Xirsys' XSDK Walkthrough:
Creating a WebRTC video chat application

Ed Rogers, Xirsys

December 22, 2015

Introduction

Welcome to XSDK; a public Javascript SDK for exploiting the power of the Xirsys platform.

XSDK is intended for use by Xirsys customers who have already familiarised themselves with at least the basic features of the Xirsys platform. Please read the following sources before this documentation;

- http://xirsys.com/developers/ and
- XirSys_v2_Platform_Documentation.pdf

This document will walk you through the process of building a full client-side WebRTC application for web browsers. You follow the progress through sequential HTML files in */webrtc_walkthough/*. The final file will mirror */examples/webrtc.html*.

It is not our purpose here to explain WebRTC itself. For that we recommend WebRTC for Beginners by Muaz Khan.

The XSDK is still developing, as is this documentation. If you would like clarification or assistance please pose us a question on Stack Overflow.

Contents

1 Signalling 3

1.1 Starting a new web application 3

1.2 Connecting to the signal server 3

1.3 Reacting to events 5

1.4 Keeping connection and account info seperate 6

1.5 An updating list of peers 7

1.6 Chat between peers 8

2 WebRTC Video 10

2.1 Introducing the p2p class 10

2.2 Taking calls 11

2.3 Making calls 13

2.4 Lights, camera, WebRTC 14

3 Tying up loose ends 16

3.1 Full screen videos 16

3.2 Server errors 16

3.3 Logging out 16

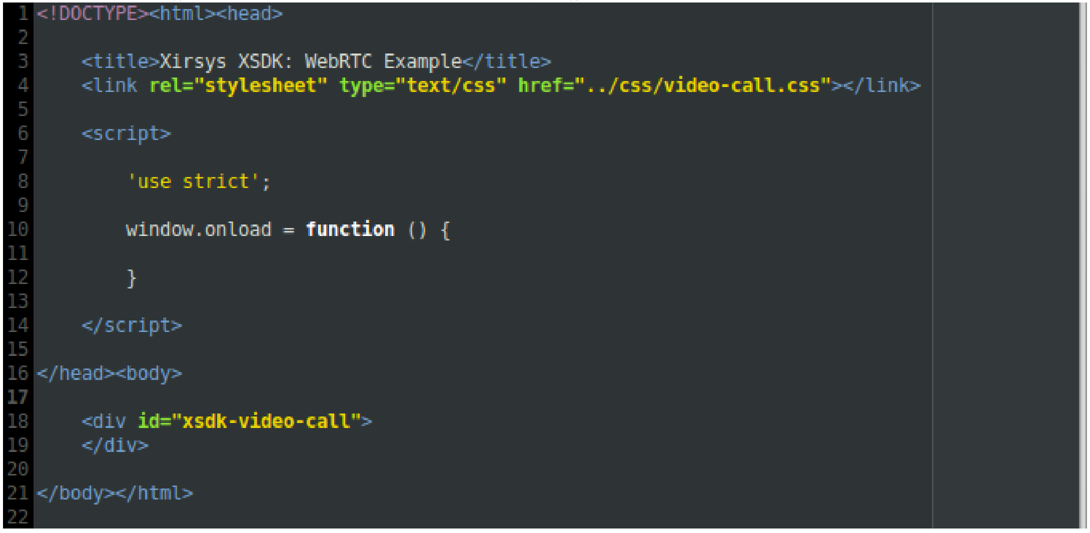
4 Next Steps 18

Chapter 1

Signalling

1.1 Starting a new web application

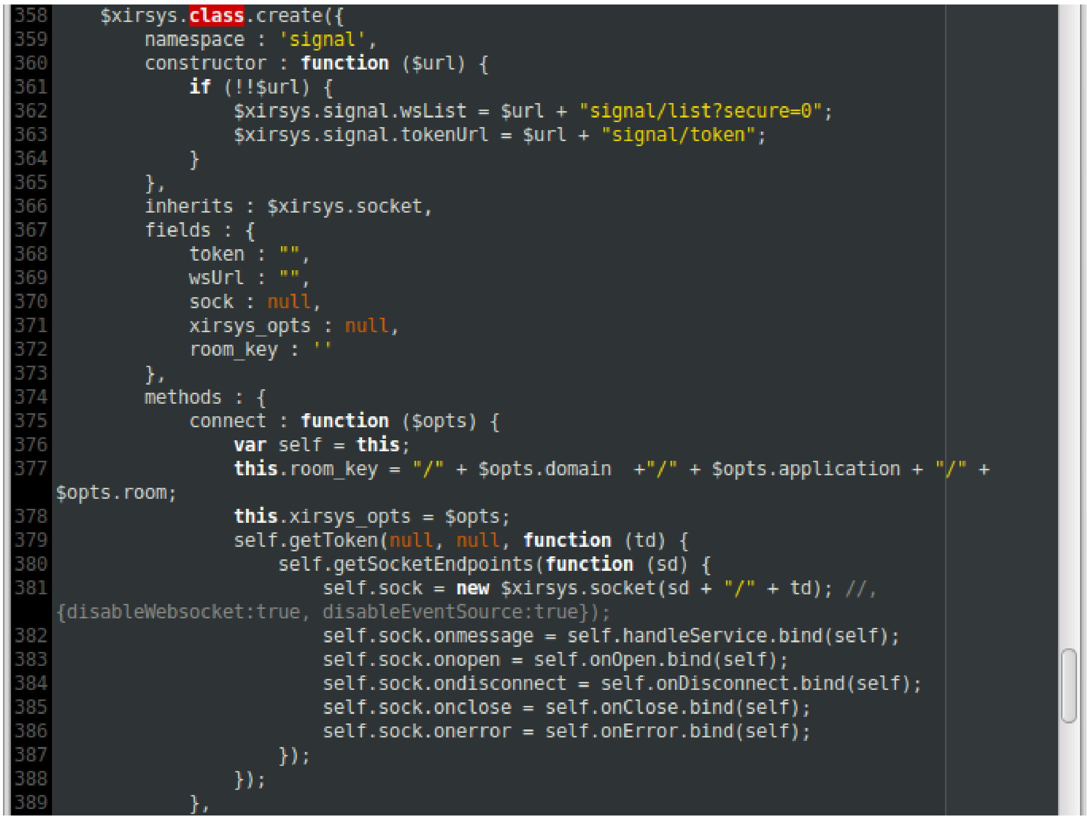
Lets start with a bare web page (*webrtc_walkthrough/1-1.html*). We will not concern ourselves here with presentation, so we're using a premade stylesheet.



```
<!DOCTYPE><html><head>

    <title>Xirsys XSDK: WebRTC Example</title>
    <link rel="stylesheet" type="text/css" href="../css/video-call.css"></link>

    <script>

        'use strict';

        window.onload = function () {

        }

    </script>

</head><body>

    <div id="xsdk-video-call">
    </div>

</body></html>

```

1.2 Connecting to the signal server

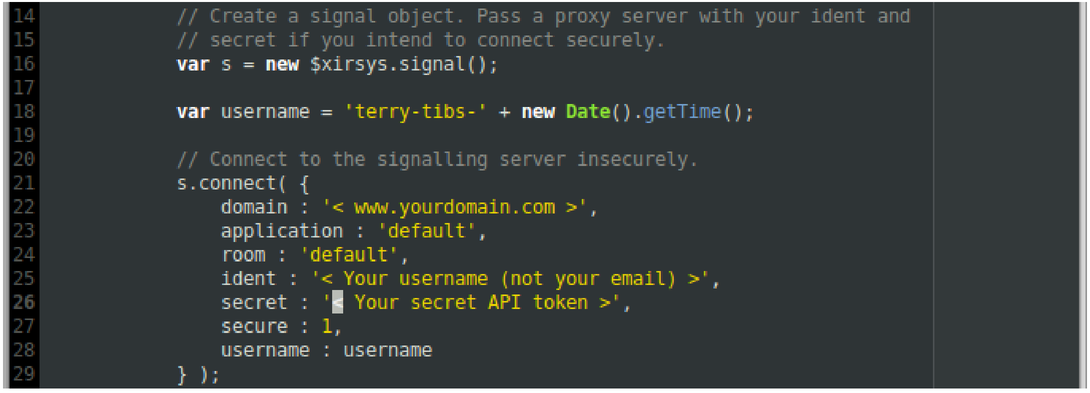
The first step is to add a connection to the Xirsys signalling server, which is made by */lib/signal.js* (all XSDK applications will also require */lib/core.js*). If we look in *signal.js* you will note that the XSDK uses a custom class system. Skip the *socket* class in that file for now and study the *signal* class further down the page.



```
    $xirsys.class.create({
        namespace : 'signal',
        constructor : function ($url) {
            if (!!$url) {
                $xirsys.signal.wsList = $url + "signal/list?secure=0";
                $xirsys.signal.tokenUrl = $url + "signal/token";
            }
        },
        inherits : $xirsys.socket,
        fields : {
            token : "",
            wsUrl : "",
            sock : null,
            xirsys_opts : null,
            room_key : ''
        },
        methods : {
            connect : function ($opts) {
                var self = this;
                this.room_key = "/" + $opts.domain  +"/" + $opts.application + "/" + $opts.room;
                this.xirsys_opts = $opts;
                self.getToken(null, null, function (td) {
                    self.getSocketEndpoints(function (sd) {
                        self.sock = new $xirsys.socket(sd + "/" + td); //, {disableWebsocket:true, disableEventSource:true});
                        self.sock.onmessage = self.handleService.bind(self);
                        self.sock.onopen = self.onOpen.bind(self);
                        self.sock.ondisconnect = self.onDisconnect.bind(self);
                        self.sock.onclose = self.onClose.bind(self);
                        self.sock.onerror = self.onError.bind(self);
                    });
                });
            },
```

You will need to instantiate a new signal object in the application and then call *connect* on it to form a connection with the server. As you will know after reading the developer information on the Xirsys website there are insecure and secure ways of connecting to the Xirsys server. In short; in order to form a connection you need get get a token from the server, which requires passing over the confidential *ident* and *secret* associated with your Xirsys account. It is best to use a proxy to do this, such as a PHP script acting as a go-between which stores the confidential information, forwarding and returning AJAX requests without having your *ident* or *secret* exposed in Javascript. To use such a proxy you will need to pass the URL when instantiating the object. However in the interests of simplicity we will by-pass this security in this walkthrough.

The *connect* method will call either the Xirsys server or your proxy (if specified) to get a token that authorises further communication. So unless you have a proxy you need to pass the *ident* and *secret* values. It also needs a *domain*, *application*, *room* and *username*, which the class uses to send messages, and *secure* for chosing between port 80 and 443. See http://xirsys.com/guide. To start with we will pass it a arbitrary name.

