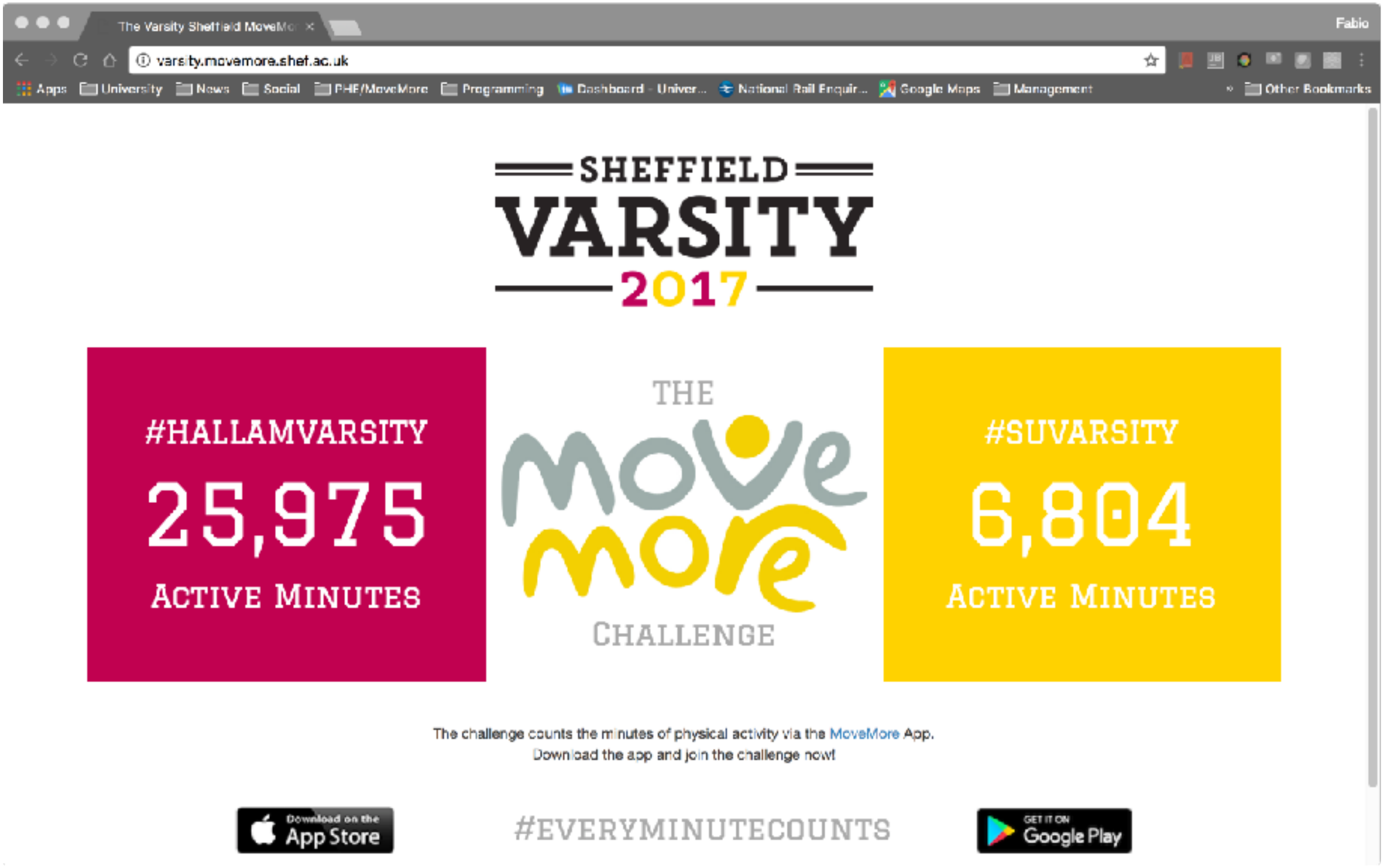
The University Of Sheffield.

Join the challenge!



http://varsity.movemore.shef.ac.uk/

Express, Socket.io and WebRTC

Express, Socket.io, WebRTC
and all the other tools you really need to know
to be the coolest kid on the block

Prof Fabio Ciravegna
Department of Computer Science
The University of Sheffield
f.ciravegna@shef.ac.uk

Routing

- Routing refers to determining how an application responds to a client request to a particular endpoint,
  - which is a URI (or path) and
  - a specific HTTP request method (GET, POST, and so on)
- Each route can have one or more handler functions
  - which is / are executed when the route is matched

Nodejs Get Routing to /

```
var http = require('http');
var url= require('url');
var server = http.createServer(function (request, response) {
   var pathname = url.parse(req.url).pathname;
   if ((pathname=='/')&& (request.method == 'GET')) {
     response.end('Hello World');
   }});
server.listen(3000);
```

The app starts a server and listens on port 3000 for connection.
It will respond with "Hello World!" for requests to the homepage.
For every other path, it will respond with a 404 Not Found.

The University Of Sheffield.

Express

http://expressjs.com/

- Node.js is great but most of its functions are rather verbose
- Express
  - A minimal and flexible node.js web application framework with a robust set of features for web applications
    - With a myriad of HTTP utility methods and middleware at your disposal
      - Creating a robust API is quick and easy
  - A thin layer of fundamental web application features, without obscuring Node features that you know and love

In express

npm install express

(create a file called app.js)

```
var express = require('express')
var app = express()
var server = app.listen(3000);

app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

Routing in Express

- Route definition:

  - app is an instance of express
  - METHOD is an HTTP request method (POST, GET)
  - PATH is a path on the server,
  - HANDLER is the callback function executed when the route is matched

```
app.METHOD(PATH, CALLBACK)
```

Routing Examples

METHOD

```
// respond with "Hello World!" on the homepage
app.get('/', function (req, res) {
  res.send('Hello World!');
})

// accept POST request on the homepage
app.post('/', function (req, res) {
  res.send('Got a POST request');
})
```

PATH

CALLBACK or HANDLER

- app is an instance of express
- is an HTTP request method (POST, GET)
- PATH is a path on the server,
- HANDLER is the callback function execute

app.all

- Special routing method not derived from any HTTP method
- Used for representing all request methods.

```
// respond with "Hello World!" to all type of
// requests (post, get, etc.) on the homepage

app.all('/', function (req, res, next) {
  res.send('Got a request');
})
```

Serving static files

http://expressjs.com/starter/static-files.html

- Serving static files is accomplished with the help of a built-in middleware in Express
  - express.static.
- Pass the name of the directory where you keep you static files
- For example, if you keep your images, CSS, and JavaScript files in a directory named **public**, you can do this:

```
app.use(express.static('public'));
```

note!

Static

- Now, you will be able to load the files under the public directory:

- http://localhost:3000/images/kitten.jpg
- http://localhost:3000/css/style.css
- http://localhost:3000/js/app.js
- http://localhost:3000/images/bg.png
- http://localhost:3000/hello.html

Route Paths

- Route paths define the endpoints at which requests can be made to.
  - e.g. '/'
- In express they can be:
  - strings
  - string patterns
  - regular expressions.
- Note!
  - Query strings are not a part of the route path.
    - In http://localhost/index.html**?index=34**
    - **?index=34** is **not** part of the route path

Examples

```
// with match request to the root
app.get('/', function (req, res) {
  res.send('root requested')
})

// will match requests to /about
app.get('/about', function (req, res) {
  res.send('about requested')
})

// will match request to /random.html
app.get('/random.html', function (req, res) {
  res.send('random.html requested')
})
```

String Patterns

```
// will match acd and abcd
app.get('/ab?cd', function(req, res) {
  res.send('ab?cd')
})

// will match abcd, abbcd, abbbcd, and so on
app.get('/ab+cd', function(req, res) {
  res.send('ab+cd')
})

// will match abcd, abxcd, abRABDOMcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
  res.send('ab*cd')
})

// will match /abe and /abcde
app.get('/ab(cd)?e', function(req, res) {
 res.send('ab(cd)?e')
})
```

Note! The characters **?, +, and ()** are subsets of their Regular Expression counterparts. The hyphen (-) and the dot (.) are interpreted literally by string-based paths. * means any char

Regular Expressions

- Following the Unix standard (also used in the vim editor)

```
// will match anything with an a in the route name:
app.get(/a/, function(req, res) {
  res.send('/a/')
})

// will match butterfly, dragonfly; but not butterflyman,
dragonfly man, and so on
app.get(/.*fly$/, function(req, res) {
  res.send('/.*fly$/')
})
```

The University Of Sheffield.

- Route handlers for a single route path can be created using

```
app.route('/book')

  .get(function(req, res) {
    res.send('Get a random book');
  })

  .post(function(req, res) {
    res.send('Add a book');
  })
```

- This reduce redundancy and typos.

Connecting to DB

MySQL

Module: mysql
Installation

http://expressjs.com/guide/database-integration.html

```
$ npm install mysql
```

Example

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'dbuser',
  password : 's3kreee7'
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function(err, rows, fields) {
  if (err) throw err;
  console.log('The solution is: ', rows[0].solution);
});

connection.end();
```

The callback has the same syntax as the standard nodejs one

app.locals

app.locals

The `app.locals` object is a JavaScript object, and its properties are local variables within the application.

```
app.locals.title
// => 'My App'

app.locals.email
// => 'me@myapp.com'
```

Once set, the value of `app.locals` properties persist throughout the life of the application, in contrast with res.locals properties that are valid only for the lifetime of the request.

You can accesss local variables in templates rendered within the application. This is useful for providing helper functions to templates, as well as app-level data. Note, however, that you cannot access local variables in middleware.

```
app.locals.title = 'My App';
app.locals.strftime = require('strftime');
app.locals.email = 'me@myapp.com';
```

- **app.locals** variables persist for the lifetime of the server
- **res.locals** for the lifetime of the request
- (all the above are user-defined variables)

The University Of Sheffield.

Request parameter

http://expressjs.com/4x/api.html

req.hostname

Contains the hostname from the "Host" HTTP header.

```
// Host: "example.com:3000"
req.hostname
// => "example.com"
```

req.ip

The remote IP address of the request.

If the `trust proxy` is setting enabled, it is the upstream address; see Express behind proxies for more information.

```
req.ip
// => "127.0.0.1"
```

The University Of Sheffield

req.path

Contains the path part of the request URL.

```
// example.com/users?sort=desc
req.path
// => "/users"
```

req.protocol

The request protocol string, "http" or "https" when requested with TLS. When the "trust proxy" setting trusts the socket address, the value of the "X-Forwarded-Proto" header ("http" or "https") field will be trusted and used if present.

```
req.protocol
// => "http"
```

req.query

An object containing a property for each query string parameter in the route. If there is no query string, it is the empty object, `{}`.

```
// GET /search?q=tobi+ferret
req.query.q
// => "tobi ferret"

// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse
req.query.order
// => "desc"
```

req.accepts(types)

Checks if the specified content types are acceptable, based on the request's `Accept` HTTP header field. The method returns the best match, or if none of the specified content types is acceptable, returns `undefined` (in which case, the application should respond with `406 "Not Acceptable"`).

