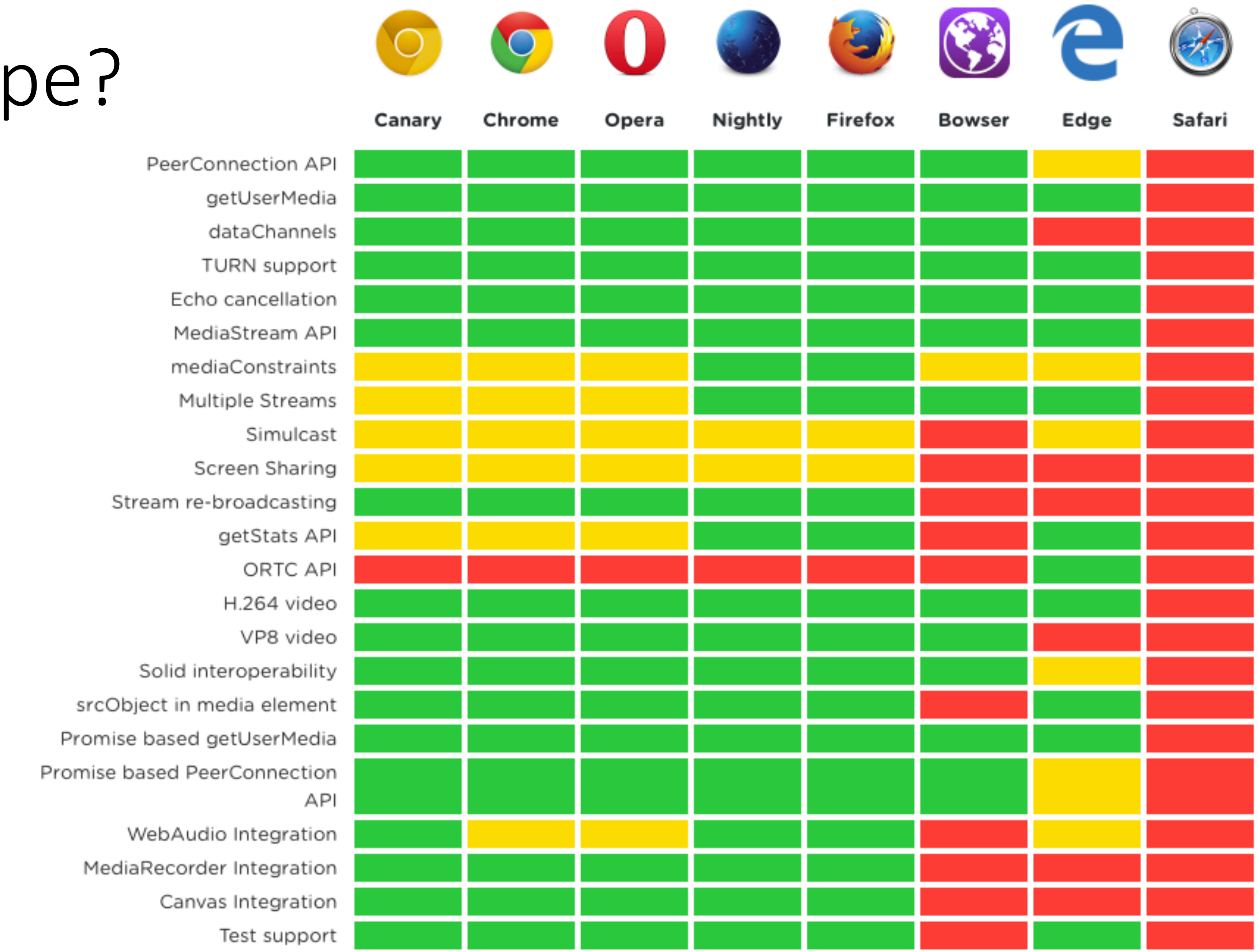
FAC 10 WebRTC Workshop

Peter Wilson, IpCortex

Making the web ‘real-time’

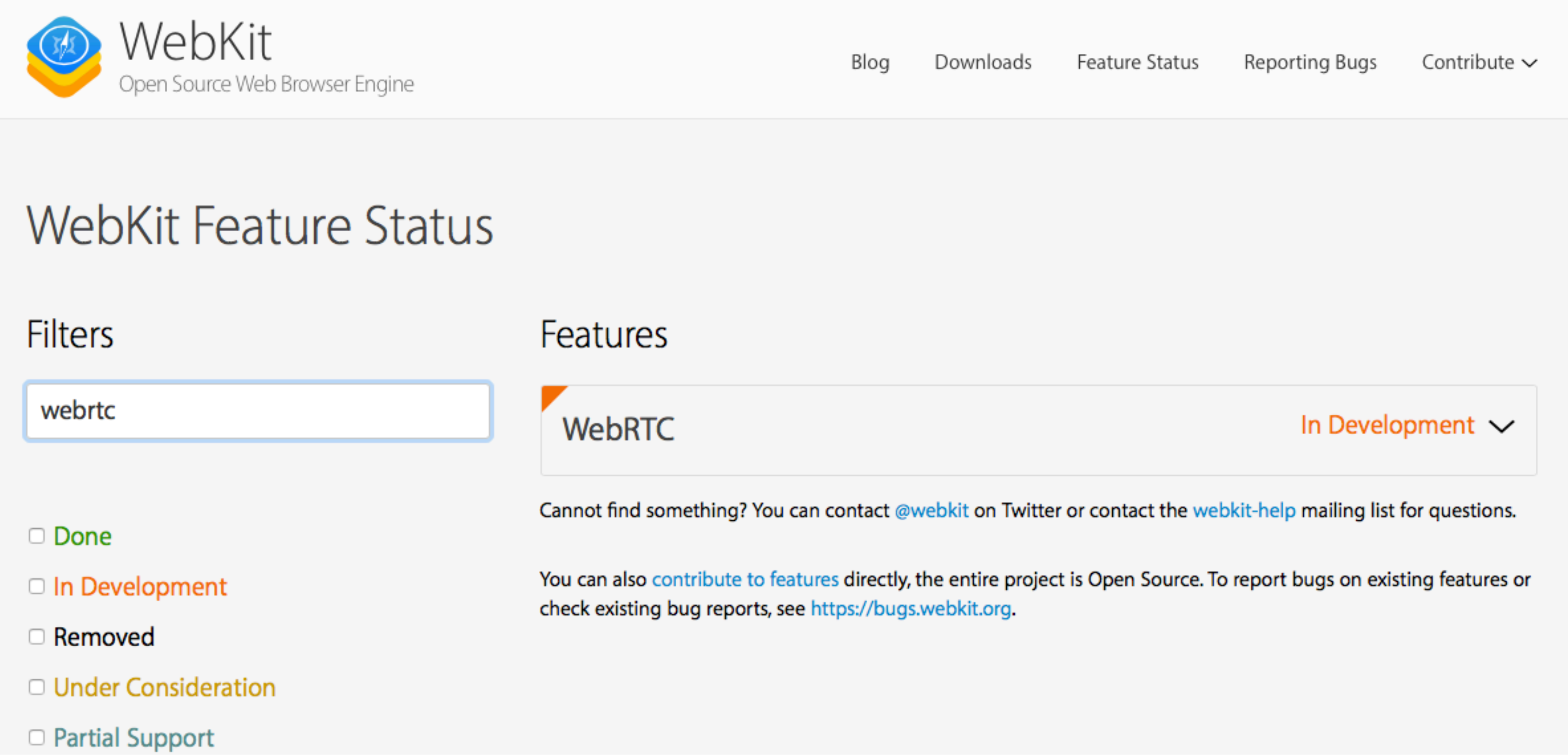
- Audio calls, Video calls, Phone calls (via gateway)
- Using ‘standard’ technologies
  - Vanilla browser
  - No proprietary plugins (Flash)
  - No proprietary protocols (Skype)
- The standardised framework is WebRTC
  - Web Real Time Communication
- Most browser are adopting this now

Is it all hype?



	Canary	Chrome	Opera	Nightly	Firefox	Bowser	Edge	Safari
PeerConnection API	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟥
getUserMedia	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
dataChannels	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
TURN support	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
Echo cancellation	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
MediaStream API	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
mediaConstraints	🟨	🟨	🟨	🟩	🟩	🟨	🟨	🟥
Multiple Streams	🟨	🟨	🟨	🟩	🟩	🟩	🟩	🟥
Simulcast	🟨	🟨	🟨	🟨	🟨	🟥	🟨	🟥
Screen Sharing	🟨	🟨	🟨	🟨	🟨	🟥	🟥	🟥
Stream re-broadcasting	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥
getStats API	🟨	🟨	🟨	🟩	🟩	🟥	🟩	🟥
ORTC API	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥
H.264 video	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
VP8 video	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Solid interoperability	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟥
srcObject in media element	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥
Promise based getUserMedia	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
Promise based PeerConnection API	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟥
WebAudio Integration	🟩	🟨	🟨	🟩	🟩	🟥	🟨	🟥
MediaRecorder Integration	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥
Canvas Integration	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥
Test support	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥