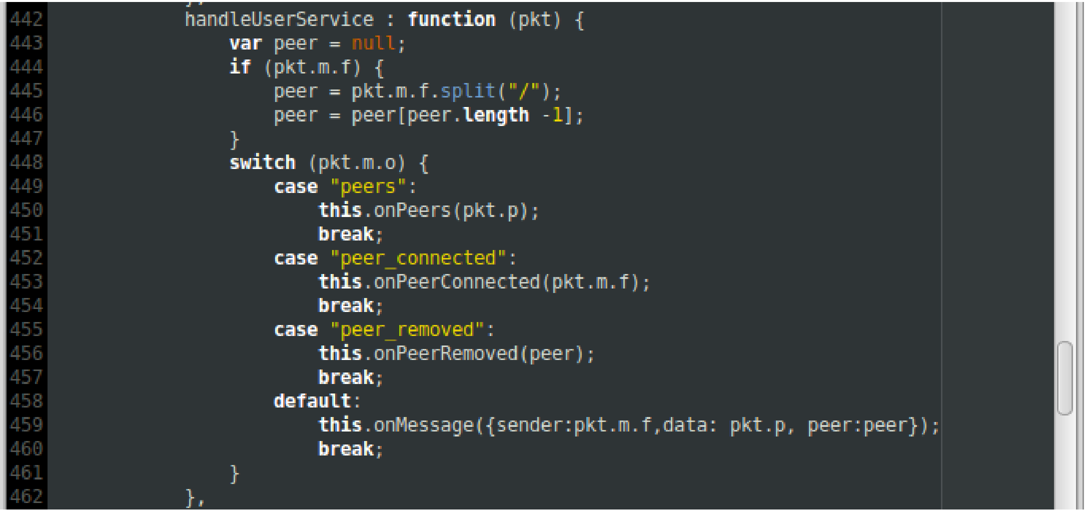


```
            // Create a signal object. Pass a proxy server with your ident and
            // secret if you intend to connect securely.
            var s = new $xirsys.signal();

            var username = 'terry-tibs-' + new Date().getTime();

            // Connect to the signalling server insecurely.
            s.connect( {
                domain : '< www.yourdomain.com >',
                application : 'default',
                room : 'default',
                ident : '< Your username (not your email) >',
                secret : '< Your secret API token >',
                secure : 1,
                username : username
            } );
```

At this point we have successfully connected to the signalling server and can pass messages back and forth.

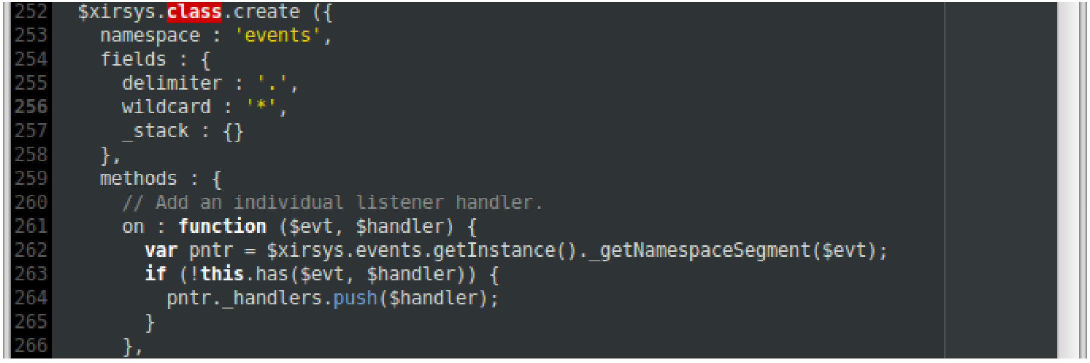
1.3 Reacting to events

The Xirsys signalling server sends a few different message types via web sockets. In the *signal* class these messages are handled by the *handleService* and *handleUserService* methods. These in turn call other methods for each message type; *onPeers*, *onPeerConnected*, *onPeerRemoved* and *onMessage* (for generic messages). Each of these emits an event through the XSDK's own events handler class. This also happens when the connection to the signalling server is opened, closed, disconnected or generates an error (*onOpen*, *onClose*, *onDisconnect* and *onError*, respectively).



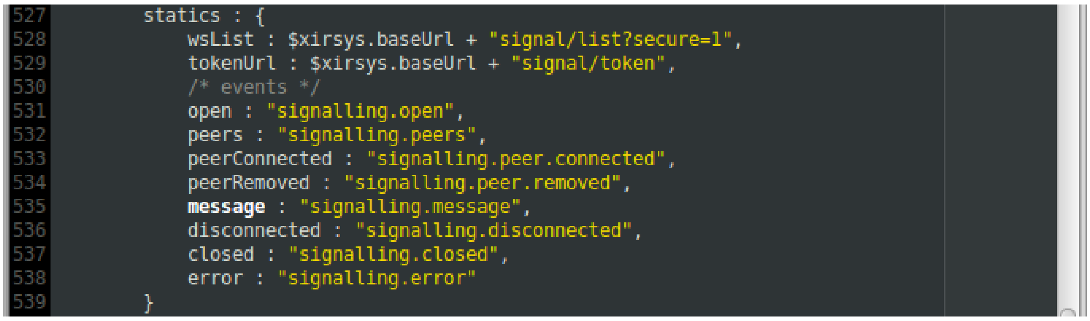
```
handleUserService : function (pkt) {
    var peer = null;
    if (pkt.m.f) {
        peer = pkt.m.f.split("/");
        peer = peer[peer.length -1];
    }
    switch (pkt.m.o) {
        case "peers":
            this.onPeers(pkt.p);
            break;
        case "peer_connected":
            this.onPeerConnected(pkt.m.f);
            break;
        case "peer_removed":
            this.onPeerRemoved(peer);
            break;
        default:
            this.onMessage({sender:pkt.m.f,data: pkt.p, peer:peer});
            break;
    }
},
```

The *events* class is in */lib/xirsys.core.js*. Its methods should be fairly self explanatory. Please note that events can include segments and wildcards, meaning you can for example listen to all events related to signalling, though in this instance we are listening for specific events only.



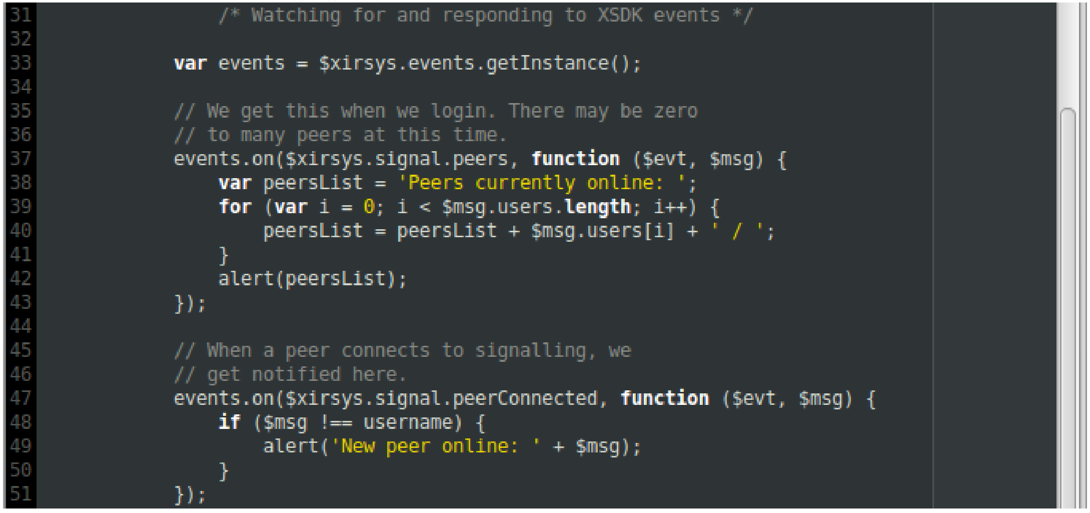
```
$xirsys.class.create ({
  namespace : 'events',
  fields : {
    delimiter : '.',
    wildcard : '*',
    _stack : {}
  },
  methods : {
    // Add an individual listener handler.
    on : function ($evt, $handler) {
      var pntr = $xirsys.events.getInstance()._getNamespaceSegment($evt);
      if (!this.has($evt, $handler)) {
        pntr._handlers.push($handler);
      }
    },
```

The events in question are strings defined as statics of the *signal* class. So in the application we will create two event listeners for the *signalling.peers* and *signalling.peersConnected* events.



```
        statics : {
            wsList : $xirsys.baseUrl + "signal/list?secure=1",
            tokenUrl : $xirsys.baseUrl + "signal/token",
            /* events */
            open : "signalling.open",
            peers : "signalling.peers",
            peerConnected : "signalling.peer.connected",
            peerRemoved : "signalling.peer.removed",
            message : "signalling.message",
            disconnected : "signalling.disconnected",
            closed : "signalling.closed",
            error : "signalling.error"
        }
```

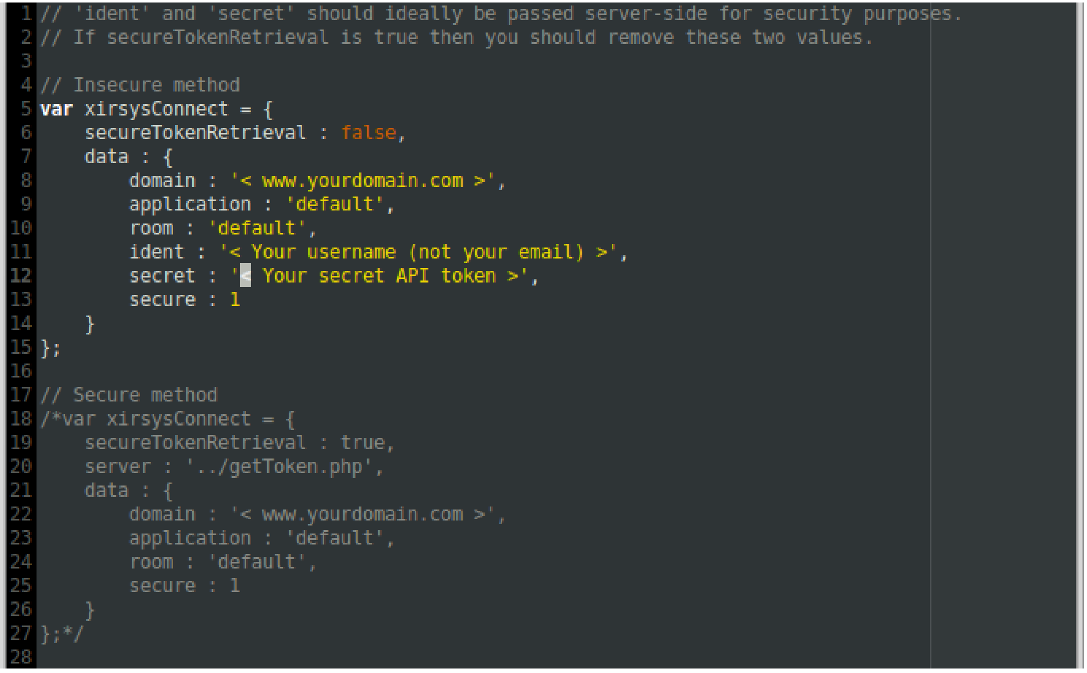
For now it suffices to alert the user to their peers occupying the same domain, application and room as them with a pop-up.



```
            /* Watching for and responding to XSDK events */

        var events = $xirsys.events.getInstance();

        // We get this when we login. There may be zero
        // to many peers at this time.
        events.on($xirsys.signal.peers, function ($evt, $msg) {
            var peersList = 'Peers currently online: ';
            for (var i = 0; i < $msg.users.length; i++) {
                peersList = peersList + $msg.users[i] + ' / ';
            }
            alert(peersList);
        });

        // When a peer connects to signalling, we
        // get notified here.
        events.on($xirsys.signal.peerConnected, function ($evt, $msg) {
            if ($msg !== username) {
                alert('New peer online: ' + $msg);
            }
        });
```

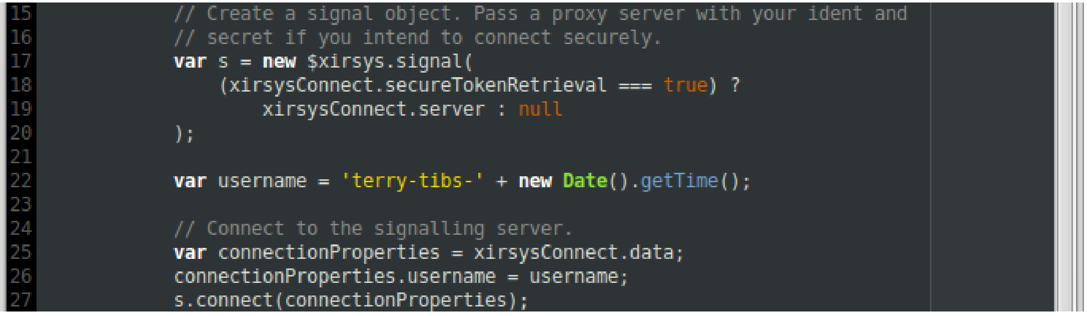
1.4 Keeping connection and account info seperate

Lets move the *ident*, *secret* and other connection information into their own file: *./xirsys_connect.js*.



```
// 'ident' and 'secret' should ideally be passed server-side for security purposes.
// If secureTokenRetrieval is true then you should remove these two values.

// Insecure method
var xirsysConnect = {
    secureTokenRetrieval : false,
    data : {
        domain : '< www.yourdomain.com >',
        application : 'default',
        room : 'default',
        ident : '< Your username (not your email) >',
        secret : '< Your secret API token >',
        secure : 1
    }
};

// Secure method
/*var xirsysConnect = {
    secureTokenRetrieval : true,
    server : '../getToken.php',
    data : {
        domain : '< www.yourdomain.com >',
        application : 'default',
        room : 'default',
        secure : 1
    }
};*/
```

This will make it easier to maintain your applications, especially if you want to run multiple applications on the same Xirsys account or switch back and forth from passing the *ident* and *secret* via Javascript for testing, to using a proxy server in anger.

