

Comunicação cliente-servidor bilateral de baixa latência aplicado a Android

Guilherme Freire Silva

TRABALHO DE CONCLUSÃO DE CURSO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO

Orientador: Prof. Dr. Marco Dimas Gubitoso

São Paulo, 2016

Lista de Figuras

Lista de Tabelas

Sumário

1 Resumo

2 Abstract

3 Introdução

Dispositivos móveis, como smartphones e tablets, estão cada dia mais presentes. Sua tecnologia se atualiza a cada momento e ainda é muito nova, muitas de suas funcionalidades ainda podem ser bastante exploradas.

O mesmo vale para tecnologias para a internet. O ambiente Web está em constante transformação, APIs ¹ e ferramentas são criadas e renovadas a todo momento e podem ser aplicadas a vários contextos de forma relevante e inovadora.

Com esse contexto em mente, o Projeto apresentado foi concebido para unir essas duas pontas de tecnologias em uma Aplicação Web que se comunica com um dispositivo móvel. O conceito básico é ter um Aplicativo de smartphone que se conecte a um servidor e passe a enviar e receber informações. O servidor lança uma página Web que recebe esses dados enviados e executa sua funcionalidade.

O conceito de “funcionalidade” é propositalmente deixado em aberto pois esse fluxo de dados encapsula uma grande variedade de funcionalidades. É possível, por exemplo, executar um jogo na página Web que utiliza o smartphone como um controle, ou manter um banco de dados no Servidor e fazer buscas utilizando o smartphone.

O Projeto tem como objetivo então abrir e esse canal de comunicação bilateral entre páginas Web e smartphones, além de facilitar o desenvolvimento de aplicações voltadas a essa estrutura.

¹*Application Programming Interface*, conjunto de rotinas e padrões de programação para acesso a um aplicativo de software ou plataforma baseado na Web

3.1 Objetivos

- Desenvolver um aplicativo para smartphones.
- Criar um servidor.
- Construir uma comunicação entre ambos.
- Encontrar uma forma eficiente de comunicação.
- Desenvolver uma página Web como Caso de Uso.

4 Desenvolvimento

Nessa seção, é apresentada a implementação do Projeto. O código será exibido e explicado, além das tecnologias necessárias.

4.1 Comunicação

A comunicação é a parte mais importante deste Projeto. O modelo cliente-servidor foi usado, mas foi necessário levar alguns pontos em consideração. O cliente de smartphone e o cliente Web precisam se comunicar com o servidor da melhor forma possível. O protocolo de comunicação é o fator que mais pode limitar as possibilidades previstas. Para não haver problemas, a comunicação precisa de:

- **Baixa latência.** No caso de o cliente Web executar um jogo, o tempo entre um comando enviado do smartphone e a reação da página precisa ser o menor possível. Caso contrário, a experiência é danificada e pode até ser impossibilitada, se o jogo for construído com base em respostas rápidas.
- **Baixo *overhead* de mensagens.** É necessário que o cabeçalho das mensagens seja o mínimo possível. Se o fluxo de mensagens for alto, ele vai consumir muita banda da conexão e recursos, tanto do servidor como do cliente. É necessário que possua a menor quantidade de informações desnecessárias. Caso a mensagem tenha muito conteúdo, ele fará pouca diferença, mas se ela for pequena, como um texto ou um conjunto de valores, o cabeçalho representa um aumento significativo no tamanho total, podendo ser até maior do que o próprio conteúdo.
- **Comunicação bilateral.** Informações podem se originar de qualquer um dos clientes ou do servidor, então é necessário que o processo em questão possa enviar as informações a qualquer momento. No caso de um cliente precisar ser atualizado caso haja dados novos, é inviável que precise consultar o servidor a todo momento.
- **Permitir múltiplas conexões simultâneas.** Não é necessário, mas permite uma liberdade ainda maior. No projeto é previsto somente que o servidor se conecte a um cliente Web e de smartphone, mas, se o servidor aceitar a conexão com vários clientes do mesmo tipo, o desenvolvimento deles pode tirar proveito disso. Um smartphone pode enviar e receber informações de outros, como em um *Chat*, por exemplo.

Com esses requerimentos em vista, a tecnologia usada foi o Framework de JavaScript Socket.io, que implementa o protocolo WebSocket.

4.1.1 Protocolo WebSocket

WebSocket é um protocolo que permite a comunicação bilateral entre cliente e servidor. Para o protocolo, não existe essa diferenciação entre os dois, o cliente e o servidor são apenas dois processos conectados. A comunicação é dada por envio de mensagens e ocorrência de eventos. Esses processos podem enviar mensagens a qualquer momento e estão ouvindo um ao outro.

A estrutura da mensagem em WebSocket é mínima. O corpo da mensagem, os dados a serem enviados, só possui dois formatos, texto ou binário. O cabeçalho contém um identificador de tipo do formato, que tem um campo para o tamanho da mensagem e nada mais. A mensagem completa é só o conteúdo mais seu tamanho. Como a conexão não muda em momento algum, todas as informações contidas no cabeçalho de requisição e resposta HTTP são invariantes e, portanto, não precisam ser reenviadas.

Um dos processos conectados, em um momento arbitrário, envia uma mensagem ao outro, talvez porque dados tenham sido atualizados ou o usuário tenha feito alguma ação em específico. Uma vez que a mensagem é enviada, esse processo volta ao que estava fazendo, sem a necessidade de esperar alguma resposta do outro processo. Esse comportamento é chamado de assíncrono, pois não um lado não age em sincronia com o outro.

O outro processo recebe a nova mensagem através de um evento. Quando esse evento acontece, ele executa um comportamento específico (*callback*) com os dados recebidos, como alteração de informações exibidos ou um cálculo com os valores novos. Por isso é dito que WebSocket é Orientado a Eventos.

4.1.2 Socket.io

Socket.io é a API de WebSocket para JavaScript utilizada. Além da implementação básica do protocolo, ele também adiciona outras funcionalidades importantes para o desenvolvimento de uma aplicação mais robusta. As que foram utilizadas no Projeto foram:

- A possibilidade de conectar múltiplos sockets em uma mesma porta. Utilizando WebSockets, cada socket precisa de uma porta exclusiva para fazer a conexão.
- Criação arbitrária de eventos. Com WebSocket, o desenvolvedor fica preso a quatro eventos pré definidos: *onopen*, *onclose*, *onmessage* e *onerror*.

Essa criação de eventos funciona da seguinte forma: o processo emissor manda a mensagem e o nome do evento que está sendo enviado. O

processo receptor define o evento que vai ouvir e a subrotina que vai ser executada na ocorrência desse evento. Se o evento lançado for ouvido pelo receptor, o *callback* é chamado.

O uso de Socket.io está presente no código do servidor e dos clientes Android e Web. Todos esses processos escutam e enviam eventos entre si, e o servidor permite a conexão de múltiplos clientes.

4.2 Aplicativo Android

O aplicativo foi desenvolvido com a proposta de servir como base para o desenvolvimento de aplicativos maiores. Ele possui os elementos básicos propostos por esse Projeto. Ele permite as seguintes ações:

- Se conectar e desconectar de um servidor, utilizando o endereço IP e Porta.
- Enviar eventos pontuais ao servidor, com objetivos variados.
- Iniciar ou parar o envio contínuo de eventos, lançados em intervalos regulares. Os dados enviados são os valores do acelerômetro² do dispositivo.
- Exibe na tela mensagens enviadas do servidor.

Ele foi implementado usando o *framework* Cordova.

4.2.1 Cordova

O Apache Cordova é um *framework open-source* para a criação de aplicativos para *mobile*. Seu desenvolvimento é dado com o uso de tecnologias Web, como HTML5, CSS3 e JavaScript.

Ele é multiplataforma, permite o desenvolvimento para sistemas como Android, iOS ou navegadores. Oferece uma API de alto nível para acessar os módulos desejados, como de sensores, arquivos e rede.

O aplicativo implementado é uma página Web, possui exatamente a estrutura. Há um arquivo principal “index.html” que referencia os recursos necessários, como CSS, JavaScript, imagens e arquivos de mídia. A parte lógica é feita em JavaScript, e a renderização em HTML5 e CSS3. Para ser portado para uma plataforma específica, o HTML é enviado para a classe “*Wrapper*” dessa plataforma, onde estão definidos os detalhes de implementação. Essa classe também tem incorporada um browser nativo, o *WebView*, que executará o programa Web dentro do dispositivo.

4.2.2 Implementação

O aplicativo implementado possui o básico de interação, conecta-se com o servidor, exibe informações recebidas na tela, utiliza alguns botões para mandar informações e faz o uso do acelerômetro do dispositivo.

Essa é uma imagem de sua interface:

²Sensor que mede a vibração ou a aceleração do movimento do dispositivo.



CONNECTED

DEVICE IS READY

IP:

192.168.0.17

PORT:

8000

Connect

LOG:

[Ping]

[Calc]

X

Y

Z

Stop

60 fps

30 fps

10 fps

1 fps

As interações da interface são:

- Um formulário para fazer a conexão com o servidor, dados IP e porta. Há um feedback visual que indica se ela foi bem sucedida.
 - Uma vez conectado, o botão serve para fechar a conexão.
- Um campo “Log” que exibe o resultado de algumas interações do usuário, ou mensagens enviadas do servidor.
- Um conjunto de botões que emitem ações pontuais ao servidor, como calcular o *Ping* ³ ou enviar um conjunto de dados.
- Um outro conjunto de botões que ligam ou desligam um fluxo contínuo de mensagens. Essas mensagens emitidas contêm os dados do acelerômetro do aparelho.
 - Ao serem pressionados, esses botões iniciam o fluxo de mensagens. Se pressionados novamente, o fluxo é interrompido.

