Universidade Federal do Piauí
Campus Senador Helvídio Nunes de Barros
Curso de Bacharelado em Sistemas de Informação

Aislan de Sousa Maia

Pesquisa e Aprendizagem Sobre a Web em Tempo Real e Uma Aplicação Protótipo

Picos
2014

Aislan de Sousa Maia

Pesquisa e Aprendizagem Sobre a Web em Tempo Real e Uma Aplicação Protótipo

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação do Campus Senador Helvídio Nunes de Barros na Universidade Federal do Pìauí como parte dos requisitos para obtenção do Grau de Bacharel em Sistemas de Informação, sob orientação do professor Ivenilton Alexandre de Souza Moura.

Picos

2014

Eu, **Aislan de Sousa Maia,** abaixo identificado(a) como autor(a), autorizo a biblioteca da Universidade Federal do Piauí a divulgar, gratuitamente, sem ressarcimento de direitos autorais, o texto integral da publicação abaixo discriminada, de minha autoria, em seu site, em formato PDF, para fins de leitura e/ou impressão, a partir da data de hoje.

Picos-PI, 14 de agosto de 2014.

Aislan de Sousa Maia
Assinatura

FICHA CATALOGRÁFICA
Serviço de Processamento Técnico da Universidade Federal do Piauí
Biblioteca José Albano de Macêdo

M217p Maia, Aislan de Sousa.
Pesquisa e aprendizagem sobre a web em tempo real e uma aplicação protótipo / Aislan de Sousa Maia. - 2014.
CD-ROM : il. ; 4 ¾ pol. (121 p.)

Monografia(Bacharelado em Sistemas de Informação) – Universidade Federal do Piauí. Picos-PI, 2014.
Orientador(A): Prof. Esp. Ivenilton Alexandre de Souza Molura

1. Web. 2. Tempo Real. 3. Node.js. 4. Meteon IS. I. Título.

CDD 005.3

Aislan de Sousa Maia

Pesquisa e Aprendizagem Sobre a Web em Tempo Real e Uma Aplicação Protótipo

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação do Campus Senador Helvídio Nunes de Barros na Universidade Federal do Piauí como parte dos requisitos para obtenção do Grau de Bacharel em Sistemas de Informação, sob orientação do professor Ivenilton Alexandre de Souza Moura.

Data de Aprovação:

04 de agosto de 2014

Ivenilton Alexandre de Souza Moura _____ UFPI

Romuere Rodrigues Veloso e Silva _____ UFPI

Flávio Duarte de Araújo _____ UFPI

Picos

2014

Dedico este trabalho em primeiro lugar a Deus, que me trouxe até aqui através de sua bondade e misericórdia. Em segundo lugar a minha mãe e meus avós, que sempre me amaram e me apoiaram desde antes de eu nascer. Dedico a meu padrasto Raimundo e toda a minha família que sempre esteve comigo em todos os momentos. A vocês eu devo isto. E por fim, a você leitor que irá poder desfrutar de meu esforço em compartilhar alguma coisa que aprendi, ainda que pouca.

Agradeço a meu Deus, minha família e meus amigos, e em destaque meu tio César, a qual devo minha inspiração e vocação, me ensinando desde muito tempo a encontrar gosto pela tecnologia e informática. Muito Obrigado!

Diante da noite, não acuse as trevas. Aprenda a fazer lume.

Em vão condenará você o pântano. Ajude-o a purificar-se.

No caminho pedregoso, não atire calhaus nos outros. Transforme os calhaus em obras úteis.

Não amaldiçoe o vozerio alheio. Ensine alguma lição proveitosa, com o silêncio.

Não adote a incerteza, perante as situações difíceis. Enfrente-as com a consciência limpa.

Debalde censurará você o espinheiro. Remova-o com bondade.

Não critique o terreno sáfaro. Ao invés disso, dê- lhe adubo.

Não pronuncie más palavras contra o deserto. Auxilie a cavar um poço sob a areia escaldante.

Não é vantagem desaprovar onde todos desaprovaram. Ampare o seu irmão com a boa palavra.

É sempre fácil observar o mal e identificá-lo. Entretanto, o que o Cristo espera de nós outros é a descoberta e o cultivo do bem para que o Divino Amor seja glorificado.

André Luiz

Resumo

Uma nova tendência está surgindo no mercado *Web*, que muitos estão chamando de *Web* em tempo real, abrindo novas possibilidades para a criação de soluções integradas e rápidas que antes não poderíamos imaginar. Esta nova tecnologia pode oferecer conteúdo aos seus usuários em tempo real, economizando o tempo que normalmente desperdiçamos e evitando problemas por não termos informação disponível em tempo hábil. Este trabalho consiste de uma pesquisa a cerca das novas metodologias de desenvolvimento *Web*, que estão tornando possível a grande comunidade de desenvolvedores entregarem conteúdo para seus usuários em tempo real. A pesquisa se divide em duas partes: primeiro é feito um levantamento a cerca do surgimento e evolução dos métodos de desenvolvimento *Web*, destacando as principais ferramentas, e em segundo é posto em prática alguns métodos através das tecnologias *Node.js*, *MeteorJS* e seus componentes, sendo desenvolvido uma aplicação de exemplo e detalhado suas funcionalidades em tempo real, além de realizado um teste comparativo entre uma aplicação *chat* em tempo real com *Node.js* e outra com *AJAX polling*.

Palavras-chave: Web, tempo real, Node.js, MeteorJS.

Abstract

A new trend is emerging in the Web market, which many are calling real time Web, opening new possibilities for the creation of integrated, timely solutions that we could not imagine before. This new technology can deliver content to your users in real time, saving the time typically wasted and avoiding problems by not having information available in a timely manner. This paper consists of a survey about the new web development methodologies that are making possible the large community of developers deliver content to your users in real time. The research is divided into two parts: first is a survey about the emergence and evolution of web development methods, highlighting key tools, and second it is put into practice some methods through Node.js, MeteorJS and its component technologies, and developed a sample application and detailed its functionality in real time, and carry out a comparative test between a live chat application with Node.js and another with AJAX polling.

Keywords: Web, real-time, Node.js, MeteorJS.

Lista de Figuras

Lista de abreviaturas e siglas

AJAX	*Asynchronous Javascript and XML*
API	*Application Programming Interface*
CSS	*Cascade Style Sheet*
DDP	*Distributed Data Protocol*
FTP	*File Transfer Protocol*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
ICE	*Interactive Connectivity Establishment*
IETF	*Internet Engineering Task Force*
IMAP	*Internet Message Access Protocol*
JDBC	*Java Database Connectivity*
JSON	*Javascript Object Notation*
LDAP	*Lightweight Directory Access Protocol*
NAT	*Network Address Translation*
POP	*Post Office Protocol*
RTP	*Real-Time Transfer Protocol*
SDP	*Session Description Protocol*
SMTP	*Simple Mail Transfer Protocol*
SOAP	*Simple Object Access Protocol*
SRTP	*Secure Real-Time Transfer Protocol*
SSE	*Server Side Events*
STUN	*Session Traversal Utilities for NAT*
TCP	*Transmission Control Protocol*
TURN	*Traversal Using Relays around NAT*
UDP	*Use Datagram Protocol*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
VoIP	Voz Sobre *IP*
WebRTC	*Web Real-Time Communication*
XML	*eXtensible Markup Language*

Sumário

1 Introdução

Conforme a *Web* foi se desenvolvendo, mais pessoas a utilizavam e, dentro deste contexto, as grandes empresas começaram a perceber novas oportunidades de negócio dentro da grande rede. A *Web* passou a se tornar uma grande plataforma, na qual serviços de diversas naturezas começaram a ser oferecidos ao público.

Os primeiros *sites* desenvolvidos eram totalmente estáticos, com conteúdo pré-definido e com pouquíssimo processamento de dados no *backend* do servidor da página. No entanto, através de muitas pesquisas na área, soluções tecnológicas foram sendo criadas, como por exemplo o *AJAX*, novas versões e melhorias no *HTML* e *CSS*, utilização de bancos de dados e processamento de conteúdo através de diversas linguagens como o *Java, Python, PHP*, entre outros, que possibilitaram os *sites* entregarem conteúdo dinâmico para os clientes.

Apesar do estágio evolutivo atingido, o que muitos chamaram de *Web 2.0*, as aplicações *Web* ainda careciam de uma maneira de oferecer conteúdo sempre atualizado sem a necessidade do navegador solicitar isto ao servidor. Com o *AJAX* e técnicas como *long pooling, reverse ajax, ajax push, HTTP Streaming, HTTP server push*, que ficaram conhecidas pelo nome *Comet*, isto pode ser contornado, mas de maneira paliativa. As soluções apresentadas causavam efeitos colaterais, como por exemplo *overhead* no cabeçalho *HTTP*, resultando em aumento da latência na conexão, o que pode ser muito prejudicial em aplicações críticas em tempo real e prejudicar também a experiência de uso do usuário. (LUBBERS; GRECO, s.d.)

Novas tecnologias aparecem com o tempo, e uma em especial já desponta com grande destaque nos últimos dois anos, a *Node.js*. criada por Ryan Dahl. *Node.js* é uma plataforma construída sob o interpretador *V8* (criada pela *Google* para o navegador *Web Google Chrome*) para facilmente desenvolver velozes e escaláveis aplicações em rede e em tempo real. Ela tem despertado bastante interesse na comunidade de desenvolvedores por usar a linguagem *Javascript* no lado servidor, um paradigma de programação orientada a eventos, chamadas de retorno assíncronas e um modelo não-bloqueante para operações de entrada e saída. Isto torna possível desenvolver aplicações leves e eficientes, que escalam muito bem e que, por isso, são perfeitas para aplicações em tempo real e que fazem uso intensivo de troca de dados. (NODEJS.ORG, 2014)

Hoje em dia se torna cada vez mais importante oferecer uma experiência de uso agradável ao usuário, e fazer o usuário esperar por informação que já deveria estar disponível alguns minutos atrás pode levar o usuário a abandonar o serviço prestado. É pensando nisso que cada vez mais vemos sites oferecendo, ou tentando pelo menos, oferecer conteúdo em tempo real

para seus usuários.

As aplicações em tempo real estão mudando a forma como recebemos e percebemos conteúdo. Elas abrem novas possibilidades para negócios e entretenimento. Por exemplo, quando navegarmos em um site ou aplicativo, ou então fazermos uma pesquisa, não será mais preciso solicitar à aplicação por novos conteúdos disponíveis, ela automaticamente se encarregará de oferecer e tornar disponível para visualização as informações de nosso interesse e no momento em que estas forem produzidas.

Este trabalho consiste de uma pesquisa e aprendizagem a cerca das novas metodologias de desenvolvimento *Web* e tecnologias *Javascript* que possibilitam a grande comunidade de desenvolvedores entregarem conteúdo para seus usuários em tempo real. A pesquisa se dividiu em duas partes: levantamento a cerca do surgimento e evolução dos métodos de desenvolvimento *Web*, destacando as principais ferramentas, e prática de alguns métodos de desenvolvimento através das tecnologias *Node.js*, *MeteorJS*, e seus componentes, sendo desenvolvido uma rede social como aplicação de exemplo, descrevendo o passo a passo de suas funcionalidades em tempo real, além de realizado um teste comparativo entre uma aplicação chat em tempo real com *Node.js* e outra com *AJAX polling*.

