

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 execute 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. Estes exemplos e outros são aprofundados na seção de Casos de Uso (TODO seção de Casos de Uso).

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 apresentado o que implementa o 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(TODO ref) 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.

- Cabeçalho mínimo de mensagens. Se o fluxo de mensagens for alto, esse cabeçalho pode consumir muita banda da conexão, além de exigir muitos recursos, tanto do servidor como dos clientes. É necessário que ele 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.

- Conexão 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. 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, o tecnologia usada foi o Framework de JavaScript Socket.io, que implementa o protocolo WebSocket.

4.1.1 Protocolo WebSocket

WebSocket é um protocolo que lida com Sockets (TODO ref) 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, não existe essa diferenciação entre os dois, ambos são apenas dois processos conectados. Essa hierarquia cliente-servidor deixa de existir e não há mais uma separação de funções entre ambos. 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 (TODO ref), 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 (TODO form).

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 uma resposta (com a exceção talvez de qualquer confirmação sobre a entrega). Esse comportamento é chamado de assíncrono, pois não necessita de uma resposta para continuar com sua execução.

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.

4.1.2 Socket.io

Socket.io é a API de WebSocket para JavaScript utilizada. Ele cuida de todos os detalhes de funcionamento do protocolo internamente, deixando uma interface simples ao usuário.

Para uma conexão ser estabelecida, tanto o cliente quanto o servidor precisam estar executando Socket.io. Sua implementação tenta primeiramente 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* abrange uma quantidade muito maior de servidores, permitindo um uso não tão restrito dessa API.

Além da implementação básica do protocolo WebSocket, ele também adiciona outras funcionalidades muito importantes para o desenvolvimento de uma aplicação maior e mais robusta. As que foram utilizadas no Projeto foram:

- É possível 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.

- Ele permite a criação arbitrária de eventos. 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.

Essa criação de eventos funciona assim: na hora de enviar uma mensagem, o processo emissor define também 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.

Um ponto negativo dessa definição de eventos é que o cabeçalho da mensagem é um pouco maior, pois o de WebSockets é mínimo ao extremo, mas esse tamanho ainda é ínfimo e o overhead causado é desprezível.

Essas funcionalidades aumentam o grau de liberdade e permitem um escopo maior na hora de criar uma aplicação robusta.

Exemplos de uso de Socket.io estão presentes no próprio código do servidor e dos clientes implementados.

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 definidos na proposta desse Projeto. 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. 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 funcionali-

dades 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:

4.2.2 Implementação

O aplicativo implementado possui o básico de interação, ele 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:

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.
- 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* ² 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.

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`:

- A divisão “`deviceready`” (TODO linha X) 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`” (TODO: linha X) 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`” (TODO linha X) exibe dados vindos do servidor.
- Os botões (TODO linha X a Y) executam diversas funções do código de enviar através de eventos para o servidor.
- Os scripts (TODO linha X a Y) 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.html`:

Para inicializar o aplicativo:

A variável “`app`” (TODO linha X) é uma classe padrão do Cordova com funções para inicializar o aplicativo. Ele é inicializado na linha X (TODO).

Para fazer a conexão com o servidor:

Na linha X (TODO), uma função é adicionada ao evento “`deviceready`” que o Cordova emite quando tudo acaba de ser inicializado. Quando o evento ocorre, 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.

²tempo necessário para uma mensagem ser enviada ao servidor e voltar dele

Eventos que o servidor ativa:

Na inicialização do Socket, dois eventos são definidos, “connect”, que chama a função “onServerConnect” e “server_ping”, que chama “onServerPing”.

“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.

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 primeira decisão de design foi fazer o *browser* executar uma computação gráfica. Para tal, 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 GPU (TODO explicar GPU) 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*.

Sua interface é simples de se usar, ela lida com detalhes técnicos de implementação de baixo nível internamente, 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 página na Wikipédia:

Three.js includes the following features:

- 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
- 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.

É criada uma esfera no centro da tela que recebe valores de *devices* vindos do servidor. Esses valores podem mudar a velocidade de rotação nos eixos X, Y e Z, e a sua cor.

