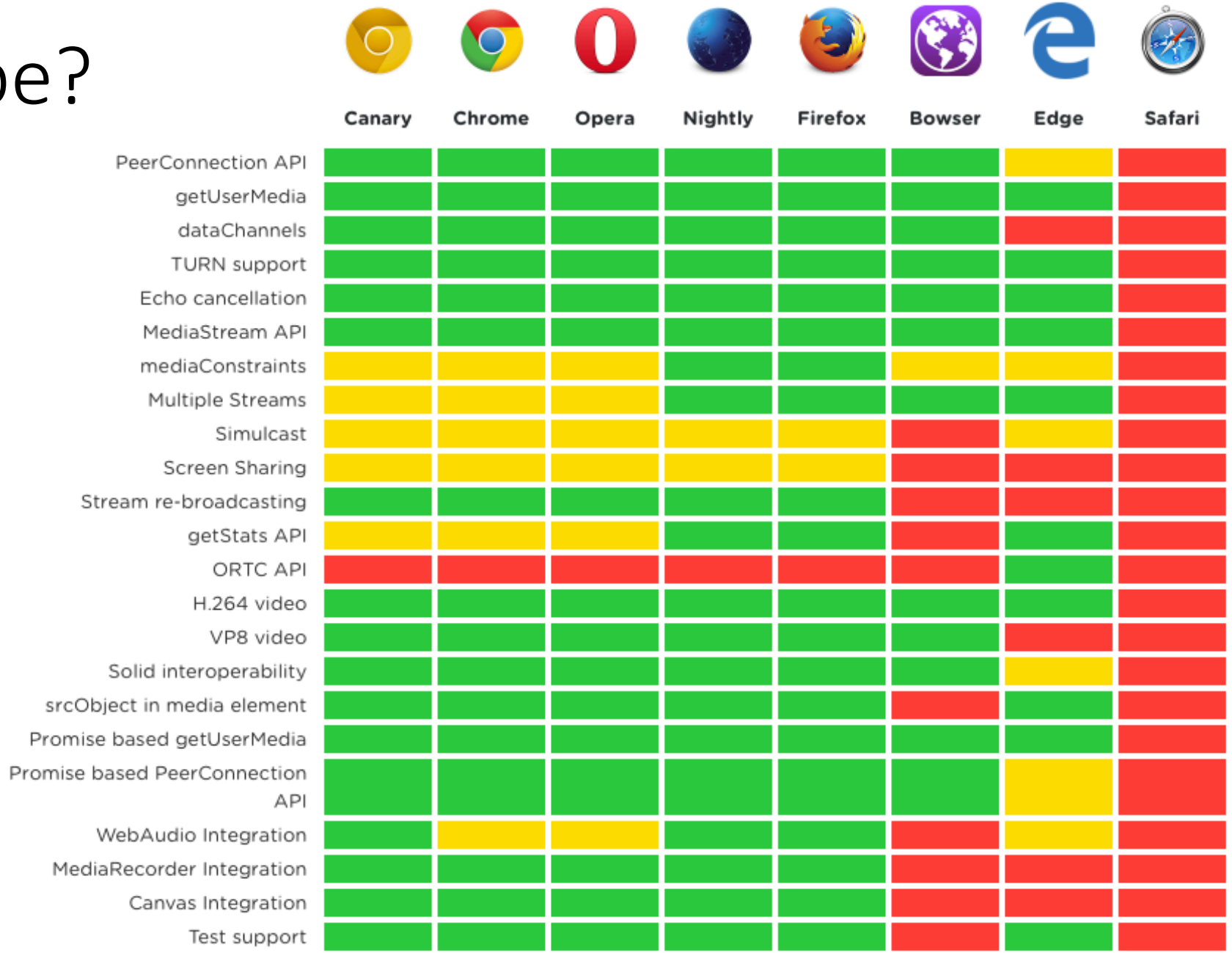
FAC 10 WebRTC Workshop

Peter Wilson, IpCortex

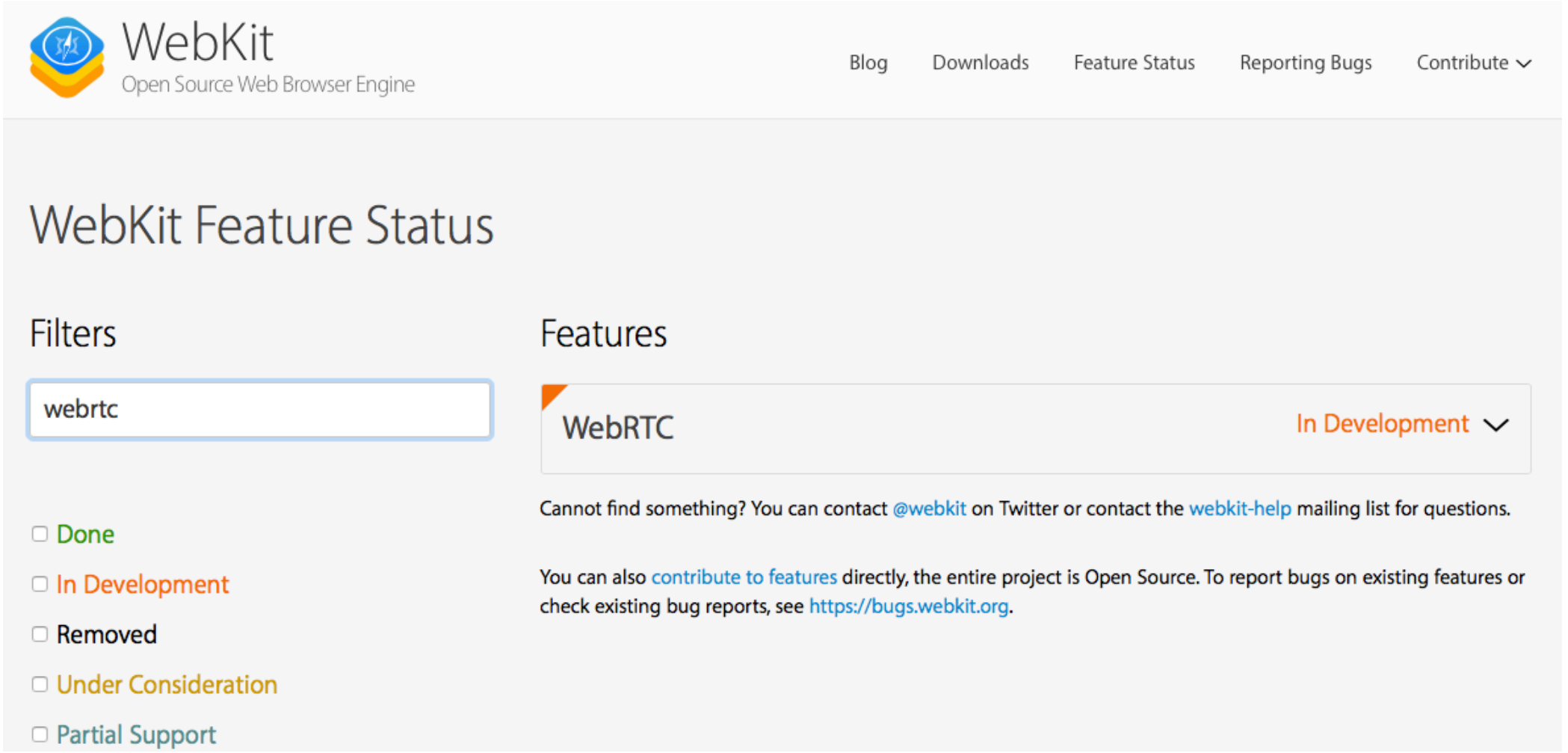
Making the web ‘real-time’