A meta deste trabalho é a de servir de referência bibliográfica para a comunidade acadêmica, fornecendo visões, conceitos e análise de casos de uso da *Web* em tempo real, além de demonstrar na prática o uso de algumas de suas tecnologias. A *Web* em tempo real continua em evolução com muitas tecnologias ainda evoluindo e outras surgindo. Dentro deste contexto, este trabalho não pretende ser uma referência exaustiva, mas servir como ponte para um novo conhecimento para estudante a cerca do assunto.

O trabalho se encontra com a seguinte divisão de capítulos:

- Capítulo 2 – A *Web* em evolução: neste capítulo é contextualizado o histórico da *Web*, suas designações iniciais e paradigma; o surgimento da chamada *Web 2.0* com a inovadora tecnologia *AJAX*; as diversas técnicas *AJAX*, sua estrutura, princípios e padrões, e suas vantagens e desvantagens;

- Capítulo 3 – A *Web* em tempo real: este capítulo começa a introduzir a *Web* em tempo real, trazendo opiniões e conceitos de diversas personalidades, além de contextualizar o mercado. Logo em seguida, é demonstrado um estudo sobre diversas tecnologias que fazem parte da Web em tempo real, como os protocolos *WebSocket, SSE, SPDY, WebRTC* e a nova versão do *HTTP*, a 2.0;

- Capítulo 4 – Contextualizando a *Web* em tempo real: neste capítulo o trabalho foca em contextualizar e sintetizar casos de uso da indústria na *Web*, e que corroboram para demonstrar os benefícios, mudanças e oportunidades obtidas com a adoção de diversas tec-

nologias que vêm permitindo a oferta para as demandas atuais, conteúdos e serviços em tempo real;

- Capítulo 5 – Estudo Prático: este capítulo aborda as implementações e estudos práticos feitos sobre o desenvolvimento de aplicações *Web* em tempo real. Aqui são estudadas as plataformas Node.js e MeteorJS, e junto a este estudo, detalhado passo a passo as funcionalidades em tempo real de uma aplicação de exemplo desenvolvida para este trabalho;
- Capítulo 6 – Um Teste Comparativo - neste capítulo é detalhado um teste comparativo realizado entre a aplicação de *chat* em tempo real, desenvolvida com *Node.js* e uma outra aplicação de *chat* desenvolvida com a técnica *AJAX polling*;
- Capítulo 7 – Conclusões: este capítulo apresenta as considerações finais a cerca de toda a pesquisa e possíveis trabalhos futuros de desenvolvimento na área da Web em tempo real.

2 A *Web* em evolução

A *Web*, constituída como um meio de compartilhar informações sobre a *Internet*, teve em seus primeiros dias seu uso baseado somente em conteúdos estáticos. Um sítio da *Web* nada mais era do que uma estrutura composta de texto e *hiperlinks*, com cada *hiperlink* ligando uma página a outra. Segundo Lengstorf e Leggetter (2013), o objetivo primário de um sítio *Web* era exibir seu conteúdo, tendo a ideia "conteúdo é o Rei". Ele acrescenta também que mesmo quando a *Web* viera a alcançar tecnologias para criar conteúdos dinâmicos, isto ainda significaria que o servidor poderia dinamicamente gerar conteúdo estático, baseado em diferentes, mas definidos valores. A aplicação que utilizamos para navegar na *Web*, conhecida como *Web Browser* (navegador *Web*), era focada em atender somente aquilo que a *Web* era capaz de entregar, renderizando *HTML*, imagens e fazendo o vínculo de *hiperlinks* entre páginas.

Neste passo, a *Web* evoluiu e passou a apresentar interfaces mais ricas com *CSS* (estilos de folha em cascata) e o *Javascript*, linguagem de programação que inicialmente fora utilizada no cliente (*Web Browser*) para manipular a interface em tempo de execução.

2.1 A *Web* 2.0

Com o mercado aquecido, e a *Internet* cada vez mais sendo utilizada, novas demandas de entretenimento e negócios apareceram, e junto com elas, novas tecnologias surgiram para atender a este crescimento. Era preciso ir além dos formulários, textos e imagens estáticas. Foi dentro deste contexto que os *Java Applets* surgiram. Por volta de 1995, numa parceria entre a *Sun* (empresa criadora da linguagem *Java*) e a *Netscape*, o navegador *Web Netscape* veio ao mercado com o *Java runtime* (*Java Applets*). Como Rai (2013) nos relata, isto foi o início da *Web* altamente interativa. Apesar de problemas com esta tecnologia, é considerada como a pioneira no campo da *Web* em tempo real, sendo esta tecnologia utilizada em vários tipos de aplicações, como *chats, games* e até anúncios em *banners*. Após algum tempo surge a linguagem *Javascript*, com suporte pelo navegador Netscape e também o *software* de animação *FutureSplash Animator*, da empresa *FutureWave Software*. Esta empresa veio a ser adquirida pela *Macromedia*, em 1996, renomeando em seguida o nome do software de animação para *Flash*. Neste tempo a *Web* foi dominada por estas duas tecnologias, tomando muito espaço dos *Java Applets*.

O *Flash* torna-se a plataforma mais largamente utilizada para a criação de diversos tipos de aplicações, como jogos, animações, tocadores de vídeo, anúncios, entre outros, até que, em

1999, a *Microsoft* vem ao mercado com a tecnologia *iframe*, lançando no mesmo ano uma extensão *ActiveX* do navegador *Internet Explorer* conhecida como *XMLHTTP*. Conforme Rai (2013) aborda, a tecnologia *XML* era o que todo mundo no momento queria utilizar para construir interatividade. Outros navegadores como o *Firefox, Safari*, e *Opera* adotaram o componente *XMLHTTP* passando a chamá-lo *XMLHttpRequest*. Esta tecnologia permitiu o carregamento de partes de uma página *Web* sem que fosse necessário atualizar a página completa. Foi o surgimento do termo *AJAX* na comunidade de desenvolvedores *Web* e o apelido utilizado nas mídias de imprensa para tudo o que girava em torno disto, a *Web 2.0*.

2.2 *AJAX - Asynchronous Javascript and XML*

Para Synodinos (2008), contribuições para aplicações *online* mais interativas devem ser creditadas a equipe de engenheiros no projeto *Microsoft Exchange*. Utilizando o elemento IFrame no desenvolvimento do aparência de seu sistema de serviço de *e-mail* a equipe buscava uma aparência similar a do programa *Outlook* e apesar de problemas de responsividade e experiência de usuário neste projeto, a equipe continuou suas pesquisas até a criação de um componente chamado *XMLHTTP* que permitia comunicação assíncrona com o servidor. Mais tarde, em 2002, a fundação *Mozilla* implementou esta tecnologia como *XMLHttpRequest* na primeira versão de seu navegador *Web*, que mais tarde viria a ser o conhecido *Firefox*.

Uma técnica conhecida pelo acrônimo *AJAX* (*Asynchronous Javascript and XML*, em português *Javascript* e *XML* Assíncronos), fazendo uso de um objeto *XMLHttpRequest* contribuiu para mudar um pouco o paradigma de solicitação / resposta dos sítios *Web* permitindo solicitações assíncronas ao servidor, ou seja, com o *Javascript*, o navegador é capaz de solicitar pequenas porções de uma página ao servidor e atualizá-la sem ser necessário recarregar a página totalmente, tornando as interações mais rápidas e a experiência de uso melhorada para o usuário. Esta técnica passou a ganhar realmente atenção com o lançamento da *Web App Gmail*, pela *Google*, que além e atualizar porções da página sem requerer atualização da página, utilizou esta mesma técnica em seu módulo de *chat*. O *AJAX* ajudou bastante a *Web* a se tornar mais interativa e dinâmica, mas apesar disto, uma questão ainda podia ser levantada que é a de que toda a comunicação *HTTP* ainda continuava sendo direcionada pelo cliente (navegador), exigindo interação do usuário ou sondagem periódica para carregamento de novos dados do servidor (UBL; KITAMURA, 2010).

2.2.1 Como Funciona a *Web*

Antes de entender como o *AJAX* funciona, foi necessário levantar como a *Web* funciona e o mecanismo de interação entre o cliente e o servidor. Para este entendimento, Tanenbaum

(2003) nos ensina que ao digitarmos o endereço de um site, o navegador busca através do *DNS* o endereço *IP* da *URL* que foi digitada, o *DNS* responde com o endereço *IP* do servidor em que a página está hospedada, o navegador estabelece uma conexão *TCP* com a porta 80 com o servidor e envia um comando de solicitação *HTTP* para o arquivo da página inicial do site. O servidor, por sua vez, obtém o nome do arquivo solicitado, faz uma busca pelo arquivo no seu disco e envia o arquivo de volta através do *HTTP* para o navegador, encerrando por fim a conexão *TCP*. Este é o ciclo inicial, porém após isso nada acontecerá até que o usuário clique em algum *link* ou interaja com o site de alguma maneira, fazendo com que o navegador faça uma nova solicitação ao servidor. Isto nos remete ao ciclo do funcionamento da *Web* que podemos chamar de solicitação / resposta.

Ainda segundo Tanenbaum (2003), do lado cliente, ou seja, do lado de quem está navegando na *Web*, esta seria uma vasta coleção de documentos, páginas, com cada qual contendo vínculos para outras páginas, partindo da ideia de uma página apontando para outra, chamada de hipertexto. O usuário então, de auxílio de um programa navegador como o *Firefox, Google Chrome*, ou *Internet Explorer*, busca a página solicitada, através de um endereço específico. O navegador então interpreta o texto da página e seus comandos de formatação, exibindo a página na tela do computador.

Conforme Holdener (2008), a cada requisição do usuário por dados, o cliente deveria consultar o servidor, esperando-o retornar com uma resposta, sendo esta na forma de uma página totalmente nova, forçando o navegador *Web* recarregá-la, junto com todos os seus componentes de imagens, folhas de estilo, scripts, entre outros. As Figuras 1 e 2 demonstram o vínculo que existe entre as diversas páginas na *Web* com o lado cliente e lado servidor, e o ciclo solicitação e resposta que ocorre entre eles.