Apple Safari



WebKit
Open Source Web Browser Engine

WebKit Feature Status

Filters

webrtc

□ Done

□ In Development

□ Removed

□ Under Consideration

□ Partial Support

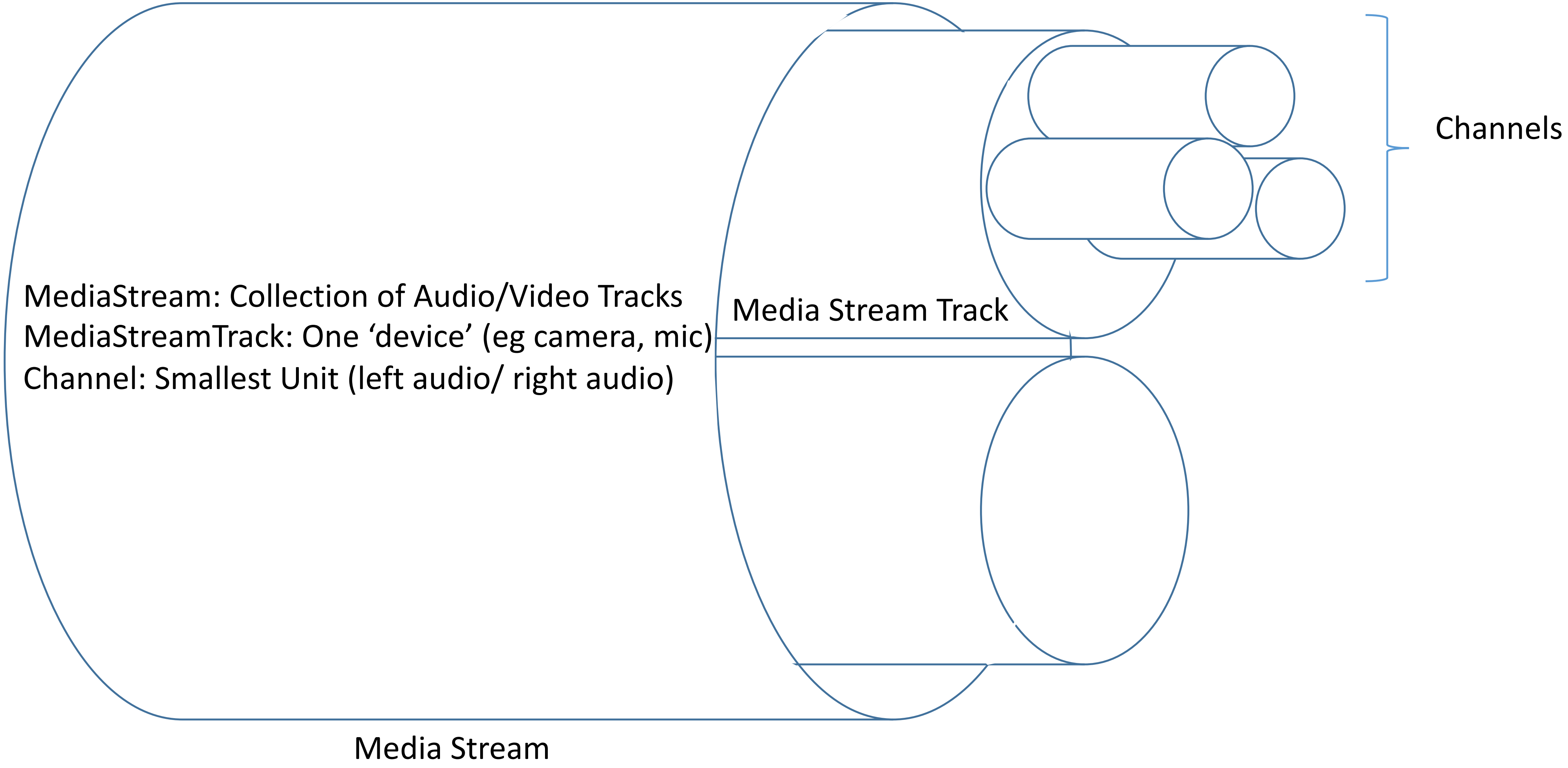
Features

WebRTC — In Development

Cannot find something? You can contact @webkit on Twitter or contact the webkit-help mailing list for questions.

You can also contribute to features directly, the entire project is Open Source. To report bugs on existing features or check existing bug reports, see https://bugs.webkit.org.

Media Streams



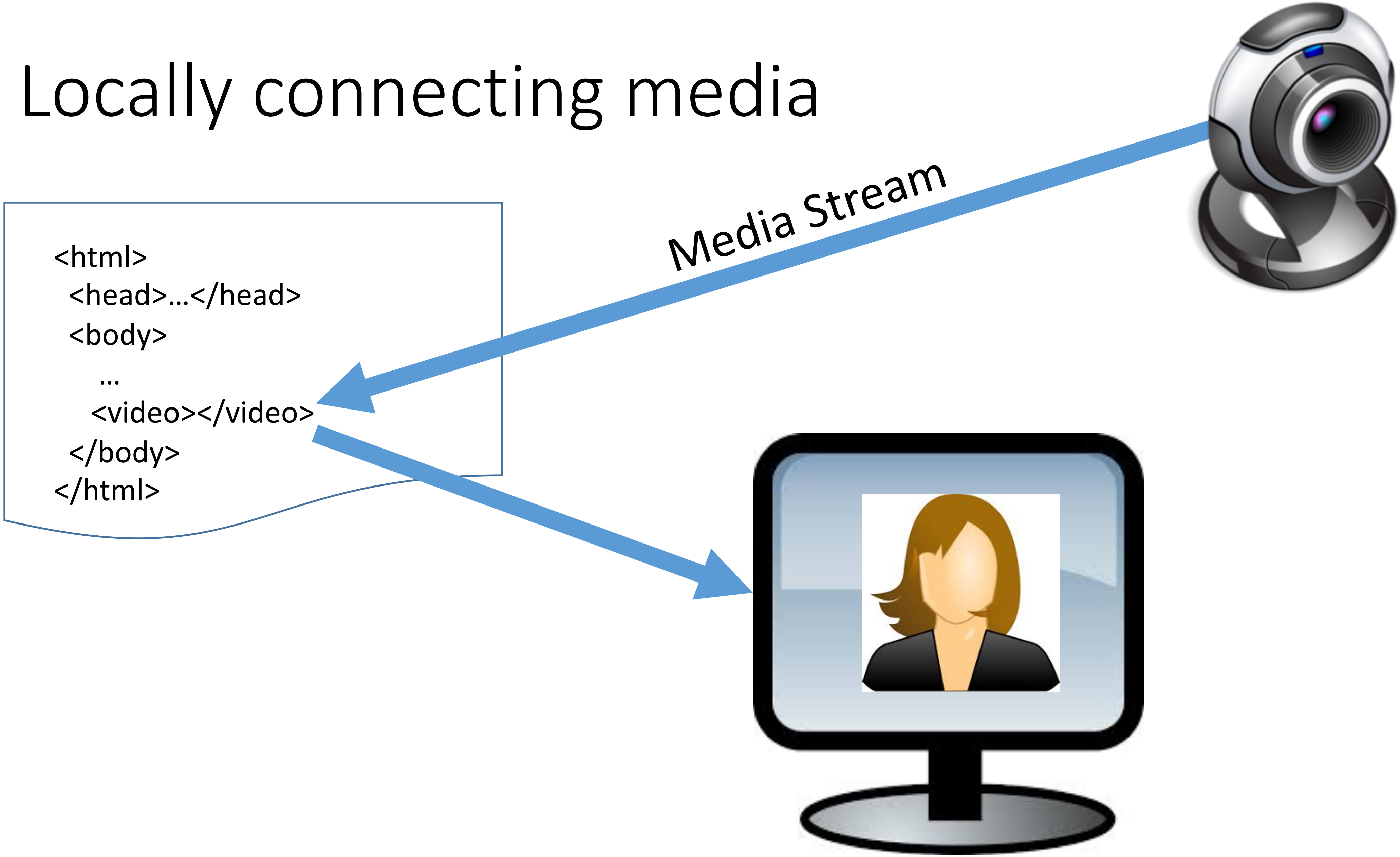
MediaStream: Collection of Audio/Video Tracks
MediaStreamTrack: One ‘device’ (eg camera, mic)
Channel: Smallest Unit (left audio/ right audio)

https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API

Media Stream

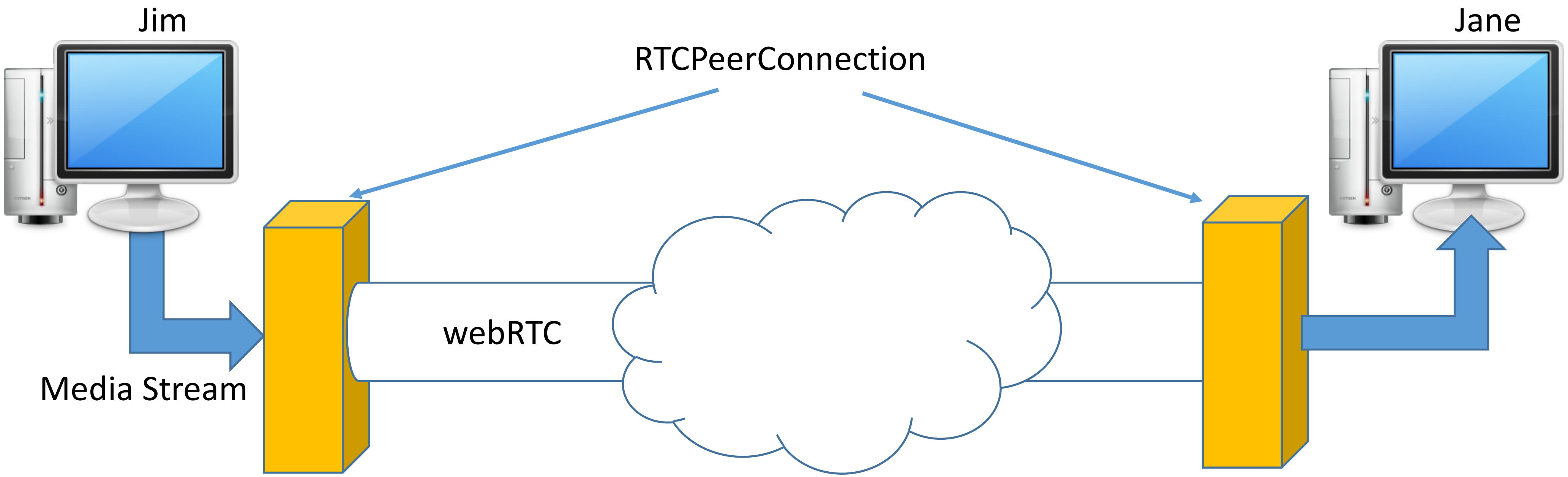
- Unidirectional
- Has one input and one output
- Local inputs:
    - Microphone
    - Camera
    - *RTCPeerConnection*
- Outputs:
    - <video> tag
    - *RTCPeerConnection*