- 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

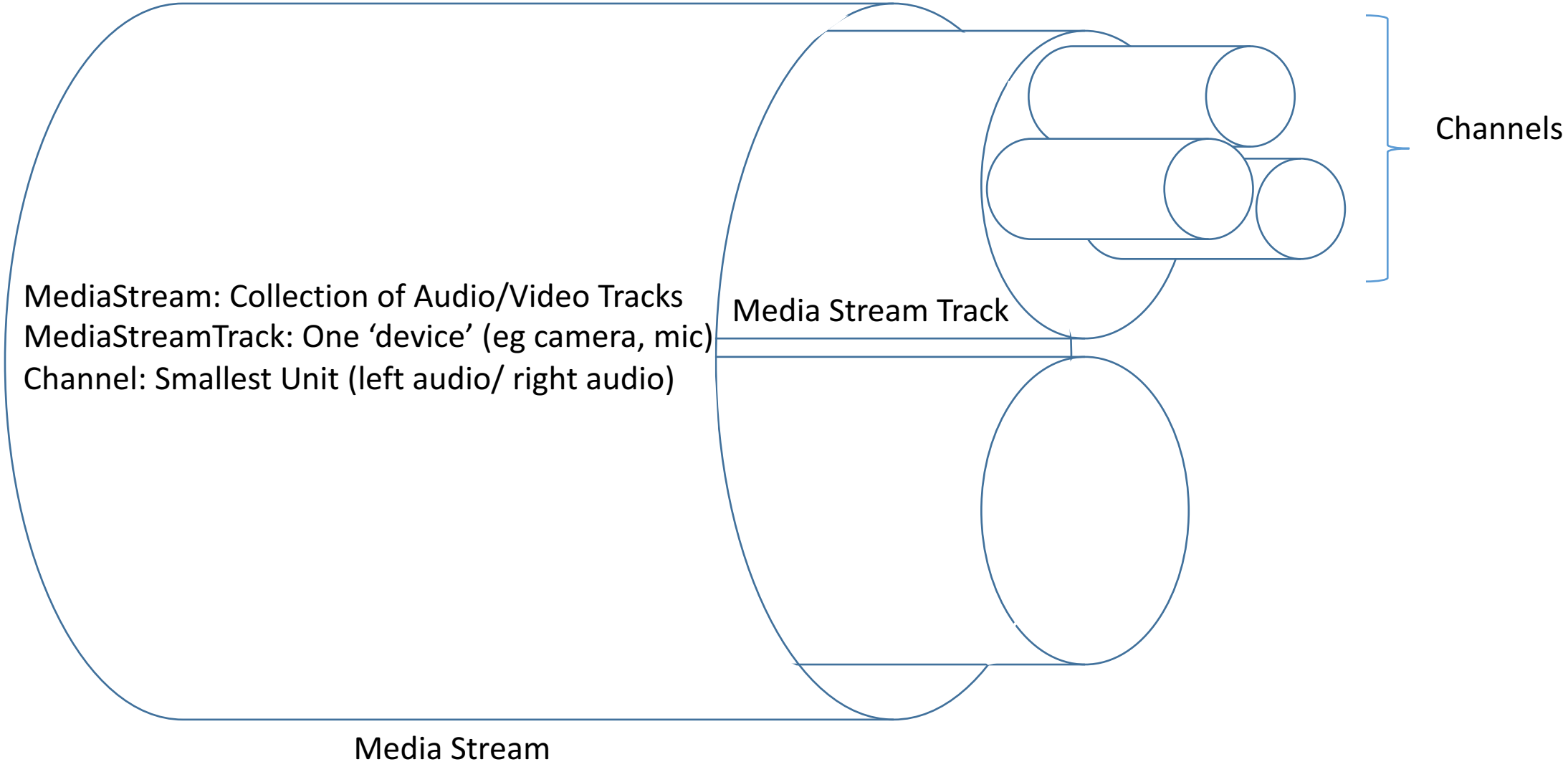
Is it all hype?