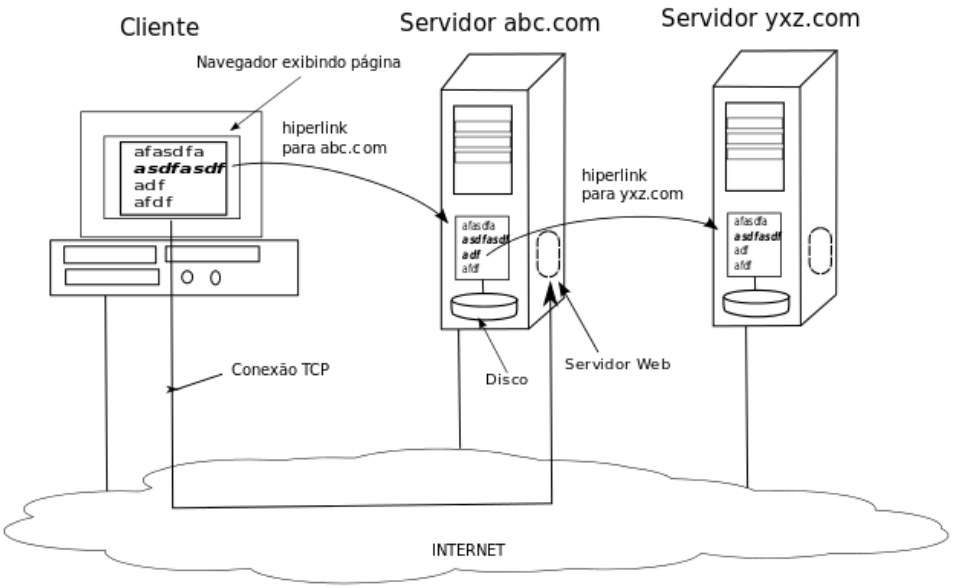

Figura 1 *– Vínculo entre páginas*

Adaptado de III

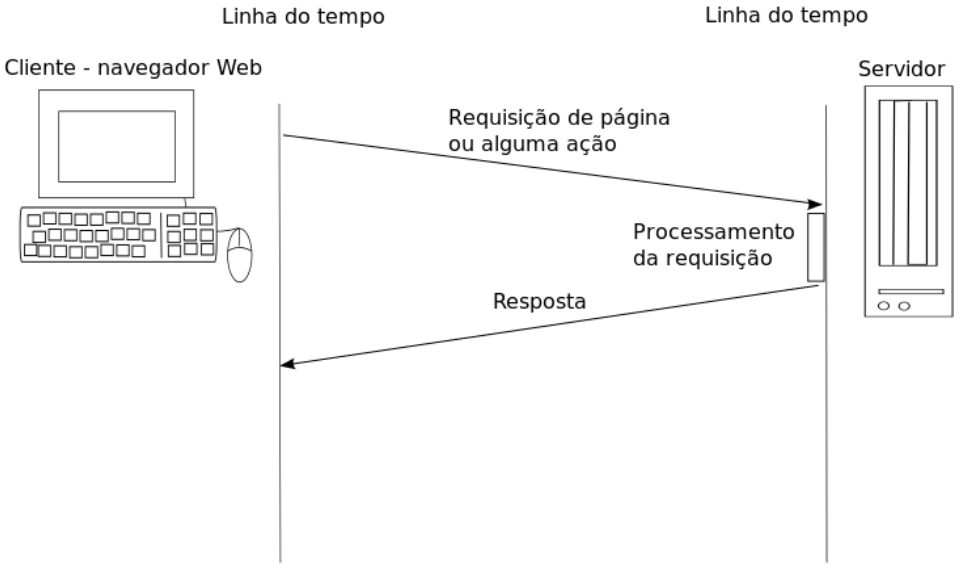

Figura 2 *– Ciclo Solicitação - Resposta*

Adaptado de III

Adiante, Wang et al. (2013) demonstra-nos também o funcionamento das antigas arquiteturas *HTTP*, que sobre qual grande parte da *Web* funciona. Segundo ela, o *HTTP* é um protocolo para requisição e resposta, dentro de um modelo cliente / servidor. O cliente submete então, na realidade, uma requisição *HTTP* para o servidor, e o servidor responde com recursos como uma página HTML e informações adicionais sobre a página e resposta. O *HTTP*, con-

forme Wang et al. (2013), foi também desenvolvido para busca de documentos. Na sua versão 1.0, atendia a um documento por requisição e era necessário abrir uma conexão para cada requisição para o servidor, gerando problemas de escalamento de recursos. Porém, a partir do crescimento da *Web* para além de busca, trocas de documentos e aumento de demandas por mais interatividade, o *HTTP* passou a ser refinado para garantir mais conectividade e respostas mais rápidas entre as requisições e respostas.

Em sua versão 1.1, o *HTTP* resolveu seu problema de uma conexão por uma requisição, e passou a reutilizar conexões já abertas. Em outras palavras, os clientes (navegadores) poderiam iniciar uma conexão com o servidor para requisitar alguma coisa, e, posteriormente, reutilizar a mesma conexão aberta para futuras solicitações, reduzindo latência e tráfego comparado à sua primeira versão. Wang et al. (2013) também esclarece que o *HTTP* é considerado um protocolo stateless, ou, melhor dizendo, é um protocolo que trata cada requisição como única e independente. Ele esquece sobre dados sobre a última requisição feita, ou seja, não mantém estado armazenado. Segundo ela, há vantagens em tratar as requisições desta maneira: um exemplo é que o servidor não precisa manter informações sobre sessões, não requisitando armazenagem de dados. Por outro lado, para cada requisição e resposta, há a necessidade de redundância de informações que o protocolo já poderia conhecer, caso mantesse informações. Nas Figuras 3 e 4 um exemplo do cabeçalho de uma requisição com o *HTTP* 1.1 enviado de um cliente para o servidor e da resposta do servidor ao cliente.

```
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://localhost:8080/PollingStock/
Cookie: showInheritedConstant=false; showInheritedProtectedConst
ant=false; showInheritedProperty=false; showInheritedProtectedPr
operty=false; showInheritedMethod=false; showInheritedProtectedM
ethod=false; showInheritedEvent=false; showInheritedStyle=false;
showInheritedEffect=false;
```

Figura 3 *– Cabeçalho de Requisição do Cliente*

Fonte: (WANG et al., 2013)

```
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: 321
Date: Wed, 06 Dec 2012 00:32:46 GMT
```

Figura 4 *– Cabeçalho de Resposta do Servidor*

Fonte: (WANG et al., 2013)

Wang et al. (2013) logo esclarece muito bem sobre os cabeçalhos enviados através da rede na ida e vinda do ciclo requisição / resposta. Nestes dois exemplos, temos um overhead (sobrecarga), de 871 bytes, e isto somente para informações de cabeçalho, ou seja, a cada iteração entre cliente e servidor, são adicionados esta quantidade de informação para a mensagem (solicitação ou resposta), não importando se o servidor tem ou não os dados que o cliente pediu.

Há dois problemas no cenário acima: primeiro que, conforme Wang et al. (2013), o protocolo *HTTP* foi designado para troca de documentos e não para as ricas e interativas aplicações que utilizamos no Desktop ou em aplicações modernas da *Web* atuais; segundo que, quanto mais interações entre clientes e servidor, mais informação é acrescentada pelo protocolo *HTTP* na comunicação.

Outra característica do *HTTP* é que ele funciona com conexões half duplex. Isto significa tráfego fluindo em uma única direção por vez. Para Wang et al. (2013), o *HTTP*, ao funcionar desta forma, é simplesmente ineficiente. Um exemplo bem claro disto seria o clássico exemplo da conversa telefônica: a cada vez que quiséssemos conversar com alguém do outro lado da linha, teríamos que enviar uma mensagem, esperar a mensagem chegar do outro lado, para só então nosso receptor enviar sua resposta, sem que pudéssemos enviar outra mensagem neste tempo, já que o caminho da mensagem já está sendo ocupado. Para esta ineficiência, há tentativas de soluções através de *AJAX* e técnicas como *HTTP streaming, long polling*, e *polling*, bem como afirma Wang et al. (2013), e que foram também objetos de pesquisa deste trabalho.

2.2.2 *AJAX* - Estrutura e Padrões

Holdener (2008) divide a *Web* em dois tempos: a clássica e a moderna. A clássica se beneficiava de somente duas ferramentas, a *HyperText Markup Language* (*HTML*) e o *HyperText Transfer Protocol* (*HTTP*), na qual o HTML formatava os textos das páginas e o *HTTP* as

transmitia. Não havia separação de apresentação, ou seja, o que é exibido, da estrutura de uma página e o servidor e cliente se comunicavam basicamente através de chamadas *HTTP GET* e *HTTP POST*. Enquanto isso, a era moderna da *Web* se daria somente após o surgimento da tecnologia *AJAX*, através de manipulações dinamicamente feitas ao *DOM* (*Document Object Model*), o qual descreve todos os elementos nós do *HTML* de uma página, com a linguagem de programação *Javascript*. A estrutura de aplicações *Web* utilizando *AJAX* se dividiria então da seguinte forma, segundo Holdener (2008):

- A linguagem de marcação *XHTML* (Extensible HyperText Markup Language);
- O *Document Object Model* (*DOM*);
- A linguagem de programação de uso geral, *Javascript*;
- As folhas de estilo *CSS* (*Cascading Style SheetsCSS*);
- e a linguagem de marcação *XML* (*Extensible Markup Language*).

Ele detalha que a linguagem de marcação *XHTML* é uma extensão da conhecida *HTML*, sendo responsável pela exibição do conteúdo, com o *DOM* sendo utilizado para navegação dos elementos contidos no *XHTML*, e o *Javascript* acessando-o diretamente. O *Javascript* é, então, a parte mais essencial da tecnologia *AJAX*, pois é o responsável por criar comunicação entre o servidor e o cliente. O *CSS* é utilizado para dar aparência à página, sendo também manipulado dinamicamente através do *DOM*, e o *XML* é utilizado como protocolo de transferência de dados entre cliente e servidor. Seguindo esta estrutura, Riordan (2008) também acrescenta mais dois componentes: o *JSON* (*Javascript Object Notation*), uma estrutura de objeto *Javascript*, e o *XMLHttpRequest*, objeto de requisição assíncrono.

Entretanto, Riordan (2008) revela que muitas aplicações *Web* que utilizam *AJAX* não seguem a estrutura de ferramentas anteriormente explanadas, e que por isso faz mais sentido não atrelar o termo *AJAX* para alguma tecnologia ou ferramenta, mas para a maneira de construir aplicações *Web* que sejam responsivas e interativas como as aplicações *Desktop*. Isso quer dizer que nem todas as aplicações *AJAX* utilizam o objeto *XMLHttpRequest*.

Holdener (2008) defende que aplicações *Web* baseadas em tecnologia *AJAX* sigam um padrão diferenciado de ciclo de desenvolvimento do que o adotado mais frequentemente na indústria de desenvolvimento de software. Este ciclo é apresentado na Figura 5.