Essas funcionalidades implementadas já cobrem muitas interações desejadas no desenvolvimento de qualquer tipo de aplicativo.

4.2.3 Código da Interface

A parte visual do aplicativo está no arquivo index.html:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <!-- <meta http-equiv="Content-Security-Policy" content="
5        default-src 'self' data: gap: https://ssl.gstatic.com '
6        unsafe-eval'; style-src 'self' 'unsafe-inline'; media-src *;
7        script-src 'self' http://cdn.socket.io/socket.io-1.0.3.js '
8        unsafe-inline' 'unsafe-eval'; connect-src localhost:*; ">
9        -->
10
11      <meta name="format-detection" content="telephone=no">
12      <meta name="msapplication-tap-highlight" content="no">
13      <meta name="viewport" content="user-scalable=no, initial-scale=1
14        , maximum-scale=1, minimum-scale=1, width=device-width">
15      <link rel="stylesheet" type="text/css" href="css/index.css">
16      <link rel="shortcut icon" href="">
17      <title>TCC</title>
18    </head>
19    <body>
20      <div class="app">
```

³Tempo necessário para uma mensagem ser enviada ao servidor e voltar dele.

```

17 <h1 id="status">Disconnected</h1>
18 <div id="deviceready" class="blink">
19   <p class="event listening">Loading Device</p>
20   <p class="event received">Device is Ready</p>
21 </div>
22
23 <form id="connect">
24   IP:<br>
25   <input type="text" name="ip" value="192.168.0.17">
26   <br>
27   Port:<br>
28   <input type="text" name="port" value="8000">
29   <br><br>
30   <input id="connect" type="submit" value="Connect">
31 </form>
32
33 <p id="log">log:</p>
34
35 <button class="btn" onclick="sendPing()">[Ping]</button>
36 <button class="btn" onclick="localCalculation()">[Calc]</
   button>
37 <button class="btn" onclick="addSpeed(speedX)">X</button>
38 <button class="btn" onclick="addSpeed(speedY)">Y</button>
39 <button class="btn" onclick="addSpeed(speedZ)">Z</button>
40 <button class="btn" onclick="resetSpeed()">Stop</button>
41
42 <button class="btn2" onclick="toggleAccelerometer(60)">60
   fps</button>
43 <button class="btn2" onclick="toggleAccelerometer(30)">30
   fps</button>
44 <button class="btn2" onclick="toggleAccelerometer(10)">10
   fps</button>
45 <button class="btn2" onclick="toggleAccelerometer(1)">1 fps<
   /button>
46
47 </div>
48 <script type="text/javascript" src="cordova.js"></script>
49 <script type="text/javascript" src="http://code.jquery.com/
   jquery-1.11.1.js"></script>
50 <script type="text/javascript" src="http://cdn.socket.io/socket.
   io-1.0.3.js"></script>
51 <script type="text/javascript" src="js/index.js"></script>
52 </body>
53 </html>

```

- A divisão “deviceready” (linha 18) exibe o estado da conexão.
 - Caso o dispositivo não tenha se conectado ainda, ele exibe “Connect Device”.
 - Ao obter uma conexão bem sucedida com o servidor, ele muda para “Device is Connected”.
- O formulário “connect” (linha 23) faz a conexão do dispositivo com o servidor.
 - Ele possui um campo para o IP e um para a Porta do servidor.

- Ao apertar o botão “Connect”, ele tenta fazer a conexão.
- O parágrafo “log” (linha 33) exibe dados vindos do servidor.
- Os botões (linha 35 a 45) executam diversas funções do código de enviar através de eventos para o servidor.
- Os scripts (linha 48 a 51) importam as bibliotecas cordova, jquery e socket.io, e a parte lógica do aplicativo, encontrado em “js/index.js”.

4.2.4 Código da parte lógica

A parte lógica do aplicativo está no arquivo index.js:

```

1  var app = {
2      initialize: function() {
3          this.bindEvents();
4      },
5      bindEvents: function() {
6          document.addEventListener('deviceready', this.onDeviceReady,
7                                  false);
8      },
9      onDeviceReady: function() {
10         app.receiveEvent('deviceready');
11     },
12     receiveEvent: function(id) {
13         var parentElement = document.getElementById(id);
14         var listeningElement = parentElement.querySelector('.listening');
15         var receivedElement = parentElement.querySelector('.received');
16
17         listeningElement.setAttribute('style', 'display:none;');
18         receivedElement.setAttribute('style', 'display:block;');
19
20         console.log('Received Event: ' + id);
21     }
22 };
23 app.initialize();
24
25 /*Variables */
26 var socket;
27 var watchID = null;
28
29 var speedX = {'x':1,'y':0,'z':0};
30 var speedY = {'x':0,'y':1,'z':0};
31 var speedZ = {'x':0,'y':0,'z':1};
32
33
34
35 /* Auxiliar functions */
36 function log(message) {
37     socket.emit('device_print', message);
38     var msg = 'Log: ' + message;
39     $('#log').text(msg);
40 }
41

```

```

42 function validateIPAddress(ipaddress) {
43     if (/^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
44         \.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
45         \.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
46         \.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/).test(ipaddress)) {
47         // if (/^192\.168?\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
48         \.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/).test(ipaddress)) {
49             return (true)
50         }
51         log("You have entered an invalid IP address!")
52         return (false)
53     }
54 }
55
56 function toggleAccelerometer(period) {
57     var freq = ~(1000 / period);
58     if (watchID == null) {
59         watchID = navigator.accelerometer.watchAcceleration(
60             sendAcceleration,
61             sendAccError,
62             { frequency: freq }
63         );
64         navigator.accelerometer.getCurrentAcceleration(sendAcceleration,
65             sendAccError);
66     }
67     else {
68         navigator.accelerometer.clearWatch(watchID);
69         watchID = null;
70     }
71 }
72
73 /* Socket emit functions */
74 function addSpeed(speed_vector) {
75     socket.emit('device_add_speed', speed_vector);
76 }
77
78 function resetSpeed() {
79     socket.emit('device_reset_speed');
80 }
81
82 function sendAcceleration(acceleration) {
83     socket.emit('device_acceleration', acceleration)
84 }
85
86 function sendAccError() {
87     log('Accelerometer error.');

```