Apple Safari



Media Streams

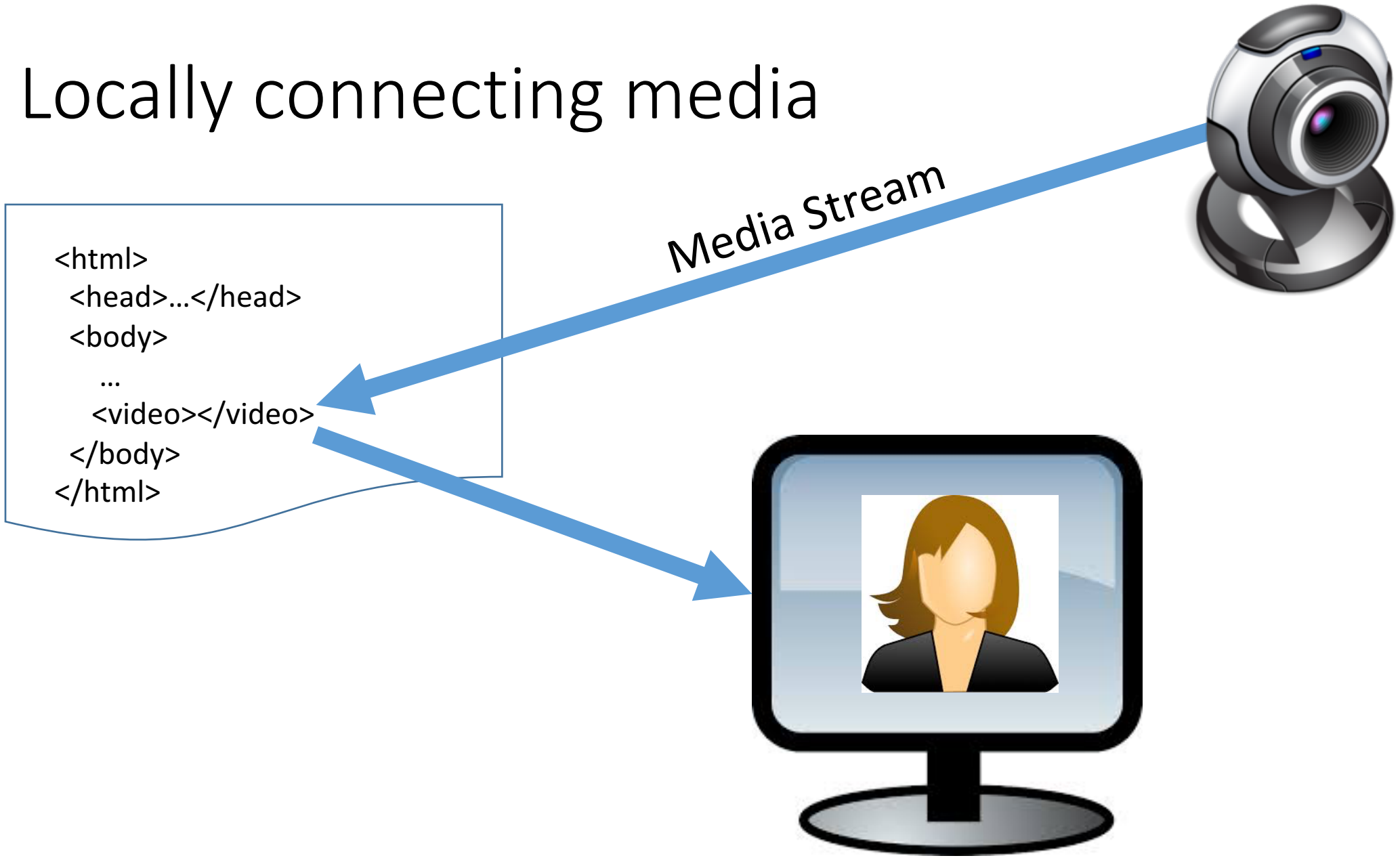


https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API

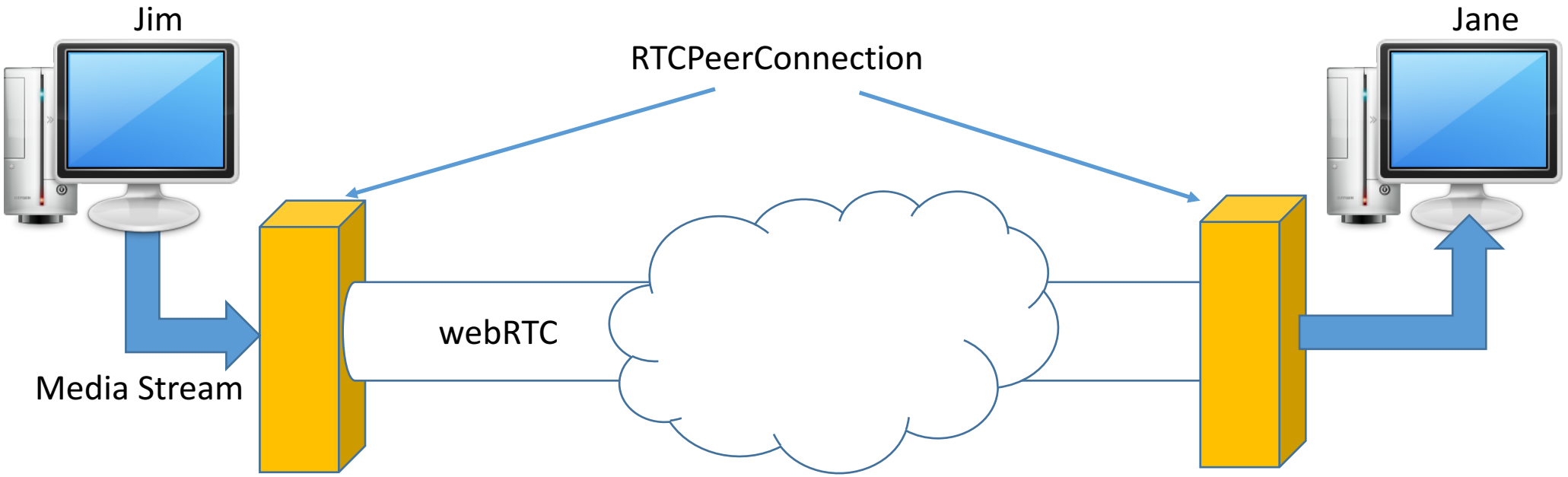
Media Stream

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

Locally connecting media

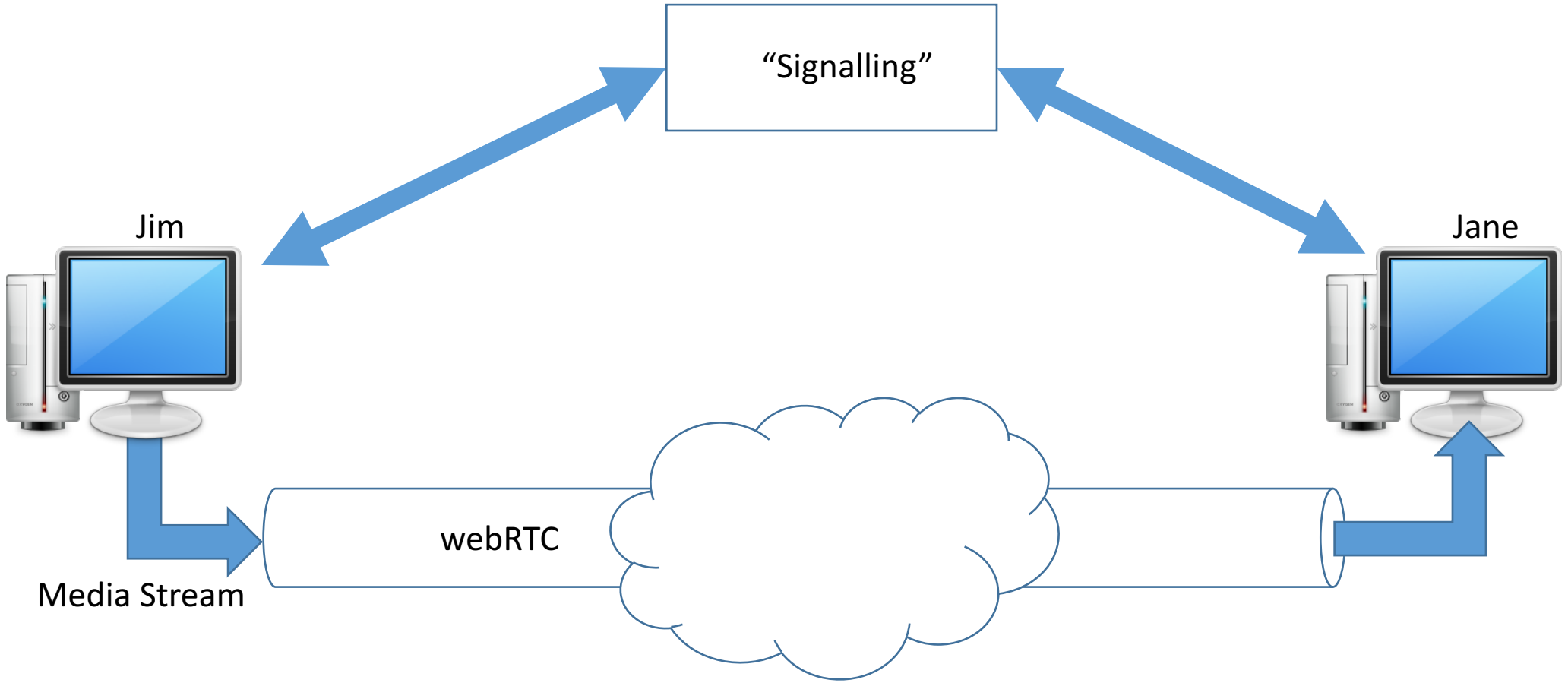


Connecting between devices