Figura 5 *– Ciclo de Vida de Desenvolvimento da Aplicação Web Ajax*

Fonte: (HOLDENER, 2008)

As aplicações *AJAX* são concebidas também em *design patterns* (padrões de *design*) bem conhecidos. O padrão de *design* ou arquitetura mais antigo e conhecido é o Cliente / Servidor. Neste *design* o navegador (cliente) é o agente ativo, sempre enviando requisições e esperando respostas do servidor, que por sua vez, faz um papel passivo, só fazendo algo quando lhe é requisitado pelo cliente. Este padrão começou a evoluir com o surgimento de formulários e scripting no lado servidor, tornando possível entrega de dados de forma mais dinâmica. A Figura 6 a seguir exemplifica esta arquitetura:

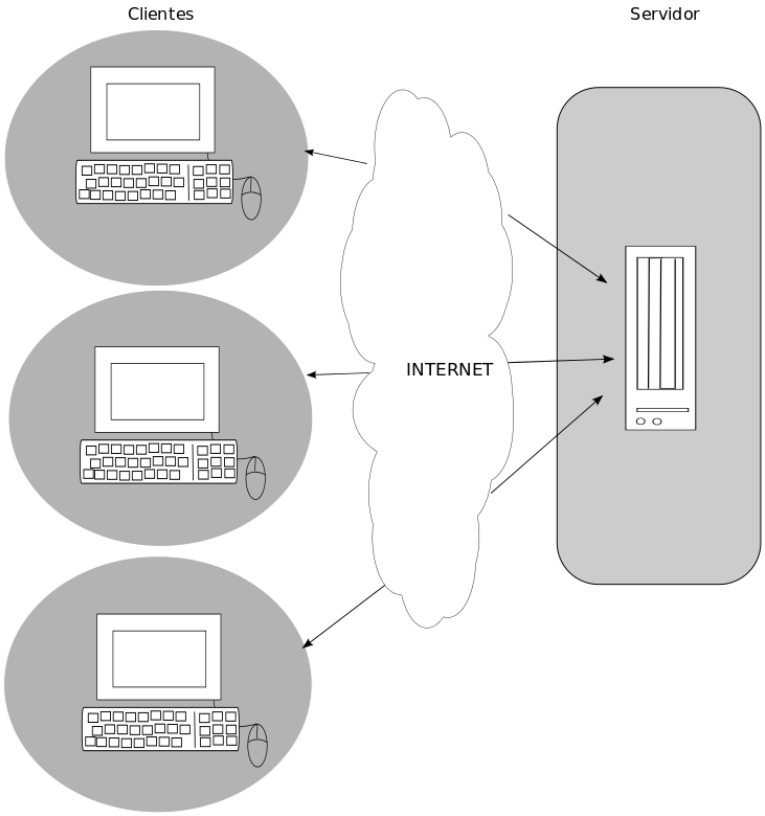

Figura 6 *– Arquitetura Cliente - Servidor para Web*

Adaptado de (HOLDENER, 2008)

Uma evolução ou derivação desta arquitetura é a arquitetura de três camadas. Esta surgiu após a aparição dos bancos de dados e seus servidores, modificando um pouco o padrão de *design* cliente / servidor. Desta vez, o cliente requisita ao servidor algum dado ou processamento, o servidor então pode, no caso de requisição por algum dado, fazer uma requisição ao servidor de banco de dados. A Figura 7 demonstra este padrão:

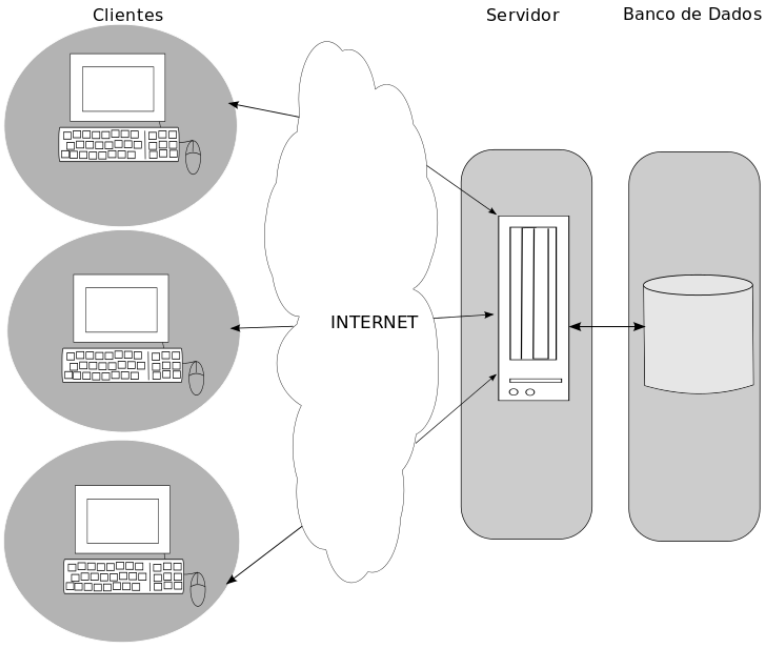

Figura 7 *– Arquitetura de Três Camadas*

Adaptado de (HOLDENER, 2008)

Holdener (2008) explica que este tipo de *design* se divide em três partes distintas: a interface de usuário, as regras de negócio ou lógica de processos, e um módulo de acesso aos dados. Ele também relata que este padrão de *design* possui o benefício de poder se estender a outras arquiteturas do tipo multi-camada, mas com a vantagem de que cada camada não exerça tanta influência sobre a outra, o que nos remete a outras vantagens, como bomo escalonamento e desacoplamento de camadas.

Ele aponta ainda uma outra evolução deste padrão de *design*, desde que este é de fácil escalonamento, evoluiu ou derivou para o padrão de *design* Modelo-Visão-Controlador, também chamado de arquitetura *MVC*. Neste arquitetura, o cliente, o servidor, e a base de dados ainda se separam em três camadas distintas, porém o que lhe diferencia é que ele é uma forma de organizar o código da aplicação em três camadas. O usuário final interage com a interface através do navegador, o controlador se responsabiliza pelo tratamento de eventos que acontecem na interface de usuário e também se responsabiliza de invocar o modelo e atualizar a visão de acordo com a ação tomada pelo usuário final. A visão, por sua vez, recebe os dados vindos do modelo, que nesta arquitetura é o módulo de acesso aos dados, e atualiza a interface de acordo com o que recebe, até que aconteça uma nova entrada ou ação por parte de usuário e o ciclo de

funcionamento comece novamente.

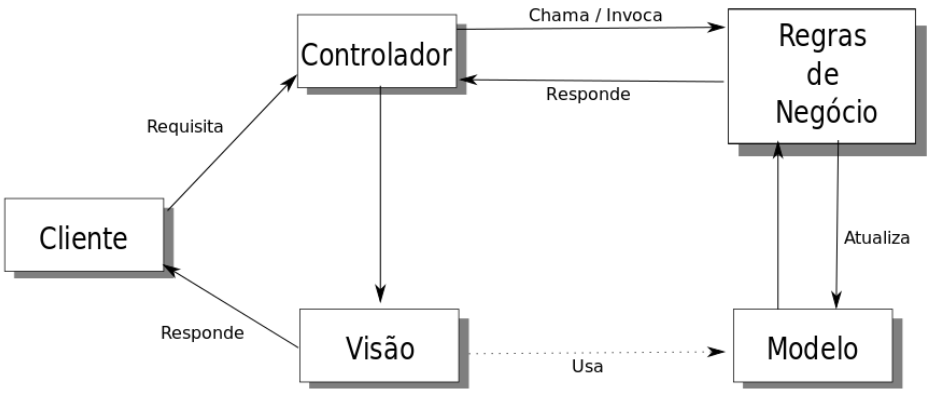

Figura 8 *– Arquitetura MVC*

Adaptado de (HOLDENER, 2008)

Podemos resumir este modelo em funções que cada módulo desta arquitetura. Voltado para o desenvolvimento *Web*, a visão se encarregaria em suma de construir o *XHTML* na presença de uma requisição do usuário. O controlador seria então todo o código de navegação da aplicação, podendo ser *scripts* do lado cliente e ou do lado servidor. O modelo, por sua vez, é o módulo de acesso aos dados, o qual trata as requisições por dados do controlador e as regras de negócio. Entretanto, para aplicações *Web* com tecnologias *AJAX*, há casos em que a visão pode responder somente com *XML* ou *JSON*.

Há um outro padrão de design mais moderno que faz pesado uso de técnicas *AJAX*, as nomeadamente *Rich Internet Applications*, também conhecidas pelo acrônimo *RIA*. Como o nome sugere, são aplicações *Web* ricas, voltadas para preencher o espaço vazio na *Web* de aplicações semelhantes ou até idênticas as aplicações *Desktop*, ou seja, aplicações com *widgets*, componentes gráficos, janelas, painéis, entre outros. Se difere das aplicações *Web* tradicionais pelo fato de que não há páginas nem mesmo troca de páginas durante a interação do usuário com a aplicação. Em outras palavras, a aplicação deve responder às requisições do usuário sem atualizar a tela e recarregar todos os componentes, dando a mesma sensação de se estar utilizando uma aplicação *Desktop*. Segundo Holdener (2008), as aplicações *RIA* atuam como tradicionais aplicações *Desktop*, modificando o navegador de um magro para um gordo cliente, através do uso de *Javascript*. Esta *RIA* ainda faz uso do padrão *MVC* para conseguir estabilidade e confiabilidade. A Figura 9 demonstra a arquitetura de aplicação *Web RIA*:

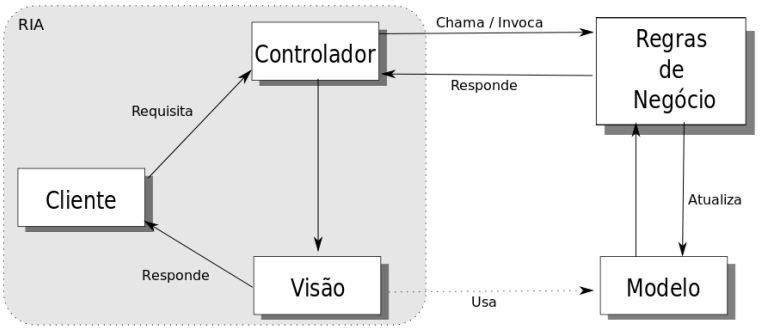

Figura 9 *– Arquitetura Web RIA*

Adaptado de (HOLDENER, 2008)

Há algumas vantagens das *RIAs* que o próprio Holdener (2008) também cita:

- As aplicações *Web AJAX* não requerem instalação, atualização ou distribuição para o usuário final, conquanto tudo isto já é oferecido pelo servidor na qual a aplicação está hospedada;
- Aplicações *Web AJAX* estão menos propensas a ataques de vírus, no geral;
- Tais aplicações possuem grande disponibilidade, conquanto podem ser acessadas de qualquer lugar, e quando escritas corretamente, são compatíveis com qualquer sistema operacional.

2.2.3 Princípios das Aplicações *AJAX*

Há algumas considerações que se devem levar em conta sobre o desenvolvimento de aplicações *Web AJAX*. Holdener (2008) cita que deve-se levar em conta questões de usabilidade em aplicações desta natureza, tais como: estrutura; simplicidade; tolerância; reusabilidade; receptividade.

Uma aplicação estruturada significa organizada em significado e utilidade para o usuário. Partes que são relacionadas devem se agrupar, enquanto partes não relacionadas se separar, o que auxilia na navegação da aplicação. A simplicidade remete que tarefas comuns para o usuário devem ser de fácil entendimento e conclusão; a tolerância se refere a quão as aplicações *AJAX* devem ser flexíveis no tratamento de erros e ocorrências inesperadas; a reusabilidade é alcançada com redução de quantidade de informação que o usuário necessite lembrar ou pensar cada vez que ela interaja com a aplicação; e a receptividade é a aplicação *AJAX* estar prepa-

rada para receber de forma adequada feedback do usuário, seja em forma de críticas, elogios ou sugestões.

Ao lado destas considerações de usabilidade, princípios orientados para o desenvolvimento de aplicações *Web AJAX* auxiliam na construção de aplicações para os usuários que são simples de navegar e utilizar. Holdener (2008) destaca seis princípios:

- Minimalismo e estrutura estética;
- Flexibilidade e eficiência;
- Consistência;
- Navegação;
- *Feedback*;
- Documentação e Ajuda.