```

99     }
100   }
101
102   var t1 = Date.now();
103   socket.emit('request_calculation');
104   log('Finished device calculation in ' + (t1 - t0));
105 }
106
107
108 /*Socket events */
109 function onServerConnect() {
110   $('#status').text("Connected");
111   app.receivedEvent('deviceready');
112   socket.emit('ready', 'device');
113 }
114
115 function onServerPing(data) {
116   var dt = Date.now() - data.t0;
117   log('Ping: ' + dt);
118 }
119
120 function onServerLog(msg) {
121   $('#log').text(msg);
122 }
123
124 function connectSocket(ipaddress, port) {
125   if(validateIPAddress(ipaddress)) {
126     if(socket !== null && socket !== undefined) {
127       socket.emit('device_disconnected');
128       socket.disconnect();
129       $('#status').text("Disconnected");
130     }
131     else{
132       socket = io('http://' + ipaddress + ':' + port);
133
134       socket.on('connect', onServerConnect);
135       socket.on('server_ping', onServerPing);
136       socket.on('device_log', onServerLog);
137     }
138   }
139 }
140
141
142 /*Device Ready Event */
143 document.addEventListener('deviceready', function() {
144
145   $('#connect').submit(function (event) {
146     var ip = this.ip.value;
147     var port = this.port.value;
148
149     connectSocket(ip, port);
150
151     return false;
152   });
153 });

```

Para inicializar o aplicativo:

- A variável `app` é uma classe padrão do Cordova com funções para inicializar o aplicativo. Ele é inicializado na linha 23.

Para fazer a conexão com o servidor:

- Na linha 143, uma função é adicionada ao evento *“deviceready”* que o Cordova emite quando toda a inicialização é finalizada. Na ocorrência desse evento, a ação do botão “Connect” é alterada para tentar conectar-se ao servidor, através da função `connectSocket`.

Essa função `connectSocket` recebe o IP e Porta vindos do formulário do botão. Se o IP for válido (a função `validateIPAddress` faz essa verificação com expressões regulares), ela faz uma conexão entre o dispositivo e o servidor definido por esses IP e Porta. Se a conexão já está feita, ele a desconecta.

Eventos que o servidor ativa:

- Na inicialização do Socket, três eventos são definidos: *“connect”* chama a função `onServerConnect`; *“server_ping”* chama `onServerPing` e *“device_log”* chama `onServerLog`.
- `onServerConnect` : Emite ao socket um evento *“ready”* para o servidor para informar o seu tipo e atualiza informações exibidas.
- `onServerPing`: Calcula o *ping* que foi enviado.
- `onServerLog`: Exibe na tela a mensagem recebida, no campo *LOG*.

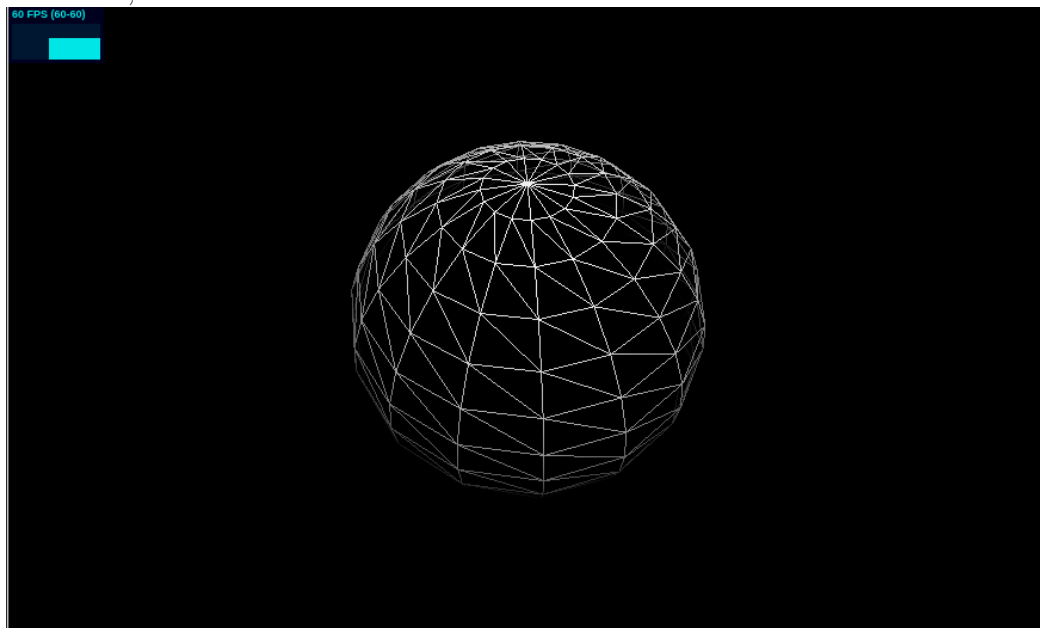
Eventos enviados ao servidor:

- As funções `addSpeed`, `resetSpeed`, `sendAcceleration`, `sendPing` e `localCalculation` enviam mensagens ao servidor com dados obtidos do input do usuário ou leitura do smartphone.
- A função `toggleAccelerometer` ativa um loop que manda leituras do acelerômetro em intervalos regulares.

4.3 Aplicação Web como Caso de Uso

A Aplicação Web criada para exemplificar o uso do sistema proposto pelo Projeto foi projetada com a ideia de servir de base para o desenvolvimento de um jogo.

A principal decisão de design foi fazer o *browser* executar uma computação gráfica. Ele renderiza uma esfera no centro da tela e seu socket escuta por eventos de movimentação. Ao receber esses eventos, as velocidades X, Y e Z de rotação dessa esfera são alterados. Um dos eventos adiciona valores à velocidade atual. O outro define a velocidade total. Além de alterar a velocidade, esses valores recebidos também alteram a cor da esfera.



Para a renderização 3D, foi utilizada a biblioteca THREE.JS.

4.3.1 THREE.JS

THREE.JS é uma biblioteca de JavaScript para a criação e execução de gráficos 3D em navegadores, de maneira leve e otimizada. Ele permite ao *browser* renderizar gráficos sem a necessidade de plugins, e utilizar a GPU⁴ do usuário para os cálculos de física e processamento de imagens. Essas funcionalidades são herdadas de WebGL, uma outra API de computação gráfica em JavaScript para *browsers*.

⁴GPU (*Graphics Processing Unit*) é um microprocessador especializado em processar gráficos. Sua estrutura de processamento paralelo os tornam mais capazes neste tipo de trabalho que CPUs normais. Uma GPU normalmente é utilizada em placas de vídeo.

Sua interface é simples de se usar, ela lida internamente com os detalhes técnicos de implementação de baixo nível, deixando o desenvolvimento menos carregado, uma vez que processamento gráfico é uma área complexa da computação.

THREE.JS é completa o bastante para a criação de cenários 3D completos, com objetos, luzes e texturas. Por utilizar a GPU para o processamento, a performance não é comprometida por suas funcionalidades.

Abaixo está uma lista de funcionalidades que possui. Por possuir muitos termos inglês sem tradução em português, a lista de forma literal da sua página⁵ na Wikipédia:

- **Effects:** Anaglyph, cross-eyed and parallax barrier.
- **Scenes:** add and remove objects at run-time; fog
- **Cameras:** perspective and orthographic; controllers: trackball, FPS, path and more
- **Animation:** armatures, forward kinematics, inverse kinematics, morph and keyframe
- **Lights:** ambient, direction, point and spot lights; shadows: cast and receive
- **Materials:** Lambert, Phong, smooth shading, textures and more
- **Shaders:** access to full OpenGL Shading Language (GLSL) capabilities: lens flare, depth pass and extensive post-processing library
- **Objects:** meshes, particles, sprites, lines, ribbons, bones and more - all with Level of detail
- **Geometry:** plane, cube, sphere, torus, 3D text and more; modifiers: lathe, extrude and tube
- **Data** loaders: binary, image, JSON and scene
- **Utilities:** full set of time and 3D math functions including frustum, matrix, quaternion, UVs and more
- **Export** and import: utilities to create Three.js-compatible JSON files from within: Blender, openCTM, FBX, Max, and OBJ

⁵<https://en.wikipedia.org/wiki/Three.js>

- **Support:** API documentation is under construction, public forum and wiki in full operation
- **Examples:** Over 150 files of coding examples plus fonts, models, textures, sounds and other support files
- **Debugging:** Stats.js, WebGL Inspector, Three.js Inspector

4.3.2 Implementação

Esse projeto explora somente o básico de THREE.js para a criação de um ambiente 3D.

Este código é mais extenso, então somente as partes principais são exibidas.

O primeiro arquivo, `global.js`, declara variáveis globais, usadas em mais de um módulo:

```
1 var socket;
2
3 var xRot, yRot, zRot;
4 var r, g, b;
5 var hasNewColors;
```

As variáveis declaradas definem o seguinte:

- O socket da conexão;
- As velocidades de rotação `xRot`, `yRot` e `zRot`;
- A cor RGB da esfera;
- Uma *flag* que aponta se a cor precisa ser atualizada.

O arquivo `socket_events.js`, define os eventos escutados e seus *callbacks*:

```
1 socket = io();
2
3 function rotate (data) {
4     xRot += data.x;
5     yRot += data.y;
6     zRot += data.z;
7 }
8
9 function reset_rotation () {
10    xRot = 0;
11    yRot = 0;
12    zRot = 0;
13 }
14
15 function update_rotation_and_colors (data) {
16    var cap = 9;
```

```

17     var x = Math.min(data.x, cap);
18     var y = Math.min(data.y, cap);
19     var z = Math.max(2, Math.min(data.z, cap));
20
21     var weight = 5;
22
23     xRot = y / 50;
24     yRot = x / 50;
25
26     r = (weight * r + (z / cap)) / (weight + 1);
27     g = (weight * g + (z / cap)) / (weight + 1);
28     b = (weight * b + (z / cap)) / (weight + 1);
29
30     hasNewColors = true;
31 }
32
33 // Socket Events
34
35 socket.on('connect', function() {
36     socket.emit('ready', 'browser');
37 });
38
39 socket.on('move', function (data) {
40     rotate(data);
41 });
42
43 socket.on('reset', function () {
44     reset_rotation();
45 });
46
47 socket.on('update_acceleration', function (data) {
48     update_rotation_and_colors(data);
49 })

```

Há 4 eventos de Sockets:

- “*connect*”: Ao se conectar, emite um evento “*ready*” com seu tipo.
- “*move*”: Recebe valores X, Y e Z, e os associa à velocidade de rotação da esfera em cada um desses eixos. Essa associação é incremental, a velocidade aumenta a cada evento.
- “*reset*”: Zera as velocidades de rotação da esfera.
- “*update_acceleration*”: Recebe valores X, Y, Z que definem a velocidade de rotação total da esfera (não incremental). Também altera sua cor incrementalmente, de acordo com alguns cálculos com as variáveis.

Por fim, o arquivo `three_sphere.js` cria todos os elementos gráficos e os renderiza.

```

1 var scene, camera, renderer;
2 var sphere, sphere;
3
4 hasNewColors = false;
5

```

```

6  function updateSphereRotation(xRot, yRot, zRot) {
7      sphere.rotation.x += xRot;
8      sphere.rotation.y += yRot;
9      sphere.rotation.z += zRot;
10 }
11
12 function updateSphereColor(r, g, b) {
13     sphere.material.color.r = r;
14     sphere.material.color.g = g;
15     sphere.material.color.b = b;
16 }
17
18
19 function init() {
20
21     scene = new THREE.Scene();
22
23     var view_angle = window.innerWidth/window.innerHeight
24     camera = new THREE.PerspectiveCamera( 75, view_angle, 0.1, 1000 );
25     camera.position.z = 5;
26     scene.add(camera);
27
28     renderer = new THREE.WebGLRenderer();
29     renderer.setSize( window.innerWidth, window.innerHeight );
30     document.body.appendChild( renderer.domElement );
31
32     //-----
33
34     // create a point light
35     var pointLight = new THREE.PointLight(0xFFFFFF);
36
37     // set its position
38     pointLight.position.x = 10;
39     pointLight.position.y = 50;
40     pointLight.position.z = 130;
41
42     // add to the scene
43     scene.add(pointLight);
44
45     //-----
46
47     // New Sphere
48     var radius = 2;
49     var segments = 16;
50     var rings = 16;
51
52     var sphereGeo = new THREE.SphereGeometry(radius, segments, rings);
53     var sphereMaterial = new THREE.MeshLambertMaterial( {
54         color: 0xCC0000,
55         wireframe : true
56     } );
57     sphere = new THREE.Mesh( sphereGeo, sphereMaterial );
58     scene.add( sphere );
59
60     //-----
61
62     xRot = 0;
63     yRot = 0;
64     zRot = 0;
65
66     r = 1;
67     g = 1;

```