Locally connecting media



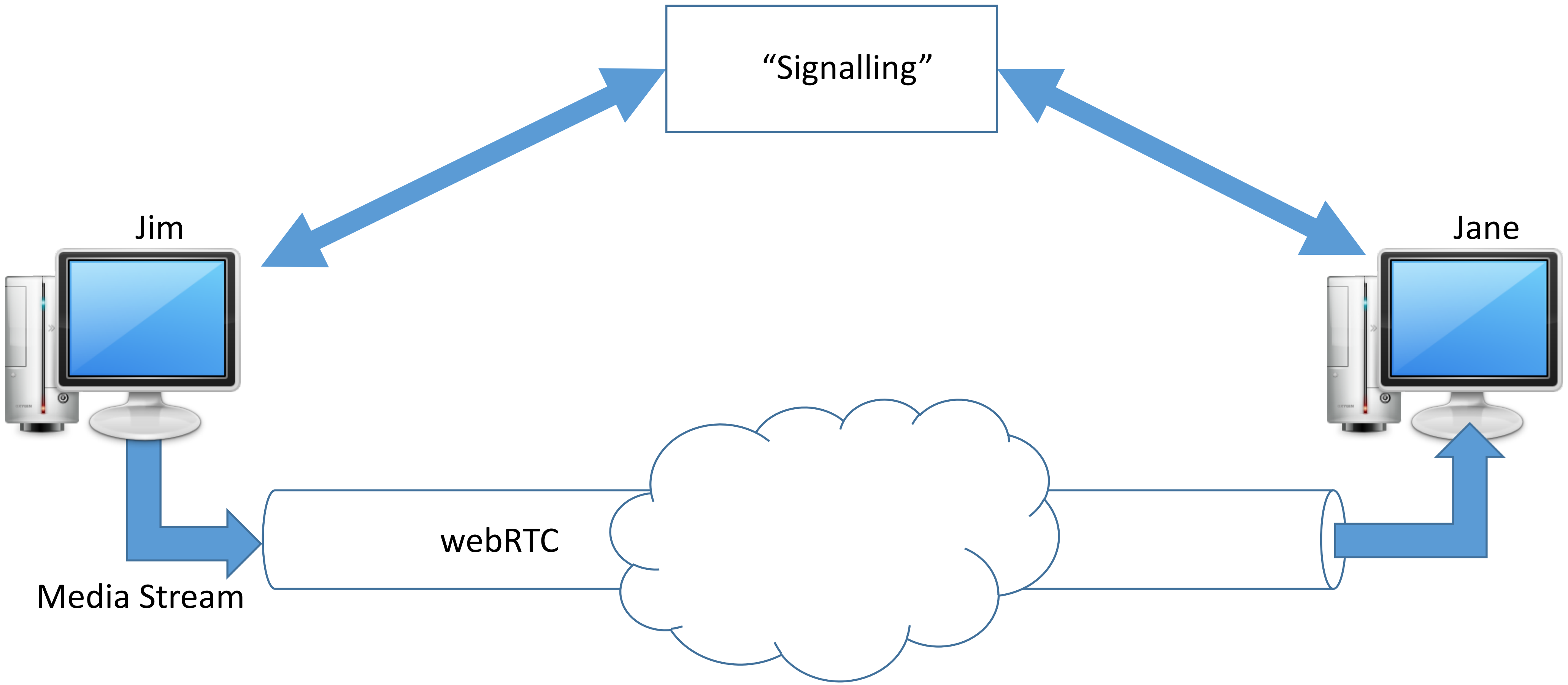
```
<html>
  <head>…</head>
  <body>
     …
     <video></video>
  </body>
</html>
```

Connecting between devices



- How do Jim and Jane find each other?
- How is their traffic routed across the Internet through corporate firewalls?
- How is permission asked to accept the call at the receiving end?

Signalling



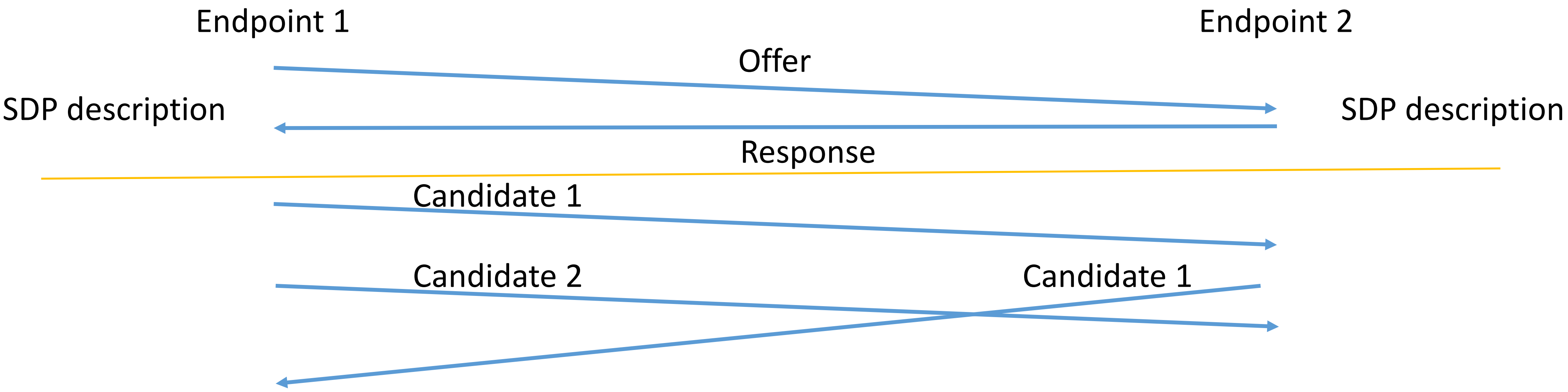
Signalling

- Allows two ends of an RTCPeerConnections to find each other
- NOT standardised
    - Not necessary and much of this is application specific
- Examples
    - IpCortex PABX – signalling, directory, presence and PSTN gateway (phone calls)
    - webRTC.io – one of the first libraries
- Must be accessible to both parties
- Provide a means of relaying information between the two parties
- Should encrypt all communications
- *Usually* a separate server

What is signalling used for?

- Very simple WebRTC requirements
    - Transfer Javascript Objects between end points (eg serialised as JSON)
    - Send media ‘offers’ and ‘responses’ between end points
    - Swap communication ‘candidates’ between end points
- Other signalling usually required for an application
    - End point discovery/name mapping (I’m Fred, I want to talk to Jane)
    - Presence (I’m Fred and I’m ‘online’)

Typical signalling exchange



- Offers are sent as Session Description Protocol messages
- Candidates give options for how to connect end points across a network
  - Accomplished via the Interactive Connectivity Establishment (ICE) framework
  - Multiple ‘candidates’ are tested concurrently with the first (fastest) used

Signalling Overhead

- Intentionally very low overhead
- Typically 24 exchanges per WebRTC session
- ~10K data exchanged
- Many techniques available
    - REST Polling
    - HTTP ‘Long Poll’
    - REST to the signalling server/ EventSource distribution to clients
    - WebSocket bi-directional ‘pipes’
- Only requires text transfer
- Not just for setup though – media can change during a call

Workshop Objectives

- Introduction to some key SW concepts
  - Classes and Objects
  - Finite State Machines
  - Promises
    - (not a key SW concept, but an unfortunate workaround for Node limitations)
- Working with protocols (signalling)
- Interoperability through protocols
- Familiarisation with AV, webRTC concepts and browser API

Workshop – what we're going to do…

1. Configure simple HTTPS server to serve scripts
2. Using streams locally
   - Create a <video> tag in a static page
   - Request media (camera and microphone)
   - Attach media to video tag
3. Local Peer Connection and Signalling
   - Connect multiple video tags together using webRTC
   - Local signalling layer
4. Remote peer-to-peer communication
   - Replace local signalling with a polled, remote signalling model
   - Node server to act as signalling relay
5. Network connections **between each team's** implementations

Workshop materials

- All materials available in GitHub

  https://github.com/ipcortex/fac-workshop-materials

- Background information in the GitHub Wiki

  https://github.com/ipcortex/fac-workshop-materials/wiki

1. HTTPS server

- Browsers only allow access to media and webRTC to ‘secure’ sites
- Need an HTTPS server
- To run an HTTPS server requires SSL certificates
- “One I prepared earlier”
  - git@github.com:ipcortex/fac-workshop-materials.git
- ./fac-workshop-materials/https
  - Run with npm run https
  - Simple ‘Hello…’ message
  - Allow unsigned SSL certificate to see page
- Basis for the rest of the workshop – build on this

Background - Promises

- Replace ‘callbacks’ for asynchronous completion
- Instead of…

```
http.get(‘http://server/mypage.html’, (res) => {
    // process response
});
```

- Use

```
http.get(‘http://server/mypage.html’)
    .then((res) => {
      // Process response
    });
```

- Seems like a simple change but can reduce ‘callback hell’

’Callback Hell’

-
  ```
  sqlExec(‘BEGIN’, (res) => {
    sqlExec(‘SELECT x FROM myTab’, (res) {
      sqlExec(‘INSERT INTO y(c1,c2) VALUES(res.v1, res.v2)’, (res) => {
        sqlExec(‘INSERT INTO z(c1,c2,c3) VALUES(res.v1, res.v4, res.v5)’, (res) => {
          sqlExec(‘COMMIT’);
        })
      });
    });
  ```

- Instead:

-
  ```
  sqlExec(‘BEGIN’)
  .then((res) => sqlExec(‘SELECT x FROM myTab’))
  .then((res) => sqlExec(‘INSERT INTO y(c1,c2) VALUES(res.v1, res.v2)’)
  .then((res) => sqlExec(‘INSERT INTO z(c1,c2,c3) VALUES(res.v1, res.v4, res.v5)’)
  .then((res) => sqlExec(‘COMMIT’))
  .catch((error) => console.error(’Something went wrong…’);
  ```

Building a Promise

- Many standard HTML5 functions return Promise
- Callback type functions can be easily wrapped

```
function myPromiseFun(params) {
  return new Promise((resolve, reject) ={
    callOldSylyAsyncFn(params, (err, res) => {
      if (err!=null)
        // Appears to caller via ‘catch’
        reject(err);
      else
        resolve(res);
    });
}
```

Classes/Object Orientated Design/Programming

- Classes and objects are inherent features of Javascript
- An object contains both state (data) and behaviour
  - State: position, mass, colour…
  - Behviour: changePosition, adjustColour
- ‘Class’ defines the attributes and behviours of all objects of that type
  - numberOfInstances
  - createInstance, findInstanceByName(‘operational’)
- Inheritance or ‘specialisation’ allows one class to build on the foundations of another.