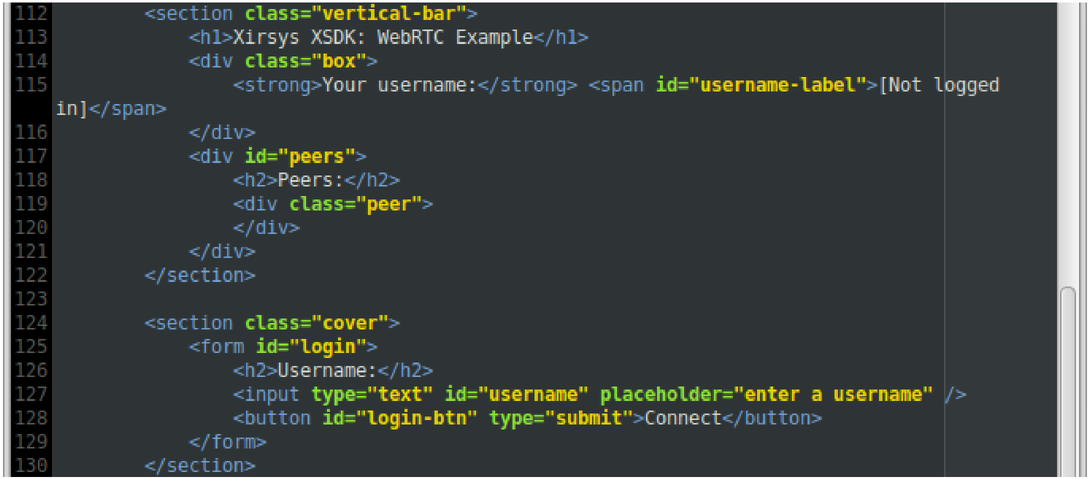


```
            // Create a signal object. Pass a proxy server with your ident and
            // secret if you intend to connect securely.
            var s = new $xirsys.signal(
                (xirsysConnect.secureTokenRetrieval === true) ?
                    xirsysConnect.server : null
            );

            var username = 'terry-tibs-' + new Date().getTime();

            // Connect to the signalling server.
            var connectionProperties = xirsysConnect.data;
            connectionProperties.username = username;
            s.connect(connectionProperties);
```

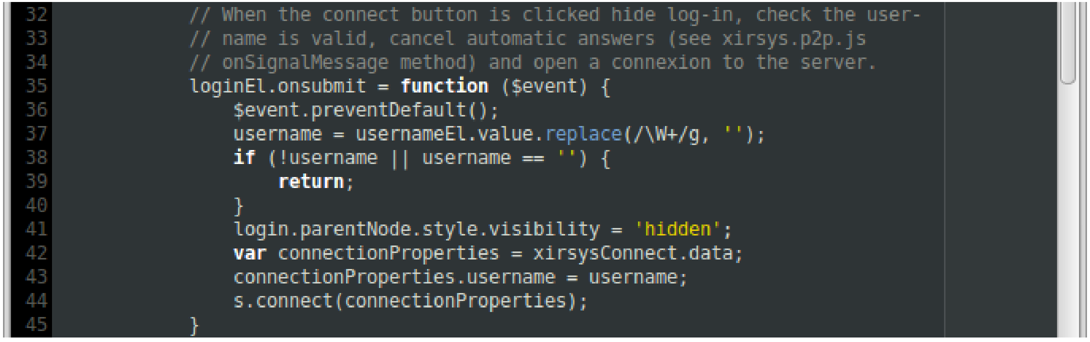
1.5 An updating list of peers

Time to add some HTML.

We'll start by asking the user to choose a valid username for themselves in a pop-up log-in form before making the connection with the signalling server.

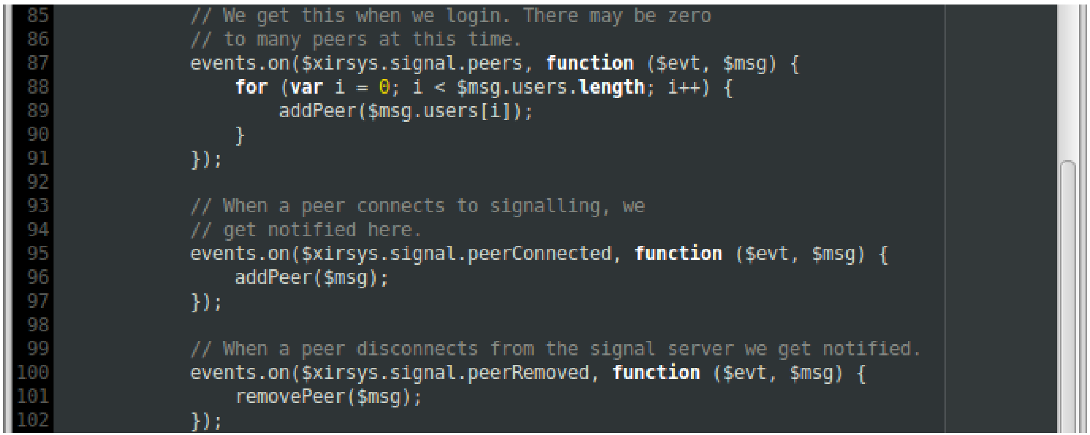


```
        <section class="vertical-bar">
            <h1>Xirsys XSDK: WebRTC Example</h1>
            <div class="box">
                <strong>Your username:</strong> <span id="username-label">[Not logged in]</span>
            </div>
            <div id="peers">
                <h2>Peers:</h2>
                <div class="peer">
                </div>
            </div>
        </section>

        <section class="cover">
            <form id="login">
                <h2>Username:</h2>
                <input type="text" id="username" placeholder="enter a username" />
                <button id="login-btn" type="submit">Connect</button>
            </form>
        </section>
```

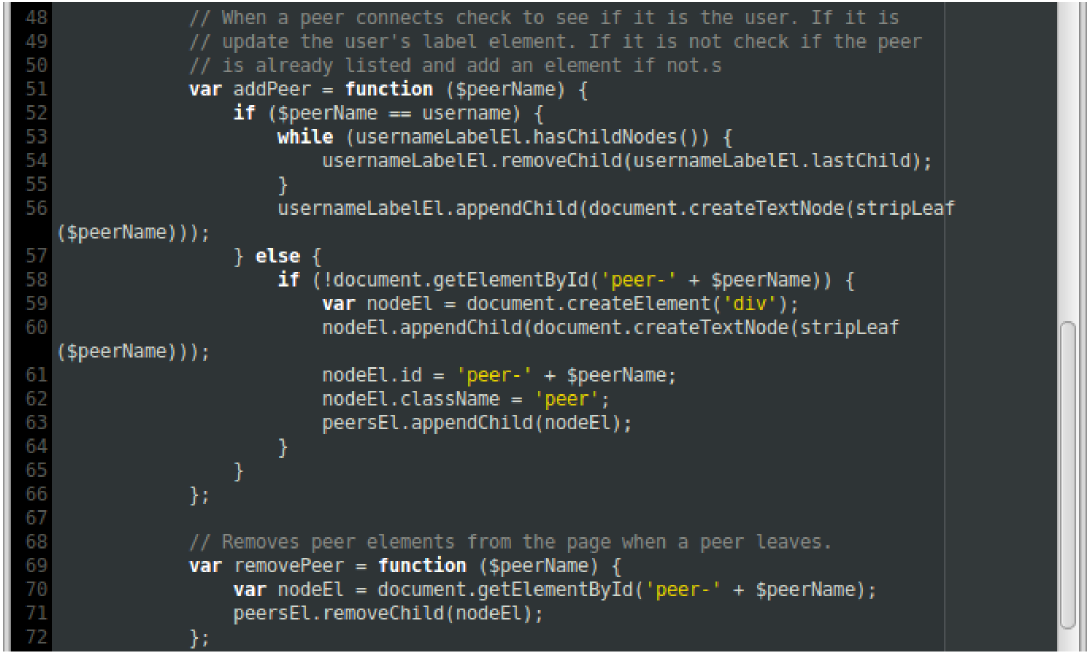


```
            // When the connect button is clicked hide log-in, check the user-
            // name is valid, cancel automatic answers (see xirsys.p2p.js
            // onSignalMessage method) and open a connexion to the server.
            loginEl.onsubmit = function ($event) {
                $event.preventDefault();
                username = usernameEl.value.replace(/\W+/g, '');
                if (!username || username == '') {
                    return;
                }
                login.parentNode.style.visibility = 'hidden';
                var connectionProperties = xirsysConnect.data;
                connectionProperties.username = username;
                s.connect(connectionProperties);
            }
```

Then we'll maintain a list of peers by adding another event listener (*peerRemoved*) and use the three events to manipulate the DOM.



```
            // We get this when we login. There may be zero
            // to many peers at this time.
            events.on($xirsys.signal.peers, function ($evt, $msg) {
                for (var i = 0; i < $msg.users.length; i++) {
                    addPeer($msg.users[i]);
                }
            });

            // When a peer connects to signalling, we
            // get notified here.
            events.on($xirsys.signal.peerConnected, function ($evt, $msg) {
                addPeer($msg);
            });

            // When a peer disconnects from the signal server we get notified.
            events.on($xirsys.signal.peerRemoved, function ($evt, $msg) {
                removePeer($msg);
            });
```