The `type` value may be a single MIME type string (such as "application/json"), an extension name such as "json", a comma-delimited list, or an array. For a list or array, the method returns the ***best*** match (if any).

```
// Accept: text/html
req.accepts('html');
// => "html"

// Accept: text/*, application/json
req.accepts('html');
// => "html"
req.accepts('text/html');
// => "text/html"
req.accepts('json, text');
// => "json"
req.accepts('application/json');
// => "application/json"

// Accept: text/*, application/json
req.accepts('image/png');
req.accepts('png');
// => undefined

// Accept: text/*;q=.5, application/json
req.accepts(['html', 'json']);
req.accepts('html, json');
// => "json"
```

req.get(field)

Returns the specified HTTP request header field (case-insensitive match). The `Referrer` and `Referer` fields are interchangeable.

```
req.get('Content-Type');
// => "text/plain"

req.get('content-type');
// => "text/plain"

req.get('Something');
// => undefined
```

Aliased as `req.header(field)`.

req.is(type)

Returns `true` if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the `type` parameter. Returns `false` otherwise.

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
// => true

// When Content-Type is application/json
req.is('json');
req.is('application/json');
req.is('application/*');
// => true
```

Getting parameters (POST)

http://stackoverflow.com/questions/5710358/how-to-get-post-query-in-express-node-js

```
var bodyParser = require('body-parser')
// to support JSON-encoded bodies
app.use( bodyParser.json() );

//OR (just one body allowed)
// to support URL-encoded bodies
app.use(bodyParser.urlencoded({
 extended: true
}));

.. in your app

// assuming POST: name=foo&color=red      <-- URL encoding
// OR POST: {"name":"foo","color":"red"}  <-- JSON encoding

app.post('/test-page', function(req, res) {
    var name = req.body.name,
        color = req.body.color;
});
```

The University Of Sheffield.

Response object

res.attachment([filename])

Sets the HTTP response `Content-Disposition` header field to "attachment". If a `filename` is given, then it sets the Content-Type based on the extension name via `res.type()`, and sets the `Content-Disposition` "filename=" parameter.

```
res.attachment();
// Content-Disposition: attachment

res.attachment('path/to/logo.png');
// Content-Disposition: attachment; filename="logo.png"
// Content-Type: image/png
```

res.end([data] [, encoding])

Ends the response process. Inherited from Node's http.ServerResponse.

Use to quickly end the response without any data. If you need to respond with data, instead use methods such as res.send() and res.json().

```
res.end();
res.status(404).end();
```

res.json([body])

Sends a JSON response. This method is identical to `res.send()` with an object or array as the parameter. However, you can use it to convert other values to JSON, such as `null`, and `undefined`. (although these are technically not valid JSON).

```
res.json(null)
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

The University Of Sheffield.

Response methods

The methods on the response object (`res`) in the following table can send a response to the client and terminate the request response cycle. If none of them is called from a route handler, the client request will be left hanging.

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

- very similar to node.js basic methods

By the way Jsonp

https://en.wikipedia.org/wiki/JSONP

- JSONP is a trick that allows a web browser
  - to fetch JSON data from a JSON server and
  - feed them to a Javascript script in the browser.

```
<script type="application/javascript"
        src="http://server.example.com/Users/1234?callback=parseResponse">
</script>
…
<script>
  function parseResponse(param){
    …
  }
</script>
```

Response: redirection

res.redirect([status,] path)

> Express passes the specified URL string as-is to the browser in the `Location` header, without any validation or manipulation, except in case of `back`.
>
> Browsers take the responsibility of deriving the intended URL from the current URL or the referring URL, and the URL specified in the `Location` header; and redirect the user accordingly.

Redirects to the URL dervied from the specified `path`, with specified HTTP status code `status`. If you don't specify `status`, the status code defaults to "302 "Found".

```
res.redirect('/foo/bar');
res.redirect('http://example.com');
res.redirect(301, 'http://example.com');
res.redirect('../login');
```

Redirects can be a fully-qualified URL for redirecting to a different site:

```
res.redirect('http://google.com');
```

Redirects can be relative to the root of the host name. For example, if the application is on `http://example.com/admin/post/new`, the following would redirect to the URL `http://example.com/admin`:

```
res.redirect('/admin');
```

A `back` redirection redirects the request back to the referer, defaulting to `/` when the referer is missing.

```
res.redirect('back');
```

note! useful for login requests!!!

res.send([body])

Sends the HTTP response.

The `body` parameter can be a `Buffer` object, a `String`, an object, or an `Array`. For example:

```
res.send(new Buffer('whoop'));
res.send({ some: 'json' });
res.send('<p>some html</p>');
res.status(404).send('Sorry, we cannot find that!');
res.status(500).send({ error: 'something blew up' });
```

This method performs many useful tasks for simple non-streaming responses: For example, it automatically assigns the `Content-Length` HTTP response header field (unless previously defined) and provides automatic HEAD and HTTP cache freshness support.

When the parameter is a `Buffer` object, the method sets the `Content-Type` response header field to "application/octet-stream", unless previously defined as shown below:

```
res.set('Content-Type', 'text/html');
res.send(new Buffer('<p>some html</p>'));
```

When the parameter is a `String`, the method sets the `Content-Type` to "text/html":

```
res.send('<p>some html</p>');
```

When the parameter is an `Array` or `Object`, Express responds with the JSON representation:

```
res.send({ user: 'tobi' });
res.send([1,2,3]);
```

The University Of Sheffield.

res.sendStatus(statusCode)

Set the response HTTP status code to `statusCode` and send its string representation as the response body.

```
res.sendStatus(200); // equivalent to res.status(200).send('OK')
res.sendStatus(403); // equivalent to res.status(403).send('Forbidden')
res.sendStatus(404); // equivalent to res.status(404).send('Not Found')
res.sendStatus(500); // equivalent to res.status(500).send('Internal Server Error')
```

If an unsupported status code is specified, the HTTP status is still set to `statusCode` and the string version of the code is sent as the response body.

```
res.sendStatus(2000); // equivalent to res.status(2000).send('2000')
```

res.set(field [, value])

Sets the response's HTTP header `field` to `value`. To set multiple fields at once, pass an object as the parameter.

```
res.set('Content-Type', 'text/plain');

res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  'ETag': '12345'
})
```

Aliased as `res.header(field [, value])`.

res.status(code)

Use this method to set the HTTP status for the response. It is a chainable alias of Node's response.statusCode.

```
res.status(403).end();
res.status(400).send('Bad Request');
res.status(404).sendFile('/absolute/path/to/404.png');
```

res.type(type)

Sets the `Content-Type` HTTP header to the MIME type as determined by mime.lookup() for the specified `type`. If `type` contains the "/" character, then it sets the `Content-Type` to `type`.

```
res.type('.html');              // => 'text/html'
res.type('html');               // => 'text/html'
res.type('json');               // => 'application/json'
res.type('application/json');   // => 'application/json'
res.type('png');                // => image/png:
```

The University Of Sheffield.

Express application generator

Use the application generator tool, `express`, to quickly create a application skeleton.

Install it with the following command.

```
$ npm install express-generator -g
```

Display the command options with the `-h` option:

```
$ express -h

  Usage: express [options] [dir]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -e, --ejs           add ejs engine support (defaults to jade)
        --hbs           add handlebars engine support
    -H, --hogan         add hogan.js engine support
    -c, --css <engine>  add stylesheet <engine> support (less|stylus|compass) (defaults to plain css)
    -f, --force         force on non-empty directory
```

For example, the following creates an Express app named ***myapp*** in the current working directory.

```
$ express myapp

   create : myapp
   create : myapp/package.json
   create : myapp/app.js
   create : myapp/public
   create : myapp/public/javascripts
   create : myapp/public/images
   create : myapp/routes
   create : myapp/routes/index.js
   create : myapp/routes/users.js
   create : myapp/public/stylesheets
   create : myapp/public/stylesheets/style.css
   create : myapp/views
   create : myapp/views/index.jade
   create : myapp/views/layout.jade
   create : myapp/views/error.jade
   create : myapp/bin
   create : myapp/bin/www
```

Then install dependencies:

```
$ cd myapp
$ npm install
```

The University Of Sheffield.

Security: login and passwords

Login/password access

- You want to make sure that when a user logs in they have right of access
  - When they request access to a restricted area
    - they are redirected to the login page
    - if they are able to provide valid credentials
      - they are redirected to the requested page

Next slides are a modification of the code at
https://github.com/strongloop/express/blob/master/examples/auth/index.js

The University Of Sheffield.

```
var express = require('express');
var hash = require('./lib/pass').hash;
var bodyParser = require('body-parser')

var app = express()
app.use(express.static('public'));

// to support URL-encoded bodies
app.use(bodyParser.urlencoded({ extended:
true }));
```


```
app.get('/', function (req, res) {
    res.redirect('/login');
});

app.get('/login.html', function (req, res) {
    res.render('login.html');
});

app.post('/login.html', function (req, res) {
    authenticate(req.body.username, req.body.password,
        function (err, user) {
            if (user) {
                res.send('Well done! Your credentials are valid!');
            } else {
//                res.send('Authentication failed, check your '
//                    + ' username and password.');
                res.redirect('/login.html');
            }
        });
});
```

Creating the login database

- Some terms:

In **cryptography**, a **salt** is random data that is used as an additional input to a one-way function that hashes a password or passphrase. The primary function of **salts** is to defend against dictionary attacks versus a list of password hashes and against pre-computed rainbow table attacks.

Salt (cryptography) - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/**Salt_(cryptography)**

Database of valid logins

```
// dummy database

var users = {
     tj: { name: 'tj' }
};

// password for tj
// when you create a user, generate a salt
// and hash the password ('foobar' is the pass here)

hash({ password: 'foobar' }, function (err, pass, salt,
hash) {
  if (err) throw err;
  // store the salt & hash in the "db"
  users.tj.salt = salt;
  users.tj.hash = hash;
});
```

The University Of Sheffield.

```
// Authenticate using our plain-object database
function authenticate(name, past, fn) {
    var user = users[name];
    // query the db for the given username
    if (!user)
        return fn(new Error('cannot find user'));
    // apply the same algorithm to the POSTed
    //  password, applying
    // the hash against the pass / salt,
    // if there is a match we found the user
     hash({ password: pass, salt: user.salt },
      function (err, pass, salt, hash) {
       if (err) return fn(err);
       if (hash == user.hash) return fn(null, user);
       fn(new Error('invalid password'));
      });}
```

Sessions

- Logging in is pointless if you do not have sessions where you can
  - log in
  - do a number of operations (e.g. access multiple pages)
  - log out
- In Express
  - npm install express-session
  - https://www.npmjs.com/package/express-session

req.session

To store or access session data, simply use the request property `req.session`, which is (generally) serialized as JSON by the store, so nested objects are typically fine. For example below is a user-specific view counter:

```
app.use(session({ secret: 'keyboard cat', cookie: { maxAge: 60000 }}))

app.use(function(req, res, next) {
  var sess = req.session
  if (sess.views) {
    sess.views++
    res.setHeader('Content-Type', 'text/html')
    res.write('<p>views: ' + sess.views + '</p>')
    res.write('<p>expires in: ' + (sess.cookie.maxAge / 1000) + 's</p>')
    res.end()
  } else {
    sess.views = 1
    res.end('welcome to the session demo. refresh!')
  }
})
```

maxAge is in millisecs

The requester must pass the session

Session.regenerate()

To regenerate the session simply invoke the method, once complete a new SID and `Session` instance will be initialized at `req.session`.

```
req.session.regenerate(function(err) {
  // will have a new session here
})
```

The University Of Sheffield.