Classes and Objects in Javascript (ES6)

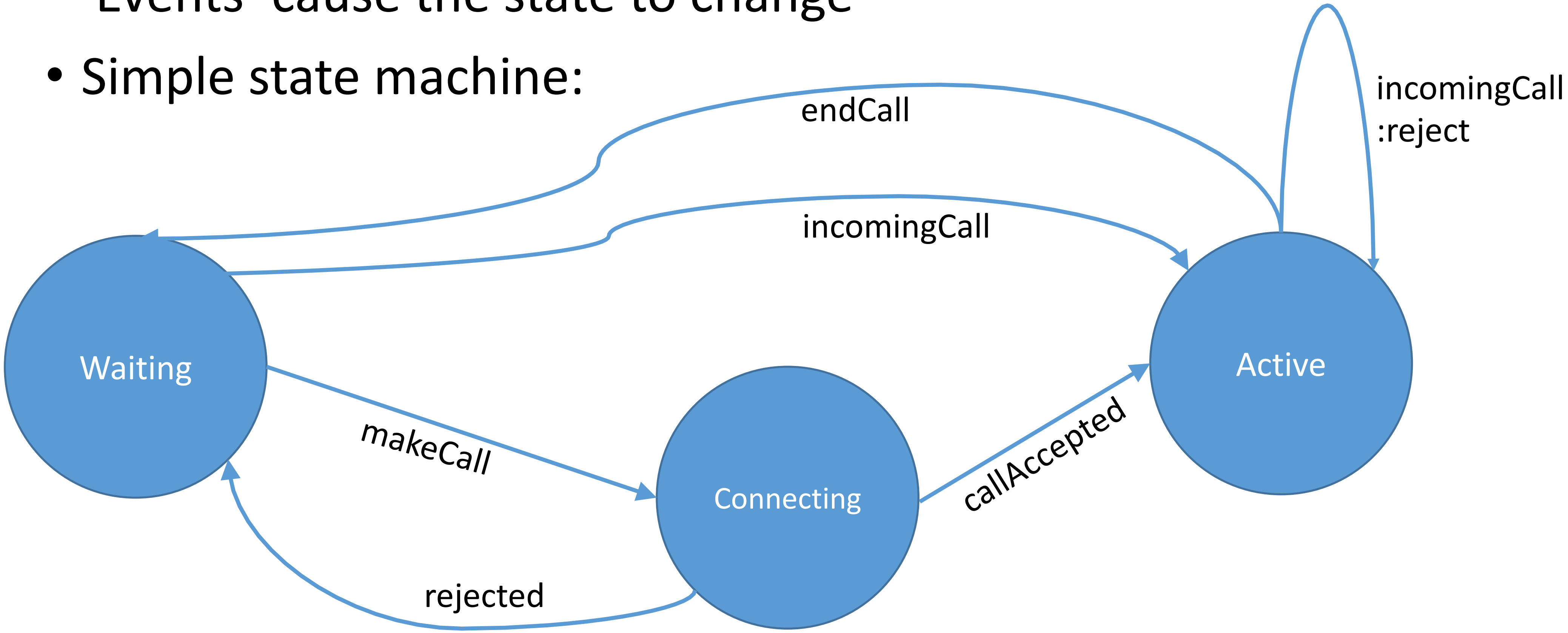
-
  ```
  class Thing extends SimpleThing {
    construct(name) {
      this.name = name;
      this.colour = ‘TRANSPARENT’;
      Thing.register[name] = this;
    }
    static findInstance(name) {
      return Thing.register[name];
    }
    setColour(newColour) {
      this.colour = newColour;
    }
    save() {…}
  };
  Thing.register ={};

  var myThing = new Thing(‘IDLE’);
  myThing.setColour(‘RED’);
  ```

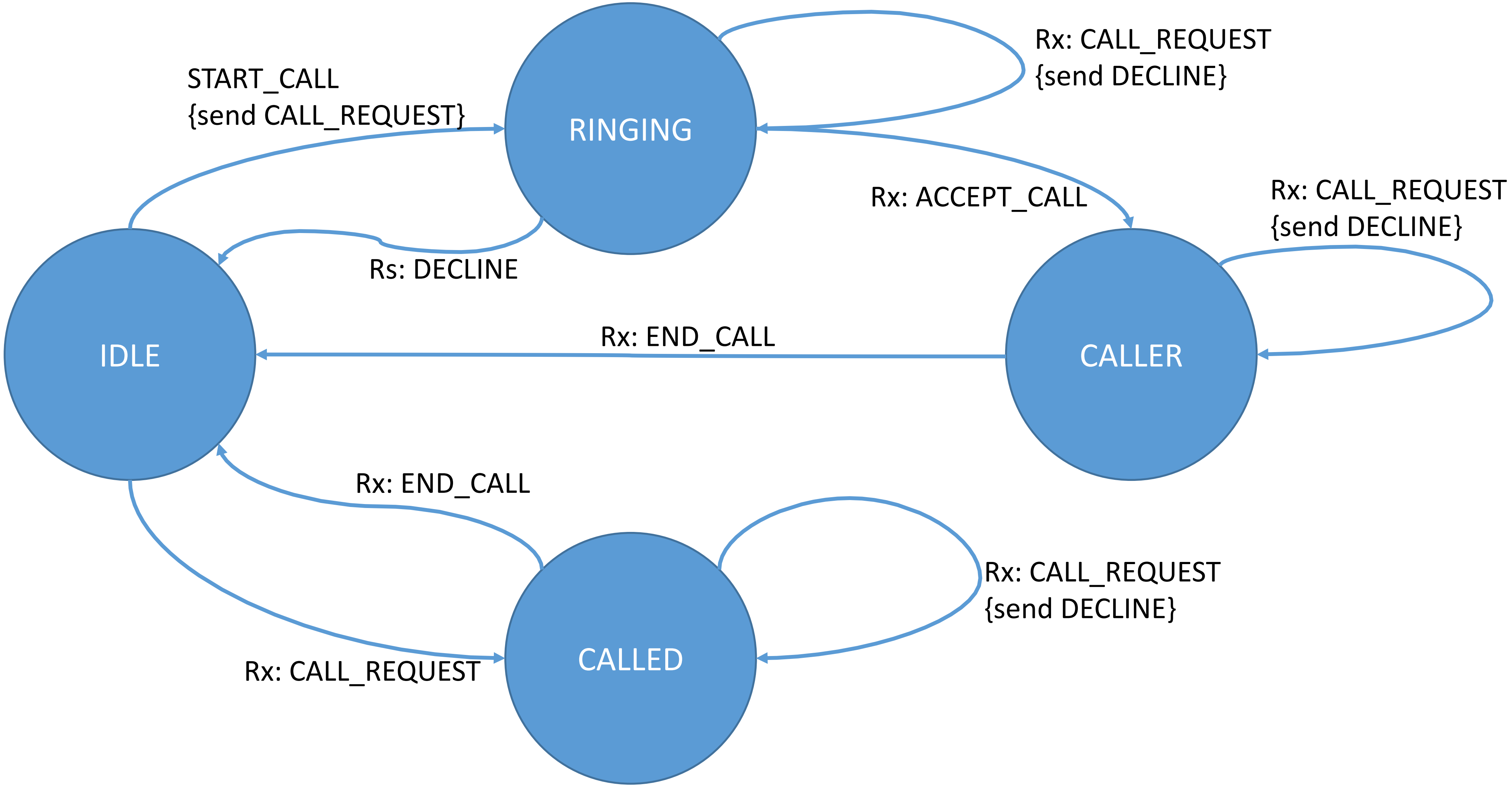
- ‘this’ references the ‘current object’
  - Thing of the **method** as a ‘message’ and the **object** as the ‘address’

State Machines

- Common SW mechanism for controlling flow
- A ‘state machine’ exists in a single state
- ‘Events’ cause the state to change
- Simple state machine:

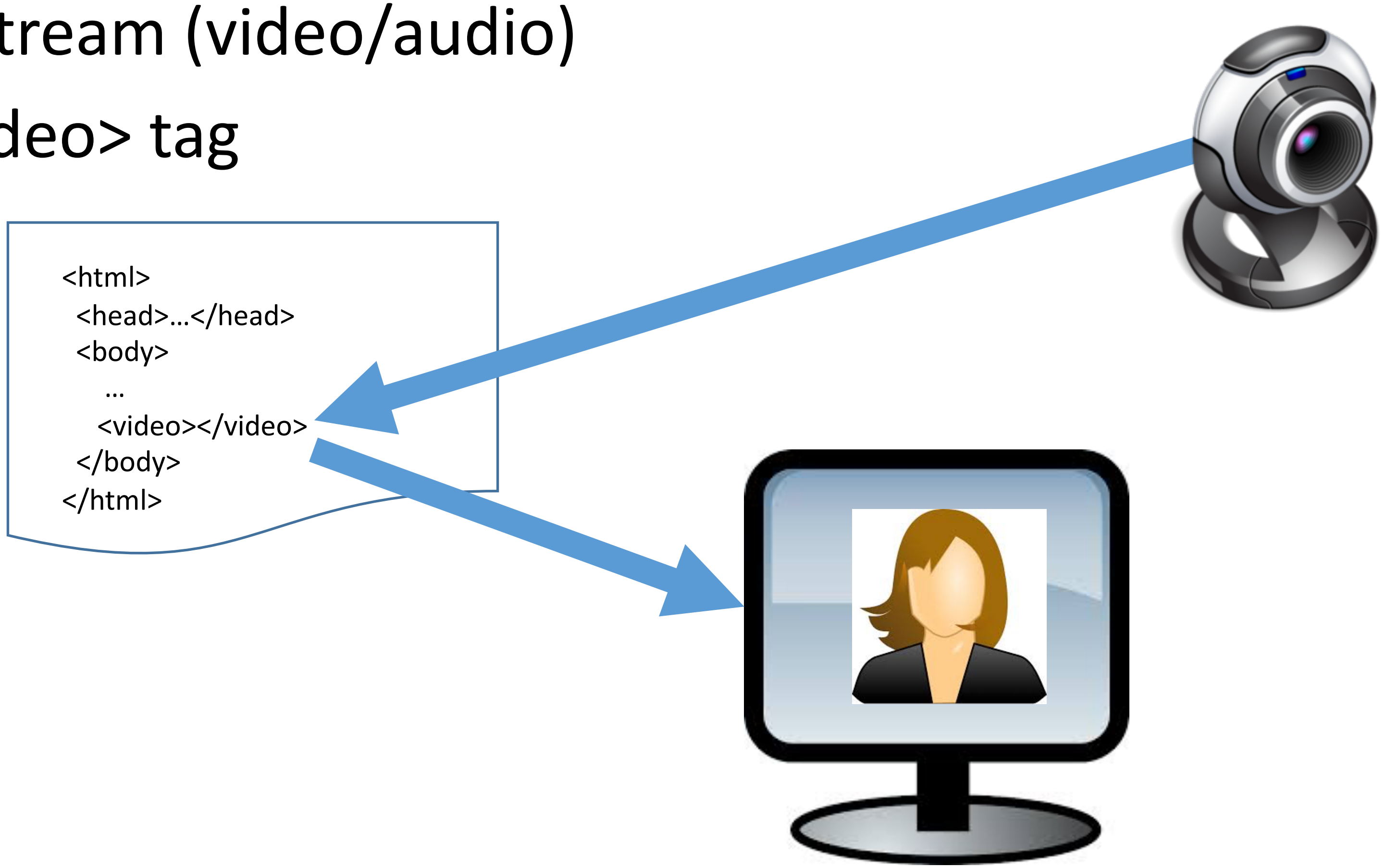


Workshop End Point State Machine



2. Local stream

- Request local media stream (video/audio)
- Attach to browser <video> tag



```
<html>
  <head>…</head>
  <body>
     …
    <video></video>
  </body>
</html>
```

2. Local media streams

```
var promise = navigator.mediaDevices.getUserMedia({
    video: true,
    audio: true
});

promise.then((avSteam) => {
  // Find my video tag…
  video = document.createElement('video');
  video.srcObject = avStream;
  video.play();

  // Add video tag to DOM
  videoContainer.append(v);
}).catch(() => {…});
```

https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

3. Local Peer Connection

```
<html>
 <head>…</head>
 <body>
   …
  <h2>Person 1</h2>
  <video id=”person_1”></video>
  <button id=”Start Call”>Call</button>

  <h2>Person 2</h2>
  <video id=”person_2”></video>

</body>
</html>
```

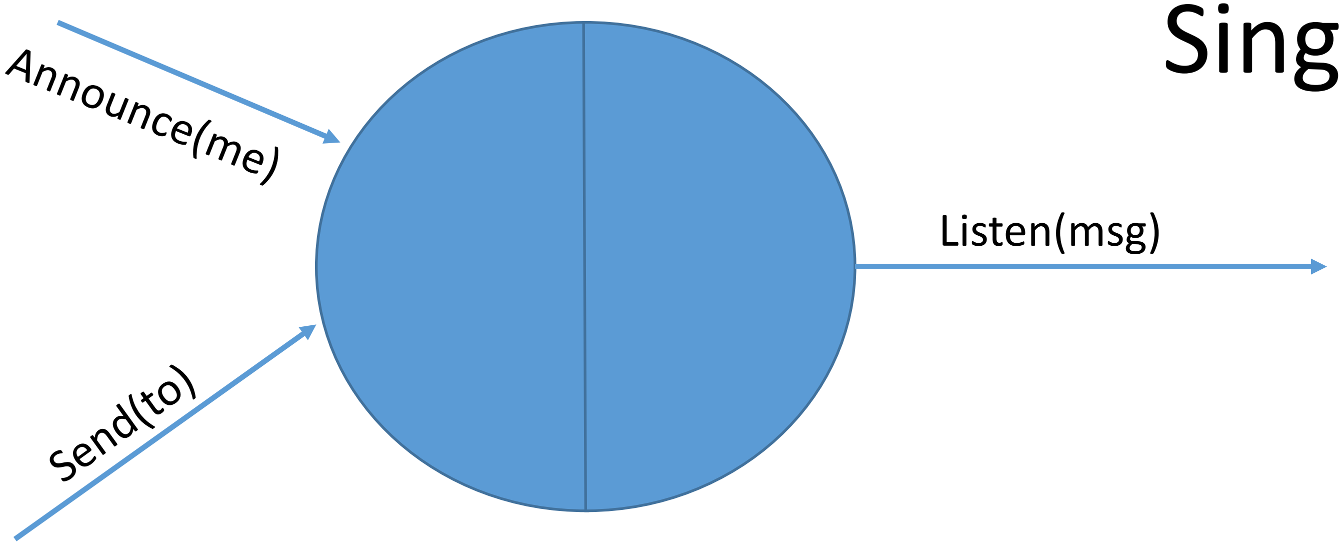
- Connecting camera/mic to a local video tag THROUGH a peer connector

- Implement our own local signalling

- Shows the basic structure of how to connect streams to each other remotely without network complexity

3. Local signalling

- Create a ‘signalling’ abstraction:
  - Announce(me), send(to), listen
  - Completely local



Single web page!

3. Code structure and aims

- Encapsulate the ‘signalling’ so it can be replaced
- Implement a standard set of messages
- Each team will interoperate with all other team’s implementation
- EndPoint base class – encapsulates the means of communication
- VideoEndPoint derived class – implements webRTC a/v sharing

- Skeleton in:
  - EndPoint class: https/assets/comms.js
  - VirtualEndPoint class: https/assets/caller.js

3. Steps to local AV calls

Building on the skeleton files in git/https:

- assets/comms.js, caller.js and driver.js

1. Add send/receive messaging to EndPoint class
   - Implement a ‘receive()’ method in VideoEndPoint class
   - Create two instances of VideoEndPoint and send a message between them
2. Create DOM->Javascript video call code
   - ‘address’ field, call button, status field, 2x<video> tags for them and me.
   - Hook button to JS onclick
   - On ‘call’ send CALL_REQUEST to target address
   - Show state changes of caller and called end points in console and in HTML
3. Add webRTC video streaming to established calls

3. Example HTML for ONE video caller

```
<div class="col-xs-12 col-md-6" id='V4’>
  <h2>Party 4:
      <span class="state">IDLE</span>
  </h2>
  <button class="pause">Pause</button>

  <!-- onclick handler for this ends a call -->
  <button class="endCall">Hangup</button>

  <!-- Somewhere to type the target address – who I want to call -->
  <input type="text" name="target" class="target" placeholder="Enter recipient call name">

  <!-- onclick handler for this trys to make a call to the name in the text field -->
  <button class="startCall">Call</button>

  <!- Video tags included here but not used until the next stage -->
  <video class="remoteVideo"></video>
  <video class="localVideo"></video>
</div>
```

3. Steps to start/end calls

1. Write HTML for 4 ‘virtual’ callers
   - 4 video tags showing person called + 4 small video tags showing local video
   - <input type=“text’> to enter the name of the person to call
   - Display the current state (in HTML)
2. For each caller, create a VideoEndPoint object
   - Pass in the video tags and the ‘status’ display tag to the constructor

```
new VideoEndPoint(
    ‘name’,remoteVideoDOMtag, localVideoDOMtag, statusDOMtag
)
```

3. Add ‘click’ event handlers for ‘call’
   - Add these into ‘driver.js’
   - When ‘call’ button clicked:
     - Work out who the caller is (who clicked call) – find the VideoEndPoint object
     - Get the value from the ‘target’ field
     - Send ‘CALL_REQUEST’ to the target

3. State Machine first steps…

- Add a setState(newState) method to VideoEndPoint
  - Use this to update the state DOM element you added to the constructor
- Add a method to VideoEndPoint to implement ‘CALL_REQUEST’
  - And call that from the switch statement in ‘receive’
- Make a call and check the states of the two end points involved

Referencing VideoEndPoints

- Each of the instance methods in VideoEndPoint has a ‘this’
  - Refers to the object that is the subject of the current request.
- In the *current* task all EndPoints are on one computer. It can be tempting to do:

```
class VideoEndPoint {
  receive(from,operation,params) {
    …
    this.doSomething();
    var fromObj = find_the_from_object(from);
    fromObj.doSomethingElse(…)
  }
}
```

Responsive HTML layout

-
  ```
  <div class=”container”>
    <div class=“row”>
      <div class=”col-xs-12 col-sm-6”>
        …
      </div>
      <div class=”col-xs-12 col-sm-6”>
        …
      </div>
      <div class=”col-xs-12 col-sm-6”>
        …
      </div>
      <div class=”col-xs-12 col-sm-6”>
        …
      </div>
   </div>
  <div>
  ```