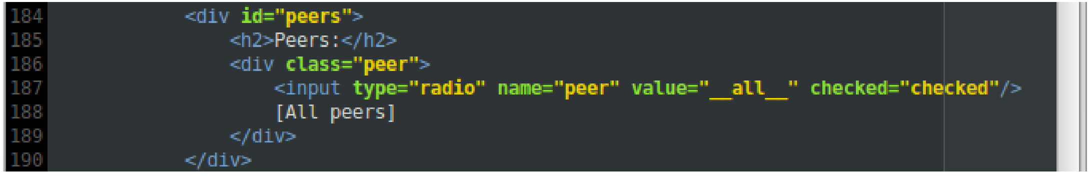


```
            // When a peer connects check to see if it is the user. If it is
            // update the user's label element. If it is not check if the peer
            // is already listed and add an element if not.s
            var addPeer = function ($peerName) {
                if ($peerName == username) {
                    while (usernameLabelEl.hasChildNodes()) {
                        usernameLabelEl.removeChild(usernameLabelEl.lastChild);
                    }
                    usernameLabelEl.appendChild(document.createTextNode(stripLeaf
($peerName)));
                } else {
                    if (!document.getElementById('peer-' + $peerName)) {
                        var nodeEl = document.createElement('div');
                        nodeEl.appendChild(document.createTextNode(stripLeaf
($peerName)));
                        nodeEl.id = 'peer-' + $peerName;
                        nodeEl.className = 'peer';
                        peersEl.appendChild(nodeEl);
                    }
                }
            };

            // Removes peer elements from the page when a peer leaves.
            var removePeer = function ($peerName) {
                var nodeEl = document.getElementById('peer-' + $peerName);
                peersEl.removeChild(nodeEl);
            };
```

1.6 Chat between peers

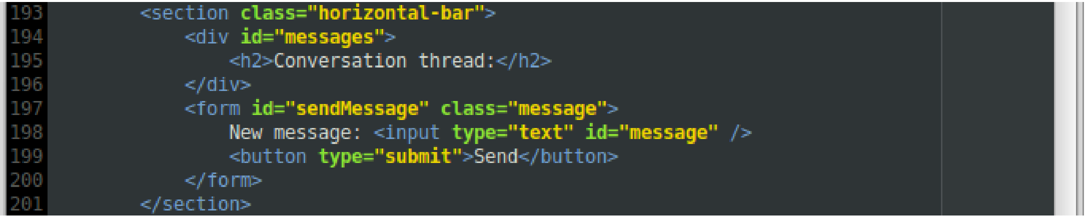
At this point lets introduce some actual functionality: a chat room.

The Xirsys signalling server can send messages to all peers or to an individual peer. To make use of this the user will need to be able to choose from the peers list, which we will do by adding an ‘all peers’ option and associating each peer with a radio button.

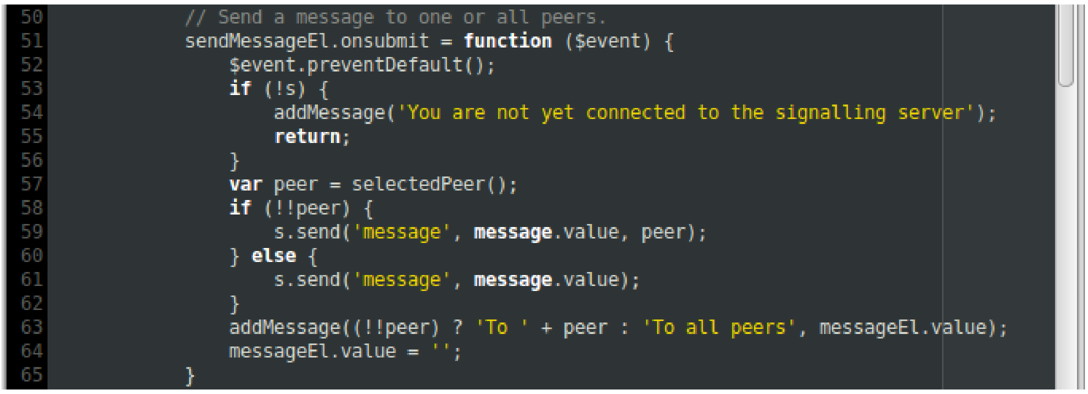


```
            <div id="peers">
                <h2>Peers:</h2>
                <div class="peer">
                    <input type="radio" name="peer" value="__all__" checked="checked"/>
                    [All peers]
                </div>
            </div>
```

Now we’ll add a second form to the application, this one for sending messages, and handle its input with a function which calls the ‘send’ method of the *signal* class.

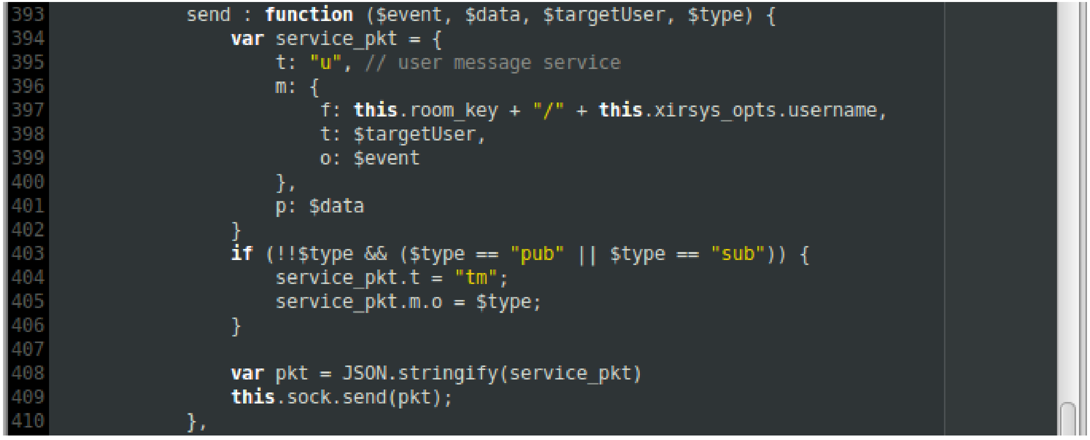


```
        <section class="horizontal-bar">
            <div id="messages">
                <h2>Conversation thread:</h2>
            </div>
            <form id="sendMessage" class="message">
                New message: <input type="text" id="message" />
                <button type="submit">Send</button>
            </form>
        </section>
```



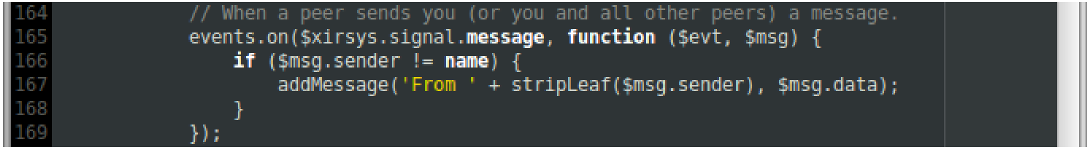
```
            // Send a message to one or all peers.
            sendMessageEl.onsubmit = function ($event) {
                $event.preventDefault();
                if (!s) {
                    addMessage('You are not yet connected to the signalling server');
                    return;
                }
                var peer = selectedPeer();
                if (!!peer) {
                    s.send('message', message.value, peer);
                } else {
                    s.send('message', message.value);
                }
                addMessage((!!peer) ? 'To ' + peer : 'To all peers', messageEl.value);
                messageEl.value = '';
            }
```

You will note that this method takes four arguments: *$event*, *$data*, *$targetUser* and *$type*. The last two are optional. If no *$targetUser* is specified the signalling server will send the message to all peers.

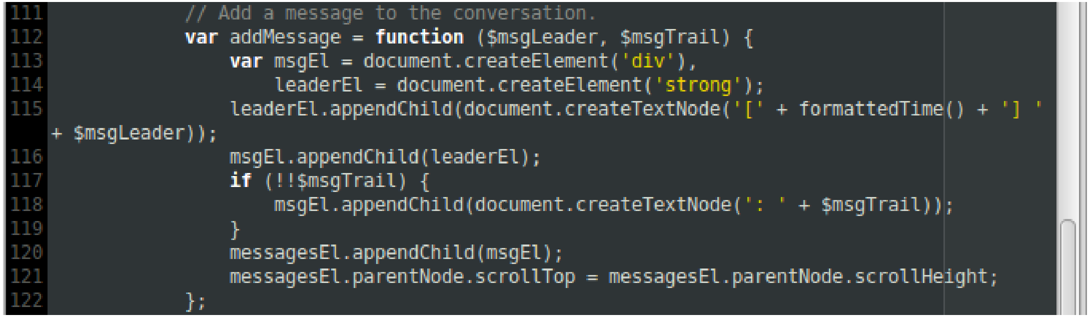


```
            send : function ($event, $data, $targetUser, $type) {
                var service_pkt = {
                    t: "u", // user message service
                    m: {
                        f: this.room_key + "/" + this.xirsys_opts.username,
                        t: $targetUser,
                        o: $event
                    },
                    p: $data
                }
                if (!!$type && ($type == "pub" || $type == "sub")) {
                    service_pkt.t = "tm";
                    service_pkt.m.o = $type;
                }

                var pkt = JSON.stringify(service_pkt)
                this.sock.send(pkt);
            },
```

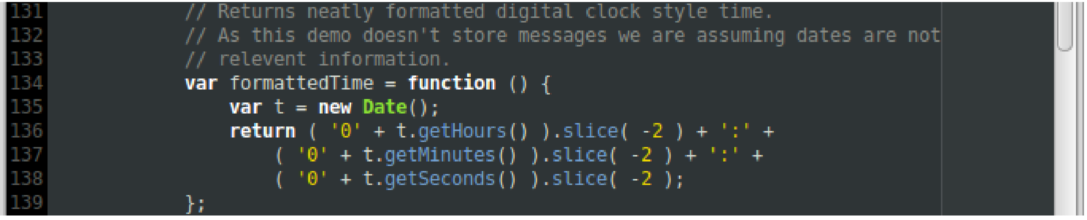
Receiving such messages requires another event listener, more DOM manipulation and few extra touches like time stamps to make it feel like a proper chat room.



```
            // When a peer sends you (or you and all other peers) a message.
            events.on($xirsys.signal.message, function ($evt, $msg) {
                if ($msg.sender != name) {
                    addMessage('From ' + stripLeaf($msg.sender), $msg.data);
                }
            });
```



```
            // Add a message to the conversation.
            var addMessage = function ($msgLeader, $msgTrail) {
                var msgEl = document.createElement('div'),
                    leaderEl = document.createElement('strong');
                leaderEl.appendChild(document.createTextNode('[' + formattedTime() + '] '
 + $msgLeader));
                msgEl.appendChild(leaderEl);
                if (!!$msgTrail) {
                    msgEl.appendChild(document.createTextNode(': ' + $msgTrail));
                }
                messagesEl.appendChild(msgEl);
                messagesEl.parentNode.scrollTop = messagesEl.parentNode.scrollHeight;
            };
```



```
            // Returns neatly formatted digital clock style time.
            // As this demo doesn't store messages we are assuming dates are not
            // relevent information.
            var formattedTime = function () {
                var t = new Date();
                return ( '0' + t.getHours() ).slice( -2 ) + ':' +
                    ( '0' + t.getMinutes() ).slice( -2 ) + ':' +
                    ( '0' + t.getSeconds() ).slice( -2 );
            };
```

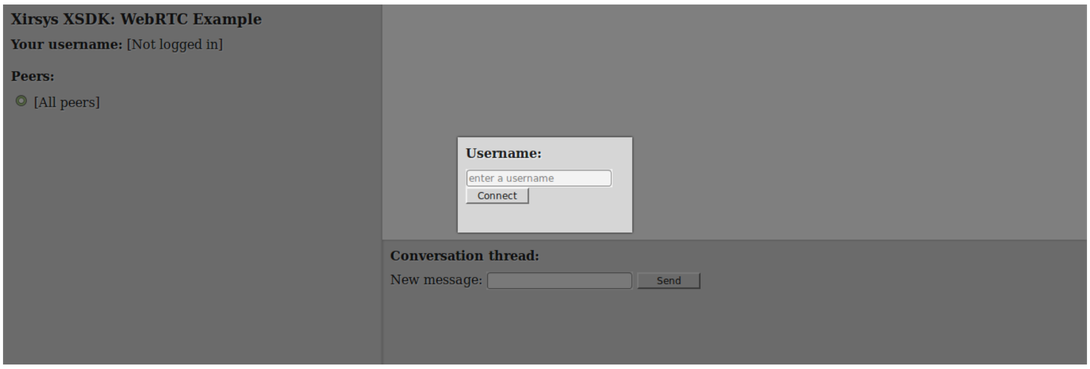
Please note that the Xirsys signalling server only acts as a conduit and as such does not store messages.