- 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



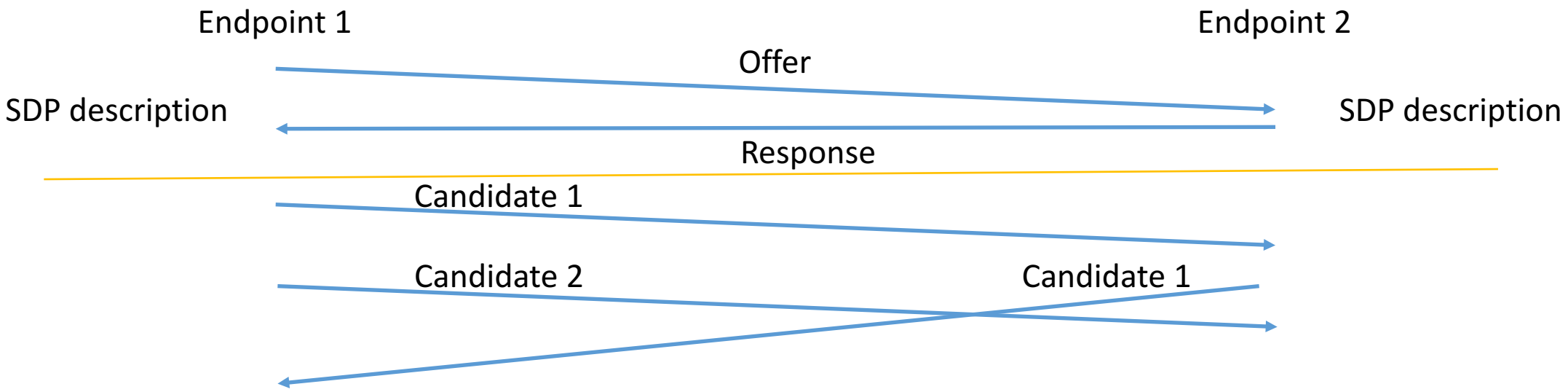
Signalling

- 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.

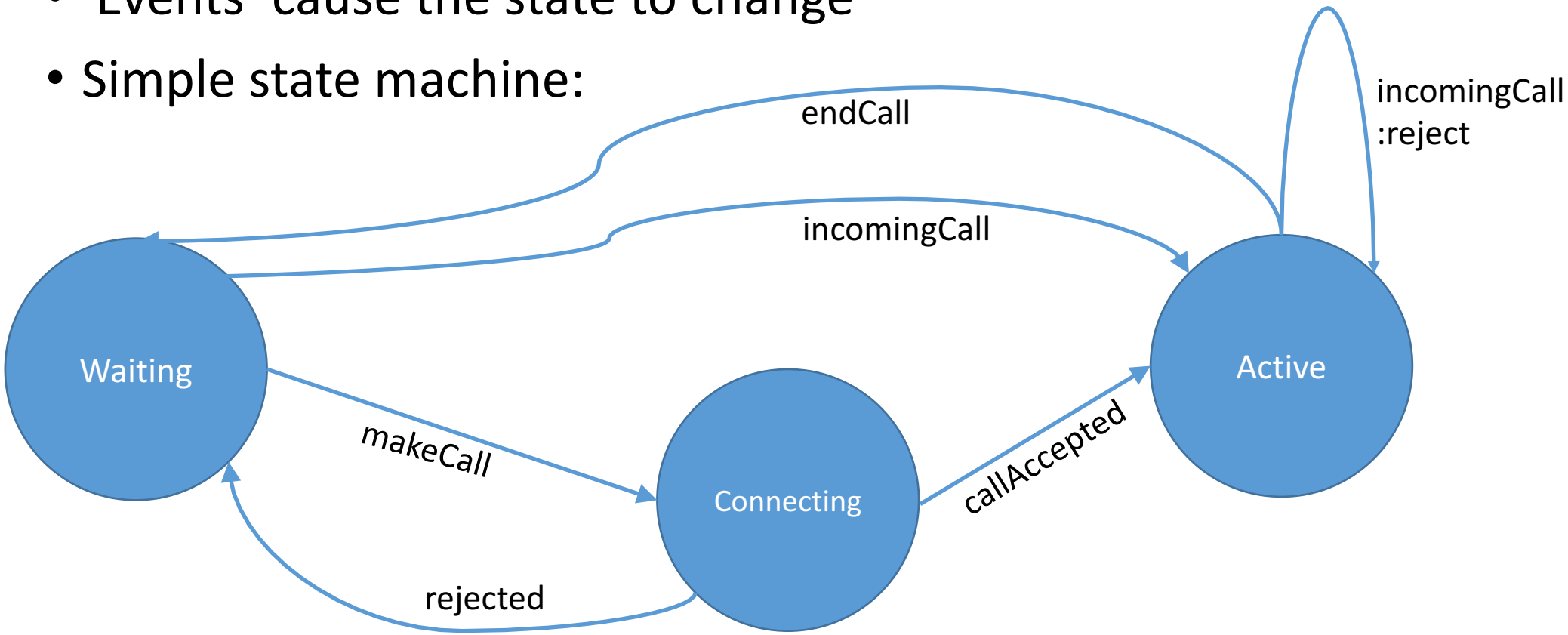
Classes and Objects in Javascript (ES6)

```
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');
```