A trick with Promises

```
promiseFunction().then((result) => doSomething);
```

A trick with Promises

```
promiseFunction().then((result) => doSomething);
```

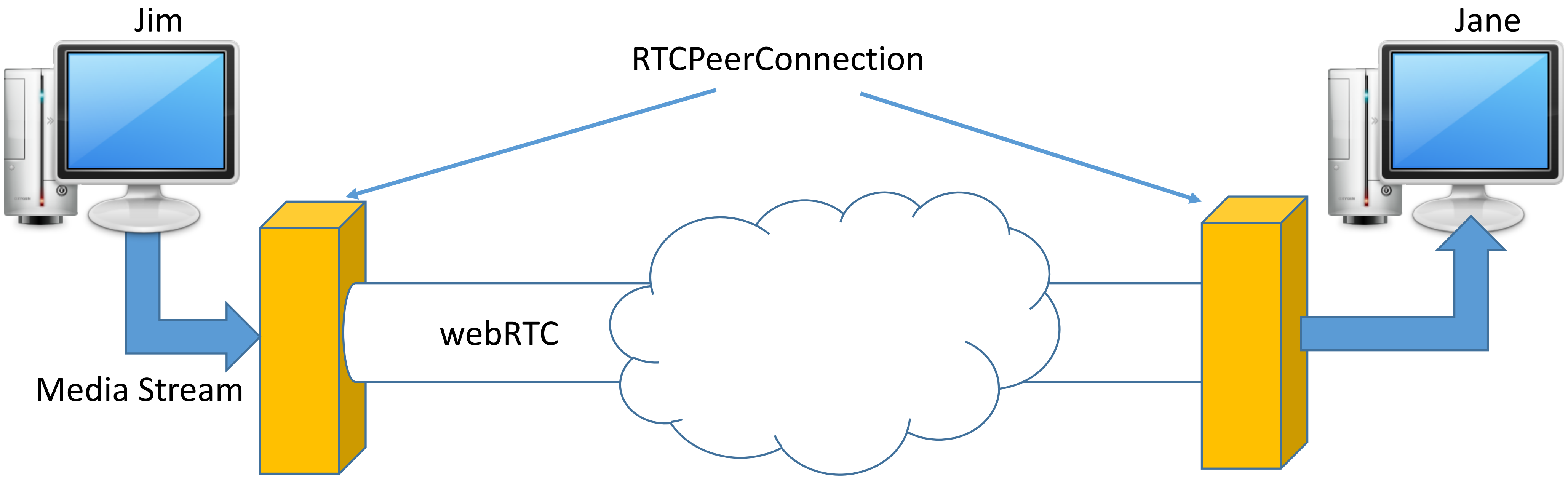
but the Promise can also be saved and be used many times:

```
this.myPromise = promiseFunction();

this.myPromise.then((result) => {doFirstThing});
…
this.myPromise.then((result) => {doAnotherThing});
```

Use this where a value that may take a time to resolve is needed in multiple places

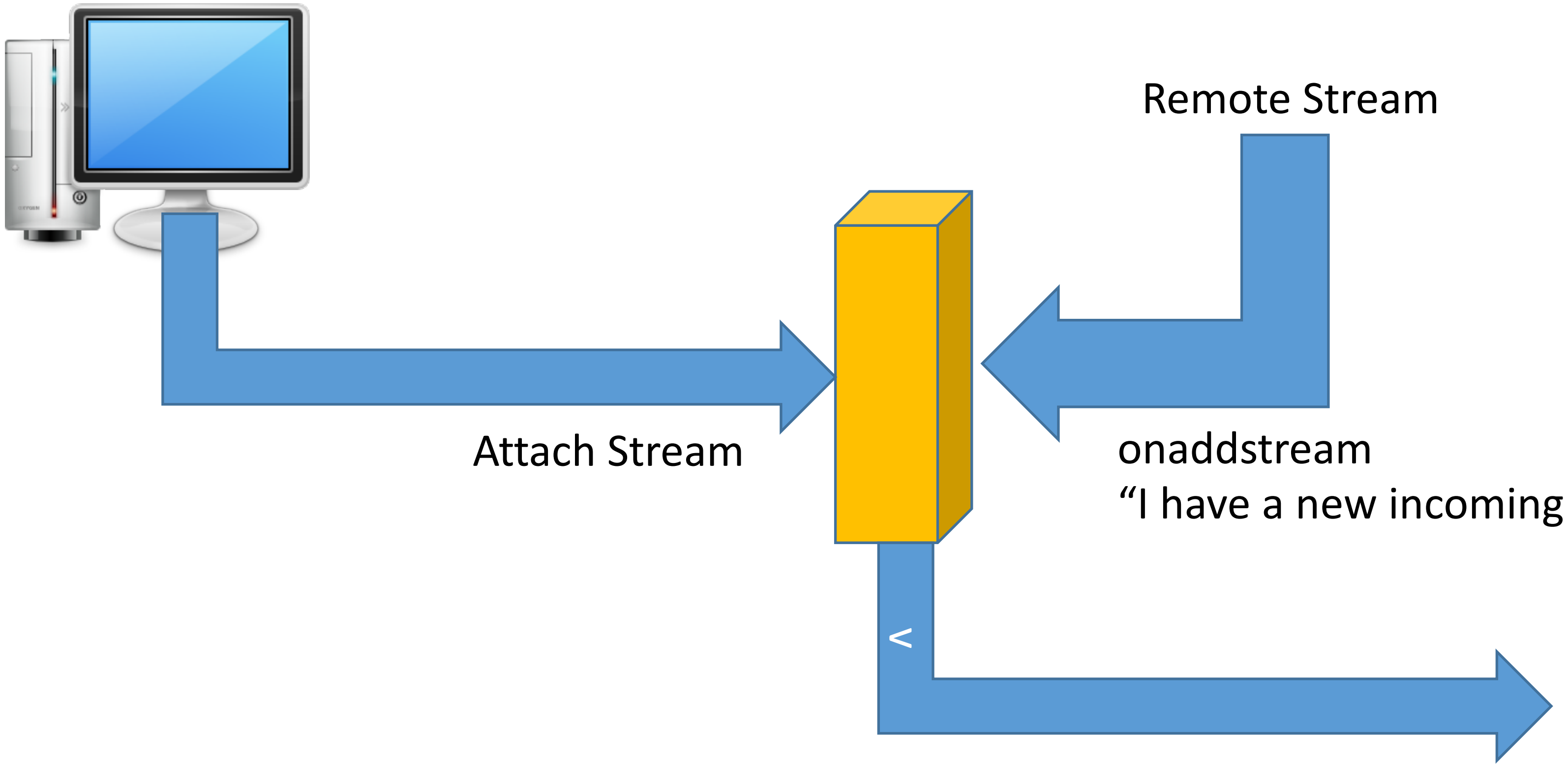
Connecting between devices



- How do Jim and Jane find each other?
- How is their traffic routed across the Internet through corporate firewalls?
- How is permission asked to accept the call at the receiving end?

https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/

RTCPeerConnection



Remote Stream

Attach Stream

onaddstream
“I have a new incoming stream for you to deal with”

onicecandidate
“I have an ICE candidate I need you to send for me”

RTCPeerConnection Events

There are ‘two’ events that can happen which need to be processed

1. The connection wants to send an ICE candidate
    - Listen for these with the onicecandidate callback function
    - When one happens send it the the other end of the call

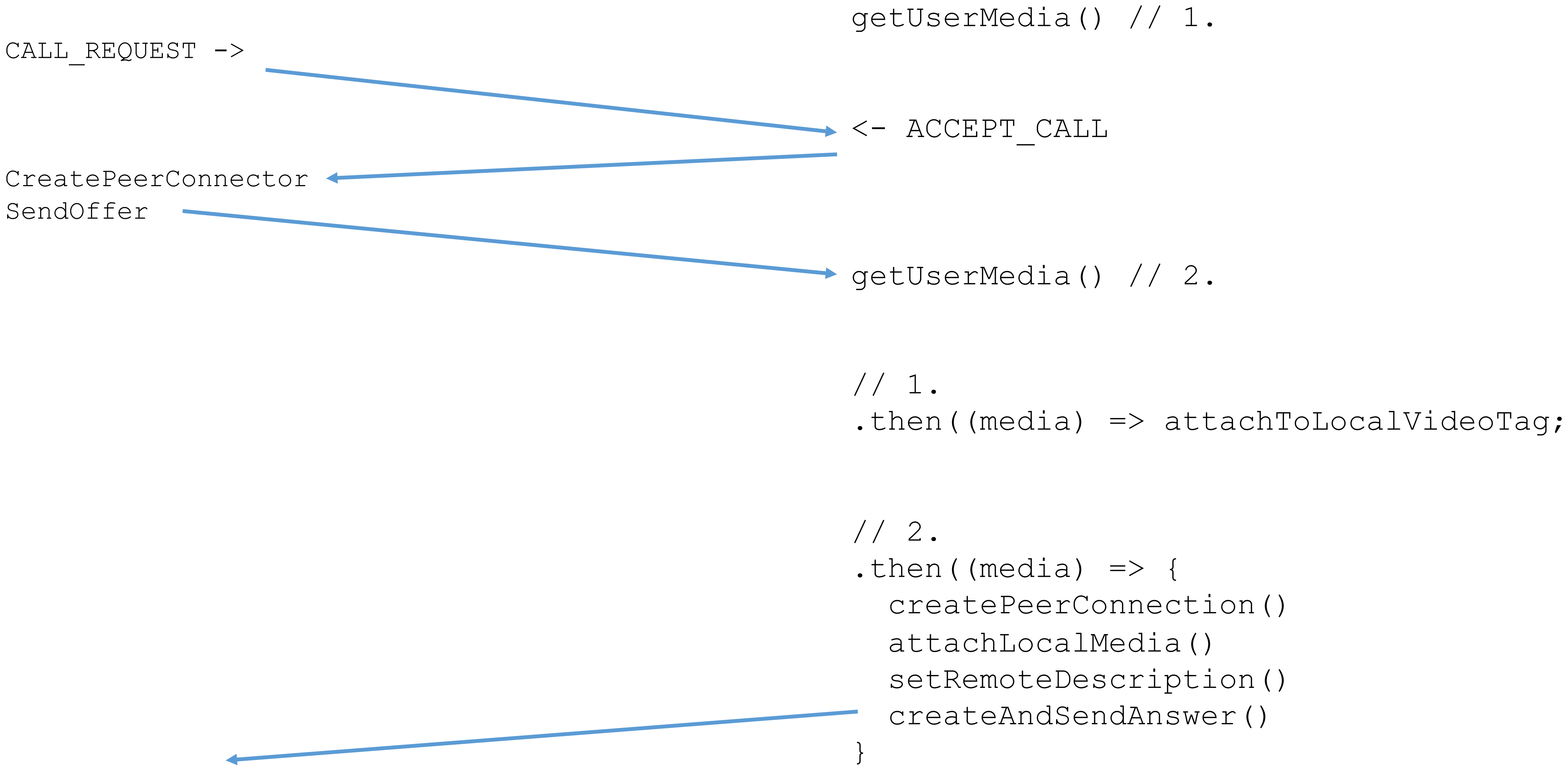
2. The connection receives an AV stream from the remote end
    - Listen for these by adding an onaddstream callback to the connection
    - When one happens, take the stream and attach it to the <video> tag