```

68     b = 0;
69 }
70
71 function render() {
72     requestAnimationFrame( render );
73
74     updateSphereRotation(xRot, yRot, zRot);
75     if (hasNewColors) {
76         updateSphereColor(r, g, b);
77         hasNewColors = false;
78     };
79
80     renderer.render(scene, camera);
81 };
82
83 init();
84 render();

```

- `init()` cria os elementos do cenário: a cena, a camera, o renderizador, a iluminação e a esfera, além de inicializar as variáveis.
- `render()` atualiza a posição da esfera de acordo com sua velocidade (com `updateSphereRotation()`), sua cor (com `updateSphereColor()`), caso haja mudanças, e renderiza a cena. Essa função é chamada a cada *frame*, o que garante que a esfera seja animada.

4.4 Servidor

O servidor foi projetado para escutar eventos dos clientes Web e Android, e responder ou encaminhar os dados recebidos. Ele é implementado com Javascript, com o auxílio de Node.js e ExpressJS.

4.4.1 Node.js e ExpressJS para criar um servidor

Node.js é uma plataforma para desenvolvimento de aplicações *server-side* (executadas no servidor) baseadas em rede utilizando JavaScript e o V8 JavaScript Engine ⁶. Com Node.js é possível criar aplicações Web utilizando apenas código em JavaScript.

Ao se levar em conta o modo em que o código em JavaScript pode ser estruturado e as demandas de aplicações Web, Node.js abre uma gama de novas possibilidades para desenvolvimento Web.

Express é um *framework* mínimo e flexível para Node.js que oferece um conjunto robusto de funcionalidades. Sua API é simples, mas oferece o conjunto de ferramentas essenciais para o desenvolvimento de para páginas e aplicações Web e Mobile, e programas de *backend*. Ela lida com detalhes de baixo nível como protocolos e processos. Além da sua funcionalidade básica, existem módulos disponíveis para adicionar novas, que são diretamente aplicados ao Express.

Para essa implementação, Node.js e Express trabalham em conjunto para criar um servidor local e lançar a página da Aplicação Web desenvolvida.

4.4.2 Implementação

O servidor criado para o Projeto tem suas configurações definidas no arquivo `package.json`. Esse arquivo permite a restauração e reprodução do servidor em outros locais.

```
1 {  
2   "name": "socket.io-server_aplication",  
3   "version": "0.9.0",  
4   "description": "A client manager using socket.io",  
5   "main": "index.js",  
6   "author": "Guilherme Freire Silva",  
7   "private": true,  
8   "license": "BSD",  
9   "dependencies": {  
10    "express": "4.13.4",  
11    "socket.io": "^1.4.8"  
12  }  
13 }
```

⁶É o interpretador de JavaScript *open source* implementado pelo Google em C++ e utilizado pelo Chrome.

- A configuração mais importante para a reprodução é a de dependências, na qual estão definidas `express` e `socket.io`.

Para a leitura do código, é importante ter em mente que o servidor faz uma distinção entre sockets de *browser*, que executam a aplicação Web, e sockets de *device*, que executam o aplicativo do Cordova.

O código que cria o servidor está em `index.js`:

```

1 // Setup basic express server
2 var express = require('express');
3 var app     = express();
4 var server  = require('http').createServer(app);
5 var io      = require('socket.io')(server);
6 var port    = 8000;
7
8 server.listen(port, function () {
9     console.log('Server listening at port %d', port);
10 });
11
12 // Routing
13 app.use(express.static(__dirname + '/public'));
14
15
16
17 // -----
18
19 var device_dict = {};
20 var browser_dict = {};
21
22 // Function to send (event, data) only to browser sockets
23 function emit_to_browser_socket(event, data) {
24     for (var key in browser_dict) {
25         if (browser_dict[key].connected) {
26             browser_dict[key].emit(event, data);
27         }
28         else {
29             console.log(" > Disconnected socket!")
30             delete browser_dict[key];
31         }
32     }
33 }
34
35 io.on('connection', function (socket) {
36
37     socket.on('ready', function (data) {
38         if (data == 'device') {
39             console.log('device connected:', socket.id);
40             device_dict[socket.id] = socket;
41         }
42         else if (data == 'browser'){
43             console.log('browser connected:', socket.id);
44             browser_dict[socket.id] = socket;
45         }
46     })
47
48     // -----
49     // Device events
50
51     socket.on('device_ping', function (sent) {

```