Chapter 2

WebRTC Video

2.1 Introducing the p2p class

Assuming you are using the default stylesheet you couldn't fail to notice that by now, although you have a fully-functional chat room application, there is a large hole on the top right of the screen. This we shall now endeavour to fill.



Xirsys XSDK: WebRTC Example

Your username: [Not logged in]

Peers:

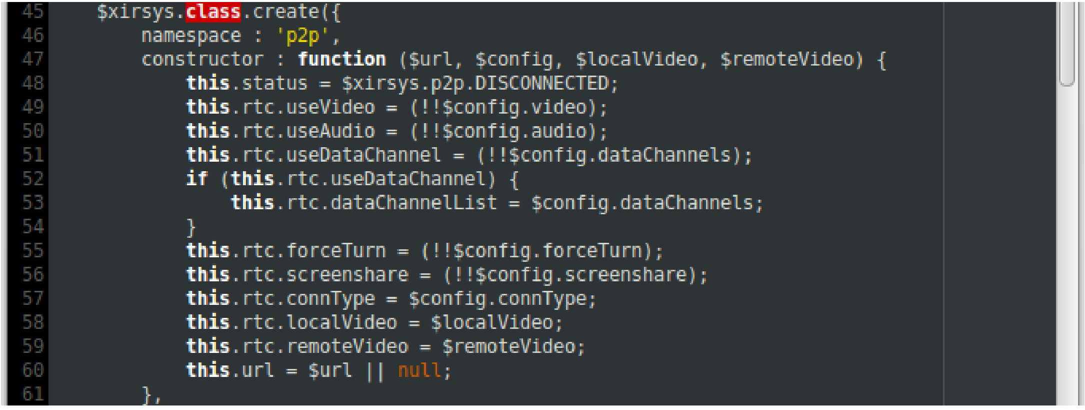
○ [All peers]

Username:

Conversation thread:

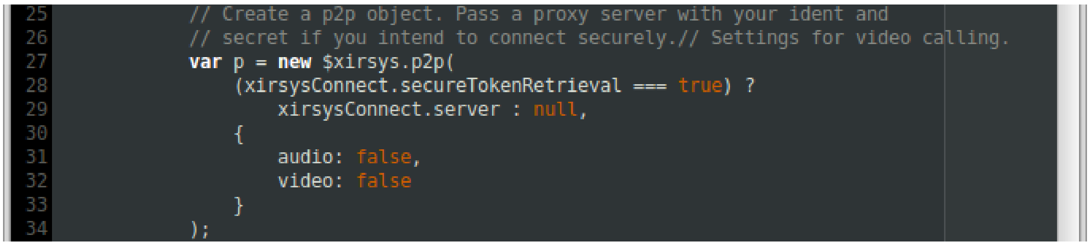
New message:

But before adding any functionality we'll convert what we've done so far to use the *p2p* class from */lib/xirsys.p2p.js*, rather than the *signal* class. The former relies upon the latter and exposes an instance of it as a property, which we shall use in our application.



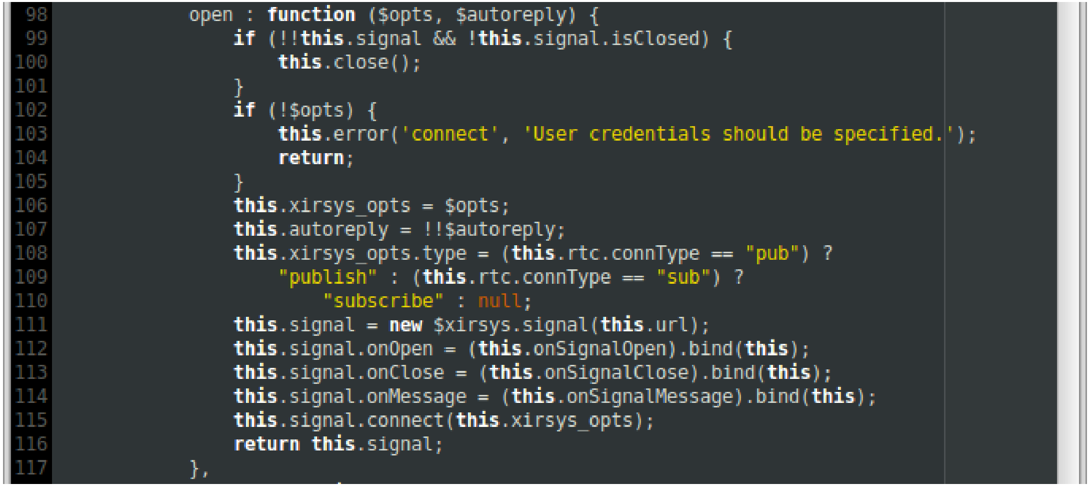
```
    $xirsys.class.create({
        namespace : 'p2p',
        constructor : function ($url, $config, $localVideo, $remoteVideo) {
            this.status = $xirsys.p2p.DISCONNECTED;
            this.rtc.useVideo = (!!$config.video);
            this.rtc.useAudio = (!!$config.audio);
            this.rtc.useDataChannel = (!!$config.dataChannels);
            if (this.rtc.useDataChannel) {
                this.rtc.dataChannelList = $config.dataChannels;
            }
            this.rtc.forceTurn = (!!$config.forceTurn);
            this.rtc.screenshare = (!!$config.screenshare);
            this.rtc.connType = $config.connType;
            this.rtc.localVideo = $localVideo;
            this.rtc.remoteVideo = $remoteVideo;
            this.url = $url || null;
        },
```

Unlike the *signal* class, instantiating *p2p* requires a second argument (*$config*). For now we will use this argument to tell the instance not to try and attach video and audio feeds (by default it attempts to do this).

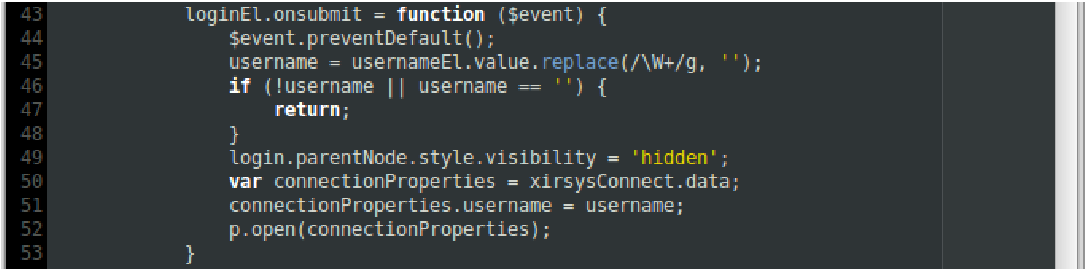


```
            // Create a p2p object. Pass a proxy server with your ident and
            // secret if you intend to connect securely.// Settings for video calling.
            var p = new $xirsys.p2p(
                (xirsysConnect.secureTokenRetrieval === true) ?
                    xirsysConnect.server : null,
                {
                    audio: false,
                    video: false
                }
            );
```

The *p2p* class' *open* method creates an instance of the *signal* class, binds its own methods to the instance's web socket events and calls *connect* on it. There is also a second argument (*$autoreply*) which we will come to later.



```
            open : function ($opts, $autoreply) {
                if (!!this.signal && !this.signal.isClosed) {
                    this.close();
                }
                if (!$opts) {
                    this.error('connect', 'User credentials should be specified.');
                    return;
                }
                this.xirsys_opts = $opts;
                this.autoreply = !!$autoreply;
                this.xirsys_opts.type = (this.rtc.connType == "pub") ?
                    "publish" : (this.rtc.connType == "sub") ?
                        "subscribe" : null;
                this.signal = new $xirsys.signal(this.url);
                this.signal.onOpen = (this.onSignalOpen).bind(this);
                this.signal.onClose = (this.onSignalClose).bind(this);
                this.signal.onMessage = (this.onSignalMessage).bind(this);
                this.signal.connect(this.xirsys_opts);
                return this.signal;
            },
```

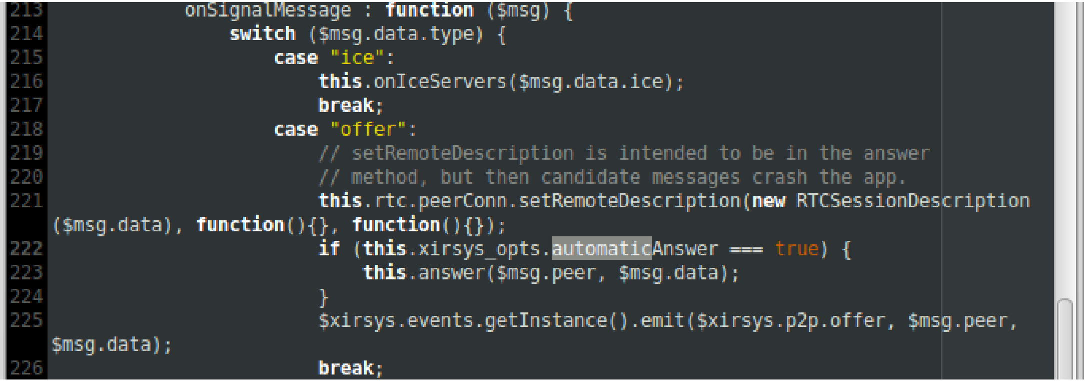


```
            loginEl.onsubmit = function ($event) {
                $event.preventDefault();
                username = usernameEl.value.replace(/\W+/g, '');
                if (!username || username == '') {
                    return;
                }
                login.parentNode.style.visibility = 'hidden';
                var connectionProperties = xirsysConnect.data;
                connectionProperties.username = username;
                p.open(connectionProperties);
            }
```

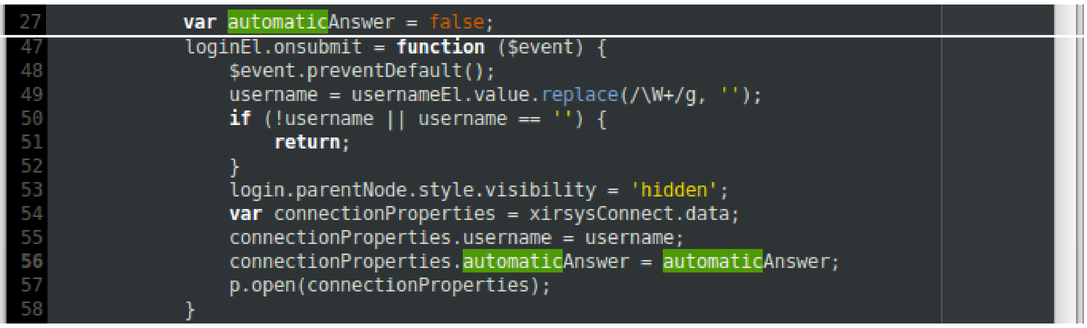
There is no need to update the event listeners we have in place so far as they will all still be coming through with the *signalling* prefix.

2.2 Taking calls

The next step is to introduce the capacity for users to take calls from other peers. This requires the */lib/xirsys.api.js* and */lib/xirsys.p2p.adapter.js* scripts. By default the *p2p* class will automatically answer calls, so we need to modify the *connectionProperties* object to cancel this behaviour.