Adding video to the demo

- Don’t create a RTCPeerConnection *until* you have access to media

- Get the media in the VideoEndPoint constructor
  - Call getUserMedia()
  - When that Promise resolves attach the stream to the local <video> tag
  - **Keep hold of the Promise returned from getUserMedia**

Asynchronous Completion: Promise Trick



```
CALL_REQUEST ->

CreatePeerConnector
SendOffer
```

```
getUserMedia() // 1.

<- ACCEPT_CALL

getUserMedia() // 2.

// 1.
.then((media) => attachToLocalVideoTag;

// 2.
.then((media) => {
  createPeerConnection()
  attachLocalMedia()
  setRemoteDescription()
  createAndSendAnswer()
}
```

Using RTCPeerConnection objects

CALLER *(wait for getUserMedia AND THEN):*

1. Create a Peer Connection when call accepted
2. Get local media (as you did for a video tag)
3. Attach media to the Peer Connection
4. Create an SDP Offer to send to the remote end
5. Close the Peer Connection at the end of the call

CALLED *(wait for getUserMedia AND THEN)*:

1. Create a Peer Connection when accepting a call
2. Get local media (as you did for a video tag)
3. Attach media to the Peer Connection
4. Wait for an SDP Offer to arrive
5. Create an SDP Answer
6. Close the Peer Connection at the end of the call

Creating an RTCPeerConnection

```
// NOW get the media stream and don’t do anything else until it’s attached to the connection
waitForUserMedia(options).then((localStream) => {
  var pc = new RTCPeerConnection;

  pc.onicecandidate = (ev) => {
    // Send to REMOTE end point
  }
  pc.onaddstream = (remoteStream) => {
    // Attach this stream to the <video> tag
  }

  // Got the LOCAL stream. Add it the the peer connection…
  pc.addStream(localStream);
});
```

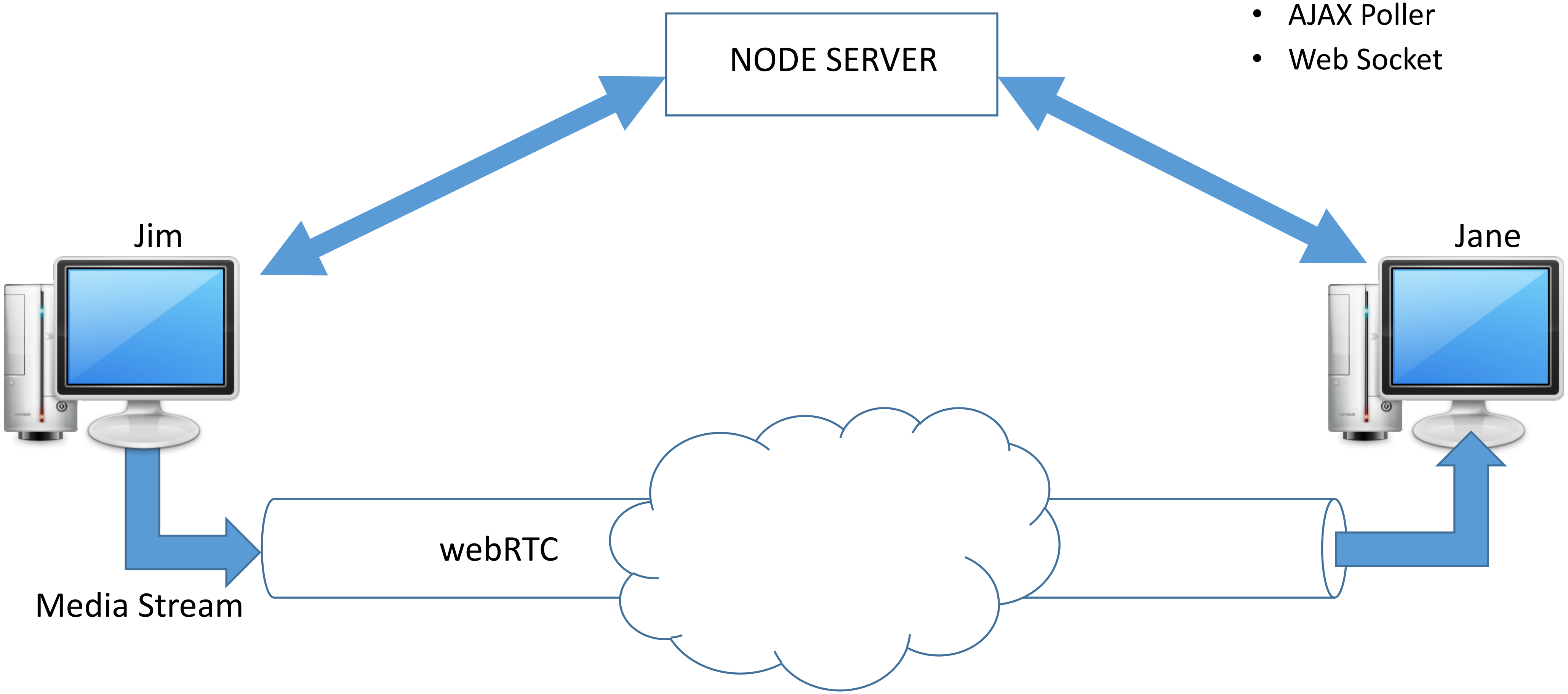
Connecting video to the call

- Similar to ‘getUserMedia’ *and then* attach to <video> tag
- Instead ’getUserMedia’ *and then* attach to Peer Connection
- To do this – we need a Peer Connection:
  - Caller: create Peer Connection when call is accepted
  - Called: create Peer Connection when accepting a call

https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/

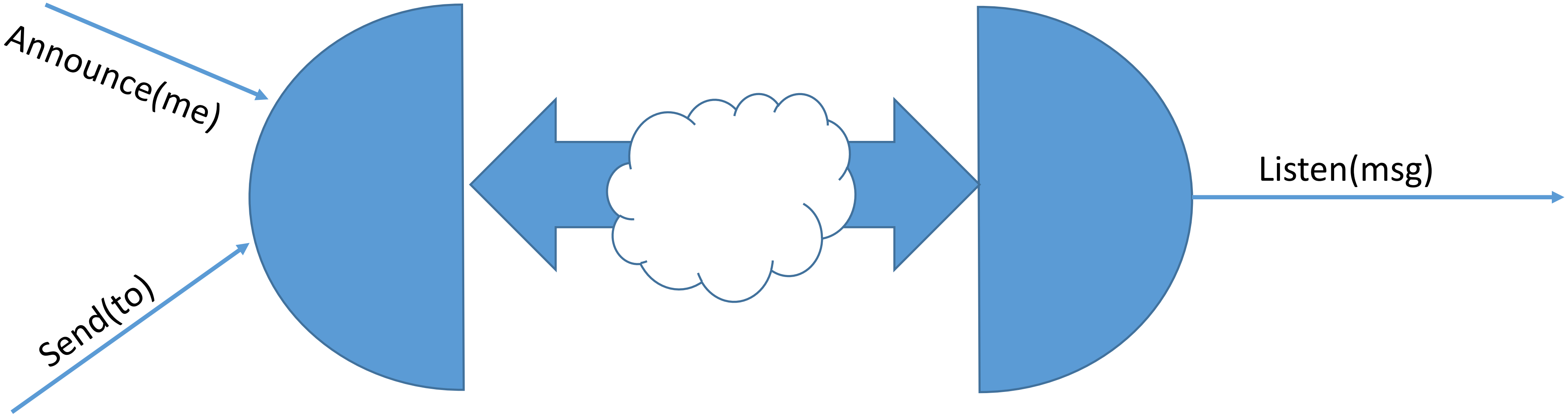
4. Remote Peer Connections

- Replace local signalling with node server proxy.
- Options for transferring signalling:
  - AJAX Poller
  - Web Socket



4. Signalling across a network

- Split your signalling into two parts:
  - Carry information across the local network
  - Modify the application to have one end point per browser