```

52     var received_in_server = Date.now();
53     socket.emit('server_ping', {
54         t0 : sent,
55         t1 : received_in_server
56     });
57 });
58
59 socket.on('device_add_speed', function (data) {
60     console.log('device_add_speed', data);
61     emit_to_browser_socket('move', data)
62 });
63
64 socket.on('device_reset_speed', function () {
65     console.log('device_reset_speed');
66     socket.emit('device_log', 'server: reset speed');
67     emit_to_browser_socket('reset', {})
68 });
69
70 socket.on('device_print', function (data) {
71     console.log(data);
72 })
73
74 // Expected Structure: { x: -0.39, y: 0.06, z: 0.2, timestamp:
75 //                       1476064230489 }
76 socket.on('device_acceleration', function (data) {
77     emit_to_browser_socket('update_acceleration', data)
78 })
79
80 // -----
81 // Client events
82
83 // Echo browser > server > browser
84 socket.on('echo_move', function (data) {
85     socket.emit('move', data);
86 });
87 });

```

O servidor é criado utilizando o ExpressJS, ouvindo na porta definida e utilizando recursos encontrados na pasta `/public`.

Quando uma conexão é iniciada, o socket criado é atribuído de eventos e seus respectivos *callbacks*. Os eventos são:

- *'ready'*: Ocorre logo após a conexão ser iniciada. O cliente envia qual o seu tipo, e o servidor o insere em um dicionário de *devices* ou de *browsers*, indexado pelo *id* desse socket.
- *'device_ping'*: É a metade do caminho do *Ping* lançado pelo *device*. Essa função emite um evento *'server_ping'* de volta ao *device*, com o *timestamp* recebido e o atual.
- *'device_add_speed'*: Recebe um conjunto de valores X, Y e Z, e os emite para todos os *browsers*, com o evento *'move'*.

- *'device_reset_speed'*: Emite um evento *'device_log'* para o *device* logar na sua tela, e emite um evento *'reset'* aos *browsers* sem dados. Esse evento é usado para parar a rotação da esfera.
- *'device_print'*: É uma função de *debug*. Somente imprime no console os dados recebidos.
- *'device_acceleration'*: Recebe os dados do acelerômetro do *device* e os emite aos *browsers* com um evento *'update_acceleration'*.
- *'echo_move'*: Recebe do *browser* um valor de aceleração X, Y e Z, e emite de volta pelo evento *'move'*.
- Há uma função auxiliar `emit_to_browser_socket(event, data)` que emite o evento *'event'* com a mensagem *'data'* para todos os sockets de *'browsers'*.

5 Estudo Aprofundado da Tecnologia

5.1 Desenvolvendo Aplicativos com Apache Cordova

O Apache Cordova é um *framework open-source* para a criação de aplicativos para *mobile*. Seu desenvolvimento é dado com o uso de tecnologias Web, como HTML5, CSS3 e JavaScript.

Ele é multiplataforma, permite o desenvolvimento para sistemas como Android, iOS ou navegadores. Por tal motivo, oferece uma API de alto nível para acessar os módulos desejados, tais como de sensores, arquivos e rede. A interface para o uso dessas funcionalidades é abstrata, de forma que código desenvolvido é executável em todas as plataformas oferecidas e o desenvolvedor não precisa se preocupar com os detalhes de cada uma na hora da implementação.

O aplicativo implementado é uma exatamente página Web, a estrutura é a mesma. Há um arquivo principal “index.html” que referencia os recursos necessários, como CSS, JavaScript, imagens e arquivos de mídia. A parte lógica é feita em JavaScript, e a renderização em HTML5 e CSS3. Para ser portado para uma plataforma específica, o HTML é enviado para a classe “*Wrapper*” dessa plataforma, onde estão definidos os detalhes de implementação. Essa classe também tem incorporada um browser nativo, o *WebView*, que executará o programa Web dentro do dispositivo.

A comunicação entre o aplicativo e os componentes nativos de cada plataforma é dada a partir de Plugins instalados. Cada Plugin é uma biblioteca adicional que permite ao *WebView* interagir com funcionalidades da plataforma nativa na qual está rodando. Eles provêm acesso a essas funcionalidades normalmente não disponíveis em aplicativos Web. Essas ferramentas são disponibilizadas para o desenvolvedor através de uma expansão da API inicial, que toma para si os detalhes da implementação em cada plataforma, simplificando o desenvolvimento.

Há um conjunto principal de Plugins, chamado de *Core*, que é mantido pelo próprio Cordova. Nele estão contidos os que acessam as principais funcionalidades de um dispositivo mobile, como acesso ao acelerômetro, câmera, bateria e geolocalização.

Há também Plugins desenvolvidos por terceiros (em geral pela própria comunidade) que trazem relações com outras funcionalidades, talvez exclusivas de uma plataforma. Enquanto no *Core* há apenas umas poucas dezenas, a lista de Plugins criada pela comunidade oferece centenas com as mais diversas funcionalidades, como um para compras dentro do App e um para enviar notificações para dispositivos vestíveis (Smartwatches, por exemplo). A criação de um Plugin é simples e incentivada, no site do Cordova há um

tutorial.

O método de desenvolvimento descrito até agora, focado em várias plataformas, é um dentre dois possíveis *Workflows* disponíveis no Cordova e seu nome é “*Cross-platform*” (CLI). Ele deve ser usado se o aplicativo desenvolvido pretende alcançar a maior variedade de Sistemas Operacionais *mobile*, com pouca ou nenhuma ênfase no desenvolvimento em uma plataforma específica.

O segundo *Workflow* é chamado “*Platform-centered*”, e deve ser usado se o projeto é focado para uma única plataforma e se pretende modificá-la em baixo nível.

A arquitetura interna está descrita no diagrama:

5.2 Utilizando Node.js e ExpressJS

Something Something

Alguns atributos chave de Node.js são:

- Assíncrono e Orientado a Eventos - Todas as APIs da biblioteca de Node.js são assíncronas, o que garante que são não bloqueantes. Isso essencialmente significa que um servidor baseado em Node.js não espera por dados de retorno da chamada da API, ele somente continua com sua execução para a próxima chamada de API. Os dados de retorno chegam ao servidor através de um mecanismo de notificações, os Eventos de Node.js.

- Muito rápido - Por ser construído com base no V8 JavaScript Engine do Google Chrome, a biblioteca de Node.js é muito veloz em execução de código.

- Possui *thread* única porém é muito escalável - Node.js possui um modelo de laço de Eventos com a utilização de uma única *thread*. O mecanismo de Eventos ajuda o servidor enviar respostas de forma não bloqueante e faz o servidor altamente escalável em comparação ao modelo tradicional de servidores que criam um número limitado de *threads* para lidar com requisições dos clientes. Node.js usa um programa de *thread* única e esse mesmo programa pode providenciar serviço a um número muito maior de requisições do que um servidor tradicional, como o Apache HTTP.

- Sem *Buffering* - Aplicações em Node.js nunca armazenam nenhum dado em *buffers*. Essas aplicações simplesmente enviam dados em blocos.

- Licenciado - Node.js está sob uma licença MIT.

5.3 Comunicação Web e Aplicações Web

O protocolo WebSocket foi utilizado para a comunicação deste Projeto, pois há vários fatores no protocolo HTTP e do modelo cliente-servidor convencional que se tornam empecilhos para as necessidades apresentadas.

Enquanto que, na prática, bastou escolher esse protocolo WebSocket, sua criação representa uma evolução no paradigma de desenvolvimento de páginas Web. A seguir é dado um panorama dos fatores que desencadearam sua criação e o que isso representa.

5.3.1 Modelo cliente-servidor

A comunicação de dados entre computadores através da Internet é algo muito comum, e necessita de uma arquitetura de rede adequada para ocorrer.

A mais utilizada é o modelo cliente-servidor. Neste modelo, existem dois tipos de processos rodando, o Cliente e o Servidor. O Cliente é o processo que roda nos computadores locais e se conecta ao Servidor. A comunicação é dada quando o Cliente manda uma requisição ao Servidor (esta requisição pode ser por uma página web, um processamento de dados, entre outros). Por sua vez, o Servidor recebe esses dados e faz o processamento necessário para completar essa requisição, e em seguida envia uma resposta ao Cliente. A resposta pode ser o que foi requisitado ou outro tipo de mensagem, como de erro, caso ocorra uma falha no processamento ou negação de permissão. O Cliente então pode continuar com o seu próprio processo.

O protocolo que acompanha esse modelo é principalmente HTTP (*Hypertext Transfer Protocol*). Ele é usado para a transferência de Recursos entre processos. Recursos são arquivos HTML, imagens, resultados de buscas, entre outros.

Um Cliente HTTP, como um navegador, abre uma conexão e envia uma mensagem de requisição ao Servidor HTTP, que retorna uma mensagem de resposta, normalmente contendo o recurso pedido. Após entregar a resposta, o Servidor encerra a conexão.

Por encerrar a conexão, é dito que HTTP um protocolo sem estado, pois não mantém informações sobre a conexão entre transações. Ele também implica que cada mensagem HTTP precisa possuir um cabeçalho com informações sobre o contexto, uma vez que o Servidor não guarda nenhum dado.

O modelo cliente-servidor é suficiente para páginas Web, pois o Cliente pede páginas estáticas e o Servidor as fornece sempre que necessário. Mas essa arquitetura não permite um uso mais dinâmico dessas páginas Web, pois sua estrutura é muito burocrática.

Para fazer uma página mais responsiva, com *feedbacks* a cada ação do usuário e dados novos, seria necessário que o Cliente fizesse uma requisição ao Servidor após cada ação executada (como preencher um formulário ou levar o cursor a algum ponto específico, por exemplo). Cada requisição HTTP dessas carregaria muitas informações em seu cabeçalho. Ao receber a página atualizada, o Cliente ainda precisaria recarregá-la para atualizar o que fosse necessário. Os dois problemas claros nessa estrutura são:

- A necessidade de recarregar a página inteira, mesmo que somente uma pequena parcela dos dados tenha sido alterada. Esse comportamento foi criado em um momento no qual não se tinha a necessidade de alterar pequenas coisas na página, e que cada HTML vindo do Servidor seria uma página completamente nova. A impossibilidade de atualizar dados individualmente remove qualquer dinamicidade desejada.
- O *overhead* gerado pelo cabeçalho das mensagens HTTP. Esse *overhead* cria uma latência (tempo de resposta) alta e faz com que um envio contante de requisições ao Servidor exija muitos recursos.

Um exemplo de comportamento impossível com essas restrições é dar um *feedback* instantâneo para o usuário na hora que ele está preenchendo um cadastro em algum site, como informar se o email inserido já foi usado ou se a senha possui os parâmetros mínimos necessários de segurança.

5.3.2 AJAX

Da discussão gerada a partir desses problemas surgiu o Ajax (*Asynchronous JavaScript and XML*). Ajax é uma técnica de desenvolvimento Web criada para possibilitar a requisição por dados parciais ao Servidor e recebê-los de forma assíncrona no *background*, sem precisar recarregar a página exibida. Com essa técnica passa a ser possível atualizar partes de uma página com base em eventos do usuário.