Minimalismo e estrutura estética estão voltados para a forma de organizar as disposições dos elementos que compõem as telas da aplicação. Alguns exemplos de estruturas estéticas são os passeios guiados, assistentes, painéis, árvores de diretórios, entre outros.

Flexibilidade é a maneira como a aplicação trata a interação do usuário. Em outras palavras, a aplicação deve ser flexível o bastante para receber as entradas do usuário seja qual for ela. Em casos de problemas, a aplicação deve tratar de maneira transparente ao usuário, sem interromper sua experiência de uso. Por outro lado, a eficiência é a capacidade da aplicação *AJAX* de ser responsiva, ou seja, rápida e leve, segundo Holdener (2008). A eficiência também está no sentido de oferecer maneiras fáceis e ao mesmo tempo mais rápidas de se conseguir concluir alguma ação para o usuário.

Consistência é a maneira com que a aplicação se parece e se comporta para o usuário. Holdener (2008) explica aqui que, o usuário deve encontrar elementos consistentes em significados por toda a aplicação. Um ícone ou botão deve realizar a mesma ação, não importando em que parte da aplicação o usuário esteja. Esta regra vale para qualquer componente que faça parte da aplicação.

Paralelamente, o princípio da navegação de uma aplicação *Web AJAX* se resume a facilidade do usuário para movimentar-se por toda a aplicação. Elementos como barras de menu, caixas de navegação, *breadcrumbs*, mapa do site, barras e campos de pesquisa, entre outros, auxiliam para uma boa usabilidade de navegação. Entretanto, Holdener (2008) alerta que este ponto ainda é um desafio para aplicações desenvolvidas com técnicas *AJAX*, pois a criação de

conteúdos dinâmicos podem não funcionar muito bem com ações nativas do browser como o botão de voltar ou o recurso de favoritos.

Ele levanta também considerações sobre as funcionalidades que uma aplicação *AJAX* deve atender. As funcionalidades principais encontradas em aplicações modernas na Web que utilizam *AJAX* como uma de suas tecnologias estão listadas abaixo:

- Coleta: aplicação acumula informações;
- Navegação: navegação fácil dentro da aplicação;
- Busca: localização de informações dentro da aplicação;
- Organização: informação similar agrupadas para fácil entendimento;
- Comunicação: compartilhamento de ideias com um grupo;
- Geração: gerar conteúdo ou informação;
- Assistência: assegurar acessibilidade para informação;
- Manipulação: revisar ou de algum modo modificar informações;
- Armazenamento: armazenar acúmulo de informação.

2.2.4 As Diversas Técnicas *AJAX*

Conforme Rai (2013), a aplicação *Web Gmail* foi a luz para a comunidade de desenvolvedores *Web* sobre as vantagens das atualizações dinâmicas nas páginas *Web*, o que abriu caminho para diversas novas implementações de técnicas *AJAX*, que traziam dados do servidor para o cliente, ou pelo menos, dando a ilusão de fazer isso.

A técnica *AJAX*, como surgida originalmente, também já fazia algum tipo de carregamento de dados do servidor para o cliente, mas com a necessidade de que o usuário tivesse ainda que interagir com algum elemento da página para que a solicitação de dados assíncrona ocorresse. Já com as novas implementações desta técnica, a tentativa era de tornar este envio de dados independente da solicitação por parte do cliente.

Rai (2013) explica que estas tecnologias foram apelidadas com o termo *Comet*, introduzida por Alex Russell em seu site em 2006. Esta palavra era uma brincadeira com o termo *AJAX*, já que ambos os termos já eram nomes conhecidos de produtos de limpeza nos Estados Unidos. O termo *Comet* não sugere somente uma tecnologia, mas várias, introduzindo mais abordagens para tornar as solicitações assíncronas em mecanismos para se obter dados do servidor para o

cliente sem solicitação explícita por parte do usuário. Em outro lado, Wang et al. (2013) cita a técnica de sondagem longa como sendo a conhecida pelo termo *Comet*, ou também *Reverse AJAX* (*AJAX* reverso).

Dentre as técnicas existentes, temos a *Hidden iframe*, *XHR polling* (sondagem *XHR*), *XHR long polling* (sondagem longa) e a *Script tag long polling* (sondagem longa com *Script tag*, ou sondagem longa com *JSONP*), além de *HTTP streaming*, sendo estas técnicas ainda hoje as mais utilizadas entre os desenvolvedores *Web*.

2.2.5 A Técnica de Sondagem Longa

Rai (2013) explica a *XHR polling* como a primeira e mais fácil técnica de se implementar; Wang et al. (2013) cita que a técnica polling é uma chamada síncrona programada regularmente. Esta técnica consiste de fazer com que o cliente (navegador *Web*) se mantenha sondando por dados periodicamente e o servidor se mantenha respondendo a esta sondagem. Caso o servidor não tenha dados para enviar, ele envia uma resposta vazia, caso tenha, então ele envia os dados solicitados de volta para o cliente. Um exemplo para este cenário pode ser uma sondagem por um dado atualizado no servidor, ou a requisição por uma resposta de recebimento de e-mail ou até mesmo uma verificação de conclusão de tarefa por parte do servidor. A seguir, a Figura 10 demonstra como funciona esta técnica quando aplicada:

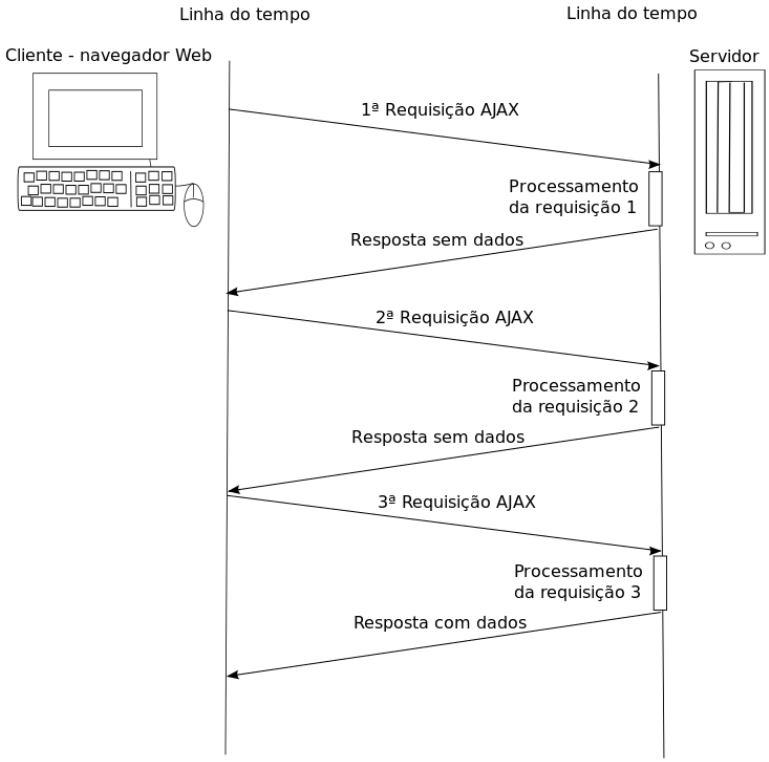

Figura 10 *– XHR polling*

Adaptado de (RAI, 2013)

Wang et al. (2013) argumenta que esta técnica é uma boa solução se quem estiver desenvolvendo a aplicação souber o exato intervalo de tempo para a entrega da mensagem solicitada, caso em que é possível sincronizar o cliente para enviar requisições somente quando se sabe que a informação requerida estará disponível no servidor. Entretanto, ela também alerta que quando se necessita de dados em tempo real, tal sincronia não é tão previsível, o que acarreta em requisições desnecessárias, na qual o cliente abre e fecha muitas conexões em situações desnecessárias.

Ainda mais, problemas com esta solução tornaram-se evidentes. O cliente, por exemplo, faz requisições para o servidor a todo instante em busca de respostas. Ainda que o servidor não possua resposta, ele sempre processa a requisição e envia uma resposta sem atender a solicitação do cliente. O cliente, por sua vez, continua insistindo na solicitação para o servidor, até que o servidor envie uma resposta que satisfaça o pedido. Claramente podemos perceber que o tráfego dentro deste ciclo será alto na rede, além do mais, o servidor atendendo a todo momento requisições em tão pouco tempo, há sobrecargas de processamento.

A tentativa de melhorar esta técnica, é modificar o servidor para alterar este ciclo de requisições do cliente pelo envio de dados extras na resposta do servidor, o chamado efeito

piggybacking. O servidor envia de volta não somente os dados solicitados pelo cliente, mas também dados adicionais que o servidor tenha. O cliente, por sua vez, necessita ser alterado para entender a agir em cima dos dados vindos adicionais. A Figura 11 a seguir mostra esta modificação:

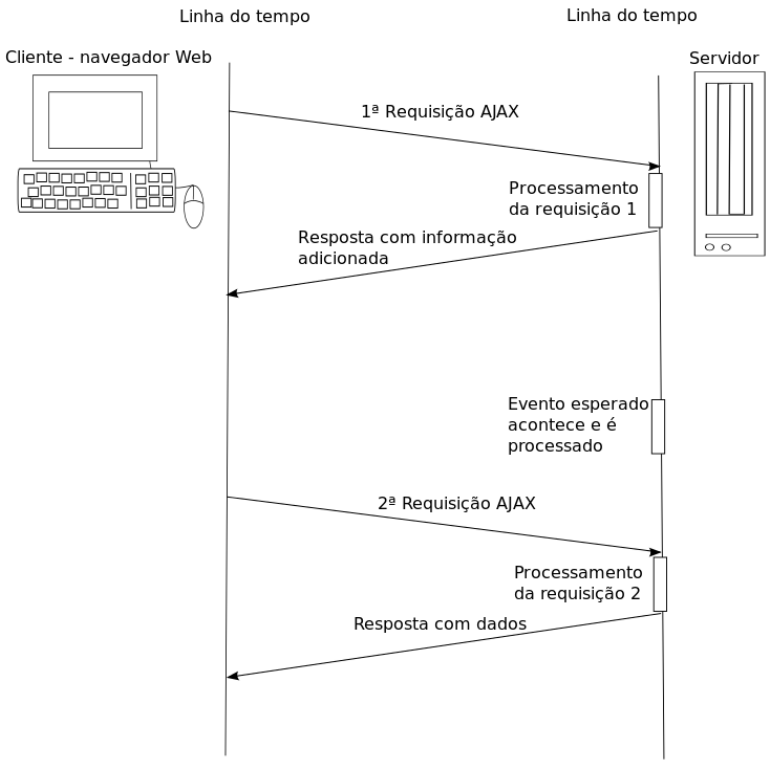

Figura 11 *– XHR polling com piggybacking*

Adaptado de (RAI, 2013)