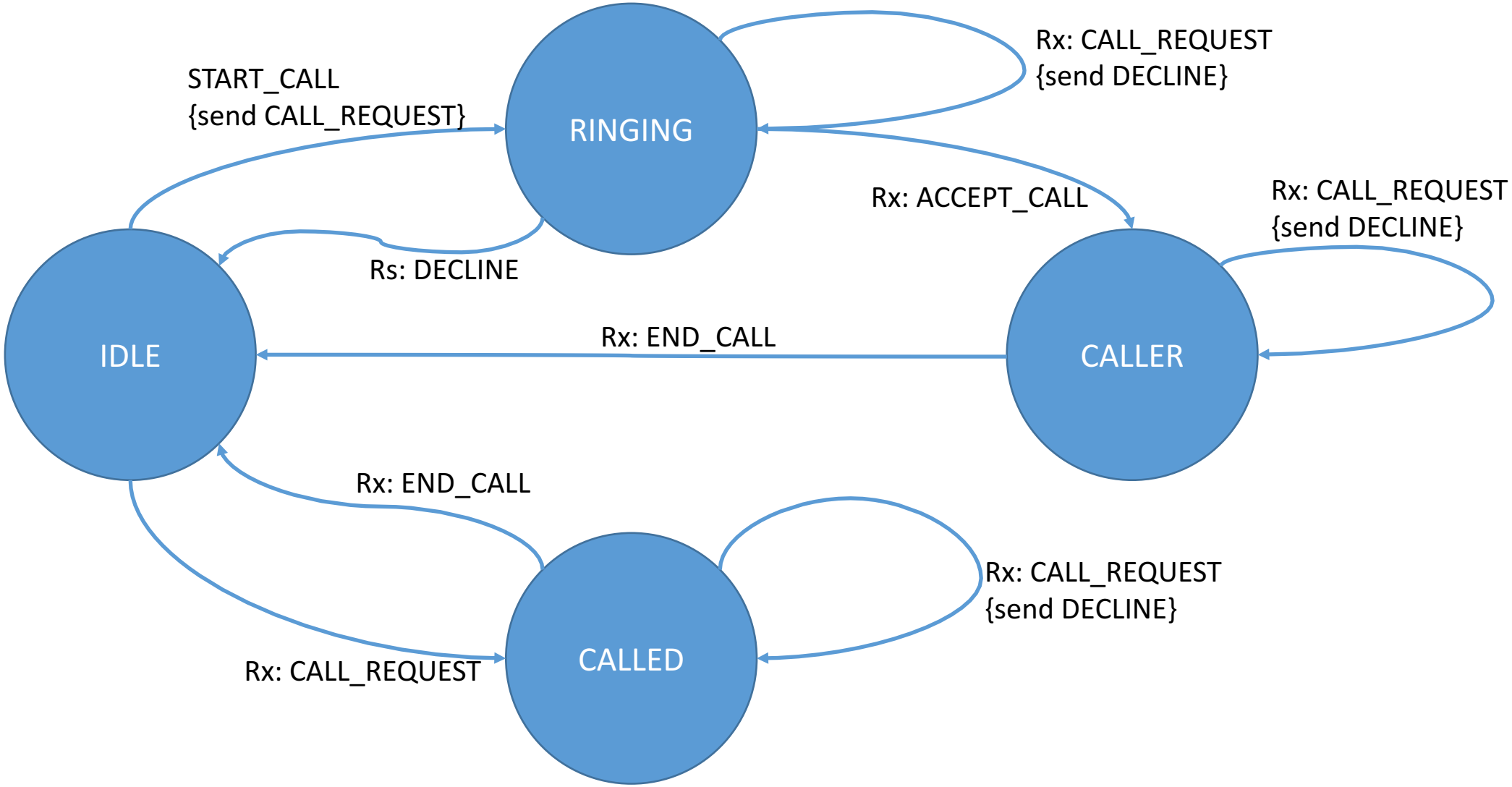
- 'this' references the 'current object'
  - Thing of the **method** as a 'message' and the **object** as the 'address'

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:

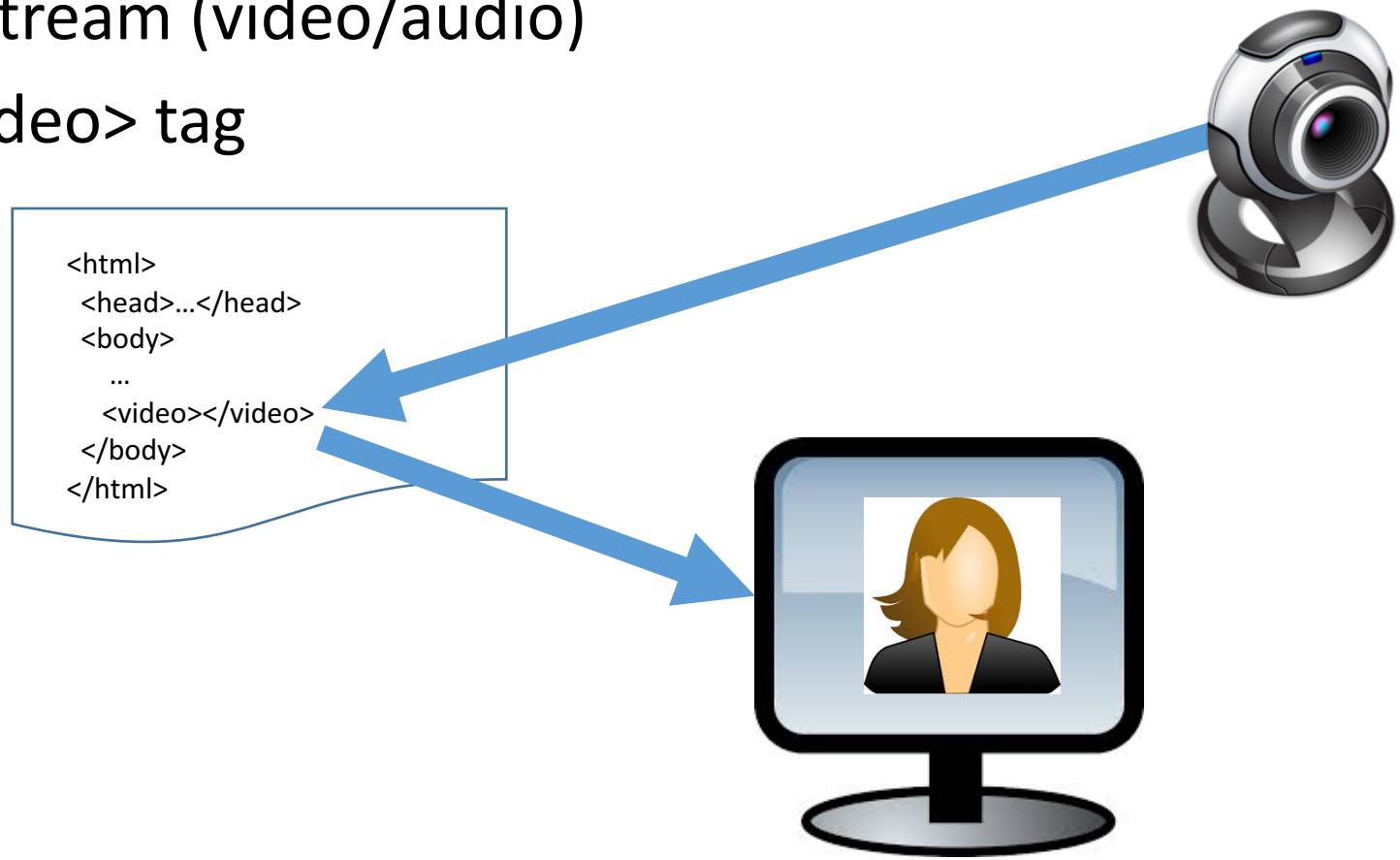


Workshop End Point State Machine



2. Local stream

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



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(() => {…});
```

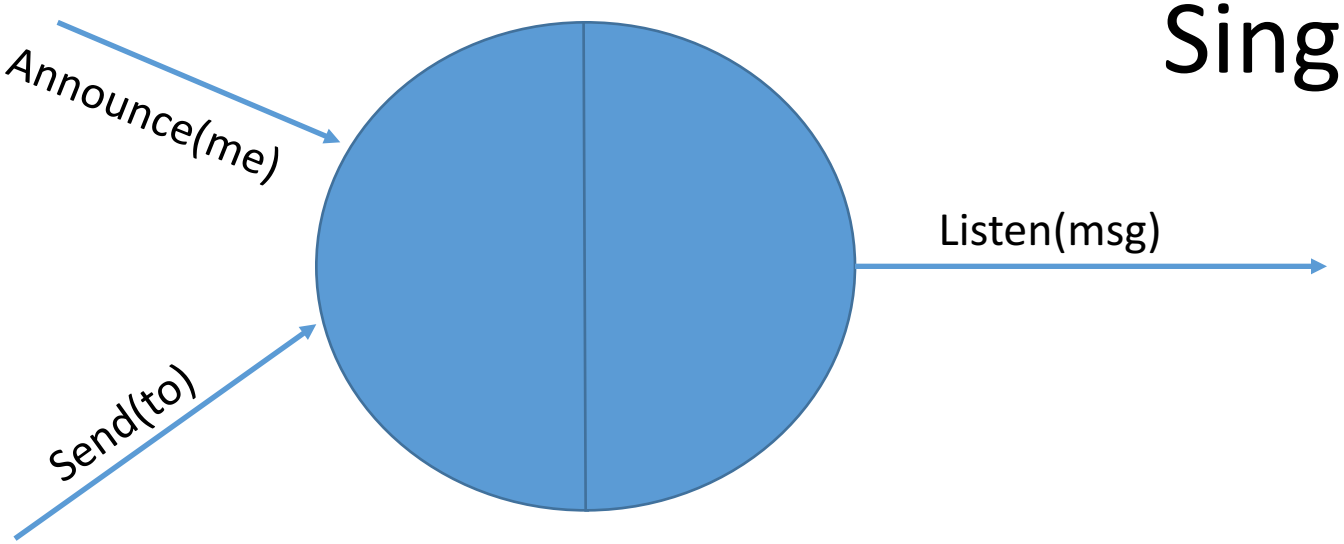
3. Local Peer Connection

```
<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!