Para ilustrar esse funcionamento com um exemplo simples, basta usar a ferramenta de busca do Google⁷. O usuário começa a digitar uma busca e resultados já aparecem sem a necessidade de atualizar a página, mesmo antes da busca estar completa (TODO imagem). O *input* do usuário ativa requisições no *background* para obter e atualizar os resultados exibidos, sem se preocupar se a busca será alterada no futuro. Assim, o usuário recebe um *feedback* mais dinâmico e menos burocrático do que o convencional (digitar a busca completa e ir para uma página de resultados).

⁷google.com

Com o Ajax, nota-se uma mudança de paradigmas na Web. Onde antes só havia a noção de página Web, agora começa a se formar uma noção de Aplicação Web. Antes, a maior interatividade possível era algo como um fluxo de telas com dados dependentes da tela passada, mas agora ações de usuários passam a ter resultados instantâneos. Sites com um fluxo de dados muito grande e dinâmico passam a ser possíveis, como Facebook ⁸, no qual é possível fazer comentários, interagir com usuários e visualizar tanto conteúdo quando desejado sem precisar atualizar a página.

Essa mudança de paradigmas fez com que muitas Aplicações tipicamente executadas em *Desktop* fossem desenvolvidas para Web, como clientes de email ou aplicações que exigissem um fluxo de dados muito grande entre o Cliente e o Servidor. Mas o que ficou claro com o tempo é que Ajax não bastava para muitas dessas Aplicações, em específico as que funcionavam em tempo real. Muitas vezes é o Servidor que precisa se comunicar ao Cliente sobre mudança nos dados. O modelo de funcionamento do Ajax não possui esse tipo de interface, então, para fazer aplicações assim funcionarem corretamente, é necessário muito esforço, lutar contra a linguagem.

Um exemplo claro disso é uma aplicação de *Chat*. Não é possível ter um padrão de quando o Servidor tem novas mensagens, então é necessário que o Cliente faça requisições a cada período de tempo. Se esse período for curto, o fluxo de dados intenso pode se tornar um problema. Se o período for longo, vai contra o princípio de ser uma comunicação em tempo real, mensagens vão demorar mais a serem entregues. Além disso, o modelo de comunicação do Ajax envia e recebe mensagens em paralelo, sem preservar sua ordem. Então, ao se fazer atualizações síncronas, é necessário ainda um tratamento de ordenação para exibir as mensagens novas.

A necessidade de vários tratamentos para uma aplicação relativamente simples funcionar deixou claro que havia necessidade de mais ferramentas para o desenvolvimento de aplicações Web.

5.3.3 Necessidade de Comunicação Bilateral

Um dos fatores mais limitantes era que a comunicação cliente-servidor não é bilateral. Não é possível para o Servidor mandar dados espontaneamente, sem que o Cliente tenha feito uma requisição antes. Para contornar esse problema foram criadas técnicas para simular essa comunicação bilateral: *Polling*, *Long-Polling* e *Streaming*.

- **Polling**

⁸facebook.com

Essa técnica já foi descrita acima no exemplo do *Chat*. Ela consiste em fazer o Cliente mandar requisições em intervalos regulares e receber a resposta logo em seguida. Ela foi a primeira tentativa de se contornar o problema, principalmente por ter a implementação mais simples e direta.

Essa solução é boa quando se sabe o tempo de atualização de dados no Servidor, pois é possível sincronizar os tempos de requisição e atualização. Porém esse é somente um dos cenários possíveis. Dados em tempo real não costumam ser tão previsíveis, então é inevitável que uma parcela dessas requisições seja desnecessária e muitas conexões sejam abertas e fechadas sem necessidade, em momentos de baixo fluxo de atualização.

- **Long-Polling**

No *Long-Polling*, o Cliente manda uma requisição e o Servidor a mantém aberta durante um determinado período de tempo. Se uma atualização chegar durante esse período, a resposta é enviada ao Cliente com os dados novos. Se o tempo acabar sem que haja mudanças, o servidor envia uma resposta para encerrar a requisição aberta. A cada resposta recebida, o Cliente envia uma nova requisição para ficar em aberto.

Essa técnica apresenta melhoras em relação ao *Polling*, porém, se existe um fluxo muito alto de atualizações no Servidor, ela não oferece nenhuma melhora substancial em relação a ele. Nessa situação aliás, o *Long-Polling* pode ser pior, pois pode ficar instável em um loop contínuo de *Polls* imediatos.

- **Streaming**

Com a técnica de *Streaming*, o Cliente manda uma requisição e o Servidor a mantém aberta continuamente. Sempre que dados são atualizados, o Servidor os envia ao Cliente, mas não encerra a conexão.

O ponto negativo dessa técnica é que, como o *Streaming* ainda é encapsulado por HTTP, é possível que *Firewalls* e servidores *Proxy* possam escolher armazenar a resposta em um *Buffer*, o que aumenta muito a latência da mensagem.

Por fim, todas essas técnicas envolvem mensagens HTTP, que contêm cabeçalhos com muitos dados desnecessários para esse uso, gerando alto consumo de recursos. Além disso, uma verdadeira conexão bilateral requer mais do que o fluxo de dados vindo do servidor, é necessário também ter o fluxo

originado no cliente. Para fazer uma simulação mais consistente com o desejado, muitas soluções hoje usam duas conexões de fluxo, uma para o cliente e uma para o servidor. A coordenação e manutenção dessas duas conexões paralelas gera um *overhead* ainda maior em termos de recursos, além do claro aumento de complexidade do código.

Por todos esses fatos, ficou claro que era necessário ter uma conexão bilateral de verdade. Para que essa conexão fosse possível, era necessário ter um controle maior da transferência de dados era necessário. Dentro do protocolo TCP, esse controle é feito por meio dos Sockets, mas, até então, eles eram indisponíveis ao desenvolvimento Web, não havia interface de interação.

5.3.4 Sockets

No modelo cliente-servidor uma conexão é definida por dois pontos finais, um no cliente e um no servidor. Cada um desses pontos finais serve para mandar e receber os dados do outro lado da conexão, e eles são nomeados como Sockets (Soquetes de Rede). Essa a conexão entre Sockets é implementada no protocolo TCP⁹. Cada Socket é identificado por um endereço de IP e uma Porta (por exemplo, 192.168.0.1:8000), independente de estar no Cliente ou no Servidor, e a conexão TCP pode ser definida de maneira única por seus dois Sockets conectados.

A conexão cliente-servidor, considerando os Sockets, é feita da seguinte maneira:

- No lado do cliente, primeiramente ele precisa saber o endereço de IP da máquina onde o servidor roda e a porta que ouve (esses dados definem o socket do servidor). Ele então envia um sinal a esse socket para pedir a conexão. Nesse sinal há um identificador, que contém seu próprio IP e uma porta escolhida na hora, para que o servidor saiba a quem enviar as respostas. É importante ter em mente que o cliente não possui um socket ainda, ele será criado caso a conexão seja bem sucedida.
- No lado do servidor, se não ocorrer nenhum erro, a conexão é aceita. Neste momento, o servidor cria um novo socket com o mesmo IP e porta local do original, para conectá-lo com o cliente. Esse novo socket é necessário para que o servidor possa continuar ouvindo a outras conexões naquela porta com o socket original, enquanto a cópia passa a ser exclusiva ao cliente.
- De volta ao cliente, se a conexão é aceita, o seu socket é criado (com as especificações que foram enviadas no pedido).

⁹Protocolo base de transporte na Internet.

- Agora o cliente e o servidor podem se comunicar com leitura ou escrita nesses novos sockets criados.

A API básica para manipulação de Sockets possui as seguintes funções:

- `SOCKET` : Cria um ponto final de uma comunicação.
- `BIND` : Associa um endereço local a um Socket.
- `LISTEN` : Anuncia que aceita conexões, e dá o tamanho da *queue*.
- `ACCEPT` : Aceita um pedido de conexão.
- `CONNECT` : Envia um pedido de conexão.
- `SEND` : Envia dados através de sua conexão.
- `RECEIVE` : Recebe dados vindos de sua conexão.
- `CLOSE` : Encerra a conexão.

Essa API não é acessível ao nível de desenvolvimento Web por fazer parte do protocolo TCP.

5.3.5 Websockets

WebSocket é um protocolo que lida com Sockets de uma conexão que foi criado para permitir a comunicação bilateral entre cliente e servidor. Ele permite mais liberdade para criar novos protocolos e maneiras de se transferir dados.

Sua interface é bastante simples, seu objetivo é o envio e recebimento de mensagens entre o cliente e o servidor. Para o protocolo, essa hierarquia cliente-servidor deixa de existir, não há diferenciação entre os dois, ambos são apenas dois processos conectados. A comunicação é dada por envio de mensagens e ocorrência de eventos. Esses processos podem enviar mensagens a qualquer momento e estão ouvindo um ao outro.

O protocolo foi pensado para funcionar bem com a infraestrutura Web já existente. Como parte desse paradigma, sua especificação determina que uma conexão é estabelecida inicialmente com o protocolo HTTP, garantindo retrocompatibilidade.

A conexão é feita da seguinte forma. Após iniciada a conexão HTTP, o cliente envia uma requisição ao servidor indicando que deseja trocar de protocolos, de HTTP para WebSocket. Se o servidor entende esse protocolo, ele envia uma resposta para permitir a troca. Nesse momento a conexão

HTTP é interrompida e a conexão WebSocket toma o seu lugar. Esse processo é chamado de *WebSocket Handshake*.

A estrutura da mensagem em WebSocket é mínima. O corpo da mensagem, os dados a serem enviados, só possui dois formatos, texto ou binário. O cabeçalho contém um identificador de tipo do formato, que tem um campo para o tamanho da mensagem e nada mais. A mensagem completa é só o conteúdo mais seu tamanho. Como a conexão não muda em momento algum, todas as informações contidas no cabeçalho de requisição e resposta HTTP são invariantes e, portanto, não precisam ser reenviadas.

Um dos processos conectados, em um momento arbitrário, envia uma mensagem ao outro, talvez porque dados tenham sido atualizados ou o usuário tenha feito alguma ação em específico. Uma vez que a mensagem é enviada, esse processo volta ao que estava fazendo, sem a necessidade de esperar alguma resposta do outro processo. Esse comportamento é chamado de assíncrono, pois não um lado não age em sincronia com o outro.

O outro processo vê a chegada dessa nova mensagem como um evento. Quando esse evento acontece, ele executa um comportamento específico (*callback*) com os dados recebidos, como alteração de informações exibidos ou um cálculo com os valores novos. Por isso é dito que WebSocket é orientado a eventos.

Existem quatro tipos de eventos e seus *callbacks* são definidos na criação do WebSocket. Esses eventos são:

- *onopen*: ocorre quando a conexão é feita;
- *onclose*: ocorre quando a conexão é fechada;
- *onmessage*: ocorre quando uma mensagem do outro WebSocket chega;
- *onerror*: ocorre se há algum erro na conexão.

Para ilustrar melhor esse comportamento de conexão, mensagens e eventos, abaixo há um exemplo em HTML de uma página que utiliza WebSockets. (TODO link) <https://www.websocket.org/echo.html>