Um problema desta técnica é que novos dados vindos do servidor ainda só chegarão após nova ação do cliente. Em consequência disto, surge uma variação desta técnica, conhecida pelo termo *long polling* (sondagem longa). O mecanismo desta técnica faz com que o servidor só responda ao cliente quando há algo para retornar na resposta, o que faz com que a espera para o resultado de uma sondagem, em comparação com a primeira técnica estudada, seja mais longa,

razão pela qual o apelido sondagem longa. A seguir, a Figura 12 demonstra o mecanismo:

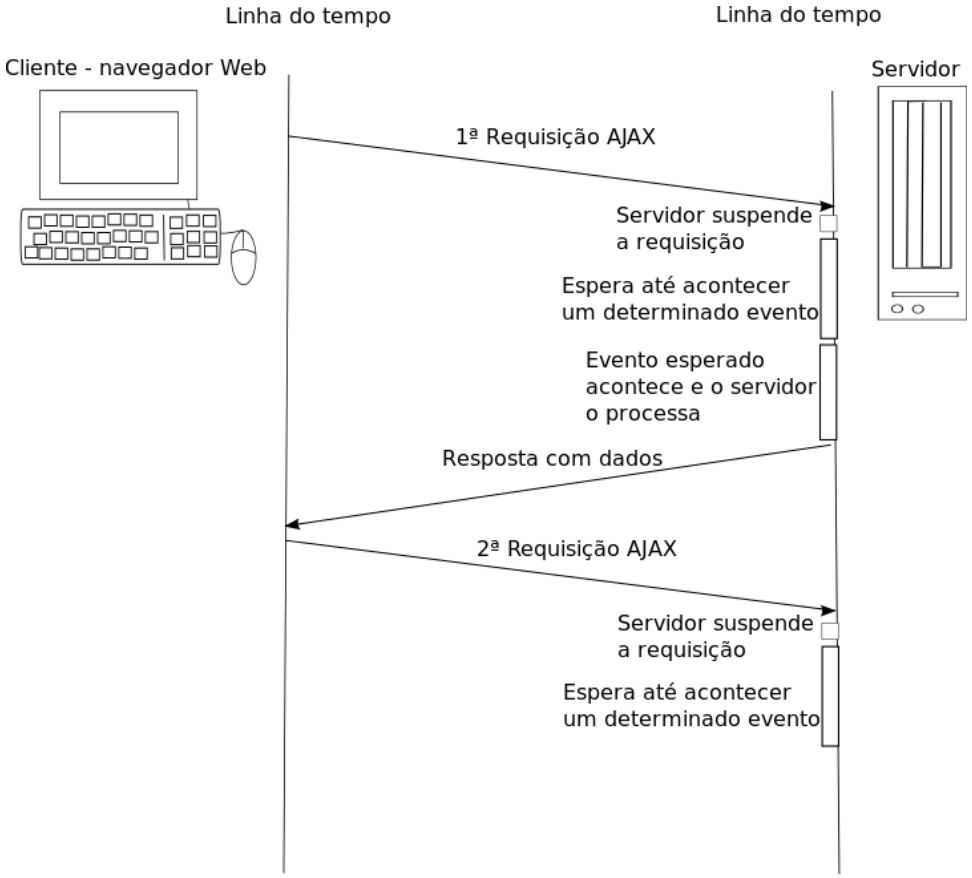

***Figura 12** – XHR long polling*

Adaptado de (RAI, 2013)

A melhoria que ocorre com esta técnica em relação a primeira, é que o cliente não mais precisa se preocupar em permanecer perguntando ao servidor se ele já possui resposta para enviar ou não, ou seja, o cliente deixa de exigir por uma resposta a curto prazo, para poder recebê-la, na pior das hipóteses, a longo prazo. Caso o servidor não tenha nada para responder, ele não devolverá imediatamente resposta para a solicitação do cliente, mantendo-se em espera para aquela solicitação, na dependência da ocorrência de um evento que a satisfaça. Wang et al. (2013) diz que esta característica é, em resumo, um atraso programado na conclusão da resposta *HTTP*, nomeando esta técnica como *hanging-get* ou *pending-post*. Há vantagens claras aqui, desde que o número de requisições do cliente para o servidor cai bastante. Outra vantagem é que livra o servidor de processar respostas até mesmo quando não há nada para satisfazer a solicitação do cliente. Conforme Rai (2013), há algumas variações que implementam esta técnica, tais como *forever iframe, multipart XHR, script tags* com *JSONP, long-living XHR*.

Outra técnica existente é a *HTTP streaming*, na qual, conforme Wang et al. (2013) , o cliente envia uma mensagem de solicitação, enquanto o servidor envia uma resposta e mantém

esta conexão aberta, sendo continuadamente atualizada indefinidamente ou por um período de tempo pré-determinado. O servidor então atualiza a resposta sempre que há dados prontos para satisfazerem a solicitação do cliente.

O problema que o ocorre com o *HTTP streaming* é que o servidor nunca sinaliza para o término da conexão aberta através do fluxo criado para o envio de respostas ao cliente, deixando a conexão aberta continuadamente. Wang et al. (2013) explica que esta situação é prejudicial em redes em que *firewalls* e *proxies* estejam presentes, pois estes podem armazenar a resposta, o que resulta em aumento de latência na entrega das mensagens.

3 A *Web* em Tempo Real

Desde que a *Web* passou a ter demandas por mais interatividade, o foco começou a transitar do conteúdo para o usuário final. Estudos e práticas sobre usabilidade nas páginas e sítios na *Web* começaram a serem levados em consideração no desenvolvimento, bem como apontou-nos Holdener (2008), ainda mais com tecnologias como o *AJAX* surgindo.

Em busca de oferecer mais interatividade, e em consequência, melhores desempenhos na experiência de uso do usuário final, o *AJAX* começou a ser muito utilizado por toda a *Web*. O *AJAX* ofereceu mais dinamismo no carregamento dos dados e na conclusão de tarefas feitas pela usuário final. Como Holdener (2008) disse, os usuários finais mereciam uma *Web* como plataforma, mais rápida, interativa e funcional. O *AJAX* trouxe aos desenvolvedores, bem como os usuários finais, custos mais baixos, melhor acessibilidade, visibilidade e percepção. Além disso, aplicações *Web* padronizadas com esta tecnologia separaram a apresentação do conteúdo, facilitando a manutenção e reduzindo o uso da largura de banda da rede, já que os dados são carregados por trás das cenas, como afirma Riordan (2008), além de carregar somente a parte da página que necessita ser atualizada, reduzindo a necessidade dos usuários de possuírem conexões muito rápidas.

Várias aplicações interativas e diferentes uma das outras apareceram, junto com a evolução da *Web* como plataforma, na qual não existiam simples trocas de documentos como antes, mas serviços e produtos eram oferecidos. Em outras palavras, um comércio muito vigoroso e jamais visto aconteceu. Ao invés de uma loja com alcance regional, agora havia uma loja com alcance mundial e com custos reduzidos, já que não se gasta com espaço físico e logística geográfica para estabelecimento de multinacionais; serviços como transmissões de vídeo e áudio em escala mundial, jogos interativos, e muitos outros mercados de nicho.

Em consequência deste mercado emergente e a *Internet* se tornando cada vez mais acessível a mais pessoas pelo mundo, naturalmente a demanda por serviços cresce e a plataforma evolui. Neste cenário, percebeu-se que as tecnologias *AJAX* já não atendiam muito bem a esta demanda, que requer conteúdo e serviço no tempo mais hábil o possível. Bem como precavê Ubl e Kitamura (2010), a comunicação *HTTP* ainda continuava sendo direcionada pelo cliente (navegador), exigindo interação do usuário ou sondagem periódica para carregamento de novos dados do servidor. Isto acaba afetando a experiência de uso do usuário e o alcance dos serviços, produtos e conteúdo fica limitada a ação do usuário. Um exemplo deste cenário é exibido na

Figura 13.

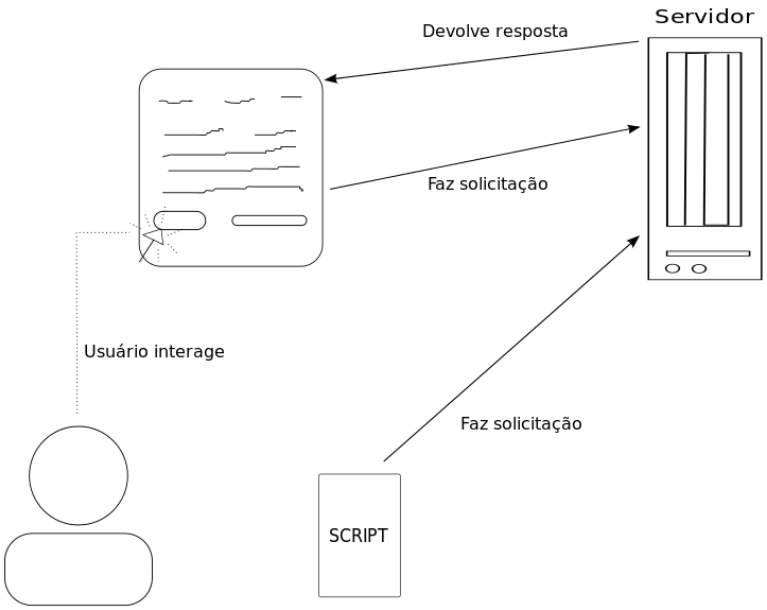

***Figura 13** – Interação Tradicional*

A Figura 13 ilustra e corrobora com a afirmação de Ubl e Kitamura (2010). Com o padrão *AJAX*, a comunicação, que funciona sobre o protocolo *HTTP*, ainda é iniciada pela interação do usuário final ou de algum *script* pré-configurado para fazer solicitações temporizadas ao servidor. Este modelo, apesar de ter contribuído bastante para carregamentos de dados dinâmicos serem possíveis e diminuído o desconforto da espera de um carregamento completo da página a cada interação do usuário, carregando somente partes da página, ainda não consegue alcançar uma verdadeira experiência em tempo real para a entrega de serviços e conteúdos. Neste cenário, se o usuário não deliberadamente solicitar por novas informações, o servidor não responderá com nada novo, já que ele funciona de modo passivo na comunicação. A tecnologia *AJAX* tenta automatizar esta tarefa, realizando solicitações automaticamente e de tempo em tempo. Em consequência, temos os problemas de sobrecargas de solicitações, analisadas neste trabalho, e o *overhead* do cabeçalho *HTTP* a cada solicitação feita, demonstrada por Wang et al. (2013).

Como Wang et al. (2013) disse, o *HTTP* trabalha com comunicação *half duplex*, o que retarda bastante a transmissão de conteúdo, dificultando uma comunicação mais ágil, já que as mensagens seguem em somente um sentido a cada solicitação e resposta. A Figura 14 demonstra esta caraterística junto com uma rede *intranet full duplex*.