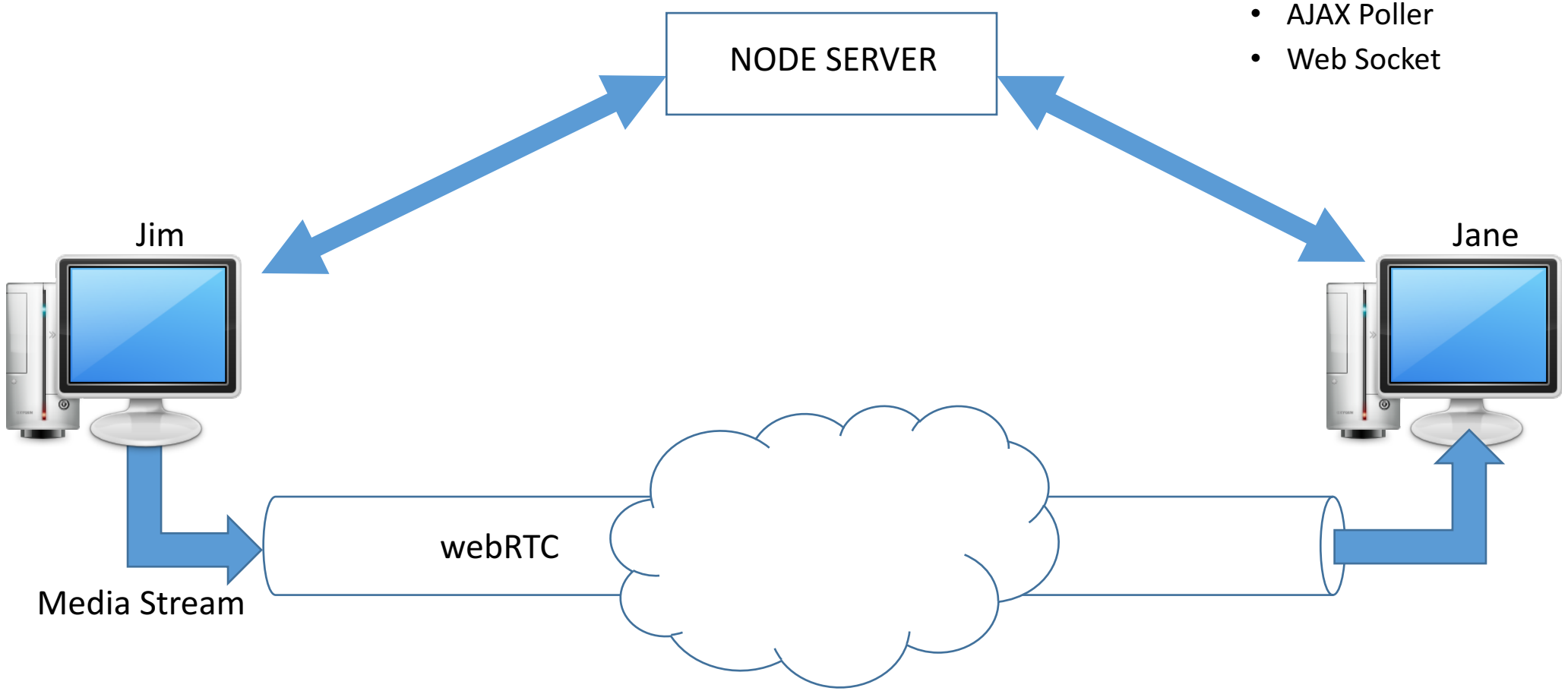
3. Code structure and aims

- 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

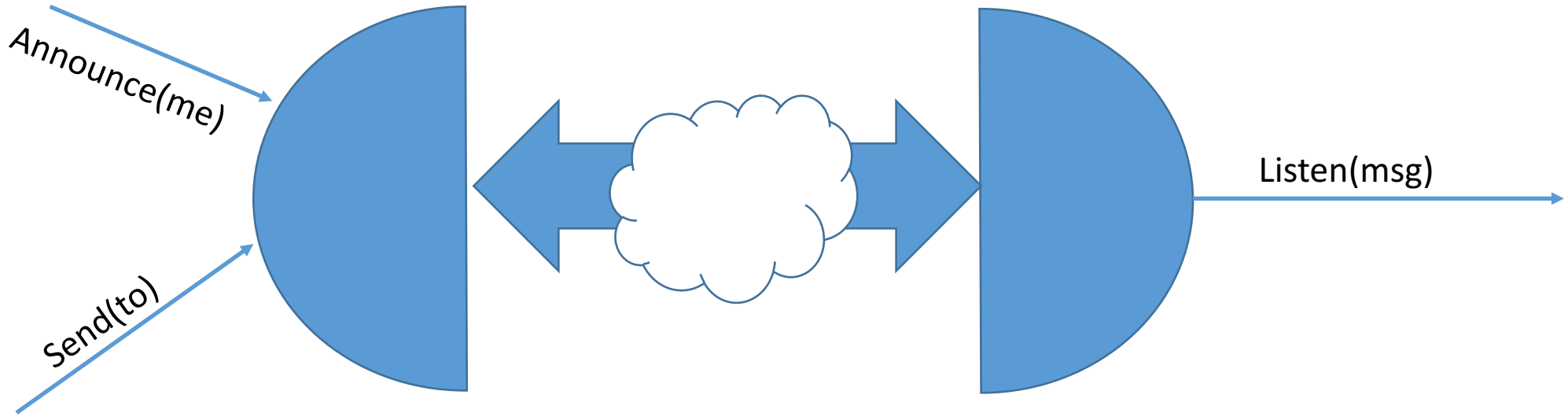
4. Remote Peer Connections

- 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)