4. Signalling across a network (2)

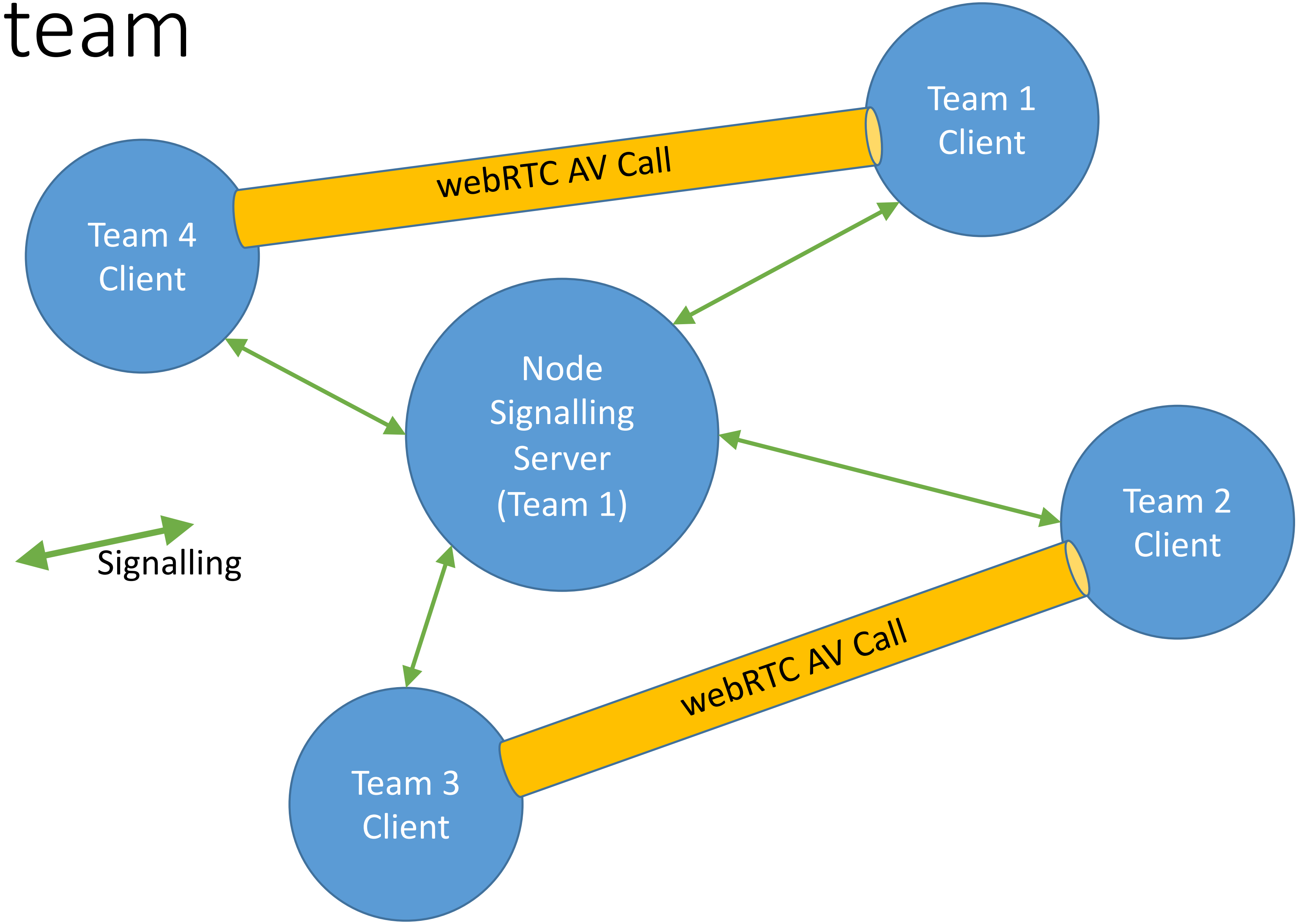
- If you’ve implemented part 3 well then
  - All you should have to do to the client/browser is rewrite EndPoint
  - There should be **no changes** required to VideoEndPoint
- Extend the node server to implement the REST interface
  - Specified in the WIKI

5. Talking between teams

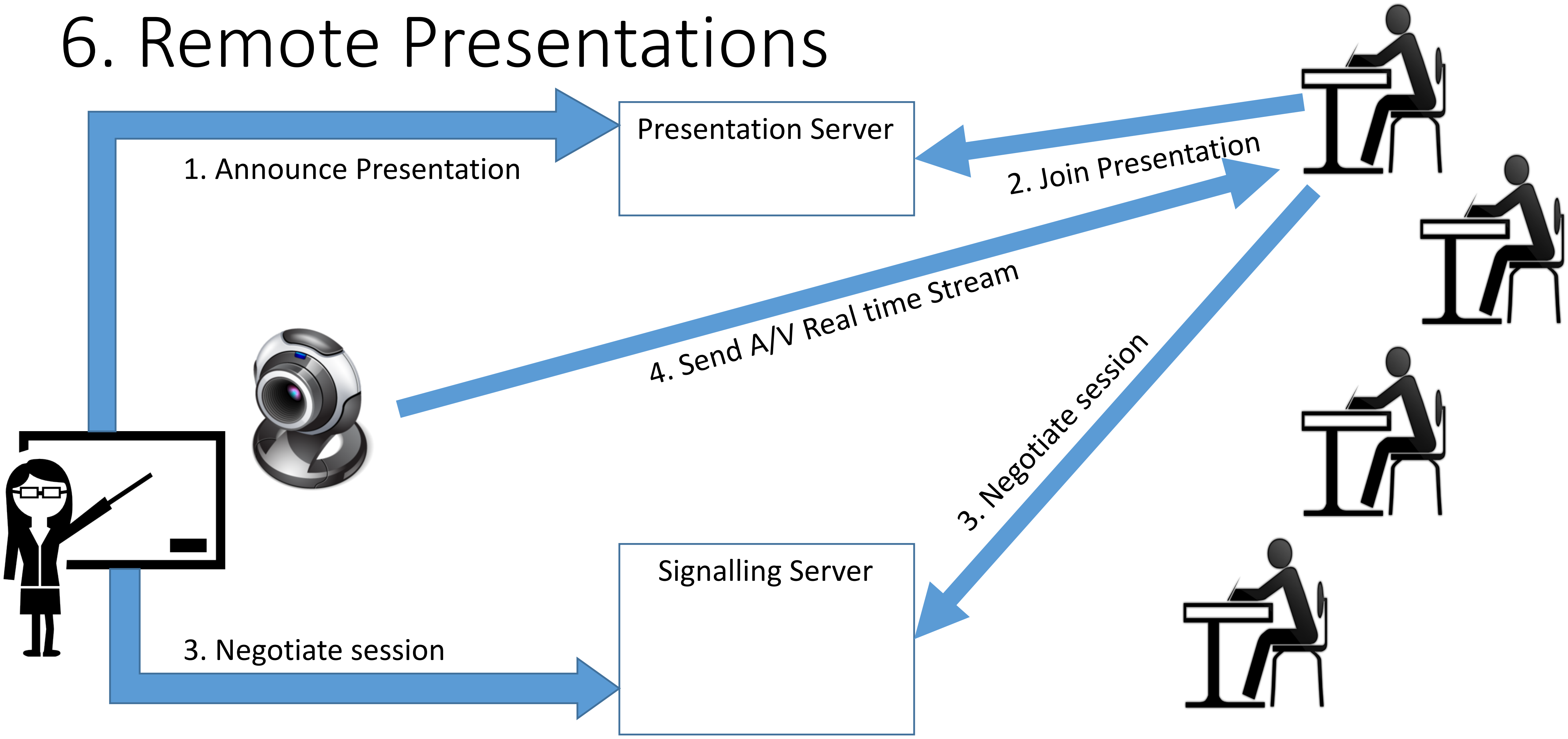
- There should be 4 teams with working clients and node servers
- The node servers are implementing a defined protocol
- So…
    - The client from any team should be able to talk to any node server
    - Two clients from two different teams should be able to connect to the same node server and make a call

- AIM:
    - Using any teams node server
    - Connect clients from all 4 teams to that server and make a call

5. Cross team


Team 1
Client
webRTC AV Call
Team 4
Client
Node
Signalling
Server
(Team 1)
Team 2
Client
Signalling
webRTC AV Call
Team 3
Client

6. Remote Presentations



Presentation Server

1. Announce Presentation

2. Join Presentation

4. Send A/V Real time Stream

3. Negotiate session

Signalling Server

3. Negotiate session

Signalling: EITHER using IpCortex api OR modified signalling from previous task

Signalling for Remote Presentations: Two Options

- Evolution of simple signalling from previous example
  - Should work on a local LAN
  - Won’t work across the Internet without TURN/STUN servers (complexity)
- IPCortex API
  - Covers all the routing across the Internet
  - More complex to configure/run

References

- mediaDevices.getUserMedia MDN:
  - https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia
- WebRTC
  - https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API
- WebRTC.org – Getting Started
  - https://webrtc.org/start/
- HTML 5 Rocks – Getting started with WebRTC (2012)
  - Illustrates local signalling *but* not is a portable way
  - https://www.html5rocks.com/en/tutorials/webrtc/basics/
- HTML 5 Rocks – WebRTC Infrastructure
  - Great overview of signalling – everything you need to know!
  - https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/
- adapter.js: https://github.com/webrtc/adapter
  - Shim to isolate applications from browser incompatibilities
- Promises:
  - https://kosamari.com/notes/the-promise-of-a-burger-party
  - https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise
- Classes/OOD/OOP: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS
- Finite State Machine: https://developer.mozilla.org/en-US/docs/Glossary/State_machine

adapter.js

https://github.com/webrtc/adapter

RTCPeerConnection

```
var pc new RTCPeerConnection();

pc.onicecandidate = (e) => {
   this.send(from, "CANDIDATE", e.candidate);
};

pc.onaddstream = (e) => {
   attachMediaStream(videoTag,e.stream);
   videoTag.play();
}

localMediaPromise.then((mediaStream) => {
   pc.addStream(mediaStream);
   console.log('PeerConnector (TX) createOffer start');

   var offerOptions = {offerToReceiveAudio: 1, offerToReceiveVideo: 1};
   pc.createOffer(offerOptions)
   .then((offer) => {
      console.log("WE HAVE AN OFFER...",offer);

      // Give the offer description to our end of the connector
      pc.setLocalDescription(offer);

      // Send the offer to the remote end of the peer connector
      this.send(from, "SDP_OFFER", offer );
   });
   // Attach this stream to a video tag...
   attachMediaStream(videoTag, mediaStream);
   // And set the 'play' state for this tag.
   videoTag.play();
});
```

RTCPeerConnection – incoming signalling

```
function receivedIncomingSDPoffer(from, data) {
   this.data.pc.setRemoteDescription(data);
   // And generate an answering offer
   this.data.pc.createAnswer().then(
      (desc) => {
         this.data.pc.setLocalDescription(desc);
         // And send this to desciption to the remote end
         this.send(from, "SDP_ANSWER", desc);
      });
   }
}

function receivedIncomingSDPanswer(from, data) {
   this.data.pc.setRemoteDescription(data);
}

function receivedCandidate(from, data) {
   var candidate = new RTCIceCandidate(data);
   this.data.pc.addIceCandidate(candidate);
}
```