling server—you can use any protocol you like.

See also

- Refer to the Implementing a chat using data channels recipe regarding the process of building a simple web chat application using WebRTC
- You can find more details on ICE on the RFC 5245 website at https://tools.ietf.org/html/rfc5245
- More information regarding SDP can be found on RFC 4566 at https://tools.ietf.org/html/rfc4566

Implementing a chat using data channels

In this recipe, we will implement a peer-to-peer private messaging service using WebRTC data channels. This method allows us to send messages directly from peer to peer, using secure and safe data channels provided by the WebRTC stack.

The schema represented in the following diagram depicts the generic feature flow:
Getting ready

We will develop a simple application, so you don't need any specific preparations for this recipe. A signaling server is necessary for this application, and it can be taken from the Building a signaling server in Erlang or Building a signaling server in Java recipe.

How to do it...

For simplicity, we will make two parts of the application: an index web page and a JavaScript library.

Creating the main HTML page of the application

1. First, create an HTML index.html page. In the following code, you can find its content. Note that the less important and obvious parts might be skipped here.

```html
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="myrtclib.js"></script>
    <script src="https://rawgit.com/GoogleChrome/webrtc/master/samples/web/js/adapter.js"></script>
</head>
<body>
    <div id="status"></div><br>
    <div id="chat"></div>
    <form name="chat_form" onsubmit="onChatSubmit(document.chat_form.msg.value); return false;">
        <input type="text" class="search-query" placeholder="chat here" name="msg" id="chat_input">
        <input type="submit" class="btn" id="chat_submit_btn"/>
    </form>
</body>
</html>
```
7. Create a connection to the signaling server and initialize the WebRTC stack. The following function is declared in the JavaScript library, which we will consider further in the recipe:

```javascript
myrtclibinit("ws://localhost:30001");
```

Note that the domain name and port might be different in your case; they should be the same as declared in the source codes of the signaling sever. By default, the signaling server is listening on local host and on port 30001.

The following function sends a message to the remote peer using the `sendDataMessage` function—we will write it as part of the JavaScript library:

```javascript
function onChatSubmit(txt) {
    var msg = JSON.stringify({"type" : "chatmessage", "txt" : txt});
    sendDataMessage(msg);
}
```

We will also declare a callback function for a catching event when a new virtual room is created:

```javascript
function onRoomReceived(room) {
    var st = document.getElementById("status");
    st.innerHTML = "Now, if somebody wants to join you, should use this link: <a href=""+window.location.href+"?room="+room+"">"+window.location.href+"?room="+room+"</a>";
}
```

To show the messages received from the remote peer, we will declare an appropriate callback function. This function gets the message and puts it in the appropriate place on the HTML page:

```javascript
function onPrivateMessageReceived(txt) {
    var t = document.getElementById('chat').innerHTML;
    t += "<br>" + txt;
    document.getElementById('chat').innerHTML = t;
}
```

Save the HTML file. This will be the main page of the applications.
Creating the JavaScript helper library

Now, create an empty myrtclib.js file and put the following content into it. Note that many parts of the following code might be used in the next chapters, so they should be well-known to you already. Such obvious parts of the code might be skipped in further.

```javascript
var RTCPeerConnection = null;
var room = null;
var initiator;
var pc = null;
var signalingURL;

The following variable will be used for handling the data channel object:

```javascript
var data_channel = null;
var channelReady;
var channel;
var pc_config = {"iceServers":
    [{url:'stun:23.21.150.121'},
    {url:'stun:stun.l.google.com:19302'}]};

function myrtclibinit(sURL) {
    signalingURL = sURL;
    openChannel();
}

function openChannel() {
    channelReady = false;
    channel = new WebSocket(signalingURL);
    channel.onopen = onChannelOpened;
    channel.onmessage = onChannelMessage;
    channel.onclose = onChannelClosed;
}

function onChannelOpened() {
    channelReady = true;
    createPeerConnection();

    if(location.search.substring(1,5) == "room") {
        room = location.search.substring(6);
        sendMessage({"type" : "ENTERROOM", "value" : room * 1});
        initiator = true;
        doCall();
    } else {
        sendMessage({"type" : "GETROOM", "value" : ""});
    }
```
Peer Connections