Session.destroy()

Destroys the session, removing `req.session`, will be re-generated next request.

```
req.session.destroy(function(err) {
  // cannot access session here
})
```

Session.reload()

Reloads the session data.

```
req.session.reload(function(err) {
  // session updated
})
```

Session.save()

```
req.session.save(function(err) {
  // session saved
})
```

```
app.use(session({
   secret: 'shhhh, very secret'
     // don't save session if unmodified
    resave: false,
     // don't create session until something stored
  saveUninitialized: false,
}));
// Session-persisted message middleware
app.use(function(req, res, next){
   var err = req.session.error;
  var msg = req.session.success;
  delete req.session.error;
  delete req.session.success;
  res.locals.message = '';
  if (err)
        res.locals.message = '<p class="msg error">'
            + err + '</p>';
  if (msg)
       res.locals.message = '<p class="msg success">'
             + msg + '</p>';
   next();});
```

goes to next callback

The University Of Sheffield

this method has 2 callbacks

```
app.get('/restricted', restrict,
  function(req, res){
    res.send('Wahoo! you entered the restricted area');
});
//first callback
function restrict(req, res, next) {
  if (req.session.user) {
    next();
  } else {
    req.session.error = 'Access denied!';
    req.session.back= req.path;
    res.redirect('login.html');  }}

app.get('/logout', function(req, res){
  // destroy the user's session to log them out
  // will be re-created next request
  req.session.destroy(function(){
    res.redirect('/');
  });});
```

goes to next callback

it saves where the user wanted to go

```
app.post('/login', function (req, res) {
  authenticate(req.body.username, req.body.password,
     function (err, user) {
       if (user) {
          var back= req.session.back||'/'
          // Regenerate session when signing in
          // to prevent fixation
          req.session.regenerate(function () {
             req.session.back=back;
                    // Store the user's primary key
                    // in the session store to be retrieved,
                    // or in this case the entire user object
          req.session.user = user;
          req.session.success = 'Logged in as ' + user.name
              + ' click to <a href="/logout">logout</a>. ' +
              ' You may now access ' +
               ' <a href="/restricted">/restricted</a>.';
          res.redirect(back);
        });
    } else {
       req.session.error = 'Authentication failed, '
           + ' please check your ' + ' username and password.';
       res.redirect('/login.html?error='+req.session.error);
          }        });});
```

save before regenerating either the target location or /

The University Of Sheffield.

Towards a different client/ server connection

socket.io

Traditional Client/Server architectures

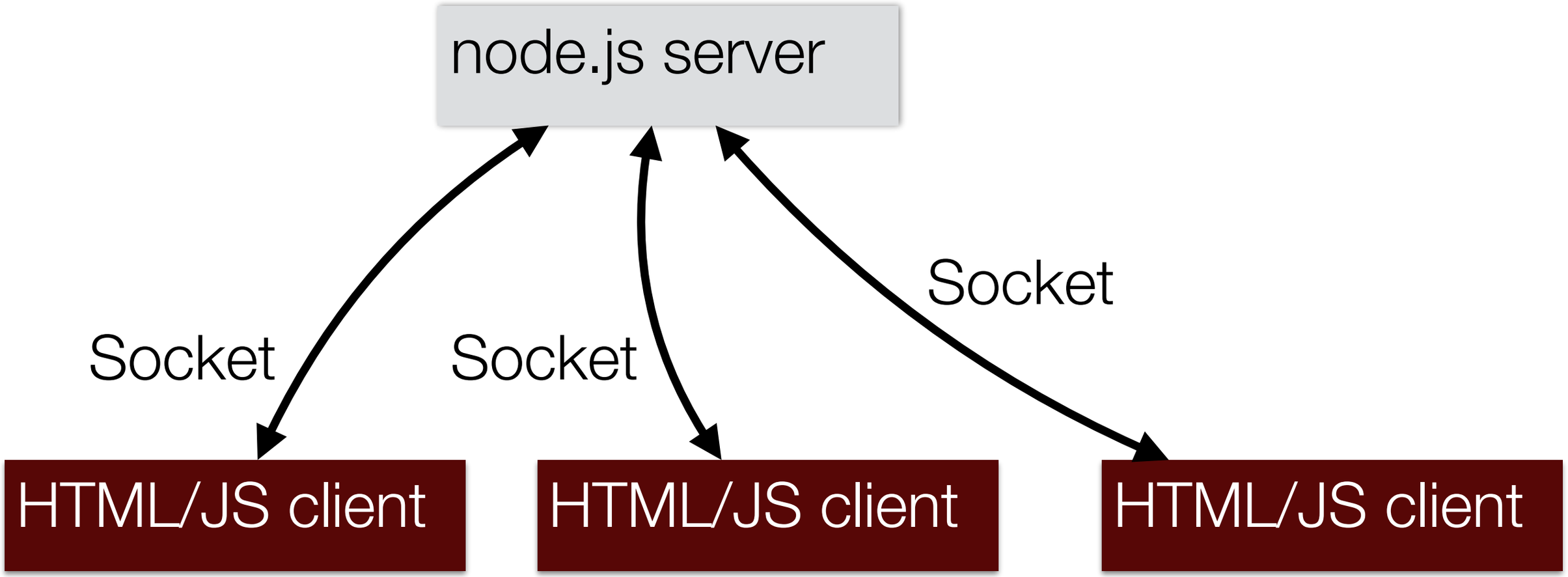
- When you make a request with Ajax/HTML/ Javascript you wait for the server to return results
- However in many cases
  - (e.g. twitter streaming API)
- you just would like the server to be able to reopen the connection and send more data
  - otherwise you need a client regularly polling the server to check if there are new results
    - rather cumbersome

Socket.io

Socket.io

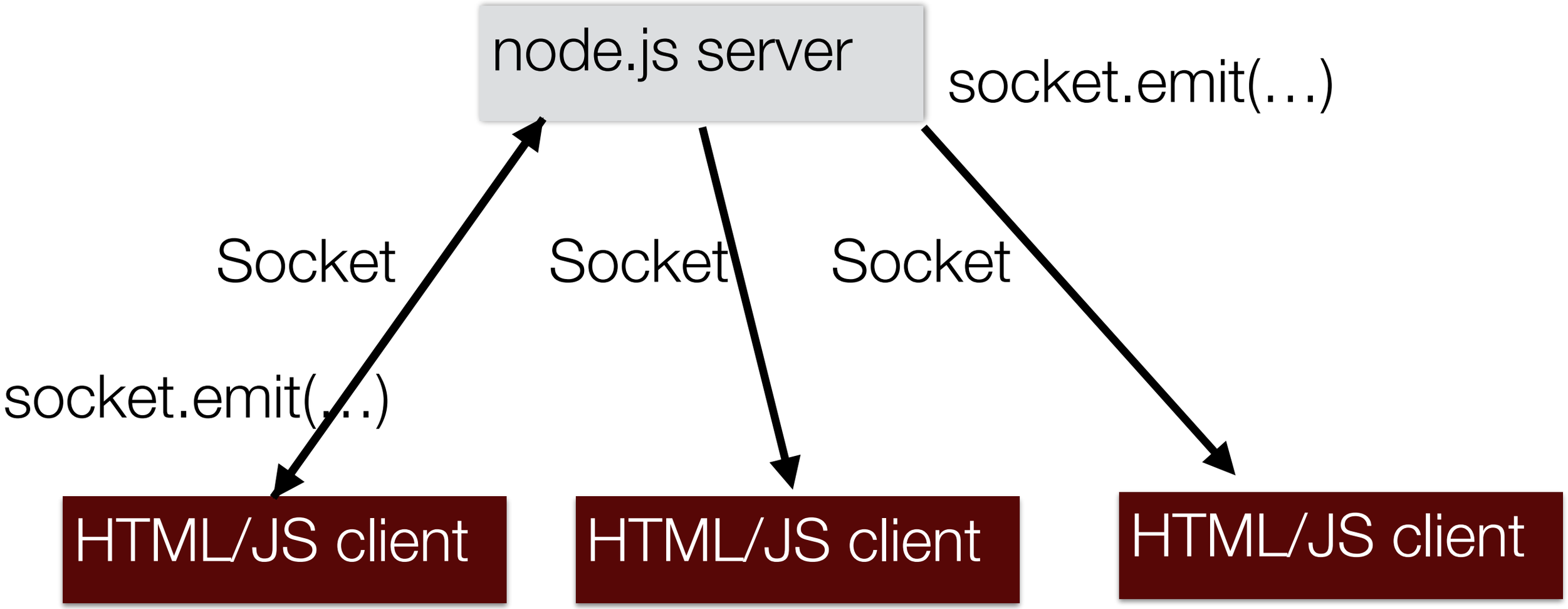
- Socket.IO enables real-time bidirectional event-based communication.
  - It works on every platform, browser or device, focusing equally on reliability and speed
  - It has two parts:
    - a client-side library that runs in the browser,
    - a server-side library for node.js.
  - Both components have a nearly identical API
- It primarily uses the WebSocket protocol
- It is possible to send any data,
  - Including blobs, i.e. Image, audio, video
- it is event based (on…)
- communication can be started by both client and server once connection is established and until it is closed from either sides

1 server, n clients, n sockets



- Socket is private channel shared by 1 client and 1 server
- However clients can communicate via the server

1 server, n clients, n sockets



- Communication happens via the command socket.emit(…) on both sides

Client Server communication

node.js server

```
var io = require('socket.io')(http);
file.serve(…);

io.on('connection', function(socket){
  socket.on ('message',
        function (param){…});
});

socket.emit ('message' param)
```

HTML/JS client

```
<script src="/socket.io/socket.io.js">
</script>
…
<script>
  var socket = io();

  socket.emit ('message' param)

  socket.on ('message', function (param){…})
```

client must open the socket

socket automatically closes when client navigates away from page

Using with Node http server

Server (app.js)

```
var app = require('http').createServer(handl
er)
var io = require('socket.io')(app);
var fs = require('fs');

app.listen(80);

function handler (req, res) {
  fs.readFile(__dirname + '/index.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading index.ht
ml');
    }

    res.writeHead(200);
    res.end(data);
  });
}

io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data
) {
    console.log(data);
  });
});
```

Client (index.html)

```
<script src="/socket.io/socket.io.js"></scri
pt>
<script>
  var socket = io('http://localhost');
  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'dat
a' });
  });
</script>
```

The University Of Sheffield.