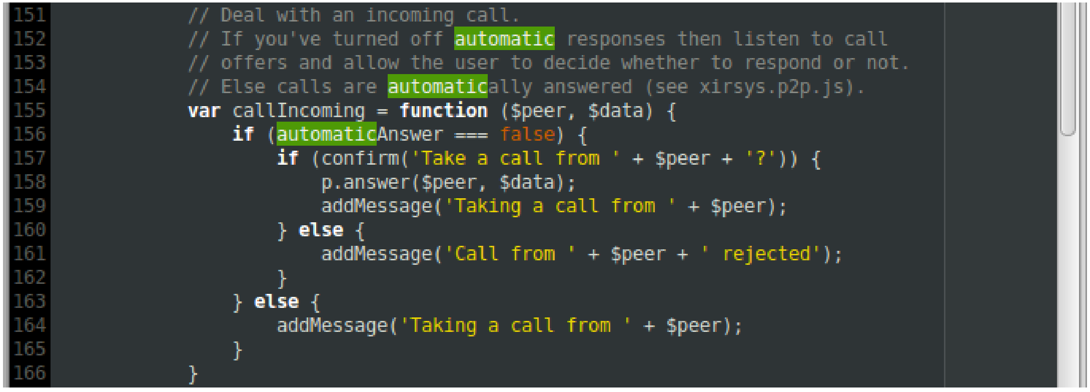


```
            onSignalMessage : function ($msg) {
                switch ($msg.data.type) {
                    case "ice":
                        this.onIceServers($msg.data.ice);
                        break;
                    case "offer":
                        // setRemoteDescription is intended to be in the answer
                        // method, but then candidate messages crash the app.
                        this.rtc.peerConn.setRemoteDescription(new RTCSessionDescription
($msg.data), function(){}, function(){});
                        if (this.xirsys_opts.automaticAnswer === true) {
                            this.answer($msg.peer, $msg.data);
                        }
                        $xirsys.events.getInstance().emit($xirsys.p2p.offer, $msg.peer,
$msg.data);
                        break;
```



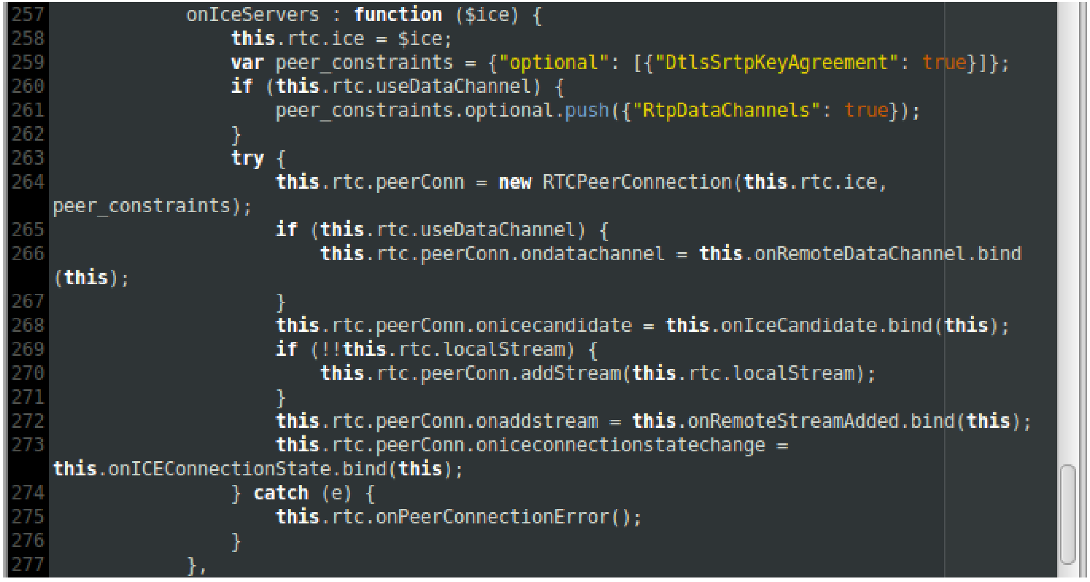
```
            var automaticAnswer = false;
            loginEl.onsubmit = function ($event) {
                $event.preventDefault();
                username = usernameEl.value.replace(/\W+/g, '');
                if (!username || username == '') {
                    return;
                }
                login.parentNode.style.visibility = 'hidden';
                var connectionProperties = xirsysConnect.data;
                connectionProperties.username = username;
                connectionProperties.automaticAnswer = automaticAnswer;
                p.open(connectionProperties);
            }
```

Incoming calls cause *offer* events, which we will listen to and present to the user as a confirmation pop-up. If the user accepts the offer then the *answer* method of the *p2p* class will call *createAnswer* on the instance's *peerConn* property.

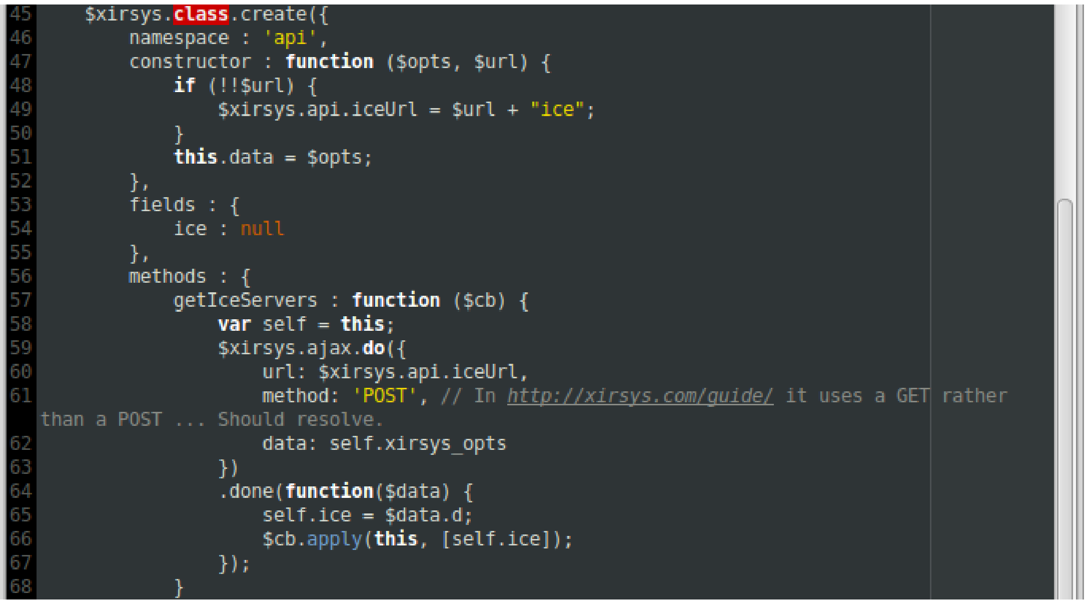


```
            // Deal with an incoming call.
            // If you've turned off automatic responses then listen to call
            // offers and allow the user to decide whether to respond or not.
            // Else calls are automatically answered (see xirsys.p2p.js).
            var callIncoming = function ($peer, $data) {
                if (automaticAnswer === false) {
                    if (confirm('Take a call from ' + $peer + '?')) {
                        p.answer($peer, $data);
                        addMessage('Taking a call from ' + $peer);
                    } else {
                        addMessage('Call from ' + $peer + ' rejected');
                    }
                } else {
                    addMessage('Taking a call from ' + $peer);
                }
            }
```

The *peerConn* property is set once the ICE servers information has been received (see *onIceServers* in the *p2p* class and *getIceServers* in the *api* class), when it is defined as an instance of *RTCPeerConnection*. The */lib/xirsys.p2p.adapter.js* script will have already ensured cross-browser compatibility for this and other WebRTC methods which are not yet stable standards.



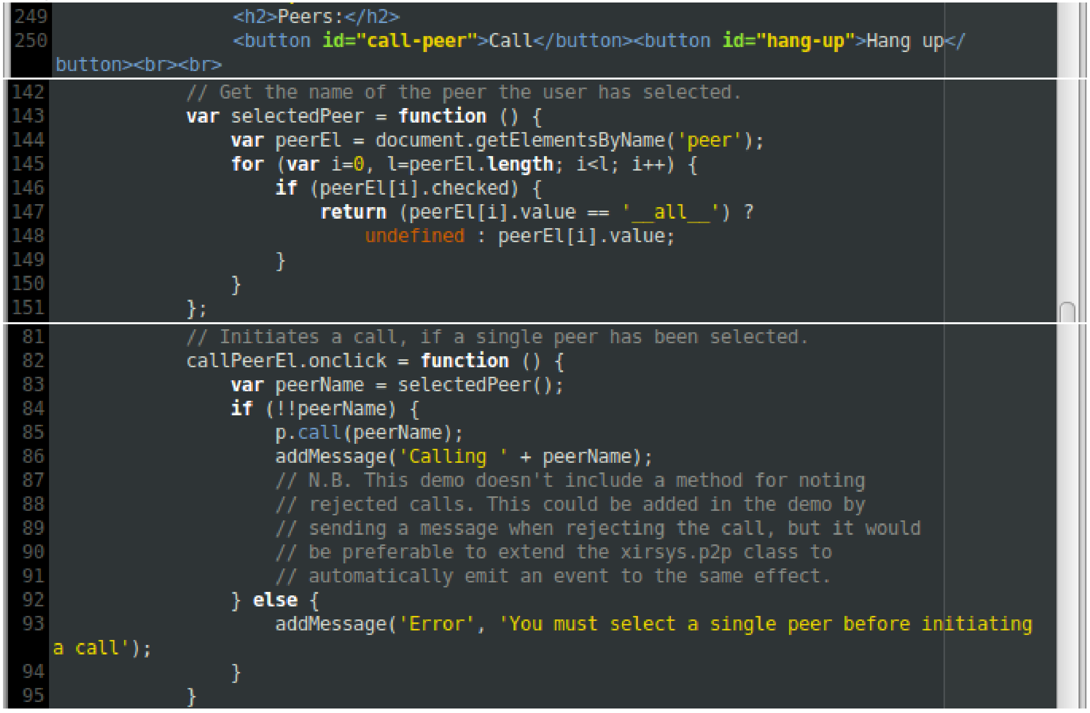
```
            onIceServers : function ($ice) {
                this.rtc.ice = $ice;
                var peer_constraints = {"optional": [{"DtlsSrtpKeyAgreement": true}]};
                if (this.rtc.useDataChannel) {
                    peer_constraints.optional.push({"RtpDataChannels": true});
                }
                try {
                    this.rtc.peerConn = new RTCPeerConnection(this.rtc.ice,
peer_constraints);
                    if (this.rtc.useDataChannel) {
                        this.rtc.peerConn.ondatachannel = this.onRemoteDataChannel.bind
(this);
                    }
                    this.rtc.peerConn.onicecandidate = this.onIceCandidate.bind(this);
                    if (!!this.rtc.localStream) {
                        this.rtc.peerConn.addStream(this.rtc.localStream);
                    }
                    this.rtc.peerConn.onaddstream = this.onRemoteStreamAdded.bind(this);
                    this.rtc.peerConn.oniceconnectionstatechange =
this.onICEConnectionState.bind(this);
                } catch (e) {
                    this.rtc.onPeerConnectionError();
                }
            },
```



```
$xirsys.class.create({
    namespace : 'api',
    constructor : function ($opts, $url) {
        if (!!$url) {
            $xirsys.api.iceUrl = $url + "ice";
        }
        this.data = $opts;
    },
    fields : {
        ice : null
    },
    methods : {
        getIceServers : function ($cb) {
            var self = this;
            $xirsys.ajax.do({
                url: $xirsys.api.iceUrl,
                method: 'POST', // In http://xirsys.com/guide/ it uses a GET rather than a POST ... Should resolve.
                data: self.xirsys_opts
            })
            .done(function($data) {
                self.ice = $data.d;
                $cb.apply(this, [self.ice]);
            });
        }
```

2.3 Making calls

Having built the receiver, we need a dialler.