```javascript
initiator = false;
}
};

function onChannelMessage(message) {
    processSignalingMessage(message.data);
};

function onChannelClosed() {
    channelReady = false;
};

function sendMessage(message) {
    var msgString = JSON.stringify(message);
    channel.send(msgString);
};

function processSignalingMessage(message) {
    var msg = JSON.parse(message);
    if (msg.type === 'offer') {
        pc.setRemoteDescription(new RTCSessionDescription(msg));
        doAnswer();
    } else if (msg.type === 'answer') {
        pc.setRemoteDescription(new RTCSessionDescription(msg));
    } else if (msg.type === 'candidate') {
        var candidate = new RTCIceCandidate({sdpMLineIndex:msg.label, candidate:msg.candidate});
        pc.addIceCandidate(candidate);
    } else if (msg.type === 'GETROOM') {
        room = msg.value;
        onRoomReceived(room);
    } else if (msg.type === 'WRONGROOM') {
        window.location.href = "/";
    }
};

function createPeerConnection() {
    try {
        pc = new RTCPeerConnection(pc_config, null);
        pc.onicecandidate = onIceCandidate;
```
Until now, the code is very similar to what we used in a typical WebRTC example application. Although, now we will add something new. We will set up a handler for the \texttt{ondatachannel} event of the \texttt{PeerConnection} object. This callback function will be called when the peer asks us to create a data channel and establish a data connection:

\begin{verbatim}
pc.ondatachannel = onDataChannel;
}
\end{verbatim}

\begin{verbatim}
catch (e) {
    console.log(e);
    pc = null;
    return;
}
\end{verbatim}

The handler function is pretty simple. We will store the reference in the data channel and initialize it:

\begin{verbatim}
function onDataChannel(evt) {
    console.log('Received data channel creating request');
    data_channel = evt.channel;
    initDataChannel();
}
\end{verbatim}

By initializing the data channel, I mean setting up a channel's event handlers:

\begin{verbatim}
function initDataChannel() {
    data_channel.onopen = onChannelStateChange;
    data_channel.onclose = onChannelStateChange;
    data_channel.onmessage = onReceiveMessageCallback;
}
\end{verbatim}

In the following function, we need to create a new data channel—not when the remote peer is asking us, but when we're the initiator of the peer connection and want to create a new data channel. After we have created a new data channel, we should ask the remote peer to do the same:

\begin{verbatim}
function createDataChannel(role) {
    try {

    data_channel = pc.createDataChannel("datachannel_"+room+role, null);
    }

    catch (e) {
        console.log('error creating data channel ' + e);
        return;
    }

    initDataChannel();
}
\end{verbatim}
When we are playing the role of the connection initiator (caller), we create a new data channel. Then, during the connection establishment, the remote peer will be asked to do the same and the data channel connection will be established:

```javascript
createDataChannel("caller");
pc.createOffer(setLocalAndSendMessage, failureCallback, null);
```

```javascript
function doCall() {

}
```

```javascript
function doAnswer() {
    pc.createAnswer(setLocalAndSendMessage, failureCallback, null);
}
```

```javascript
function setLocalAndSendMessage(sessionDescription) {
    pc.setLocalDescription(sessionDescription);
    sendMessage(sessionDescription);
}
```

To send text messages via the data channel, we need to implement the appropriate function. As you can see in the following code, sending data to the data channel is pretty easy:

```javascript
function sendDataMessage(data) {
    data_channel.send(data);
}
```

The following handler is necessary to print the state of the data channel when it is changed:

```javascript
function onChannelStateChanged() {
    console.log('Data channel state is: ' +
                data_channel.readyState);
}
```
When the remote peer sends us a message via the data channel, we will parse it and call the appropriate function to show the message on the web page:

```javascript
function onReceiveMessageCallback(event) {
    console.log(event);
    try {
        var msg = JSON.parse(event.data);
        if (msg.type === 'chatmessage')
            onPrivateMessageReceived(msg.txt);
    } catch (e) {};
}
```

Save the JavaScript file.

Now, start the signaling server and open the HTML file in a web browser—you should see an input field and a button on the page. At the top of the page, you should see a URL to be shared with the remote peer.

On another browser's window, open the sharing link. In the web browser's console, you should see the Data channel state is open message. Now, enter something in the input box and click on the Submit query button. You should see the message printed on another browser's window.

**How it works...**

When the application starts, it establishes a connection with the signaling server and gets a virtual room number. Then, another peer starts the application and enters the virtual room. The second peer is the caller. When the peer connection is established, the caller creates a new data channel and another peer receives this event notification. So, both peers get a data channel reference and can use it for data exchanging.

In our example, when the customer enters a message and clicks on the Submit query button, we will wrap the message into a JSON object and send it via the data channel. The remote peer gets the JSON object, parses it to the message, and displays it on the page.

**There's more...**

Using data channels, peers can exchange any kind of data. It can be plain text, for example, or binary data. Moreover, the same data channel can be used to exchange different sorts of data between peers. In this recipe, we used JSON to format messages, and every packet has a type field. To send text messages, we used the chatmessage type, but you can use your own custom type system to distinguish messages. You can also use something other than JSON. So, data channels are a good tool to exchange data between peers, using a secure and safe direct connection.
Implementing a chat using a signaling server

In this recipe, we will cover the process of implementing private, peer-to-peer web chat using signaling server as the middle point. Peers will send chat messages via the signaling server. In the schema represented in the following diagram, you can see the flow:

How to do it...

To implement the chat feature via the signaling server, we need to add some methods to the client code with the following steps:

1. We need to add appropriate code to the function that processes the messages from the signaling server:

   ```javascript
   function processSignalingMessage(message) {
     var msg = JSON.parse(message);
     if (msg.type === 'CHATMSG') {
       onChatMsgReceived(msg.value);
     } else if (msg.type === 'offer') {
       pc.setRemoteDescription(new RTCSessionDescription(msg));
       doAnswer();
     } else if (msg.type === 'answer') {
       // Add code for handling answer messages
     }
   }
   ```
pc.setRemoteDescription(new RTCSessionDescription(msg));
}

} else if (msg.type === 'candidate') {
  var candidate = new RTCIceCandidate({sdpMLineIndex: msg.label,
                                          candidate: msg.candidate});
  pc.addIceCandidate(candidate);
}

} else if (msg.type === 'GETROOM') {
  room = msg.value;
  onRoomReceived(room);
}

} else if (msg.type === 'WRONGROOM') {
  window.location.href = "/";
}

};

2. We will check whether the received message is of the CHATMSG type and if so, we will call the onChatMsgReceived method to process it:

function onChatMsgReceived(txt) {
  var chatArea = document.getElementById("chat_div");
  chatArea.innerHTML = chatArea.innerHTML + txt;
  chatArea.scrollTop = chatArea.scrollHeight;
}

Here, we will get the chat_div element by its ID and alter its content by adding the chat message received from the remote peer via the signaling server.

3. To send a chat message, we should implement a method like the following:

function chatSendMessage(msg) {
  if (!channelReady) return;
  sendMessage({"type" : "CHATMSG", "value" : msg});
}

This function checks whether the WebSocket channel is up and sends a chat message to the signaling server using the channel. To use this function, we can use the HTML input tag with the submit button and call it on the submit event.

How it works...

The basic principle of this solution is pretty simple:

- One peer sends a text message to the signaling server, marking it as the CHATMSG type
- The signaling server retransmits the message to another peer
- Another peer gets the message from the signaling server, checks whether it is of the CHATMSG type and if so, shows it to the user
To distinguish chat messages from WebRTC messages, you can use any word to mark the message type. It can be CHATMSG or whatever you prefer.

There's more...

This way of implementing web chat is usually not secure because the data will go via the signaling server and not directly through the peers. Nevertheless, it is suitable for public chat rooms where there can be several people at a time. For private peer-to-peer chats, it is usually better to use WebRTC data channels, and that way it is more secure.

See also

- To implement the chat feature using data channels, follow the Implementing a chat using data channels recipe

Configuring and using STUN

Your WebRTC application can work without STUN or TURN servers if all the peers are located in the same plain network. If your application is supposed to work for peers that might be located in different networks, it will definitely need to use at least the STUN server to work.

Getting ready

In this recipe, we will install a STUN server on a Linux box. STUN server can be installed under the other platform as well, but for simplicity, we will consider only the Linux case. So, please prepare a Linux machine.

In this recipe, we will use a very basic and simple STUN server implementation, so you probably will not need to install additional libraries or do some difficult configuration.

STUN needs two IP addresses to work correctly. Thus, when experimenting with your Linux box, take care that the Linux box should have at least two IP addresses that are available for all possible peers (WebRTC clients).

How to do it...

The following set of steps will lead you through the process of configuring and building a STUN service:

1. Download the STUN server from its home page at http://sourceforge.net/projects/stun/.
2. Unpack the archive and go into the STUN server folder:

```
tar -xzf stund-0.97.tgz
```

```
cd stund
```

3. Build it with the following command:

```
make
```

The last command will build the server. After that, you can start the STUN server by using the following command:

```
./server -h primary_ip -a secondary_ip
```

Note that instead of `primary_ip` and `secondary_ip`, you should use actual IP addresses that are available on the machine. This software can't detect such network parameters automatically, so you need to set it up explicitly.

If you want to start the server in the background, add the `-b` option to the preceding command.

Now, when the STUN server is configured and running, we can utilize it in the WebRTC application. When your application wants to create a peer connection object, it uses something like the following code:

```javascript
var pc;
pc = new RTCPeerConnection(configuration);
```

Here, `configuration` is an entity that contains different options for creating peer connection object. To utilize your freshly installed STUN server, you should use something like the following code:

```javascript
var configuration = {
  'iceServers': [
    {
      'url': 'stun:stun.myserver.com:19302'
    }
  ]
}
```

Here we inform the web browser that it can use the STUN server if necessary. Note that you should use the real domain name or IP address of the STUN server. You can also explicitly set the port number as shown in the preceding code, in case it is distinguished from the default value.
How it works...

STUN server can help peers determine their network parameters and thus establish a direct communication channel. If your clients are located behind NAT or firewall, your application should use at least the STUN service to make the direct connection possible. Nevertheless, in many cases that might not be enough, and using TURN might be necessary.

The following diagram might be helpful to you to imagine how the STUN server is located in the whole infrastructure, and how all the components interoperate with each other:

There's more...

As an alternative to this, you can use rfc5766-server—it is a free and open source implementation of both STUN and TURN servers. It also supports many additional features that might be quite useful. You can find it at https://code.google.com/p/rfc5766-turn-server/.

See also

- For details on how STUN works, you can refer to RFC #3489 http://www.ietf.org/rfc/rfc3489.txt.
- In the Configuring and using TURN recipe, we will use a TURN server based on the rfc5766-server software. That application can serve as a STUN server as well.
Configuring and using TURN

In most cases, it is enough to use a STUN server to establish a peer-to-peer direct connection. Nevertheless, you will often need to utilize TURN servers—mostly for clients located in big companies (because of firewall policy and tricky NAT) and some specific countries (because of firewalls and access limits).

Getting ready

In this section, we will download, install, and do the basic configuration of a TURN service. Then, we will utilize it in our WebRTC application. A TURN server can be installed under different platforms, although we will cover a Linux box use case only. Thus, for this recipe, you will need a Linux box installed.

For this recipe, we will use rfc5766-turn-server—a free and open source implementation of the TURN and STUN servers. Download its source code from its home page at https://code.google.com/p/rfc5766-turn-server/.

How to do it...

First, we will shortly cover the installation and basic configuration of the TURN server. After that, we will learn how to use it in the application.

If you have TURN server already installed, you can skip this section and go directly to the next one.

Installing the TURN server

I assume that you have downloaded rfc5766-server already and unpacked it. So, let’s install and configure your own TURN server:

1. Go to the rfc5766-server folder with the following command:
   
   cd ~/turnserver-4.1.2.1

2. Build the server:
   
   ./configure

   make

   sudo make install
Note that rfc5766-server needs some libraries that might be not installed on your system—in particular, libssl-dev, libevent-dev, and openssl. You should install the absent libraries to compile the software successfully.

3. After that, you can start the server—it will detect all the network options automatically:

```
turnserver
```

You will see diagnostic messages in the console:

```
0: ===========Discovering relay addresses: =============
0: Relay address to use: x.x.x.x
0: Relay address to use: y.y.y.y
0: Relay address to use: ::1
0: ==============------------------------------------------
0: Total: 3 relay addresses discovered
0
0: ==============------------------------------------------
```

To stop the server, just press Ctrl + C; you will get back to console.

Now it is time to perform some configuration steps and tune your fresh TURN server for your requirements.

By default, the TURN server doesn't have any configuration file. We need to create this configuration file from the default configuration file supplied with the server:

```
sudo cp /usr/local/etc/turnserver.conf.default /usr/local/etc/turnserver.conf
```

Open the turnserver.conf file and edit it according to your requirements. We will not cover all the TURN options here, but just basic configuration items that might be important:

- **Listening IP**: This option determines the IP addresses that will be used by the TURN server while operating. By default, this option will do it automatically. Nevertheless, it is a good idea to set the obvious IP addresses you would like the server to use:

  ```
  listening-ip=
  ```
Note that the TURN server needs at least two public IP addresses to operate correctly.

- **Relay IP**: In this option, you can explicitly set up IP address that should be used for relay. In other words, if you have two IP addresses, one of them can be `listening-ip` and the second one `relay-ip`.

  ```
  relay-ip=
  ```

- **Verbosity**: In this option, you can set a level of verbosity. By default, the TURN server will not print extra details on its work, but for debugging and diagnostic purposes, it might be very useful to set the verbose level to normal. For that, you should place the word `verbose` in the configuration file. If you would like to refer to more details, you should write the word with capital V—`Verbose`—so the server will print as much debugging details as possible.

  ```
  no-auth
  ```

- **Anonymous access**: You can enable anonymous access during the development process, if you're sure that your TURN server is protected by network firewall and nobody can use it. Otherwise, you should not enable this option especially on production systems:

  ```
  In this recipe, we haven't covered TURN authentication—this topic is covered in Chapter 2, Supporting Security.
  ```

At this stage, you have your own TURN server with basic configuration, which can be used in WebRTC applications.

### Using TURN in WebRTC application

When you create a peer connection object, you usually use some construction like the following one:

```javascript
var pc;
pc = new RTCPeerConnection(configuration);
```

Here, `configuration` is an entity that contains different options to create a peer connection object. To utilize your TURN server, you should use something like the following:

```javascript
var configuration = {
    'iceServers': [
    {
      'url': 'stun:stun.l.google.com:19302'
    },
    {
```
Here, we will ask the WebRTC API (actually, we will ask the web browser) to use one of three ways when establishing a peer connection:

- Public STUN server provided by Google.
- TURN server with anonymous access. You will use this notation to utilize the TURN server installed and configured in this recipe.
- TURN server with authentication. In Chapter 2, Supporting Security, we will cover the topic of security and authentication within the scope of a TURN server. To utilize a server that uses authentication, you should use this notation.

Note that you can ask the web browser to use a UDP or TCP protocol while establishing a peer connection through the TURN server. To do that, set up the transport parameter as shown in the preceding bullet points.

How it works...

In some cases, when clients use NAT and firewalls, it is impossible to establish a peer connection using STUN. This situation often appears when a client is located in a corporative network with a strict policy. In such a case, the only way to establish the connection is to use the TURN server.

The TURN server works as a proxy—all the data between peers (including audio, video, and service data) goes through the TURN server.
The following diagram shows how all the components operate with each other:

In this recipe, we covered only one TURN solution, open source and popular, but there are other solutions in the world that could also be suitable for you:

- **TurnServer**: This is also free and open source. For more information, refer to http://turnserver.sourceforge.net.
- **Numb**: This is not software that you can download and install, but a service where you can create an account and get access to a configured TURN server. For more details, refer to http://numb.viagenie.ca.

Of course, there are even more different solutions and services available.

**See also**

- For details on TURN servers, refer to RFC 5766 at http://tools.ietf.org/html/rfc5766
- For details regarding STUN (another useful technology with the scope of developing WebRTC-based services), you can also take a look at the *Configuring and using STUN* recipe
Where to buy this book
You can buy WebRTC Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.