```
1  <!DOCTYPE html>
2  <meta charset="utf-8" />
3  <title>WebSocket Test</title>
4  <script language="javascript" type="text/javascript">
5
6      var wsUri = "ws://echo.websocket.org/";
7      var output;
8
9      function init()
10     {
11         output = document.getElementById("output");
12         testWebSocket();
```

```

13 }
14
15 function testWebSocket()
16 {
17     websocket = new WebSocket(wsUri);
18     websocket.onopen = function(evt) { onOpen(evt) };
19     websocket.onclose = function(evt) { onClose(evt) };
20     websocket.onmessage = function(evt) { onMessage(evt) };
21     websocket.onerror = function(evt) { onError(evt) };
22 }
23
24 function onOpen(evt)
25 {
26     writeToScreen("CONNECTED");
27     doSend("WebSocket rocks");
28 }
29
30 function onClose(evt)
31 {
32     writeToScreen("DISCONNECTED");
33 }
34
35 function onMessage(evt)
36 {
37     writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data+'</span>');
38     websocket.close();
39 }
40
41 function onError(evt)
42 {
43     writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
44 }
45
46 function doSend(message)
47 {
48     writeToScreen("SENT: " + message);
49     websocket.send(message);
50 }
51
52 function writeToScreen(message)
53 {
54     var pre = document.createElement("p");
55     pre.style.wordWrap = "break-word";
56     pre.innerHTML = message;
57     output.appendChild(pre);
58 }
59
60 window.addEventListener("load", init, false);
61
62 </script>
63
64 <h2>WebSocket Test</h2>
65
66 <div id="output"></div>

```

Essa página executa um código simples. Ela cria um WebSocket conectado à “ws://echo.websocket.org/”.

O evento de conexão *'onOpen'* ativa uma função que escreve na tela e

manda uma string para o outro WebSocket.

O evento de recebimento de mensagem '*onMessage*' escreve na tela o conteúdo que foi recebido e fecha a conexão.

O evento de encerramento de conexão e de erro ('*onClose*' e '*onError*') só escrevem na tela um status.

5.3.6 Teste de eficiência de Websockets

No experimento a seguir, é mostrado a diferença de tráfego de dados e latência entre WebSocket e *Polling* para requisitar dados em tempo real.

Informações para entender o exemplo:

- *RabbitMQ Message Broker*: um simples programa que recebe e encaminha mensagens. Nesse exemplo ele está em um servidor, recebe dados de um mercado de ações fictício.
- *Java Servlet*: uma classe Java usada para estender as funcionalidades de um servidor. Pode ser definido como um componente semelhante um servidor, que gera dados HTML e XML para a camada de apresentação de uma aplicação Web. Ele processa dinamicamente requisições e respostas.
- Mozilla Firefox: *Browser* muito utilizado.
- Firebug: Um *add-on* do Mozilla Firefox que permite fazer debug de páginas Web e monitorar o tempo que leva para carregar páginas e executar scripts.
- *Live HTTP Headers*: *add-on* do Mozilla Firefox que mostra ao vivo o tráfego de cabeçalhos HTTP.

O exemplo foi retirado do site <https://www.websocket.org/quantum.html> e foi transcrito em tradução livre:

Então, o quão dramática é a redução de tráfego de dados desnecessários e latência? Vamos comparar uma aplicação *Polling* e uma WebSocket lado a lado.

Para o exemplo de *Polling*, eu criei uma simples aplicação Web, na qual a página manda requisições a um *RabbitMQ Message Broker* pedindo dados de um Mercado de Ações em tempo real, usando um modelo *publish/subscribe* tradicional. Ele requisita esses dados por fazer *Polling* para um *Java Servlet* hospedado no Servidor Web. O *RabbitMQ Message Broker* recebe

os dados de um *feed* de Mercado de Ações fictício, atualizado continuamente. A página Web conecta e se inscreve em um canal específico do Mercado (um Tópico do *Message Broker*) e usa uma chamada `XMLHttpRequest` para pedir (fazer um *Poll*) por atualizações uma vez por segundo. Quando elas chegam, alguns cálculos são feitos e os dados do Mercado são mostrados em uma tabela, como na imagem a seguir:

| COMPANY | SYMBOL | PRICE | CHANGE | SPARKLINE | OPEN | LOW | HIGH |
|--------------------------|--------|-------|--------|--|-------|-------|-------|
| THE WALT DISNEY COMPANY | DIS | 27.65 | 0.56 |  | 27.09 | 24.39 | 29.80 |
| GARMIN LTD. | GRMN | 35.14 | 0.35 |  | 34.79 | 31.31 | 38.27 |
| SANBISK CORPORATION | SNDK | 20.11 | -0.13 |  | 20.24 | 18.22 | 22.26 |
| GOODRICH CORPORATION | GR | 49.99 | -2.35 |  | 52.34 | 47.11 | 57.57 |
| NVIDIA CORPORATION | NVDA | 13.92 | 0.07 |  | 13.85 | 12.47 | 15.23 |
| CHEVRON CORPORATION | CVX | 67.77 | -0.53 |  | 68.30 | 61.49 | 75.11 |
| THE ALLSTATE CORPORATION | ALL | 30.88 | -0.14 |  | 31.02 | 27.92 | 34.12 |
| EXXON MOBIL CORPORATION | XOM | 65.66 | -0.86 |  | 66.52 | 59.87 | 73.17 |
| METLIFE INC. | MET | 35.50 | -0.15 |  | 35.73 | 32.16 | 39.30 |

Figura 1 — Uma aplicação de valores de Mercado em JavaScript

Nota: O *feed* no Tópico do Mercado produz muitas atualizações de preço por segundo, então usar *Polling* com um segundo de intervalo é mais prudente do que *Long-Polling*, pois ele resultaria em uma série de *Polls* contínuos. O *Polling* controla de forma efetiva a vinda de atualizações.

Tudo parece certo, mas ao se olhar o funcionamento, é revelado que há problemas sérios com essa aplicação. Por exemplo, com o *Firebug*, você consegue ver que a requisição `GET` martela o Servidor em intervalos de 1 segundo. Ativando o *Live HTTP Headers*, é revelado o enorme *overhead* causado por cabeçalhos associados a cada requisição. Os dois próximos exemplos mostram o cabeçalho HTTP para somente uma requisição e uma resposta.

Exemplo 1 — cabeçalho de requisição HTTP:

```

1 GET /PollingStock//PollingStock HTTP/1.1
2 Host: localhost:8080
3 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
  rv:1.9.1.5)
4 Gecko/20091102 Firefox/3.5.5
5 Accept: text/html,application/xhtml+xml,application/
  xml;q=0.9,*/*;q=0.8
6 Accept-Language: en-us
7 Accept-Encoding: gzip,deflate
8 Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
9 Keep-Alive: 300

```



```

10 | Connection: keep-alive
11 | Referer: http://www.example.com/PollingStock/
12 | Cookie: showInheritedConstant=false;
13 | showInheritedProtectedConstant=false;
14 | showInheritedProperty=false;
15 | showInheritedProtectedProperty=false;
16 | showInheritedMethod=false;
17 | showInheritedProtectedMethod=false;
18 | showInheritedEvent=false;
19 | showInheritedStyle=false;
20 | showInheritedEffect=false

```

Exemplo 2 — cabeçalho de resposta HTTP