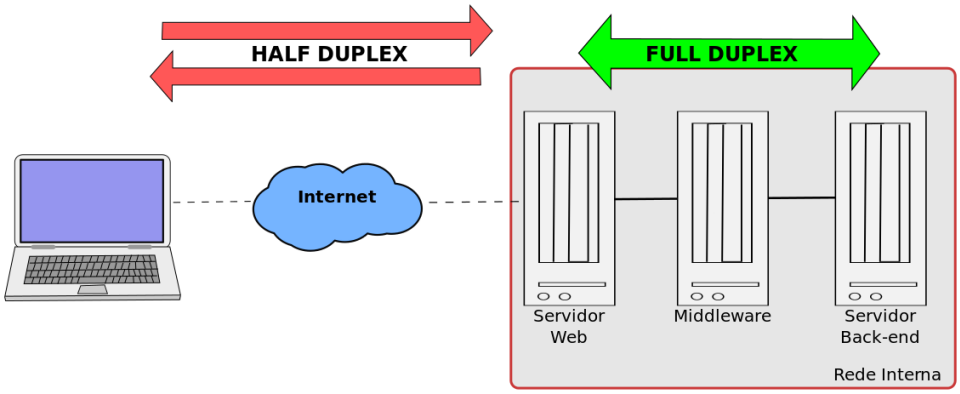

Figura 14 *– Comunicação HTTP half duples e full duplex*

Adaptado de (WANG et al., 2013)

Segundo Wang et al. (2013), a solução proposta está na especificação do *HTML 5*, na seção *Connectivity*, o *WebSocket*. Esta solução visa atingir aquilo que o *AJAX* não conseguiu: tornar a rede interativa, sem perca de recursos e desempenho.

Dentro do contexto atual, para Grigorik (2013) hoje temos, através da rápida evolução e inovação na infraestrutura de rede dos navegadores, mecanismos eficientes para *streaming*, comunicação bidirecional e em tempo real, possibilidade para customização de protocolos de aplicações e videoconferência *peer-to-peer* através da *Web*.

Ele afirma que hoje o navegador *Web* é a mais larga plataforma de distribuição disponível para os desenvolvedores de software. Está instalado em todo *smartphone, tablet, laptop, desktop*, entre outros dispositivos. Ele ainda afirma que as projeções de crescimento da indústria atual sinaliza para cerca de 20 bilhões de aparelhos conectados até o ano de 2020, com cada um possuindo um navegador *Web*, ou pelo menos conectado na rede, não importando o tipo de dispositivo, fabricante ou versão de sistema operacional. A *Web*, muito ainda através dos navegadores, está, então, espalhada por todo dispositivo, com os navegadores recebendo a cada dia mais recursos.

Grigorik (2013) relata que o resultado de toda esta expansão é uma crescente base de usuários, novas demandas para serviços online e alta demanda para aplicações *Web* de alta performance, com velocidade e desempenho sendo um recurso muito importante para estas aplicações, sendo dependente, intrinsicamente, de como o navegador e a rede interagem entre si.

Segundo Ingebrigtsen (2013), a tecnologia está se movendo rapidamente à frente, fazendo-nos pensar sobre como desenvolver aplicações que trabalham mais inteligentemente, entregam serviços mais rápido, e empurram dados e alterações diretamente para o usuário sem o deixar interagir com qualquer coisa ou sondar por informações novas. Para ele, a busca por novas

maneiras para conseguir tais resultados tem mudado a *Web* através do *HTML5* e *Javascript* tornando-se mais rápidos e mais largamente disponíveis. Seria então a hora para, aproveitando as tecnologias atuais e demandas dos usuários, mudar padrões e pensar diferente.

Aplicações *Web* em tempo real estão crescimento e cada vez mais em evidência. O *Facebook* e o *Twitter* são exemplos de aplicações que utilizam esta tecnologia, o que demonstra o enorme potencial mercadológico e inovador da *Web* em tempo real. Para Fromm (2009), a web em tempo real é um fenômeno de mídia e que, assim como as recentes inovações na área, a *Web 2.0* e a computação na nuvem, ainda não possui uma única definição do que o termo "*Web* em tempo real"significa, mas sugere que possamos identificar a Web em tempo real como:

1. uma nova forma de comunicação;
2. criação de um novo corpo de conteúdo;
3. é tempo real;
4. é público e possui um gráfico social associado a ele;
5. carrega um modelo implícito de federação.

Iskold (2008) comenta que a *Web* em tempo real é inevitável, e que ela já está em nosso meio, mesmo que inicialmente através de *Tv*, rádio, jornais, mas que agora essa tecnologia de tempo real está na *Web*, adentrando em nosso cotidiano através do *Twitter*. Ele comenta:

> O capitalismo é sobre as oportunidades. Sempre que há uma lacuna, há uma oportunidade para preenchê-la. O clássico negócio de jornal funcionava assim. Pessoas se reuniram notícias ao longo do dia e, em seguida, uma vez a cada 24 horas, cometido que eles tivessem obtido o jornal. Isso foi bom o suficiente para um longo período de tempo, mas com o surgimento do rádio e da televisão, e mais tarde de *blogs* e *RSS*, uma vez por dia parece uma piada. Claramente, nós exigimos notícias mais frequentemente do que uma vez por dia. Blogueiros de notícias de política, notícias do mundo, e em particular de tecnologia, reconheceram que a velha maneira de entregar notícia tinha uma falha - não era em tempo real. Eles transformaram a falha em uma oportunidade. (ISKOLD, 2008, tradução nossa)

Iskold (2008) também explica que o *Twitter* não fez sucesso por acidente. O *Twitter* seria, segundo ele, um dos mais vívidos exemplos de como a Web em tempo real está se infiltrando em nossa cultura. Desta plataforma ele faz exemplo de como nós estamos famintos por saber o que está acontecendo agora, o que chats e e-mail não suprem e que ouvir rádio ou assistir televisão é chato, pois não é personalizado, enquanto ao passo que o *Twitter* traz exatamente o tipo de notícias em tempo real que gostamos: personalizado e curto.

Segundo Kirkpatrick (2009a), o investidor Paul Buchheit comenta que a *Web* em tempo real está para ser a próxima grande coisa. Argumenta que é muitas vezes uma maneira mais fácil e mais eficiente maneira de comunicação porque uma inteira conversação pode acontecer em questão de minutos ou segundos, que é similar para a diferença entre uma chamada telefônica

e uma série de e-mails de voz. A chamada telefônica ocorre em tempo real de modo que toda a conversação muitas vezes pode ser concluída muito rapidamente. Kirkpatrick (2009a) divide a *Web* em tempo real em três conceitos:

1. Ambiente;
2. Automação:
3. Emergência;

Para o ambiente, ele explica que a *Web* é feita de páginas lincadas juntas, mas envolta de muitas daquelas páginas que agora são mídias sociais sinalizadas como postagens de *blogs, bookmarks, tweets* e outros endereços que referem-se para uma página mas não são visíveis quando estamos procurando.

No conceito de automação e emergência, argumenta que a *Web* em tempo real seria alguma coisa que estamos procurando constantemente, pelo fato de que ela está constantemente trazendo novas informações. Neste contexto, ela poderia nos notificar, em tempo real, dos eventos importantes e mudanças no mundo. Acredita que colocando tempo real no centro da *Web*, todo tipo de automatização de monitoração de informação seria possível.

3.1 O que é a *Web* em Tempo Real

Conforme relatório de Kirkpatrick (2009c), muitas das diferentes formas da *Web* em tempo real leva a alguns benefícios comuns, elementos de experiência de usuário, lições aprendidas, armadilhas e possibilidades.

Durante esta pesquisa, primeiramente buscou-se uma definição do que de fato é ou se trata a *Web* em tempo real. Kirkpatrick (2009c) nos traz algumas definições e visões diferenciadas a cerca do assunto. Para ele, a *Web* em tempo real é muito mais do que percebemos no *Facebook* ou *Twitter*, ainda que estes sejam os melhores exemplos do que podemos nos referir como *Web* em tempo real. Segundo ele, o *Facebook*, por exemplo, pode expandir suas funcionalidades em tempo real além do modelo adotado de stream (fluxo) que utiliza hoje. O *Twitter*, por sua vez, pode descobrir como reter mais usuários e encorajar mais do que o pequeno número de pessoas quem criam a maior parte do conteúdo da plataforma. Engenheiros estimam que se publica no *Twitter* cerca de mil mensagens por segundo e entre 5 a 10 milhões de *links* compartilhados por dia, antes de duplicações (KIRKPATRICK, 2009c).

Kirkpatrick (2009c) afirma que, hoje, a *Web* em tempo real é muito maior do que o *Twitter*. Além disso, relata algumas outras diferentes visões sobre a *Web* em tempo real[1](informação

[1] Entrevista concedida em (KIRKPATRICK, 2009c)

verbal):

Segundo o fornecedor de infraestrutura, *Kaazing*, a *Web* em tempo real está usando a tecnologia *HTML5 Web Sockets* para enviar informações financeiras ao vivo para navegadores *Web* de clientes bancários que tem sido sempre limitado para aplicações financeiras devido a razões de segurança[1].

Para *Pip.io* (aplicativo cliente *Web*), a *Web* em tempo real está criando uma experiência parecida com um chat construído com tecnologia *XMPP* para usuários se comunicarem com amigos em volta de objetos como um *Google Map* ou rodando um vídeo *Netflix* no *OS Pip.io*[1].

Segundo a companhia de recomendação semântica, *Evri*, a *Web* em tempo real é o refluxo e fluxo de dados de tráfego sobre *Wikipedia*. Estes dados apontam para tópicos interessantes que *Evri* necessita para construir páginas de tópicos para servir seus clientes publicadores.[1]

Para a máquina de pesquisa *OneRiot*, a *Web* em tempo real é feita de *links* que pessoas compartilham no *Twitter*, assim como *Digg, Delicious* e os fluxos de *click* de mais de um milhão de usuários que tem optado por expor o que eles vêem online através da barra de ferramentas do *OneRiot*[1].

Conforme o serviço de Q&A, *Aardvark*, a *Web* em tempo real são as pessoas dentro do círculo social de um usuário quem vêem a estar disponível online em um dado momento e interessados no tópico de uma questão do usuário[1].

Kirkpatrick (2009c) relata que existem milhares de blogs que entregam conteúdo atualizado para qualquer aplicação que inscreva para um *PubSubHubhub* (um protocolo em tempo real) ou fonte *RSSCloud*, imediatamente após o conteúdo ser publicado[1].

Jay Rosen, Professor de Jornalismo do *NYU* diz que a *Web* em tempo real cria um sentido de fluxo para usuários que é comparável a maneira que a televisão mantém nossa atenção. Já para o desenvolvedor do protocolo *PubSubHubhub*, Brett Slatkin, do *Google*, a *Web* em tempo real é a fundação para eficiente computação e casos de usos que nós ainda podemos imaginar[1].

3.2 *HTML 5 WebSocket*

WebSocket é a tentativa de eliminar as limitações de comunicação que acontecem com o protocolo *HTTP*. Conforme Wang et al. (2013), *WebSocket* é naturalmente *full duplex*, bidirecional e conexão de único *socket*. As requisições *HTTP* tornam-se uma única requisição de abertura de conexão *WebSocket*, ou seja, a primeira requisição é endereçada para a abertura de uma conexão *WebSocket*, enquanto todas as outras reusarão a mesma conexão, tanto via cliente – servidor, quanto servidor – cliente. Sergiienko (2014) e Ingebrigtsen (2013) dizem que o *WebSocket* é um protocolo que oferece canais de comunicação *full duplex* sobre uma única conexão

TCP e que o *WebSocket* é a devida substituição para as técnicas de *long polling* (sondagem longa).