Este código é mais extenso, então algumas partes serão omitidas.

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

```
var socket;

var xRot, yRot, zRot;
```

```
var r, g, b;
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*:

```
socket = io();

function rotate (data) {
    xRot += data.x;
    yRot += data.y;
    zRot += data.z;
}

function reset_rotation () {
    xRot = 0;
    yRot = 0;
    zRot = 0;
}

function update_rotation_and_colors (data) {
    var cap = 9;
    var x = Math.min(data.x, cap);
    var y = Math.min(data.y, cap);
    var z = Math.max(2, Math.min(data.z, cap));

    var weight = 5;

    xRot = y / 50;
    yRot = x / 50;

    r = (weight * r + (z / cap)) / (weight + 1);
    g = (weight * g + (z / cap)) / (weight + 1);
    b = (weight * b + (z / cap)) / (weight + 1);

    hasNewColors = true;
}

// Socket Events
```

```

socket.on('connect', function() {
    socket.emit('ready', 'browser');
});

socket.on('move', function (data) {
    rotate(data);
});

socket.on('reset', function () {
    reset_rotation();
});

socket.on('update_acceleration', function (data) {
    update_rotation_and_colors(data);
})

```

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.

```

var scene, camera, renderer;
var sphere, sphere;

hasNewColors = false;

function updateSphereRotation(xRot, yRot, zRot) {
    sphere.rotation.x += xRot;
    sphere.rotation.y += yRot;
    sphere.rotation.z += zRot;
}

function updateSphereColor(r, g, b) {
    sphere.material.color.r = r;

```



```

    sphere.material.color.g = g;
    sphere.material.color.b = b;
}

function init() {

    scene = new THREE.Scene();

    var view_angle = window.innerWidth/window.innerHeight
    camera = new THREE.PerspectiveCamera( 75, view_angle, 0.1, 1000 );
    camera.position.z = 5;
    scene.add(camera);

    renderer = new THREE.WebGLRenderer();
    renderer.setSize( window.innerWidth, window.innerHeight );
    document.body.appendChild( renderer.domElement );

    //-----

    // create a point light
    var pointLight = new THREE.PointLight(0xFFFFFF);

    // set its position
    pointLight.position.x = 10;
    pointLight.position.y = 50;
    pointLight.position.z = 130;

    // add to the scene
    scene.add(pointLight);

    //-----

    // New Sphere
    var radius = 2;
    var segments = 16;
    var rings = 16;

    var sphereGeo = new THREE.SphereGeometry(radius, segments, rings);
    var sphereMaterial = new THREE.MeshLambertMaterial( {
        color: 0xCC0000,
        wireframe : true
    } );

```

```

    sphere = new THREE.Mesh( sphereGeo, sphereMaterial );
    scene.add( sphere );

    //-----

    xRot = 0;
    yRot = 0;
    zRot = 0;

    r = 1;
    g = 1;
    b = 0;
}

function render() {
    requestAnimationFrame( render );

    updateSphereRotation(xRot, yRot, zRot);
    if (hasNewColors) {
        updateSphereColor(r, g, b);
        hasNewColors = false;
    };

    renderer.render(scene, camera);
};

init();
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 é implementado com Javascript, com o auxílio de Node.js e ExpressJS

4.4.1 Node.js

Node.js é uma coisa louca e terá um capítulo separado. Eu acho.

4.4.2 ExpressJS

4.4.3 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.

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`:

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 Comunicação Web e Aplicações Web

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

(TODO explicar TCP e/ou TCP/IP) 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.