Using with Express 3/4

Server (app.js)

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

server.listen(80);

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});

io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data
) {
    console.log(data);
  });
});
```

Client (index.html)

```
<script src="/socket.io/socket.io.js"></scri
pt>
<script>
  var socket = io.connect('http://localhost'
);
  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'dat
a' });
  });
</script>
```

Sending and receiving events

Socket.IO allows you to emit and receive custom events. Besides `connect`, `message` and `disconnect`, you can emit custom events:

Server

```
// note, io(<port>) will create a http server for you
var io = require('socket.io')(80);

io.on('connection', function (socket) {
  io.emit('this', { will: 'be received by everyone'});

  socket.on('private message', function (from, msg) {
    console.log('I received a private message by ', from, ' saying ', msg);
  });

  socket.on('disconnect', function () {
    io.emit('user disconnected');
  });
});
```

Restricting yourself to a namespace

If you have control over all the messages and events emitted for a particular application, using the default / namespace works. If you want to leverage 3rd-party code, or produce code to share with others, socket.io provides a way of namespacing a socket.

This has the benefit of `multiplexing` a single connection. Instead of socket.io using two `WebSocket` connections, it'll use one.

We have two namespaces here: /chat and /news

Server (app.js)

```
var io = require('socket.io')(80);
var chat = io
  .of('/chat')
  .on('connection', function (socket) {
    socket.emit('a message', {
        that: 'only'
      , '/chat': 'will get'
    });
    chat.emit('a message', {
        everyone: 'in'
      , '/chat': 'will get'
    });
  });

var news = io
  .of('/news')
  .on('connection', function (socket) {
    socket.emit('item', { news: 'item' });
  });
```

Client (index.html)

```
<script>
  var chat = io.connect('http://localhost/ch
at')
    , news = io.connect('http://localhost/ne
ws');

  chat.on('connect', function () {
    chat.emit('hi!');
  });

  news.on('news', function () {
    news.emit('woot');
  });
</script>
```

The University Of Sheffield.

Sending and getting data (acknowledgements)

Sometimes, you might want to get a callback when the client confirmed the message reception.

To do this, simply pass a function as the last parameter of `.send` or `.emit`. What's more, when you use `.emit`, the acknowledgement is done by you, which means you can also pass data along:

Server (app.js)

```
var io = require('socket.io')(80);

io.on('connection', function (socket) {
  socket.on('ferret', function (name, fn) {
    fn('woot');
  });
});
```

DO NOT return a private message via socket.emit - it would be a public message!!!!
Return it within a callback!

Client (index.html)

```
<script>
  var socket = io(); // TIP: io() with no ar
gs does auto-discovery
  socket.on('connect', function () { // TIP:
 you can avoid listening on `connect` and li
sten on events directly too!
    socket.emit('ferret', 'tobi', function (
data) {
      console.log(data); // data will be 'wo
ot'
    });
  });
</script>
```

Broadcasting

- Broadcasting means sending a message to everyone else
  - except for the socket that starts it

Broadcasting messages

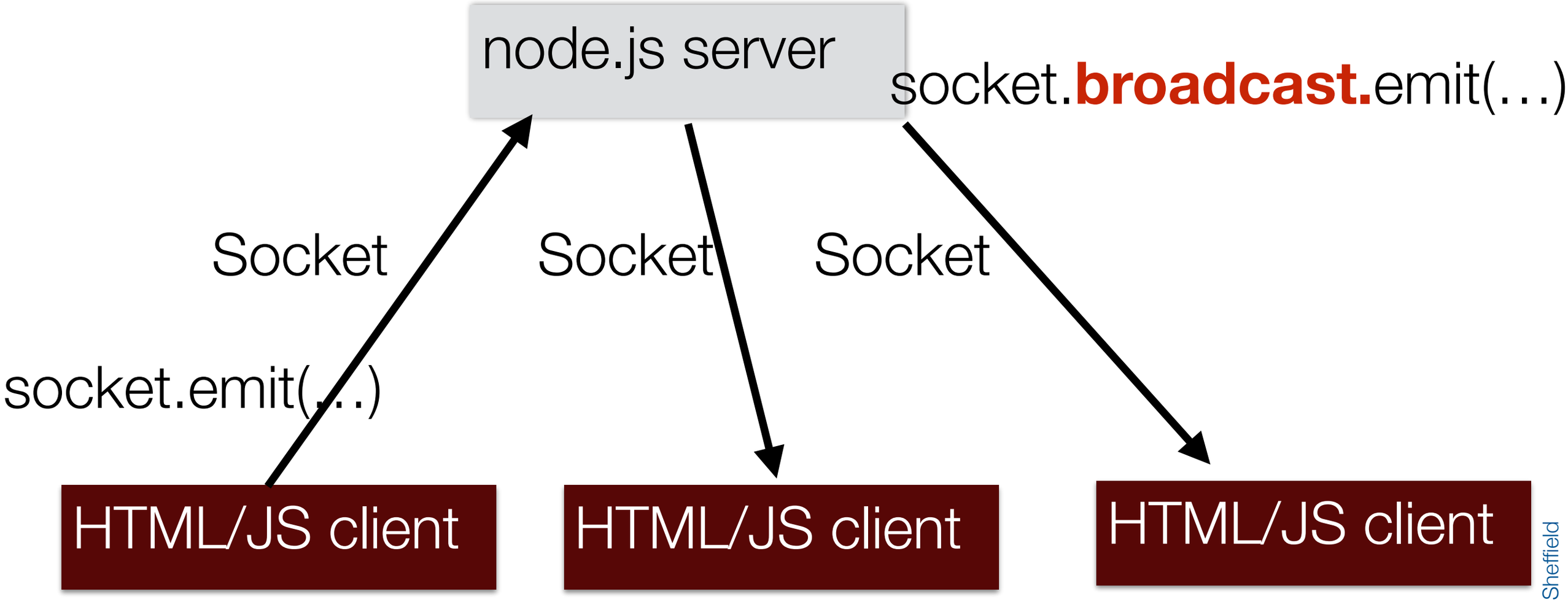
To broadcast, simply add a `broadcast` flag to `emit` and `send` method calls. Broadcasting means sending a message to everyone else except for the socket that starts it.

Server

```
var io = require('socket.io')(80);

io.on('connection', function (socket) {
  socket.broadcast.emit('user connected');
});
```

socket.broadcast.emit



- Communication is not returned to the originating client

Rooms

Within each namespace, you can also define arbitrary channels that sockets can `join` and `leave`.

Joining and leaving

You can call `join` to subscribe the socket to a given channel:

```
io.on('connection', function(socket){
  socket.join('some room');
});
```

And then simply use `to` or `in` (they are the same) when broadcasting or emitting:

```
io.to('some room').emit('some event');
```

This is on the server side
The client can be in just one room at a time

To leave a channel you call `leave` in the same fashion as `join`.

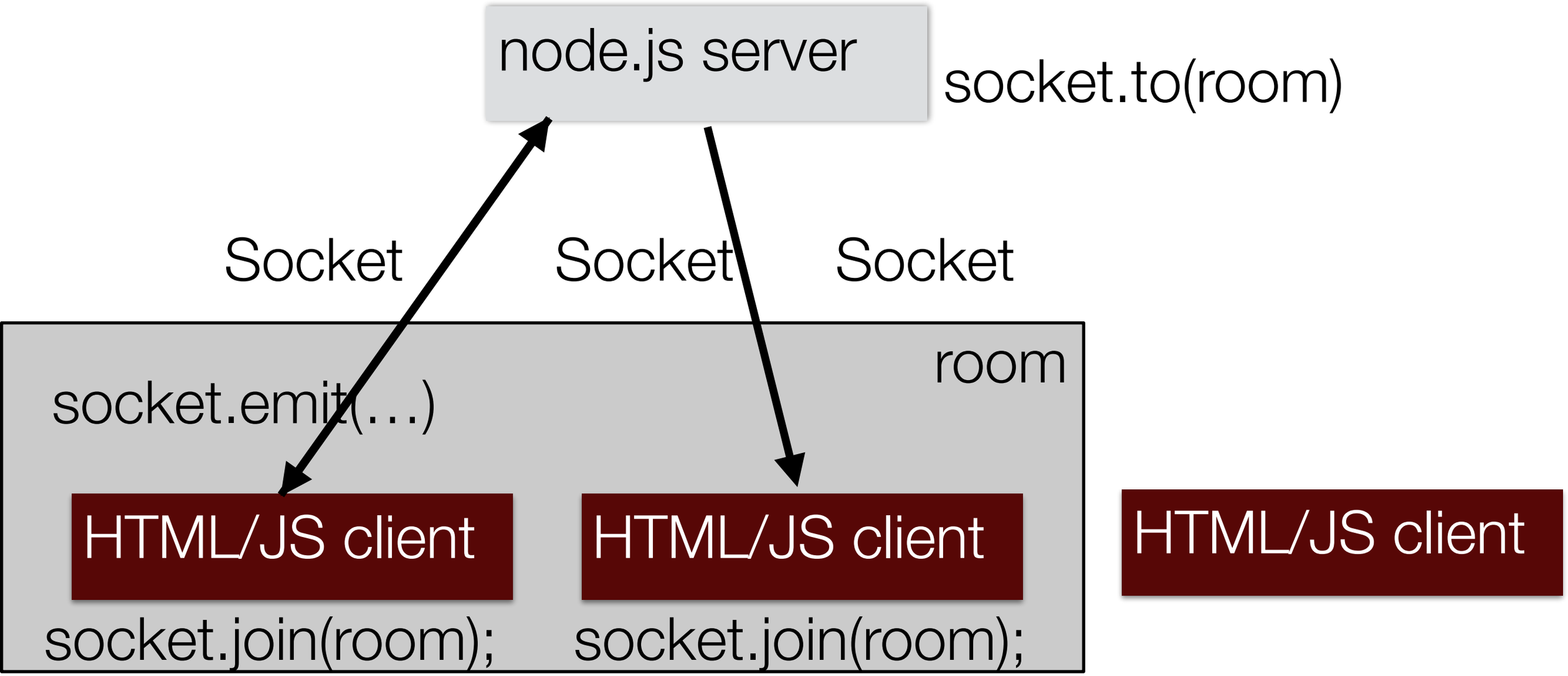
Default room

Each `Socket` in Socket.IO is identified by a random, unguessable, unique identifier `Socket#id`. For your convenience, each socket automatically joins a room identified by this id.

This makes it easy to broadcast messages to other sockets:

```
io.on('connection', function(socket){
  socket.on('say to someone', function(id, msg){
    socket.broadcast.to(id).emit('my message', msg);
  });
});
```

1 server, n clients, n sockets



- Once you are in a room, socket.emit(…) just reaches those in the same room

Disconnection

- e.g. when client moves away from page

```
io.on('connection', function(socket){
  console.log('a user connected');
  socket.on('disconnect', function(){
    console.log('user disconnected');
  });
});
```

The University Of Sheffield.

Instant messaging and chat

Goal

- Creating an instant messaging and chat system
- Design:
  - Node.js/Express serves a file index.html
  - Index.html opens a socket and joins a room
    - the client tells the server it is joining a room
    - the server opens the room if not existing and joins the client to it
    - the server tells everybody in the room the client has joined
- every time the user writes and sends a message
    - the client sends the text to the server
    - the client writes on its own message panel
    - the server broadcasts it to everybody else
    - the other clients in the room write the message onto their message panels

joining a room

node.js server