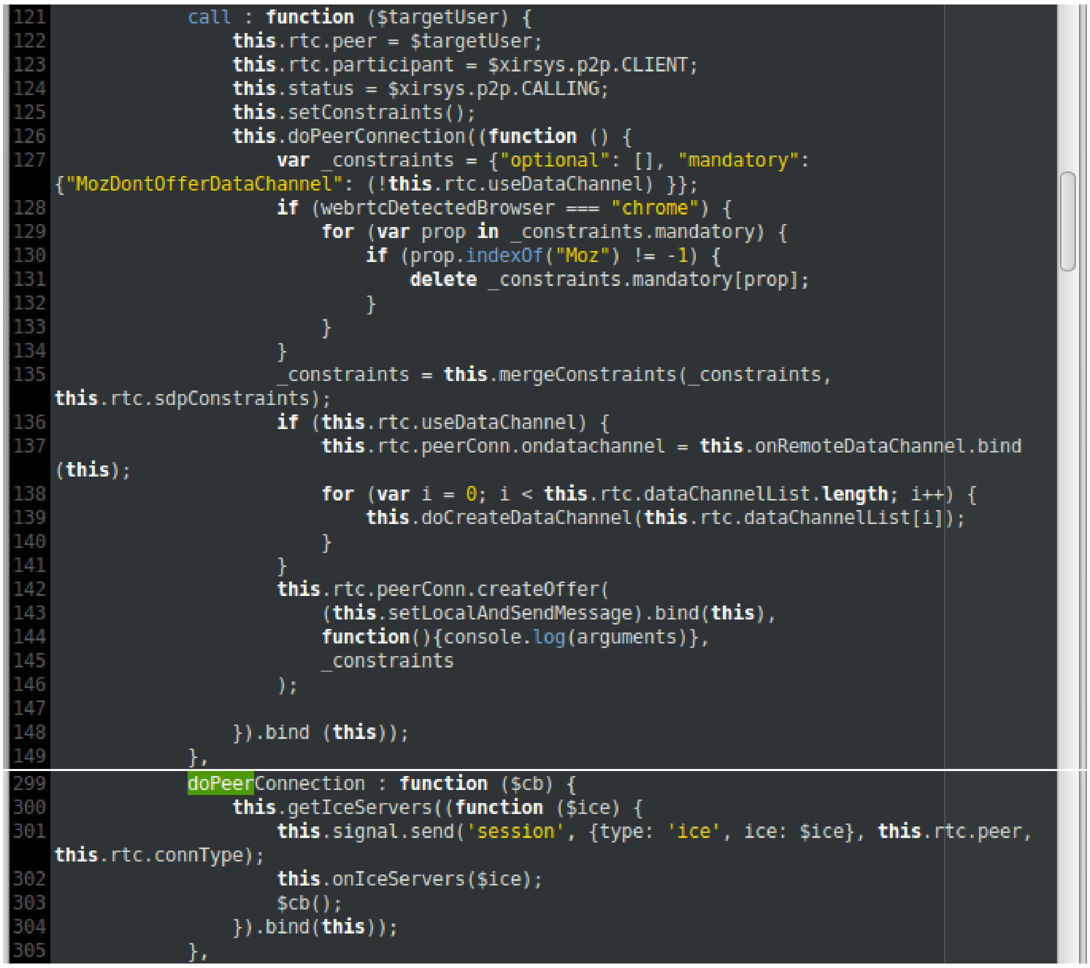


```
<h2>Peers:</h2>
<button id="call-peer">Call</button><button id="hang-up">Hang up</button><br><br>
```

```
// Get the name of the peer the user has selected.
var selectedPeer = function () {
    var peerEl = document.getElementsByName('peer');
    for (var i=0, l=peerEl.length; i<l; i++) {
        if (peerEl[i].checked) {
            return (peerEl[i].value == '__all__') ?
                undefined : peerEl[i].value;
        }
    }
};
```

```
// Initiates a call, if a single peer has been selected.
callPeerEl.onclick = function () {
    var peerName = selectedPeer();
    if (!!peerName) {
        p.call(peerName);
        addMessage('Calling ' + peerName);
        // N.B. This demo doesn't include a method for noting
        // rejected calls. This could be added in the demo by
        // sending a message when rejecting the call, but it would
        // be preferable to extend the xirsys.p2p class to
        // automatically emit an event to the same effect.
    } else {
        addMessage('Error', 'You must select a single peer before initiating a call');
    }
}
```

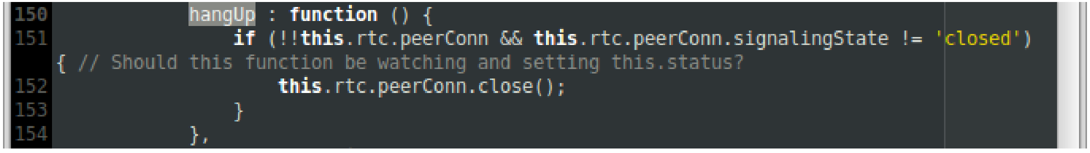
When a user calls the peer they have selected from the list the *call* method of *p2p* forms a peer connection with the *doPeerConnection* method. When the connection is made an offer is sent to the peer in question, and received via the event listener described above.



```
            call : function ($targetUser) {
                this.rtc.peer = $targetUser;
                this.rtc.participant = $xirsys.p2p.CLIENT;
                this.status = $xirsys.p2p.CALLING;
                this.setConstraints();
                this.doPeerConnection((function () {
                    var _constraints = {"optional": [], "mandatory":
{"MozDontOfferDataChannel": (!this.rtc.useDataChannel) }};
                    if (webrtcDetectedBrowser === "chrome") {
                        for (var prop in _constraints.mandatory) {
                            if (prop.indexOf("Moz") != -1) {
                                delete _constraints.mandatory[prop];
                            }
                        }
                    }
                    _constraints = this.mergeConstraints(_constraints,
this.rtc.sdpConstraints);
                    if (this.rtc.useDataChannel) {
                        this.rtc.peerConn.ondatachannel = this.onRemoteDataChannel.bind
(this);
                        for (var i = 0; i < this.rtc.dataChannelList.length; i++) {
                            this.doCreateDataChannel(this.rtc.dataChannelList[i]);
                        }
                    }
                    this.rtc.peerConn.createOffer(
                        (this.setLocalAndSendMessage).bind(this),
                        function(){console.log(arguments)},
                        _constraints
                    );

                }).bind (this));
            },
```

```
            doPeerConnection : function ($cb) {
                this.getIceServers((function ($ice) {
                    this.signal.send('session', {type: 'ice', ice: $ice}, this.rtc.peer,
this.rtc.connType);
                    this.onIceServers($ice);
                    $cb();
                }).bind(this));
            },
```

The *hangUp* method simply closes the peer connection, if there is one.

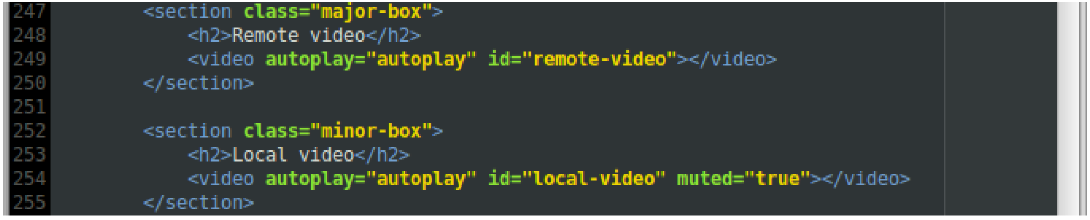


```
            hangUp : function () {
                if (!!this.rtc.peerConn && this.rtc.peerConn.signalingState != 'closed')
{ // Should this function be watching and setting this.status?
                    this.rtc.peerConn.close();
                }
            },
```

2.4 Lights, camera, WebRTC

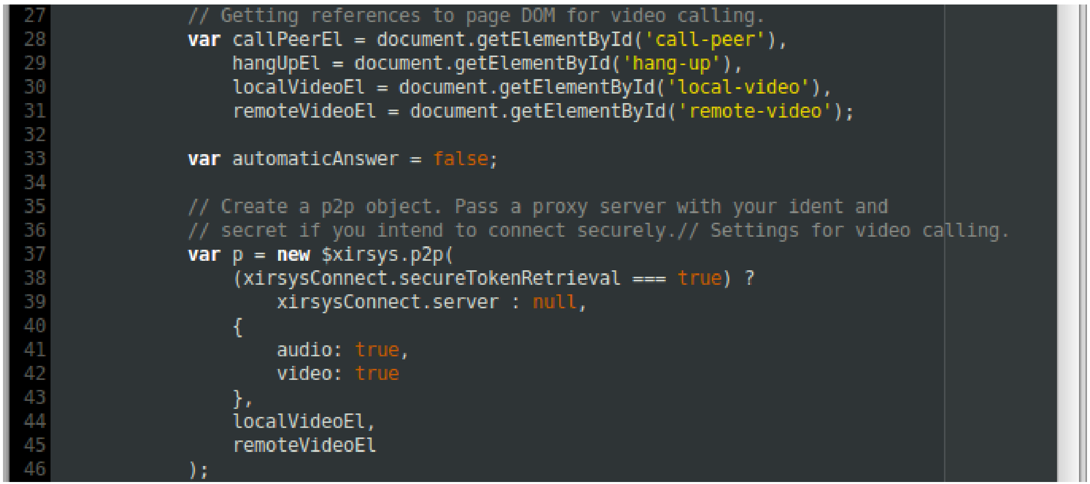
If you attempted to run the application at the last stage you will have noted that no offer was, in fact, made. The WebRTC API requires a stream or data channel to be associated with the offer, otherwise it cannot be created.

So lets add some media streams; one video element for local streams, and one for remote streams.