(TODO a requisição é um HTTP, explicar HTTP)

<https://www.jmarshall.com/easy/http/>

(TODO nota de rodapé) A outra grande arquitetura muito utilizada também é a peer-to-peer (P2P). Nessa arquitetura, não há um computador central para funcionar como servidor, cada computador conectado (Peer) na rede realiza funções tanto de cliente como de servidor na aplicação que está sendo executada. Essa aplicação tem suas tarefas organizadas e divididas entre os Peers. Essa arquitetura é conhecida principalmente por ser usada para a transferência de arquivos grandes, como músicas e vídeos. Nessa transmissão, os Peers que tem o arquivo são conectados aos que não tem, e começam a transferência de pequenos pacotes de dados. Esses pacotes não precisam vir ordenados e o Peer receptor os armazena localmente. Uma vez que a transferência é completada, ele ordena os pacotes e monta o arquivo final. Uma vantagem dessa arquitetura é que a transferência não é limitada pela capacidade de banda de Servidor, e Peers podem se conectar e desconectar sem que haja problemas para o receptor, o arquivo não será corrompido por eventuais problemas de conexão. Um ponto negativo é que, sem um Servidor, não há um controle de que tipos de arquivos estão sendo transferidos (o que abre uma porta para pirataria) e não é fácil interromper uma transferência, uma vez que ela pode ser composta de milhares de conexões. Outro

ponto é que não é fácil de se conhecer a procedência dos dados recebidos, a segurança não pode ser garantida.

Esta arquitetura não serve para as necessidades do projeto, a hipótese de uso foi descartada após o estudo de sua estrutura.

Este modelo 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 de websites, 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 Cliente fizesse uma requisição ao Servidor após o usuário ativar algum gatilho (como preencher um formulário ou levar o cursor a algum ponto específico, por exemplo). Como o protocolo dessa requisição é HTTP, ela possui em seu cabeçalho toda a estrutura de dados para ser processado e enviado de volta, também com um cabeçalho equivalente. Ao receber a página atualizada, o Cliente ainda precisa recarregar a página para atualizar o que for necessário. Há dois problemas claros nessa estrutura:

- 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 pelas pelo cabeçalho das requisições e repostas com o protocolo HTTP. Esse overhead cria uma latência 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.

(TODO verificar se) A conexão só dura durante o processo de requisição entre ambos, ela é fechada uma vez que a requisição é satisfeita.

Da discussão gerada à partir desses problemas, surgiu o Ajax (Asynchronous JavaScript e XML). Ajax é uma técnica de desenvolvimento Web usada para fazer requisições ao Servidor para receber dados no background de forma assíncrona, sem precisar recarregar a página inteira. Com ele, é possível receber dados novos e atualizá-los no código sem alterar o resto da página. Isso 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 Google (TODO link). O usuário começa a digitar uma busca e os resultados já começam a aparecer sem a necessidade de atu-

alizer 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 muito mais dinâmico e menos burocrático do que o convencional (digitar a busca completa e ir para uma página de resultados).

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 (TODO link), 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 Chats ou aplicações que exigissem um fluxo de dados muito grande entre o Cliente e o Servidor. 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 muito com 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 enviar e recebe mensagens em paralelo, sem preservar sua ordem. Então, ao se fazer atualizações síncronas, é necessário ainda mais uma verificação para exibir mensagens em ordem cronológica.

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

6 Necessidade de Comunicação Bilateral

Um dos fatores mais limitantes era que a comunicação cliente-servidor não era bilateral. Não era possível para o Servidor mandar dados espontane-

amente, sem que o Cliente tivesse feito uma requisição antes. Para contornar esse problema foram criadas técnicas para simular essa comunicação bilateral: Polling, Long-Polling e Streaming.

6.1 Polling

Essa técnica já foi descrita acima no exemplo do Chat. Ela consiste em fazer o cliente mandar requisições em intervalos regulares de tempo e receber a resposta logo em seguida. Ela foi a primeira tentativa de se contornar o problema, principalmente por ser a implementação mais simples e direta. Essa solução é boa quando se sabe o tempo de atualização de dados no servidor, pois então 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 do servidor.

6.2 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 esses dados novos. Se o tempo acabar sem que haja mudanças, o servidor envia uma resposta para encerrar a requisição aberta. Essa técnica apresenta melhoras em relação ao Polling, porém, se existe um fluxo 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.