```
file.serve()

io.on('connection', function(socket){
  socket.on ('joining',
        function (userId, roomId){
          socket.join(room);
          socket.to(room).emit
                ('updatechat',
                socket.username +
                ' has joined this room', '');});
});
```

socket.to(socket.room) broadcasts to all sockets in the room but the calling one

HTML/JS client 1

```
<script src="/socket.io/socket.io.js">
</script>
…
<script>
  var socket = io();

  socket.emit('joining', userId, roomId);
…write on message panel
```

HTML/JS client 2

```
<script src="/socket.io/socket.io.js">
</script>
…
<script>
  var socket = io();
  socket.on ('updatechat',
          function (message){
            …write on message panel
          });
```

sending a message

node.js server

```
io.on('connection', function(socket){
  socket.on ('joining',
        function (userId, roomId){
          socket.join(room);
          socket.broadcast.to(room).emit
                ('updatechat',
                socket.username +
                ' has joined this room', '');});});
socket.on('sendchat', function (data) {
    io.sockets.in(socket.room).emit
        ('updatechat', socket.username,
        data);
 });});
```

io.sockets.in broadcasts to all sockets in the room including the sender (if you use *io.sockets.in* the sender does not need to write independently onto its own panel)

HTML/JS client 1

```
socket.emit('sendchat', message);

socket.on ('updatechat',
        function (message){
          …write on message panel
        });
```

HTML/JS client 2

```
<script src="/socket.io/socket.io.js">
</script>
…
<script>
  var socket = io();
  socket.on ('updatechat',
          function (message){
            …write on message panel
          });
```

The University Of Sheffield.

The rest is just a form!

You are in room: 3946

User 1494 has joined this room:

User 1494: hello!

me: hello to you!

User 1494: it is good to see you

me: indeed!

Anyway!

Send

The University Of Sheffield.

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>My Chat App</title>
    <link href="../css/style.css" rel="stylesheet">
</head>
<body onload="initialiseUserAndRoom();">
<script src="/socket.io/socket.io.js"></script>
<script src="../js/main.js"></script>

<h1 id="roomNo"></h1>
<div>
    <div id="chat"></div>
    <form action="" onsubmit="return sendText()">
        <input id="text" autocomplete="off" autofocus/>
        <button>Send</button>
    </form>


</div>
</body>
```

The University Of Sheffield.

```
/**
 * It sends a message when the user presses the button or return
 * @returns {boolean}
 */
function sendText() {
    var inpt = document.getElementById('text');
    var text = inpt.value;
    if (text == '')
        return false;
    socket.emit('sendchat', text);
    inpt.value = '';
    return false;
}
```

The University Of Sheffield

```
var inColour = false;
var previousWriter = '';
var roomId;
var userId;


var socket = io();
socket.on('updatechat', function (who, text) {
    var divI = document.getElementById('chat');
    var dv2 = document.createElement('div');
    divI.appendChild(dv2);
    dv2.style.backgroundColor = getChatColor(who);
    var whoisit = (who == userId) ? 'me' : who;
    dv2.innerHTML = '<br/>'+whoisit + ': ' + text+'<br/><br/>';
});

/**
 * it initialises the user and the room
 * here you should include the login/password request
 */
function initialiseUserAndRoom() {
    var rndmId = 'User ' + Math.floor((Math.random() * 10000) + 1);
    userId = prompt("Please enter your name", rndmId);
    // if cancel is selected
    if (userId == null) userId = rndmId;

    var randomRoomId = Math.floor((Math.random() * 10000) + 1);
    roomId = prompt("What room would you like to join?", randomRoomId);
    // if cancel is selected
    if (roomId == null) roomId = randomRoomId;
    socket.emit('joining', userId, roomId);
    document.getElementById('roomNo').innerHTML= 'You are in room: '+roomId;
}
```

Remember that if you use this program in Cordova you cannot have in-line declaration.
Define an init function doing this

WebRTC

WebRTC

- WebRTC (Web Real-Time Communication) is
  - an API definition drafted by the World Wide Web Consortium (W3C)
  - that supports browser-to-browser applications
    - for voice calling, video chat, and P2P file sharing
    - **without the need of either internal or external plugins**
    - WebRTC is a free, open project
- This means that:
  - With WebRTC it is possible to create a Skype-like application that works in a browser
- WebRTC is made possible by the availability of bidirectional channels like socket.io

WebRTC

https://tokbox.com/about-webrtc

- WebRTC is made up of three APIs:
  - GetUserMedia
    - Camera, microphone, screen, etc. access
  - PeerConnection
    - Sending and receiving media
  - DataChannels
    - sending non-media direct between browsers
- The development of WebRTC is supported by the W3C, Google, Mozilla, and Opera
  - supported in Opera, Google Chrome versions 23+, and Mozilla Firefox versions 22+.

The University Of Sheffield.

Why is WebRTC important?

The WebRTC project is incredibly important as it marks the first time that a powerful real-time communications (RTC) standard has been open sourced for public consumption. It opens the door for a new wave of RTC web applications that will change the way we communicate today.

Significantly better video quality	WebRTC video quality is noticeably better than Flash.
Up to 6x faster connection times	Using JavaScript WebSockets, also an HTML5 standard, improves session connection times and accelerates delivery of other OpenTok events.
Reduced audio/video latency	WebRTC offers significant improvements in latency through WebRTC, enabling more natural and effortless conversations.
Freedom from Flash	With WebRTC and JavaScript WebSockets, you no longer need to rely on Flash for browser-based RTC.
Native HTML5 elements	Customize the look and feel and work with video like you would any other element on a web page with the new video tag in HTML5.

GetUserMedia

http://www.html5rocks.com/en/tutorials/webrtc/basics/

- navigator.getUserMedia()
  - Webcam and microphone input are accessed without a plugin.
- Checking if browser supports it

```
function hasGetUserMedia() {
  // !! converts a value to a boolean and ensures a boolean type.
  return !!(navigator.getUserMedia ||
            navigator.webkitGetUserMedia ||
            navigator.mozGetUserMedia ||
            navigator.msGetUserMedia);}


if (hasGetUserMedia()) {
  // Good to go!
} else {
  alert('getUserMedia() is not supported in your browser'
}
```

getUserMedia is now supported by most browsers. Use the OR to support older browsers

- The first parameter to getUserMedia() is an object specifying the details and requirements for each type of media you want to access.
- For example {video: true, audio: true} will access both video and audio

```
<video autoplay></video>

<script>
  var errorCallback = function(e) {
    console.log('Rejected!', e);
  };

  // Not showing vendor prefixes.
  navigator.getUserMedia({video: true, audio: true},
     function(localMediaStream) {
       var video = document.querySelector('video');
       video.src =
          window.URL.createObjectURL(localMediaStream);

 }, errorCallback);
</script>
```

HTML5 container for video

callback function returns a videostream

window.URL.createObjectURL(). This method creates a simple URL string which can be used to reference data stored in a Blob object

Video parameters

- Video and audio can have parameters
  - Instead of just indicating basic access to video
    - e.g. {video: true})
  - You can additionally require the stream to be HD

```
var hdConstraints = {
  video: {
    mandatory: {
      minWidth: 1280,
      minHeight: 720
    }
  }
};

navigator.getUserMedia(hdConstraints, successCallback,
errorCallback);
```

Choosing the camera

```
cameras = [];
function getSources(sourceInfos) {
    var fillCameras = 0;
    for (var i = 0; i !== sourceInfos.length; ++i) {
        var sourceInfo = sourceInfos[i];
        if (sourceInfo.kind === 'video') {
            var text = sourceInfo.label ||
                    'camera ' + (cameras.length + 1);
            cameraNames[fillCameras] = text;
            cameras[fillCameras++] = sourceInfo.id;
        } else if (sourceInfo.kind === 'audio') {
            audioSource = sourceInfo.id;
        }    }
    videoSource = cameras[cameras.length - 1];
}

MediaStreamTrack.getSources(getSources);
```

In phones the first camera is the front facing one.
The last camera is the back camera

choosing (2)

```
function sourceSelected(audioSource, videoSource) {
  var constraints = {
    audio: {
      optional: [{sourceId: audioSource}]
    },
    video: {
      optional: [{sourceId: videoSource}]
    }
};

navigator.getUserMedia(constraints,
        successCallback, errorCallback);
```

Taking snapshot via canvas

```
<video autoplay></video>
<img src="">
<canvas style="display:none;"></canvas>
<script>
  var video = document.querySelector('video');
  var canvas = document.querySelector('canvas');
  var ctx = canvas.getContext('2d');
  var localMediaStream = null;
  video.addEventListener('click', snapshot, false);
  navigator.getUserMedia({video: true}, function(stream) {
    video.src = window.URL.createObjectURL(stream);
    localMediaStream = stream;
  }, errorCallback);

  function snapshot() {
    if (localMediaStream) {
      ctx.drawImage(video, 0, 0);
      document.querySelector('img').src
        = canvas.toDataURL('image/png');
    }  }

</script>
```

event click on video

saving the local stream

creating png image
from localMediaStream

The University Of Sheffield.

Applying effects

```
<style>
video { background: rgba(255,255,255,0.5); border: 1px solid #ccc;}
.grayscale { +filter: grayscale(1);}
.sepia { +filter: sepia(1);}
.blur { +filter: blur(3px);}
</style>
<video autoplay></video>
<script>
var idx = 0;
var filters = ['grayscale', 'sepia', 'blur', 'brightness',
               'contrast', 'hue-rotate', 'hue-rotate2',
               'hue-rotate3', 'saturate', 'invert', ''];

function changeFilter(e) {
  var el = e.target;  el.className = '';
    // loop through filters.
  var effect = filters[idx++ % filters.length];
  if (effect) { el.classList.add(effect);  }}

document.querySelector('video').addEventListener(
    'click', changeFilter, false);
</script>
```

The University Of Sheffield.