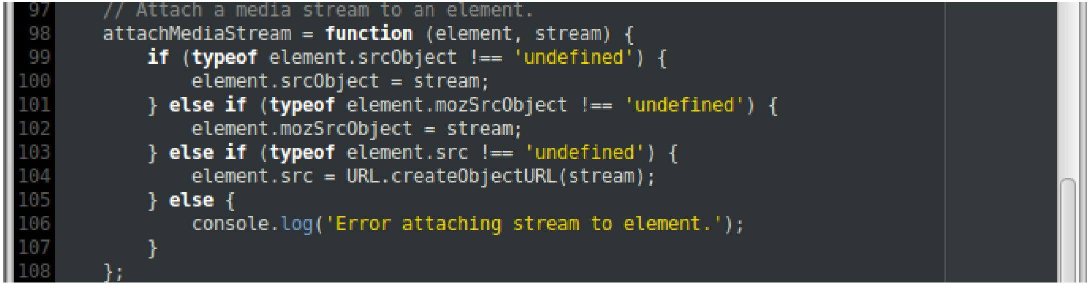


```
        <section class="major-box">
            <h2>Remote video</h2>
            <video autoplay="autoplay" id="remote-video"></video>
        </section>

        <section class="minor-box">
            <h2>Local video</h2>
            <video autoplay="autoplay" id="local-video" muted="true"></video>
        </section>
```

Send references to them when instantiating *p2p* and the *attachMediaStream* function in */lib/xirsys.adapter.js* will add a *src* properties to both videos.



```
// Getting references to page DOM for video calling.
var callPeerEl = document.getElementById('call-peer'),
    hangUpEl = document.getElementById('hang-up'),
    localVideoEl = document.getElementById('local-video'),
    remoteVideoEl = document.getElementById('remote-video');

var automaticAnswer = false;

// Create a p2p object. Pass a proxy server with your ident and
// secret if you intend to connect securely.// Settings for video calling.
var p = new $xirsys.p2p(
    (xirsysConnect.secureTokenRetrieval === true) ?
        xirsysConnect.server : null,
    {
        audio: true,
        video: true
    },
    localVideoEl,
    remoteVideoEl
);
```



```
// Attach a media stream to an element.
attachMediaStream = function (element, stream) {
    if (typeof element.srcObject !== 'undefined') {
        element.srcObject = stream;
    } else if (typeof element.mozSrcObject !== 'undefined') {
        element.mozSrcObject = stream;
    } else if (typeof element.src !== 'undefined') {
        element.src = URL.createObjectURL(stream);
    } else {
        console.log('Error attaching stream to element.');
    }
};
```

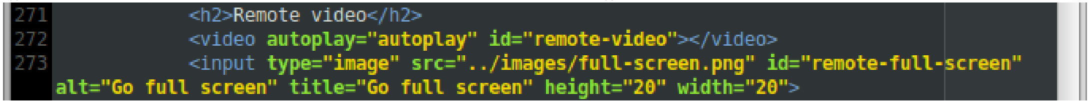
There you have it, a fully functional, client-side WebRTC video chat application. Just a couple more things...

Chapter 3

Tying up loose ends

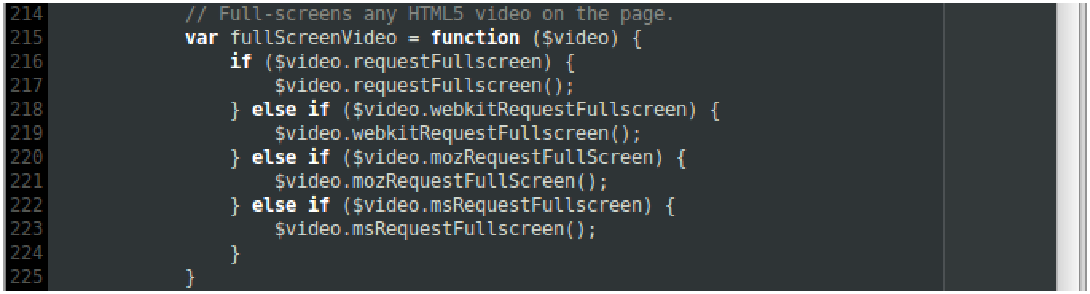
3.1 Full screen videos

A user can request a HTML5 video element goes full screen with the context menu, but that's not particularly discoverable. We'll include a couple of buttons that do the same.

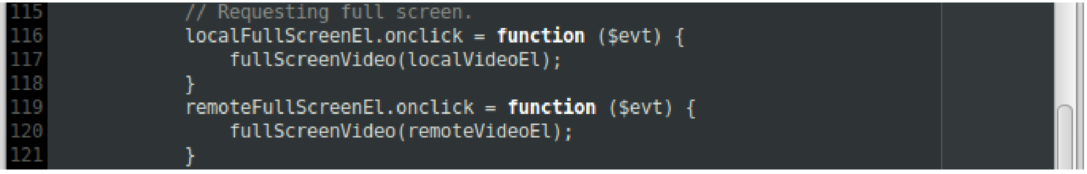


```
<h2>Remote video</h2>
<video autoplay="autoplay" id="remote-video"></video>
<input type="image" src="../images/full-screen.png" id="remote-full-screen" alt="Go full screen" title="Go full screen" height="20" width="20">
```

At the moment requesting full screen by Javascript currently requires a shim, so lets add that, and call it when the buttons are clicked.



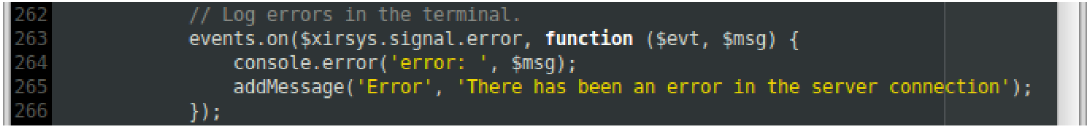
```
// Full-screens any HTML5 video on the page.
var fullScreenVideo = function ($video) {
    if ($video.requestFullscreen) {
        $video.requestFullscreen();
    } else if ($video.webkitRequestFullscreen) {
        $video.webkitRequestFullscreen();
    } else if ($video.mozRequestFullScreen) {
        $video.mozRequestFullScreen();
    } else if ($video.msRequestFullscreen) {
        $video.msRequestFullscreen();
    }
}
```



```
// Requesting full screen.
localFullScreenEl.onclick = function ($evt) {
    fullScreenVideo(localVideoEl);
}
remoteFullScreenEl.onclick = function ($evt) {
    fullScreenVideo(remoteVideoEl);
}
```

3.2 Server errors

We should let the user know when there has been a connection problem by listening for the *signalling.error* event.

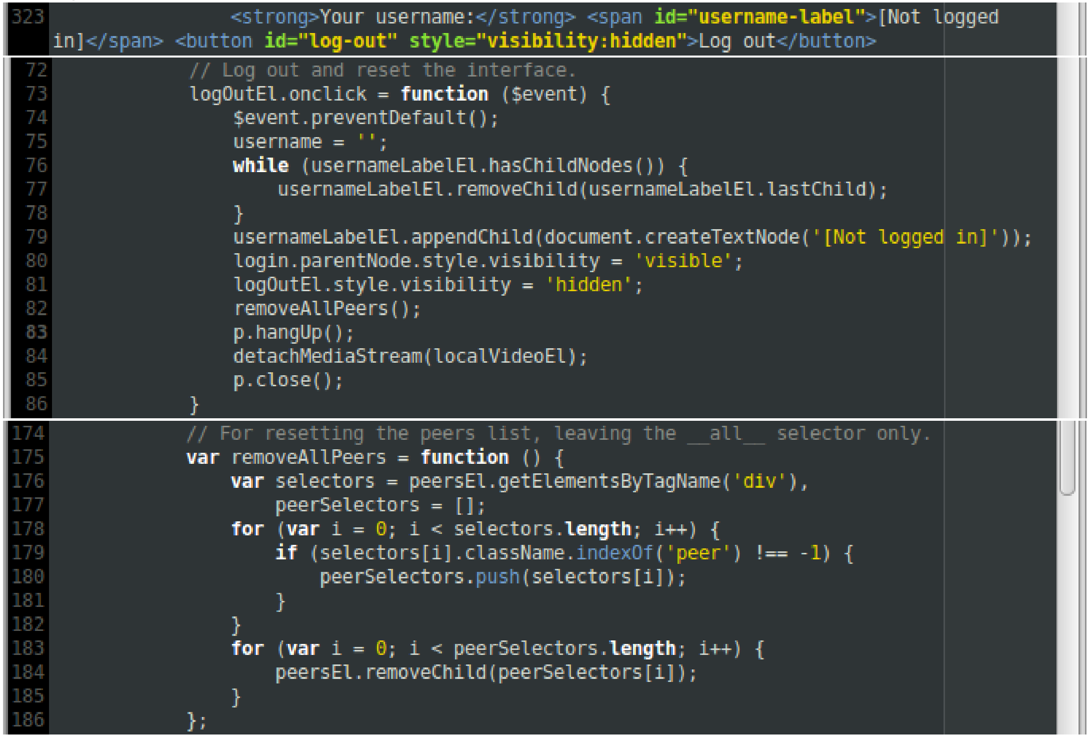


```
// Log errors in the terminal.
events.on($xirsys.signal.error, function ($evt, $msg) {
    console.error('error: ', $msg);
    addMessage('Error', 'There has been an error in the server connection');
});
```

3.3 Logging out

There are two ways of doing this. In the context of this particular application it might be simpler to reload the page with Javascript. But as there are circumstances where you wouldn't want an application to lose its state because of a log out we'll do the slightly more involved approach this time.

So now when a user clicks on the *log-out* button it removes the list of peers, hangs up on any existing calls, detaches the user's media streams and closes the connection with the signalling server.



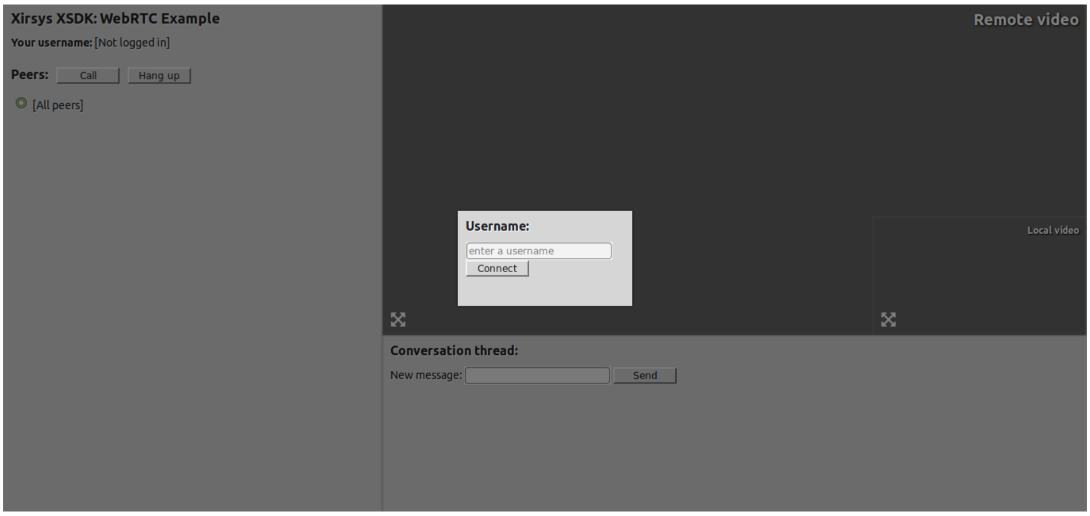
```
            <strong>Your username:</strong> <span id="username-label">[Not logged in]</span> <button id="log-out" style="visibility:hidden">Log out</button>
```

```
            // Log out and reset the interface.
            logOutEl.onclick = function ($event) {
                $event.preventDefault();
                username = '';
                while (usernameLabelEl.hasChildNodes()) {
                    usernameLabelEl.removeChild(usernameLabelEl.lastChild);
                }
                usernameLabelEl.appendChild(document.createTextNode('[Not logged in]'));
                login.parentNode.style.visibility = 'visible';
                logOutEl.style.visibility = 'hidden';
                removeAllPeers();
                p.hangUp();
                detachMediaStream(localVideoEl);
                p.close();
            }
```

```
            // For resetting the peers list, leaving the __all__ selector only.
            var removeAllPeers = function () {
                var selectors = peersEl.getElementsByTagName('div'),
                    peerSelectors = [];
                for (var i = 0; i < selectors.length; i++) {
                    if (selectors[i].className.indexOf('peer') !== -1) {
                        peerSelectors.push(selectors[i]);
                    }
                }
                for (var i = 0; i < peerSelectors.length; i++) {
                    peersEl.removeChild(peerSelectors[i]);
                }
            };
```

Chapter 4

Next Steps



This walkthrough has only covered a handful of the possibilities of the XSDK. Please peruse through scripts in the *lib* directory to get a greater understanding of it, and perhaps also see the */examples/peerjs.html* demonstration of how to integrate Xirsys and the XSDK with other services.