6.3 Streaming

Com a técnica de streaming, o cliente manda uma requisição, e o servidor manda e mantém uma resposta aberta, que é atualizada continuamente, sem ter um momento certo para ser fechada (um intervalo de tempo pode ser definido, mas normalmente a conexão é mantida indefinidamente). Essa resposta é atualizada sempre que há novos dados no servidor, mas ele nunca manda um sinal para completar a resposta e encerrar a conexão. O ponto negativo dessa técnica é que, como o streaming ainda é encapsulado no 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 requisições e respostas com um cabeçalho HTTP, que contém muitos dados desnecessários para esse uso, gerando latência. 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(TODO verificar informação), esse controle é feito por meio dos Sockets, mas, até então, eles eram indisponíveis ao desenvolvimento Web, não havia interface de interação.

7 Sockets

No modelo cliente-servidor uma conexão é dada por dois pontos finais, um no Cliente e um no Servidor. Essa é a conexão de baixo nível definida pelo protocolo TCP. Cada um desses pontos finais serve para mandar e receber os dados em relação ao outro lado da conexão e eles são nomeados como Soquetes de Rede (Sockets). Cada Socket é definido por um endereço de IP e uma Porta (por exemplo, 192.168.0.1:8000), independente se está no Cliente ou no Servidor, e a conexão TCP é definida de maneira única por seus dois Sockets.

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 contem 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).

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

8 Websockets

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>

```
<!DOCTYPE html>
<meta charset="utf-8" />
<title>WebSocket Test</title>
<script language="javascript" type="text/javascript">

var wsUri = "ws://echo.websocket.org/";
var output;

function init()
{
    output = document.getElementById("output");
    testWebSocket();
}

function testWebSocket()
{
    websocket = new WebSocket(wsUri);
    websocket.onopen = function(evt) { onOpen(evt) };
    websocket.onclose = function(evt) { onClose(evt) };
    websocket.onmessage = function(evt) { onMessage(evt) };
    websocket.onerror = function(evt) { onError(evt) };
}

function onOpen(evt)
{
    writeToScreen("CONNECTED");
    doSend("WebSocket rocks");
}

function onClose(evt)
{

```

```

        writeToScreen("DISCONNECTED");
    }

    function onMessage(evt)
    {
        writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data+'</span>');
        websocket.close();
    }

    function onError(evt)
    {
        writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
    }

    function doSend(message)
    {
        writeToScreen("SENT: " + message);
        websocket.send(message);
    }

    function writeToScreen(message)
    {
        var pre = document.createElement("p");
        pre.style.wordWrap = "break-word";
        pre.innerHTML = message;
        output.appendChild(pre);
    }

    window.addEventListener("load", init, false);

</script>

<h2>WebSocket Test</h2>

<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.

8.1 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. Ele foi retirado do site <https://www.websocket.org/quantum.html>, e é somente transcrito aqui, em tradução livre:

(TODO REF <https://www.websocket.org/quantum.html>)

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. (TODO REF: <https://www.rabbitmq.com/tutorials/tutorial-one-python.html>) 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. (TODO REF: <https://pt.wikipedia.org/wiki/Servlet>) Mozilla Firefox: Browser muito utilizado. Firebug: .

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 uma 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 updates uma vez por segundo. Quando updates chegam, alguns cálculos são feitos e os dados do Mercado são mostrados em uma tabela, como na imagem a seguir

(TODO imagem)

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 (um add-on do Mozilla Firefox que permite fazer o debug de uma página Web e monitorar

o tempo que ela para carregar páginas e executar scripts), você consegue ver que a requisição GET martela o Servidor em intervalos de 1 segundo. Ativando o Live HTTP Headers (outro add-on do Mozilla Firefox que mostra ao vivo o tráfego de cabeçalhos HTTP), 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

```
GET /PollingStock//PollingStock HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/PollingStock/
Cookie: showInheritedConstant=false;
showInheritedProtectedConstant=false;
showInheritedProperty=false;
showInheritedProtectedProperty=false;
showInheritedMethod=false;
showInheritedProtectedMethod=false;
showInheritedEvent=false;
showInheritedStyle=false;
showInheritedEffect=false
```

Exemplo 2 — cabeçalho de resposta HTTP

```
HTTP/1.x 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Content-Type: text/html; charset=UTF-8
Content-Length: 21
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 possui 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 deploy (TODO I18n) essa 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.