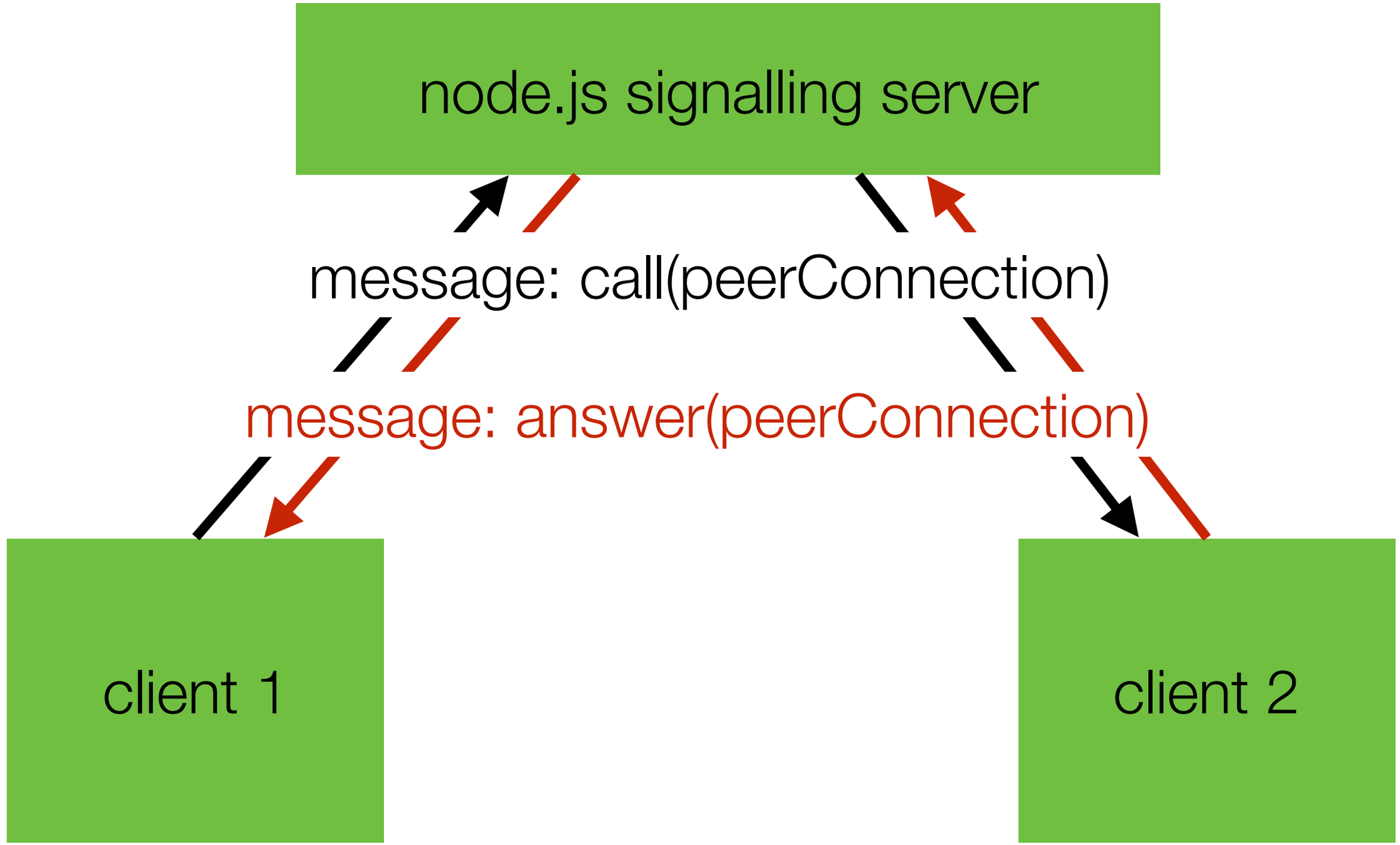
PeerConnection

- It creates the connection between the two clients via a signalling server
  - the role of the signalling servers is to pass the messages between the two clients via socket.io

```
function createPeerConnection() {
  try {
   pc = new webkitRTCPeerConnection(
           { "iceServers":
             [{ "url": "stun:stun.l.google.com:19302" });
  pc.onicecandidate = handleIceCandidate;
                                  callback when remote stream is added
  pc.onaddstream = handleRemoteStreamAdded;
                                   callback when remote stream is removed
  pc.onremovestream = handleRemoteStreamRemoved;

} catch (e) {
     alert('Cannot create RTCPeerConnection object.');
    return;  }}
```

Calling via socket.io



The University Of Sheffield.

Calling

on the initiating partner

```
peerConnection.createOffer(callRemote);
function callRemote(description) {
  peerConnection.setLocalDescription(description);
  socket.emit('message', description)
```

on the server, send it to the partner…

```
socket.on('message', function (message) {
        socket.to(room).emit('message', message);
        });
```

on the remote client…

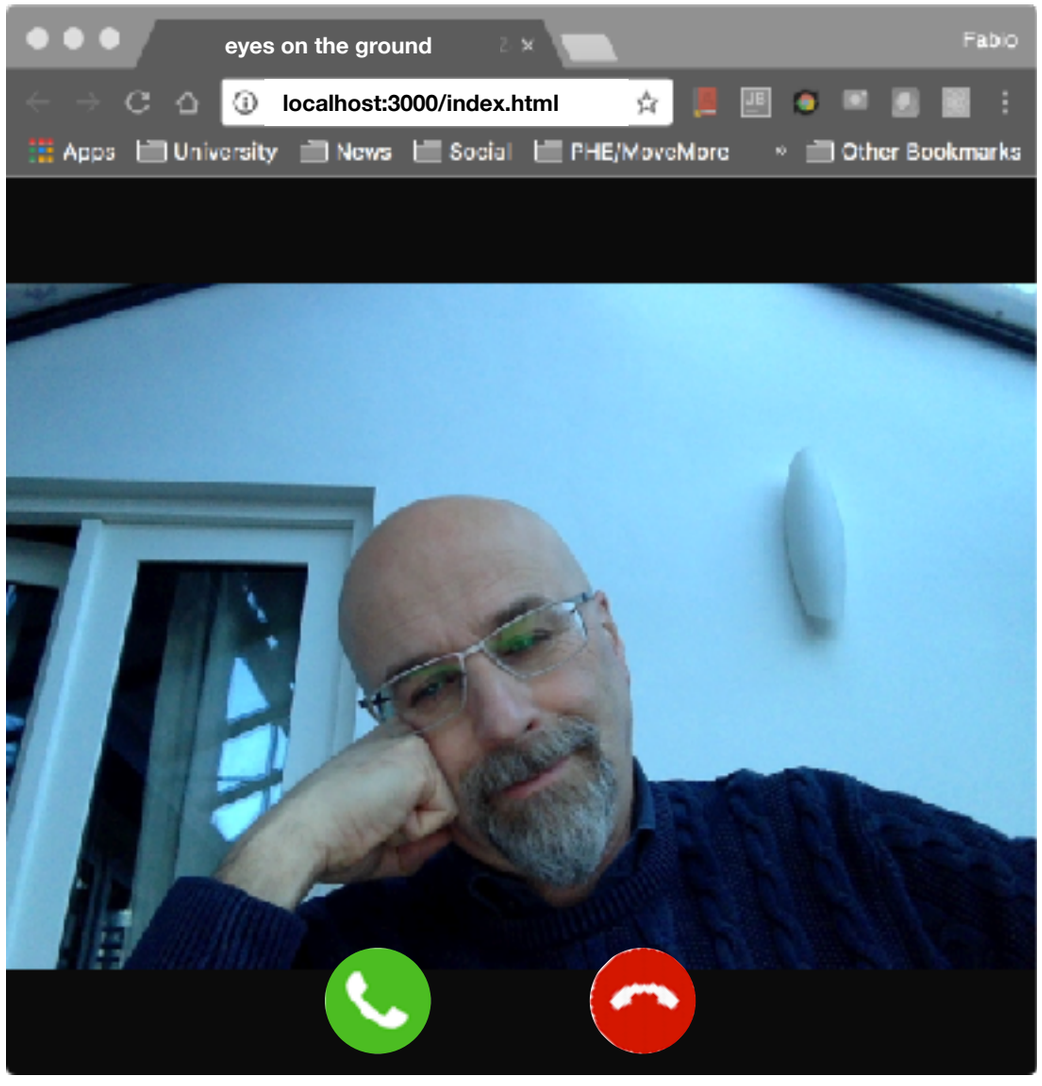
```
socket.on('message', function (message){
    if (message.type === 'offer') {
      if (!activeRTPConnection)
        peerConnection= createPeerConnection();
      peerConnection.createAnswer(answerRemote);
  });
function answerRemote(description) {
 peerConnection.setLocalDescription(description);
 socket.emit(‘message', description);}
```

Showing stream

- Receiving the connection description fires the event ‘pc.onaddstream’
  - Its callback function will receive as input the event that has caused the callback to be activated
  - the event has a stream field containing the userMedia stream (audio, video…) of the remote client
    - we create a blob and assign it as src of our HTML video element
    - now the remote stream is visible in our browser

```
function handleRemoteStreamAdded(event) {
  var remoteVideo= document.getElementById(‘remote_video’);
  remoteVideo.src = window.URL.createObjectURL(event.stream);
   remoteStream = event.stream;
}
```

Done!



A complete step by step guide can be found at

https://codelabs.developers.google.com/codelabs/webrtc-web/#0

Firewalls!

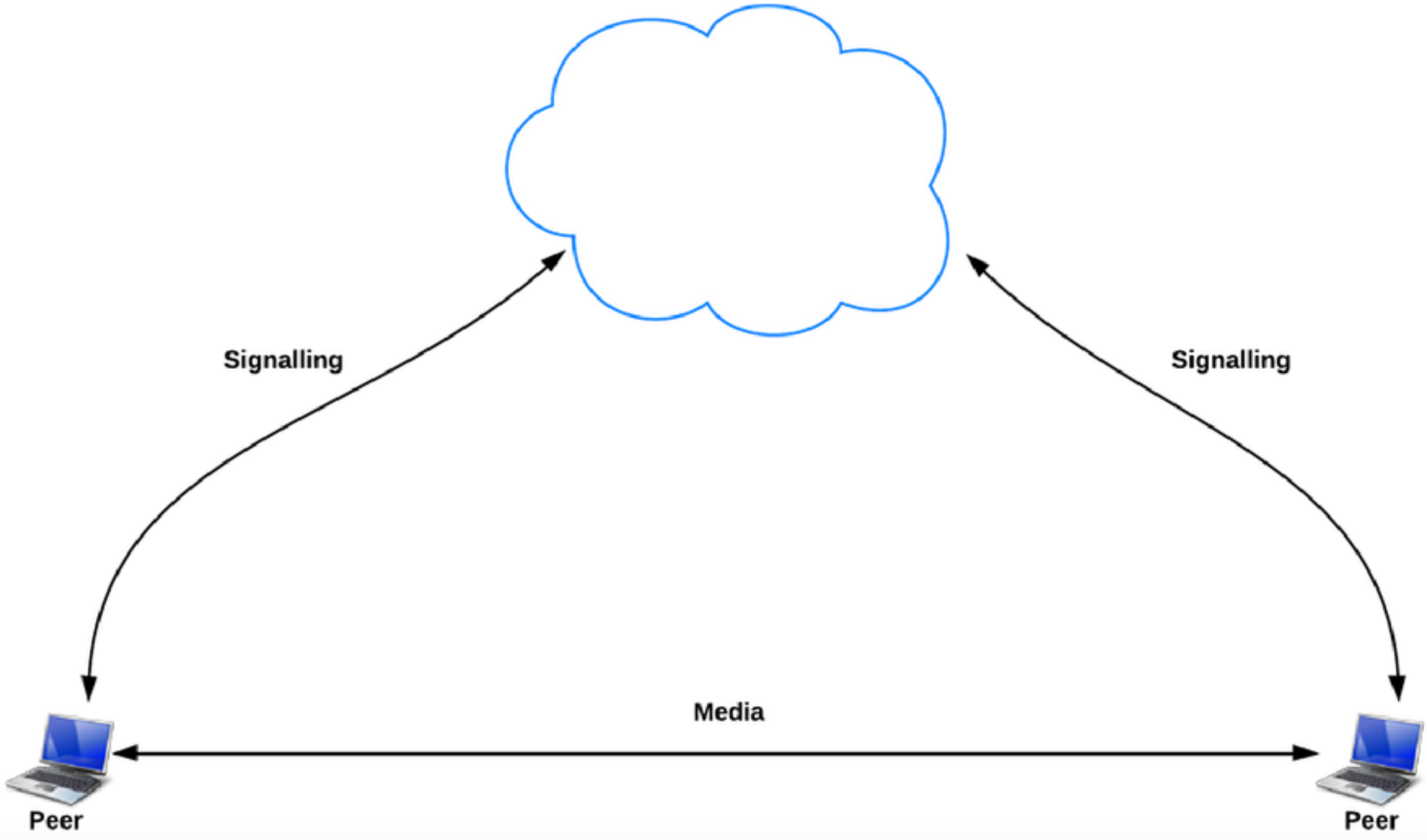
http://www.html5rocks.com/en/tutorials/webrtc/basics/#toc-signalling