```

1 | HTTP/1.x 200 OK
2 | X-Powered-By: Servlet/2.5
3 | Server: Sun Java System Application Server 9.1_02
4 | Content-Type: text/html; charset=UTF-8
5 | Content-Length: 21
6 | Date: Sat, 07 Nov 2009 00:32:46 GMT

```

Só por diversão, eu contei todos os caracteres. O total de *overhead* de informação na requisição e resposta HTTP é de 871 bytes, sem incluir nenhum dado! Claro, esse é somente um exemplo e você pode ter menos de 871 bytes de cabeçalho, mas eu também vi casos onde ele ultrapassava 2000 bytes. Nessa aplicação de exemplo, uma mensagem típica do Tópico de Mercado contém em torno de 20 caracteres. Como você pode ver, ela é efetivamente afogada pelo excesso de informação do cabeçalho, que nem é necessário no final das contas!

Então, o que acontece quando você dá *deploy* nessa aplicação para um grande número de usuários? Vamos observar o tráfego de dados somente dos dados do cabeçalho HTTP de requisição e resposta associados a essa aplicação de *Polling* em três casos diferentes.

- **Caso de uso A:** 1,000 clientes fazendo *Polling* a cada segundo: tráfego de dados é $(871 \times 1,000) = 871,000$ bytes = 6,968,000 bits por segundo. (6.6 Mbps)
- **Caso de uso B:** 10,000 clientes fazendo *Polling* a cada segundo: tráfego de dados é $(871 \times 10,000) = 8,710,000$ bytes = 69,680,000 bits por segundo. (66 Mbps)
- **Caso de uso C:** 100,000 clientes fazendo *Polling* a cada segundo: tráfego de dados é $(871 \times 100,000) = 87,100,000$ bytes = 696,800,000 bits por segundo. (665 Mbps)

Essa é uma quantidade enorme de tráfego de dados desnecessários! Imagina só se fosse possível transferir a informação

necessária. Então, com HTML5 Web Sockets você pode! Eu re-construí a aplicação usando WebSockets, adicionando um manipulador de evento para que a página Web possa assincronamente ouvir por mensagens do *Message Broker* de atualizações do preço do Mercado. Cada uma dessas mensagens é um WebSocket frame que possui só 2 bytes de overhead (ao invés de 871)! Veja como isso afeta o tráfego de dados de overhead naqueles três casos.

- **Caso de uso A:** 1,000 clientes recebem 1 mensagem por segundo: Tráfego de dados é $(2 \times 1,000) = 2,000$ bytes = 16,000 bits por segundo (0.015 Mbps)
- **Caso de uso B:** 10,000 clientes recebem 1 mensagem por segundo: Tráfego de dados é $(2 \times 10,000) = 20,000$ bytes = 160,000 bits por segundo (0.153 Mbps)
- **Caso de uso C:** 100,000 clientes recebem 1 mensagem por segundo: Tráfego de dados é $(2 \times 100,000) = 200,000$ bytes = 1,600,000 bits por segundo (1.526 Mbps)

Como você pode ver na figura abaixo, WebSockets provê uma redução dramática no tráfego de dados desnecessários em relação ao método de *Polling*.

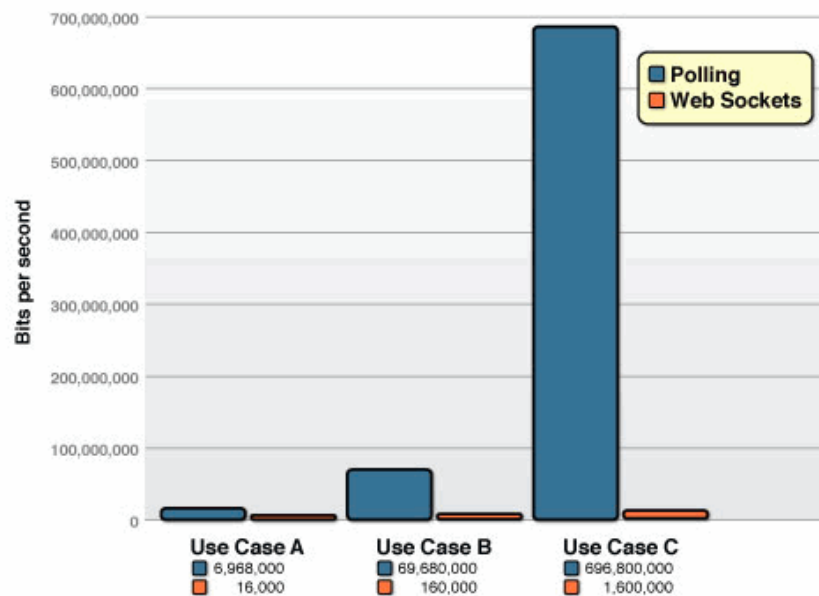


Figura 2 — Comparação de *overhead* entre aplicações de *Polling* e *WebSocket*

E em relação à redução na latência? Veja a figura abaixo. Na metade de cima, você consegue a latência do método de *Polling*.

Se assumirmos, nesse exemplo, que é necessário 50 milissegundos para uma mensagem chegue do servidor ao *browser*, então a aplicação de *Polling* introduz muita latência extra, porque cada nova requisição precisa ser enviada ao servidor quando a resposta está completa. Essa nova requisição requer outros 50 milissegundos, e, durante esse tempo, o servidor não consegue mandar qualquer outra mensagem ao browser, resultando em um consumo de memória adicional ao servidor.

Na metade de baixo da figura, você vê a redução na latência proporcionada pelo uso de WebSocket. Uma vez que a conexão é promovida a WebSocket, as mensagens podem fluir do servidor ao browser no momento em que surgem. Ainda leva 50 milissegundos para as mensagens atravessarem do servidor ao cliente, mas a conexão WebSocket permanece aberta para que não haja necessidade de enviar outra requisição ao servidor.

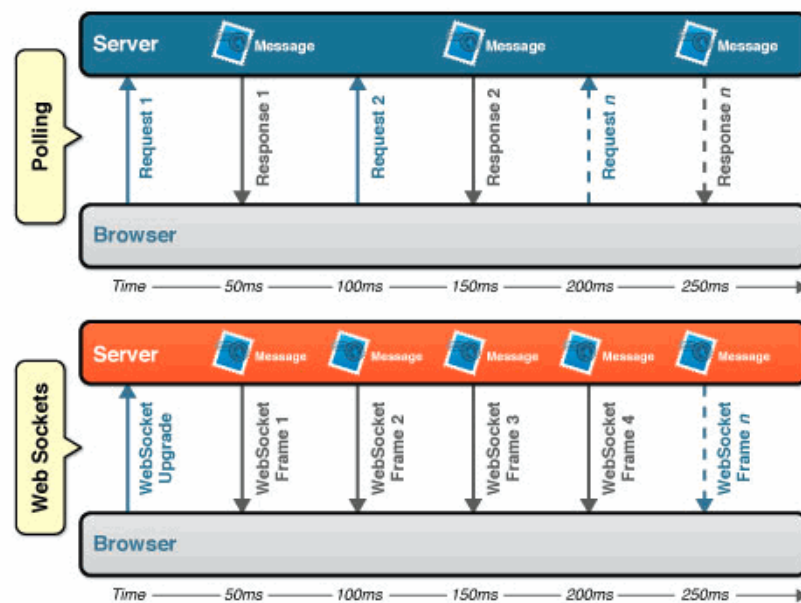


Figura 3 — Comparação de latências entre *Polling* e *WebSocket*

Em conclusão WebSocket é a solução ideal para problemas apresentados anteriormente. É um protocolo feito para a real comunicação plenamente bilateral e obtém isso com o menor gasto de recursos possível. Com ele, a criação de aplicações Web se torna muito mais viável e popular.

Mas existe um problema nesse meio de desenvolvimento Web, que é o fato de que a todo momento a infraestrutura é alterada. Novos protocolos são criados, outros são atualizados, antigos deixam de funcionar corretamente. Esse é um problema inerente ao meio e que afeta a todos.

Para ilustrar esse fato, vamos tomar como exemplo o protocolo *SPDY* (pronunciado “speedy”), desenvolvido principalmente pela Google.¹⁰

Em linhas gerais, esse protocolo busca velocidade e se baseia no fato de que, se uma conexão é estabelecida com o servidor e o cliente começa a enviar muitas requisições HTTP, essas requisições vão incluir informações repetidas, comuns àquela sessão. Ele define então que, ao invés de enviar essas informações desnecessárias repetidamente durante a sessão, o servidor passa a salvar em um dicionário os usuários conectados e utilizar essas informações salvas. Por conta disso, fazer requisições passa a ser mais rápido e consumir menos recursos.¹¹

Também para aumentar a velocidade, o *SPDY* multiplexa requisições para permitir que sejam feitas em paralelo. Atualmente no protocolo HTTP as requisições são todas feitas em série. O problema aparece nesse momento, pois o WebSocket esperava ter um socket TCP dedicado e agora passa a funcionar em cima de uma camada de conexão *SPDY* multiplexada.

O WebSocket (assim como outros protocolos e ferramentas) necessita de atualizações de tempos em tempos, por situações imprevisíveis como essa. Essa mudança constante aumenta o nível de dificuldade para o desenvolvedor estar sempre atualizado com seus detalhes de implementação.

Uma das maneiras de se contornar essa dificuldade é utilizar outras APIs que cuidam desses detalhes internamente.

5.3.7 Socket.io

Por fim, foi criado o Socket.io, uma API de WebSocket para JavaScript. Ela cuida de todos os detalhes de funcionamento do protocolo internamente, deixando uma interface simples ao usuário. Assim, mudanças no funcionamento do protocolo não chegam ao desenvolvedor.

Para uma conexão ser estabelecida, tanto o cliente quanto o servidor precisam estar executando Socket.io. Sua implementação tenta primeira-

¹⁰Ele não é um protocolo padrão, porém é bastante promissor e está sendo utilizado para o desenvolvimento do protocolo HTTP 2.0.

¹¹Note a semelhança com WebSockets.

mente fazer a conexão usando WebSocket, mas se o servidor não suporta esse protocolo, ele recua (faz *fallback*) para outras tecnologias, como *AJAX long-polling*, *AJAX multipart streaming*, *IFrame*, *JSONP polling*, entre outros (todos utilizando a mesma interface). Esse *fallback* permite à API abranger uma quantidade muito maior de servidores, reduzindo possíveis problemas numa implementação em larga escala.

Além da implementação básica do protocolo WebSocket, ele também adiciona outras funcionalidades importantes para o desenvolvimento de uma aplicação mais robusta:

- **A possibilidade de criação arbitrária de eventos.** Além dos eventos padrões *'connect'*, *'message'* e *'disconnect'*, o desenvolvedor pode criar quaisquer outros eventos que achar necessário.

Essa criação de eventos funciona da seguinte forma: o processo emissor manda a mensagem e o nome do evento que está sendo enviado. O processo receptor define o evento que vai ouvir e a subrotina que vai ser executada na ocorrência desse evento. Se o evento lançado for ouvido pelo receptor, o *callback* é chamado.

Com WebSocket, o desenvolvedor fica preso aos quatro eventos pré definidos: *'onopen'*, *'onclose'*, *'onmessage'* e *'onerror'*. Internamente, os quatro são eventos de mensagem, disparados em momentos e situações específicas. O que o Socket.io faz é dar a liberdade de definir quaisquer eventos desejados.

- **A possibilidade de conectar múltiplos sockets em uma mesma porta.** Internamente, ele faz uma multiplexação das várias conexões abertas em uma mesma porta, mas os processos entendem que tem um socket dedicado a eles. Utilizando WebSockets, cada socket precisa de uma porta exclusiva para fazer a conexão.

A possibilidade de vários sockets em uma mesma aplicação permite uma modularização muito maior, pois não é necessário que um módulo saiba se tem uma conexão dedicada ou se está compartilhando com mais dezenas de outros sockets.

Um ponto negativo dessa definição de eventos é que o cabeçalho da mensagem é um pouco maior, pois o de WebSockets é reduzido ao extremo, mas esse tamanho ainda é ínfimo e o *overhead* causado não se torna um problema.

- **Definição de *namespaces*.** Dada a existência de múltiplos sockets, é possível organizá-los em *namespaces*. Cada *namespace* vai conter um

conjunto de sockets conectados, e eventos emitidos a um *namespace* serão enviados somente à esse conjunto.

Por padrão, caso nenhum *namespace* seja definido, todos os sockets são conectados ao *namespace* `'/'`. Assim, esse funcionamento fica invisível ao desenvolvedor que não os está utilizando.

- **Definição de *rooms*.** *Rooms* são um segundo nível de organização, definidos dentro de *namespaces*. Um socket pode entrar e sair de *rooms* com funções `join` e `leave`. Cada socket pode definir uma *room* específica na hora de emitir um evento, da forma:

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

É fácil notar a preocupação com organização dessas funcionalidades. Com o protocolo WebSocket puro, é inviável fazer uma aplicação que faz o uso de múltiplos sockets com execuções distintas, mas que interagem entre si. Um dos propósitos da criação de Socket.io foi aumentar o grau de liberdade do desenvolvedor e tornar esse tipo de aplicação muito simples de ser desenvolvida.

6 Casos de Uso

7 Parte Subjetiva

7.1 Desafios e frustrações

7.2 A contribuição do curso de Computação

7.3 Próximo Passos

7.4 Agradecimentos

8 Bibliografia