(TODO formatação) 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)

(TODO explicar HTML5?) 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 reconstruí a aplicação usando HTML5 Web Sockets, 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, HTML5 Web Sockets provê uma redução dramática no tráfego de dados desnecessários em relação ao método de Polling. (TODO I18n de polling solution)

(TODO Figura 3)

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 para o 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 upgraded (TODO I18n) para 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.

(TODO Figura 4)

Final do exemplo

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. (TODO rodapé: Ele não é um protocolo padrão, porém é bastante promissor e é está sendo utilizado para o desenvolvimento do protocolo HTTP 2.0)

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. (TODO rodapé: note a semelhança com WebSockets)

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/IP dedicado e agora passa a funcionar em cima de uma camada de conexão SPDY multiplexada.

O WebSocket (assim como outros protocolos e ferramentas) necessitam de atualizações de tempos em tempos, por situações inevitáveis 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.

9 Socket.io

Abaixo, é exibido o código de um chat simples como exemplo. O código está completo, nada foi omitido para explicitar a simplicidade de se fazer uma aplicação com Socket.io.

No cliente:

```
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <style>
      * { margin: 0; padding: 0; box-sizing: border-box; }
      body { font: 13px Helvetica, Arial; }
      form { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }
      form input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }
      form button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }
      #messages { list-style-type: none; margin: 0; padding: 0; }
      #messages li { padding: 5px 10px; }
      #messages li:nth-child(odd) { background: #eee; }
    </style>
  </head>
  <body>
    <ul id="messages"></ul>
    <form action="">
      <input id="m" autocomplete="off" /><button>Send</button>
    </form>
    <script src="https://cdn.socket.io/socket.io-1.2.0.js"></script>
    <script src="http://code.jquery.com/jquery-1.11.1.js"></script>
    <script>
      var socket = io();
      $('form').submit(function(){
        socket.emit('chat message', $('#m').val());
        $('#m').val('');
        return false;
      });
      socket.on('chat message', function(msg){
        $('#messages').append($('- ').text(msg));
      });
    </script>
  </body>
</html>

```

- No browser, a página HTML exhibe o histórico da conversa e um for-

mulário para o usuário escrever uma mensagem.

- Ao inserir uma mensagem, o socket envia um evento 'chat message' com a texto.

- O socket ouve um evento com o mesmo nome 'chat message'. Ao ocorrer, o conteúdo recebido é inserido na página e exibido ao usuário.

- Note que não há um evento 'connection' definido, mas ainda assim ele é disparado quando a conexão é definida, e envia seus dados de identificação.

No servidor:

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

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

io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

- No servidor, o evento 'connection' ativa uma subrotina que recebe o socket e registra nele o evento 'chat message'.

- O evento 'chat message' ativa uma subrotina que emite a mensagem recebida para todos os sockets conectados ao servidor.

Esse exemplo mostra múltiplos sockets em uso no servidor, e a definição de eventos arbitrários, ainda que semelhantes aos de WebSocket.

Por fim, é possível dizer que resolve os problemas apresentados. Ele torna possível a comunicação bilateral cliente-servidor com mínimo custo e baixa latência. Sua interface simples torna o desenvolvimento de aplicações Web rápido e direto, mas suas funcionalidades são poderosas o bastante para se criar algo consistente, robusto e, principalmente, escalável.

Socket.io é a principal ferramenta usada para fazer esse Projeto funcionar.

10 Device

Device: O aplicativo é criado usando o Cordova. Cordova é uma IDE (?) que faz o porte de programas em JavaScript para várias plataformas, como Android e iOS.

- ¿ Tiveram mil problemas até o Cordova ser escolhido.

- O device se conecta ao servidor utilizando Socket.io

- Estruturação de um código Web (JavaScript, CSS, HTML5)

11 Server

Server: Usa socket.io para fazer a comunicação com os clients. ¿ Usa node.js para something

12 Browser

Browser: Primeiro client, lançado pelo server; Recebe dados por Socket; Não envia dados;

Visualização de dados; Visualização 3D: Usa Three.js (Biblioteca 3D para js); Usa os dados recebidos para alteração de objetos 3D.