- A realistic situation however prevents this from working because firewalls prevent clients to see each other directly
- WebRTC needs four types of server-side functionality:
  - User discovery and communication.
  - Signalling.
  - NAT/firewall traversal.
  - Relay servers in case peer-to-peer communication fails
- The STUN protocol and its extension TURN are used by the ICE framework to enable RTCPeerConnection to cope with NAT traversal

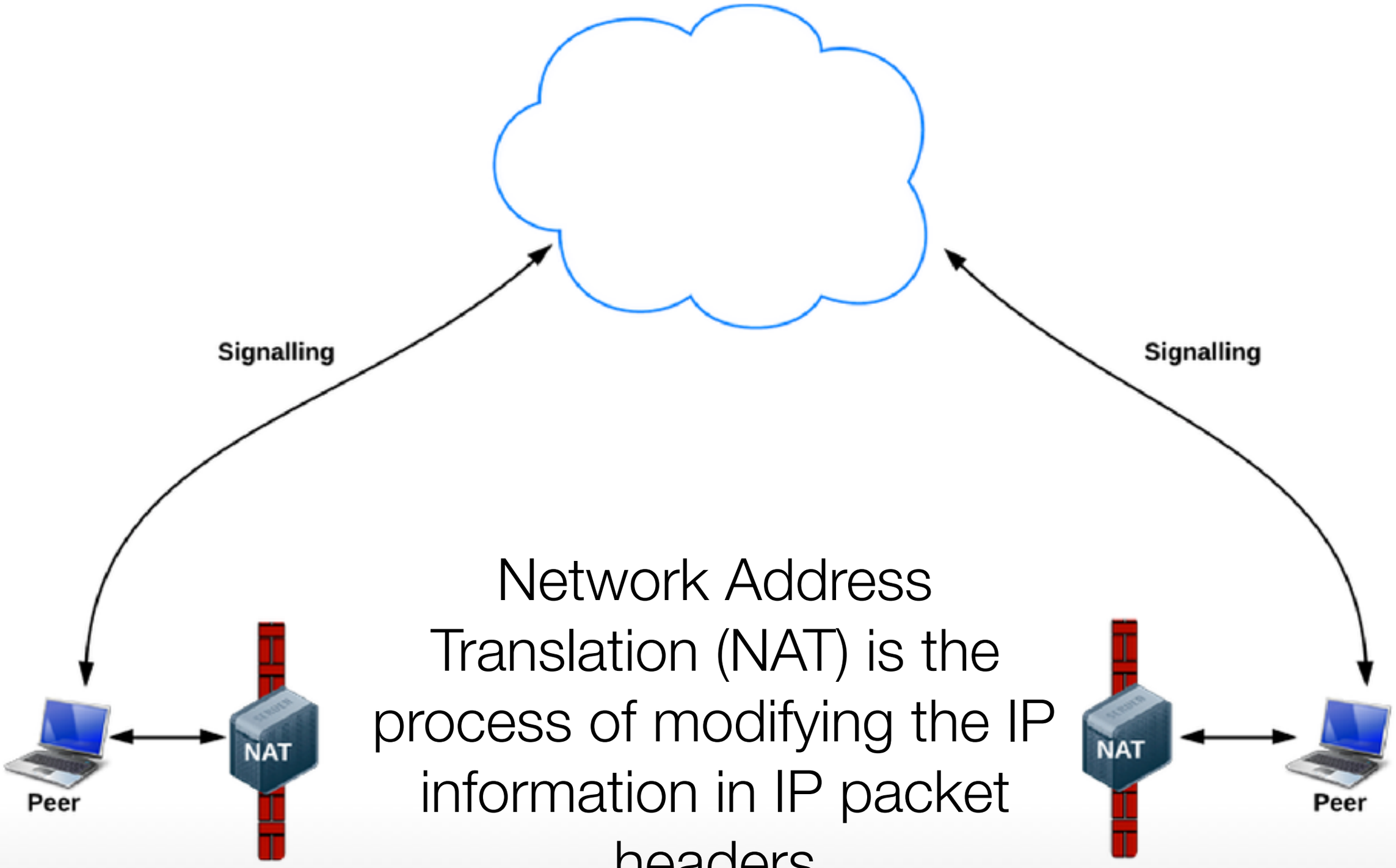
The University Of Sheffield.

An Ideal World

http://io13webrtc.appspot.com/#45



The Real World

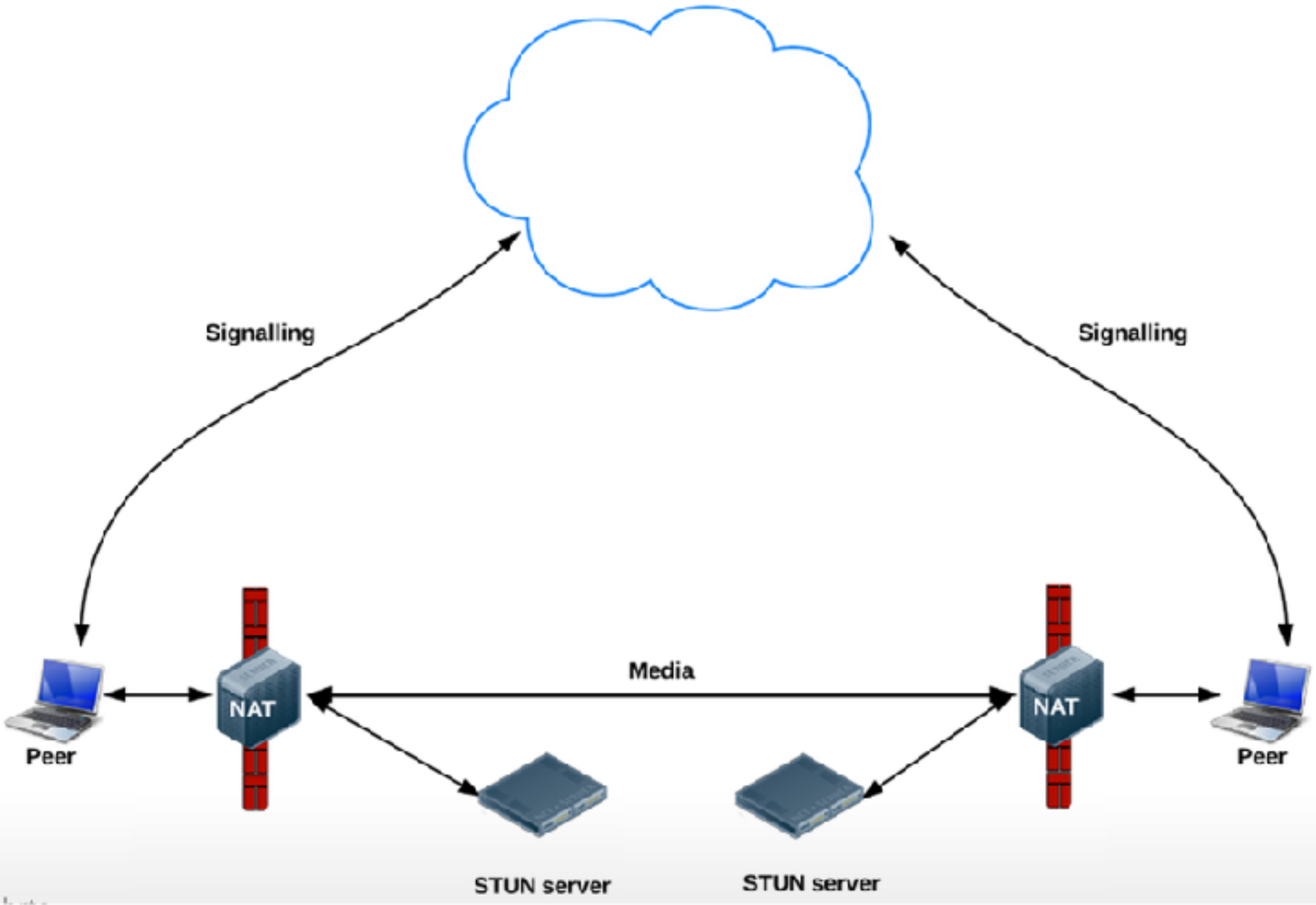


Network Address Translation (NAT) is the process of modifying the IP information in IP packet headers