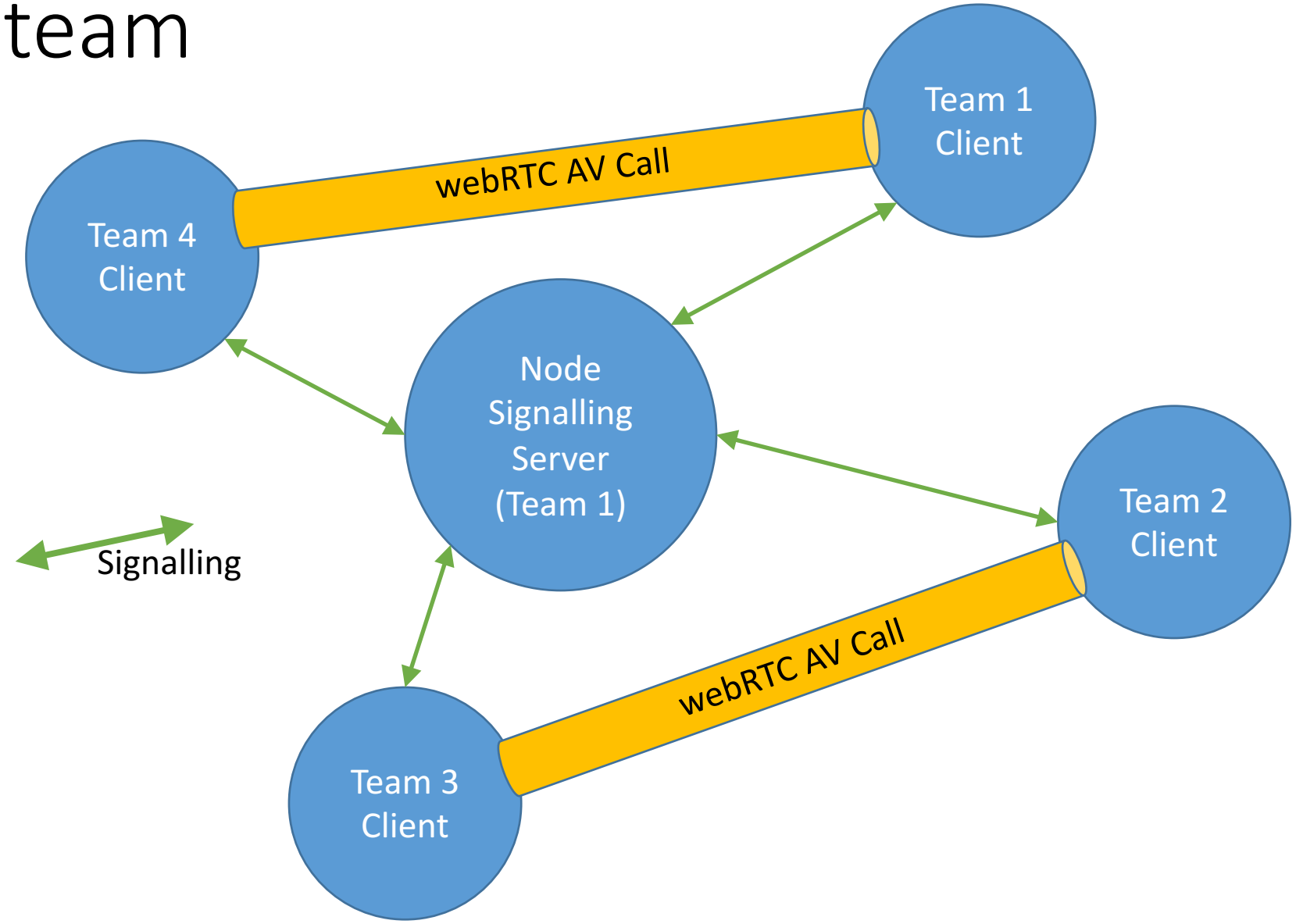
- If you've implemented part 3 well then
  - All you should have to do to the client/browser is rewrite EndPoint
  - There should be **no changes** required to VideoEndPoint
- Extend the node server to implement the REST interface
  - Specified in the WIKI

5. Talking between teams

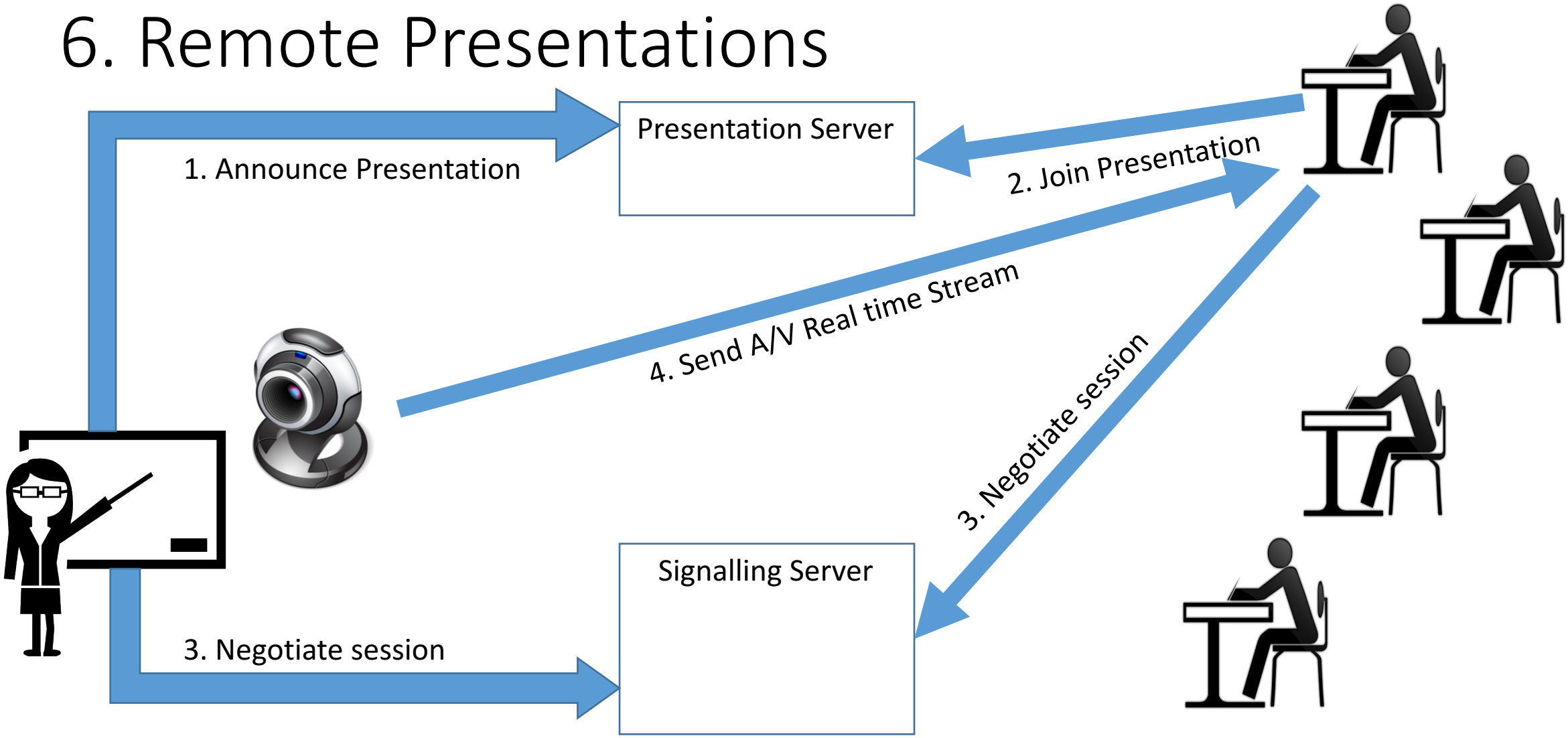
- There should be 4 teams with working clients and node servers
- The node servers are implementing a defined protocol
- So…
  - The client from any team should be able to talk to any node server
  - Two clients from two different teams should be able to connect to the same node server and make a call

- 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



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);
}
```