Foram feitas duas páginas de Three.js: 1) Um simples objeto é rotacionado e tem sua cor alterada de acordo com dados recebidos.

2) É a aplicação de um algoritmo de ruído para geração procedural de terreno em um grid. Utiliza os dados recebidos para alterar os parâmetros do algoritmo. No teste inicial ele altera em tempo real o grid, de forma que o envio constante de dados cria um movimento também constante.

- ¿ Talvez possa fazer um teste de performance, e rodar o algoritmo em um grid muito maior.

13 Parte Subjetiva

13.1 Desafios e frustrações

13.2 A contribuição do curso de Computação

13.3 Próximos Passos

13.4 Agradecimentos

Evolução do Projeto:

Esse projeto utiliza várias tecnologias que eu nunca tive contato.

Inicialmente a ideia inicial do TCC era usar um Arduino e um conjunto de sensores para coletar dados do meio e mandar de maneira síncrona para um servidor. O servidor então usaria esses dados para fazer algum controle. Numa primeira discussão com o orientador, a ideia foi rapidamente descartada, pelos seguintes motivos:

Os sensores dele normalmente são imprecisos; A montagem de um dispositivo como o idealizado seria desnecessariamente custosa e muito propensa a erros; Sem uma montagem muito bem executada, o dispositivo ficaria muito frágil; É possível conseguir muitas leituras de sensores utilizando um smartphone.

Com os argumentos apresentados, a decisão foi criar um aplicativo para Android com o mesmo propósito. Todos os argumentos apresentados são resolvidos com essa solução. A implementação é extremamente mais simples; os sensores são muito mais precisos; a estrutura física já está pronta e é um objeto que já está presente em todo o mundo (o produto final atinge grande parte da população); e, por fim, é muito viável a adição ou remoção de funcionalidades, além de muito mais suporte para a plataforma.

Uma vez que a decisão de utilizar um smartphone, foi necessário saber como implementar um aplicativo que tenha acesso aos sensores. A primeira ideia foi utilizar serviço MIT App Inventor.

14 Tecnologias utilizadas

14.1 Node.js

15 Tecnologias citadas

15.1 MIT App Inventor

É um serviço disponibilizado pelo MIT para a criação de aplicativos. Ele é extremamente didático e inclusivo, sua interface não é dada por linhas de código, e sim por blocos lógicos que se encaixam e formam um algoritmo (na prática, é bastante ruim não poder escrever linhas de código livremente). Sua API possui interface para o uso de sensores, além de outras funcionalidades que não foram exploradas neste TCC, como acesso à ferramenta de reconhecimento de voz e ferramentas sociais, como email, mensagem e Twitter. O serviço é bom, porém possui várias restrições, então é difícil de ser usado.

Com certa dificuldade um primeiro aplicativo de teste foi feito, para pegar valores do giroscópio. O próximo passo seria criar um servidor e estabele-

cer uma conexão entre ambos, porém as opções de conectividade do App Inventor também são muito restritas, então, dada a dificuldade prevista, eu achei melhor deixar essa plataforma de lado no momento e procurar outras alternativas para a criação de um app.

15.2 Kivy

É um framework Open Source de Python. Seu objetivo é o desenvolvimento rápido de aplicações multiplataforma que fazem o uso de interfaces inovadoras, como telas com Multitouch e sensores de movimento. Sua API, por exemplo, lida com eventos de mouse, teclado e toques de tela. Ele também possui um foco na criação de apps com NUI (Natural User Interface). NUI é uma metodologia de interface cuja proposta é ser invisível ao usuário, e apresentar uma experiência natural e intuitiva. Seu principal foco é evitar grandes barreiras durante o aprendizado. Através de decisões de design, o usuário tem um entendimento do software sem grandes dificuldades, à medida que a complexidade da interação aumenta.

[TODO] Para exemplificar a facilidade de desenvolvimento de lógica e de uma interface gráfica, abaixo tem o código para se criar um jogo de Pong.

15.3 Cordova

16 Bibliografia