https://www.bestvpn.com/blog/4246/what-is-a-nat-firewall/

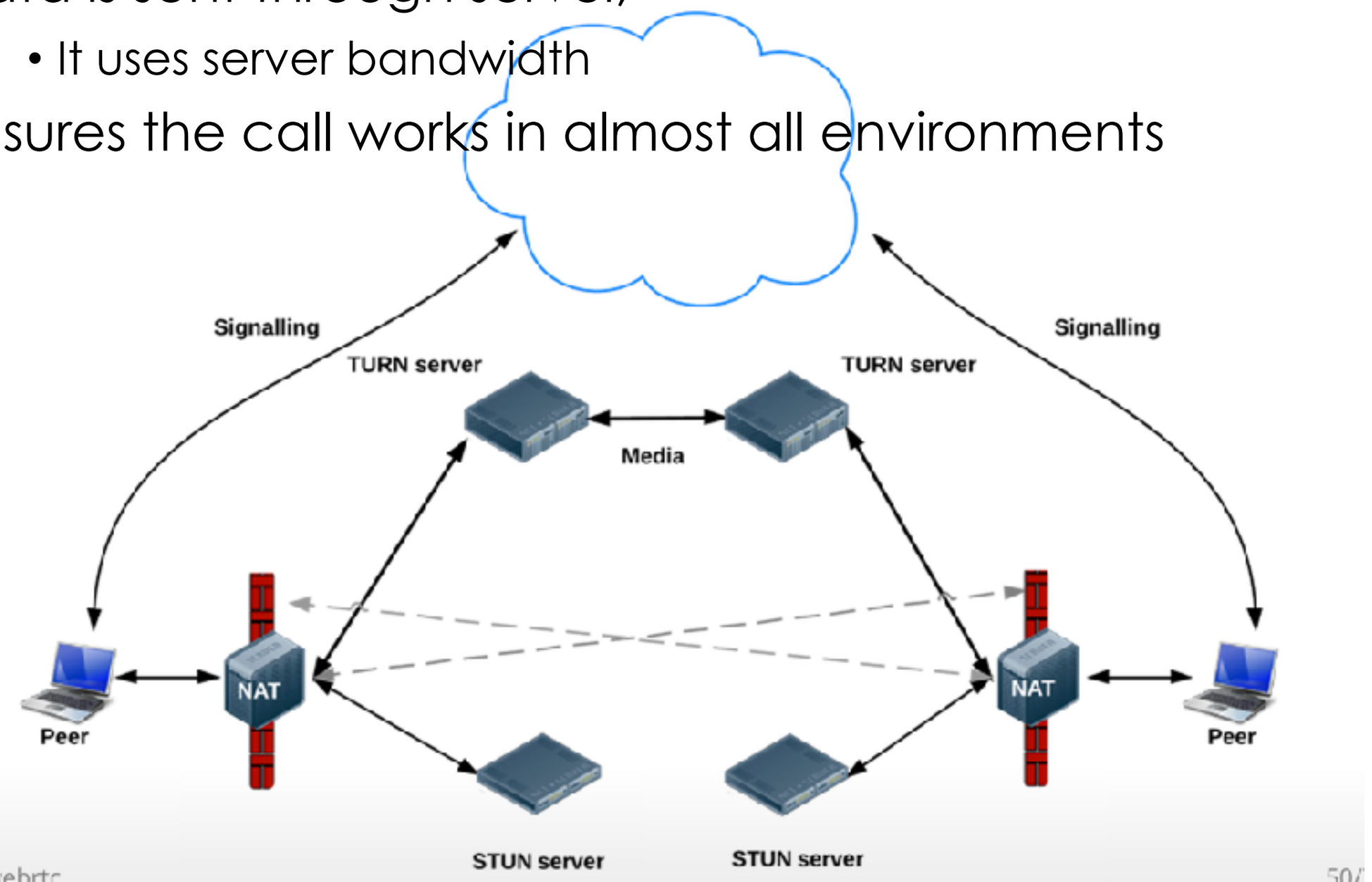
STUN Server

- Tell me what my public IP address is
  - Simple server, cheap to run
  - Data flows peer-to-peer



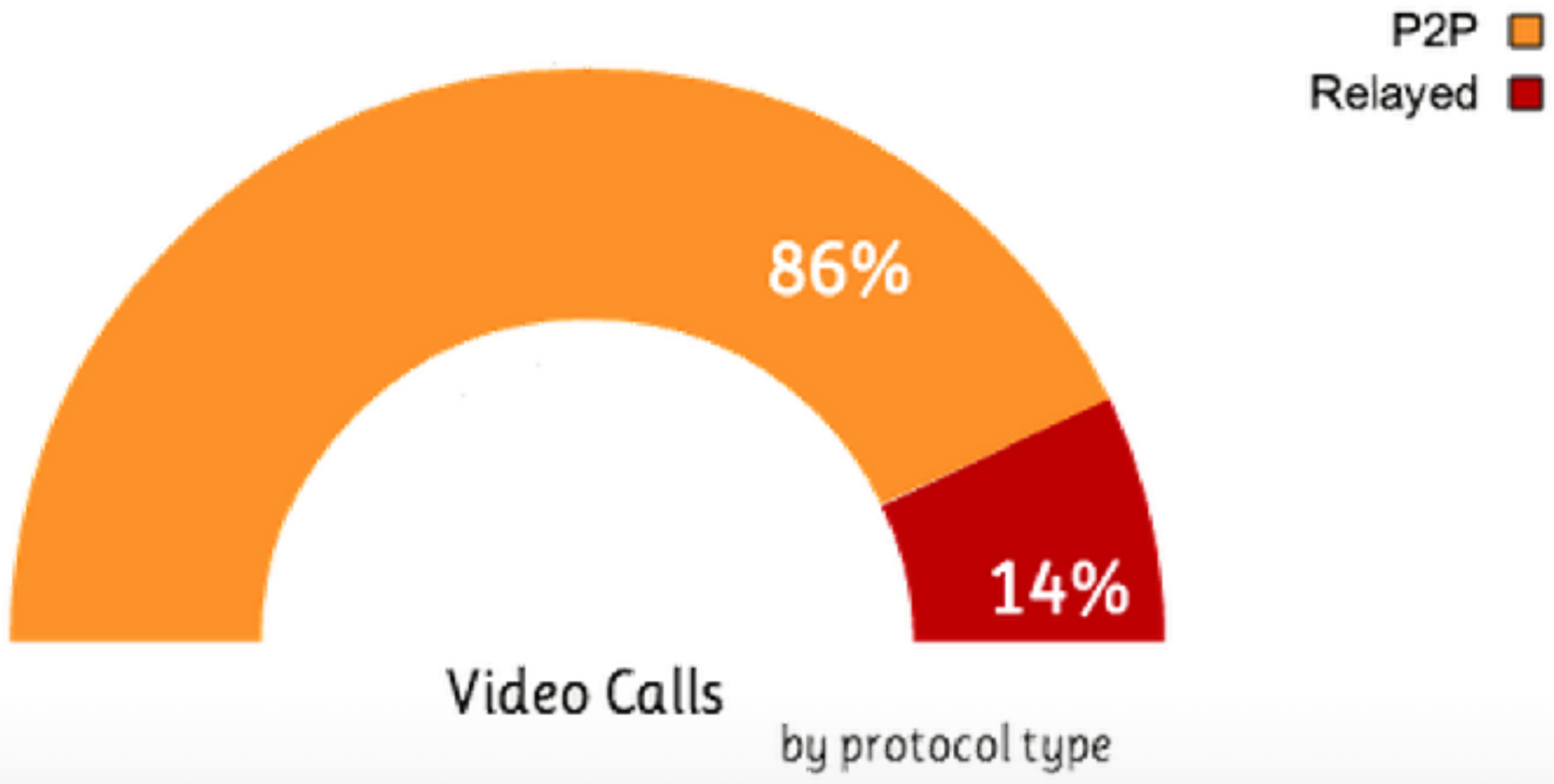
TURN Server

- Provide a cloud fallback if peer-to-peer communication fails
  - Data is sent through server,
    - It uses server bandwidth
  - Ensures the call works in almost all environments



ICE

- ICE: a framework for connecting peers
  - Tries to find the best path for each call
  - Vast majority of calls can use STUN (webrtcstats.com):



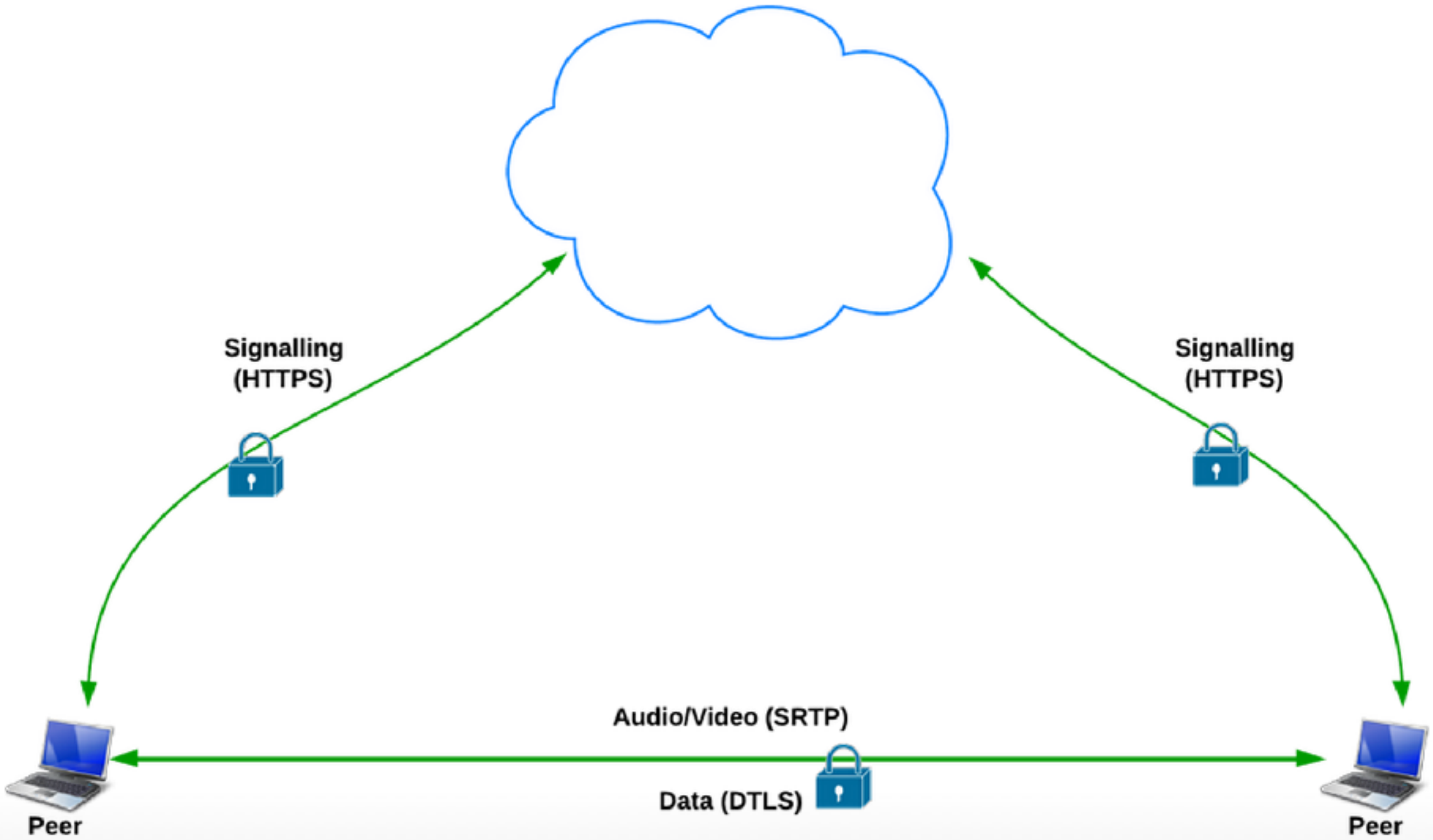
Declaring turn/stun servers

- In the RTCPeerConnection declaration

```
pc = new webkitRTCPeerConnection(
    { "iceServers":
       [
        { "url": "stun:stun.l.google.com:19302" },
        { "url":"turn:<<login>>@<<give TURN url>>>",
          "credential":<<password>>}],
          optional: [{RtpDataChannels: true}
       ]
    });
```

Always Use HTTPS

- it is just a small change in the node.js setting



Questions?

We will see applications of WebRTC in the next lecture!