Segundo Websocket.org (2014a), o protocolo *WebSocket* representa o próximo passo evolucionário na comunicação dentro da *Web* comparado para tecnologias *Comet* e *AJAX*, apesar de ressaltar que cada tecnologia tem suas próprias capacidades e casos de uso.

Wang et al. (2013) defende que o *WebSocket* reduz latência na rede, pois o servidor passa a ser ativo, ou, em outras palavras, passa a poder enviar mensagens (dados) conforme elas se tornam disponíveis. Esta característica do servidor nunca foi possível obter através do *AJAX*, o qual tentava fazer o servidor respondendo de tempos em tempos através de sondagens. Em pé de igualdade, o cliente também pode enviar mensagens para o servidor a qualquer tempo, já que o caminho (conexão) passou a ser bidirecional e *full duplex*. Esta característica, em comparação com as técnicas de sondagem do *AJAX*, traz um ganho em redução de latência, já que passa a existir somente uma requisição contra as várias requisições que são feitas temporalmente pelas

técnicas de sondagem. Na Figura 15 há uma comparação entre os dois cenários.

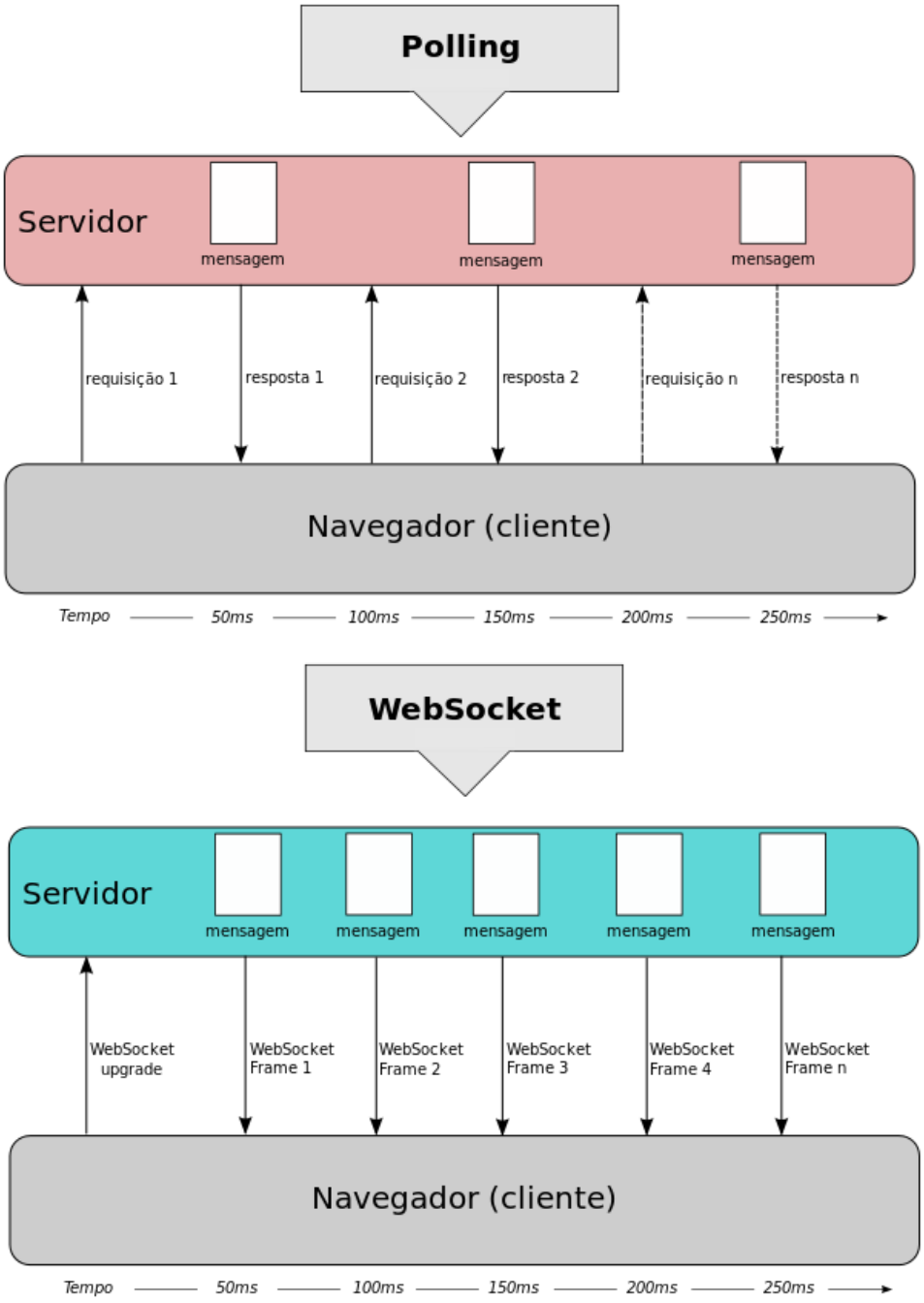

Figura 15 *– Diferença entre WebSocket e AJAX polling*

Adaptado de (WANG et al., 2013)

Wang et al. (2013) cita quatro características inovadoras do *WebSocket*:

1. Performance: *WebSocket* economiza largura de banda da rede, processamento da *CPU* (no servidor não é mais necessário tanto processamento para checagem de seguidas e

repetidas requisições), e latência na transmissão das mensagens, tornando possível uma comunicação em tempo real, além de ainda ser possível fazer *polling* ou *streaming* sobre o protocolo *HTTP*;

2. Simplicidade: Wang et al. (2013) afirma que para se tentar chegar a uma comunicação em tempo real na *Web*, sob o protocolo *HTTP* e técnicas anteriores ao *WebSocket* é muito difícil e complexo. Por manter estado da sessão através de requisições stateless (não mantem estado) adiciona complexidade ao código fonte. Também ele diz que tratar *AJAX* é complicado e requer considerações especiais e que fazer o *HTTP* funcionar para algo que ele não foi designado aumenta a complexidade do software. Em sua vez, *WebSocket* ajuda a desenvolver aplicações em tempo real de maneira simplificada sob uma comunicação orientada a conexão;

3. Padronizado: *WebSocket*, por ser um protocolo de rede básico, é extensível e permite a construção de protocolos sobre ele. *WebSocket* trabalha de maneira bem mais desacoplada do que seu predecessor *AJAX*. Isto permite uma modularização de protocolos de alto nível, permitindo maior desacoplamento entre cliente e servidor. *WebSocket* é interoperável. Com ele é possível termos componentes reusáveis, como por exemplo, o *XMPP*, que pode ser usado sobre um cliente que utiliza *WebSocket* e logar-se em diferentes servidores de *chat*, se aproveitando da característica do *XMPP* entender o mesmo padrão de protocolo;

4. Especificação do *HTML5*: *WebSocket* faz parte da especificação *Connectivity*, do *HTML5*. É um esforço para tornar os navegadores *Web* em uma plataforma de aplicações totalmente capazes, independente dos sistemas operacionais e suas diferenças, e dos modelos de segurança ou design da *Web*. Nesta especificação, o *WebSocket* oferece uma moderna *API* que não afeta a segurança dos navegadores, utilizando uma interface de rede semelhante ao *TCP*.

3.2.1 Os Métodos *Pull* e *Push*

Uma característica marcante do protocolo *WebSocket* é que ela torna a comunicação entre o cliente e o servidor *full duplex*, em contraste com a natureza do protocolo *HTTP* versão 1.0 e 1.1. Dentro desta comunicação *full duplex*, tanto o servidor quanto o cliente são ativos e podem enviar mensagens um ao outro a qualquer momento. Neste contexto, é perceptível a diferença entre a característica das tecnologias anteriores, baseadas no modelo *Pull* (puxar ou retirar) e o modelo *Push* (empurrar) utilizado em aplicações em tempo real.

Roden (2010) relata que o método *Pull* é o qual a maioria das interações na *Web* tem se baseado. Ele ensina que neste modelo, o usuário clica em um *link* e o conteúdo puxa conteúdo a

partir do servidor. Caso o servidor tenha dados a mais para entregar, será enfileirado e terá que esperar até que o cliente (navegador) faça outra requisição. Diferentemente deste modelo *Pull*, o método *Push* consiste de que, tão quão breve o servidor tenha novos dados para o cliente, ele os envia imediatamente sem necessidade de requisição.

Estes dois modelos são a base de como muitas tecnologias funcionam hoje na *Web*. Seja através do padrão de uso ainda mais convencional, na qual o cliente é o agente ativo e o servidor o passivo, ou seja no mais moderno, na qual os dois são agentes ativos na comunicação. A seguir, a Figura 16 exemplifica estes dois modelos dentro de um contexto usuário – *site*:

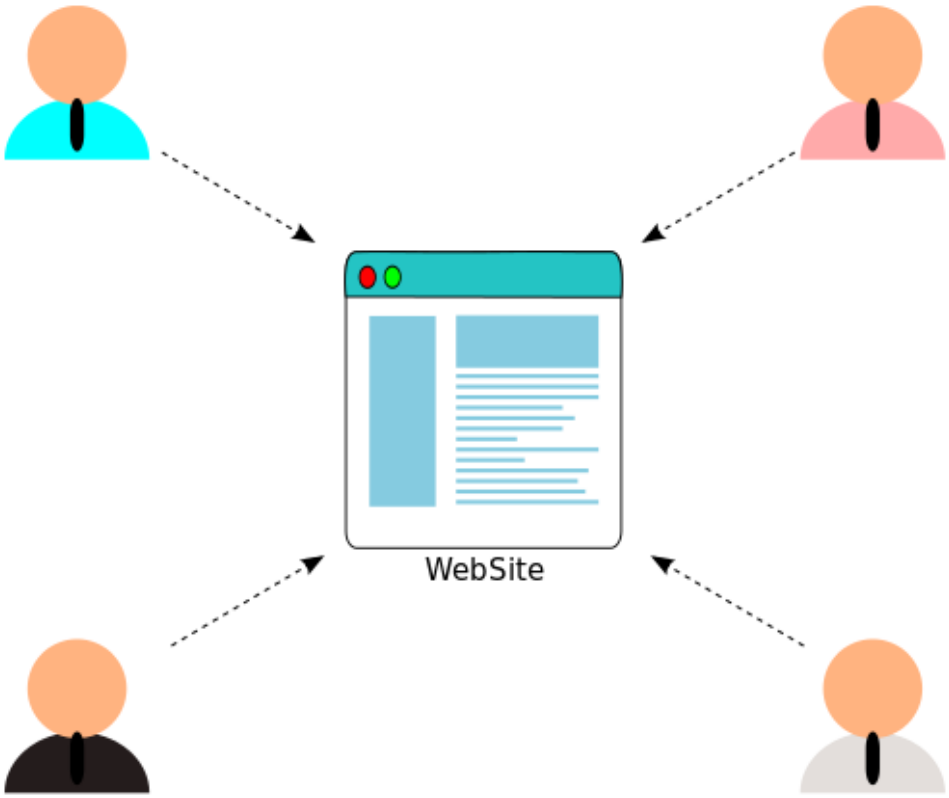

Figura 16 *– Contexto Pull: Usuário busca informação*

Adaptado de (RODEN, 2010)

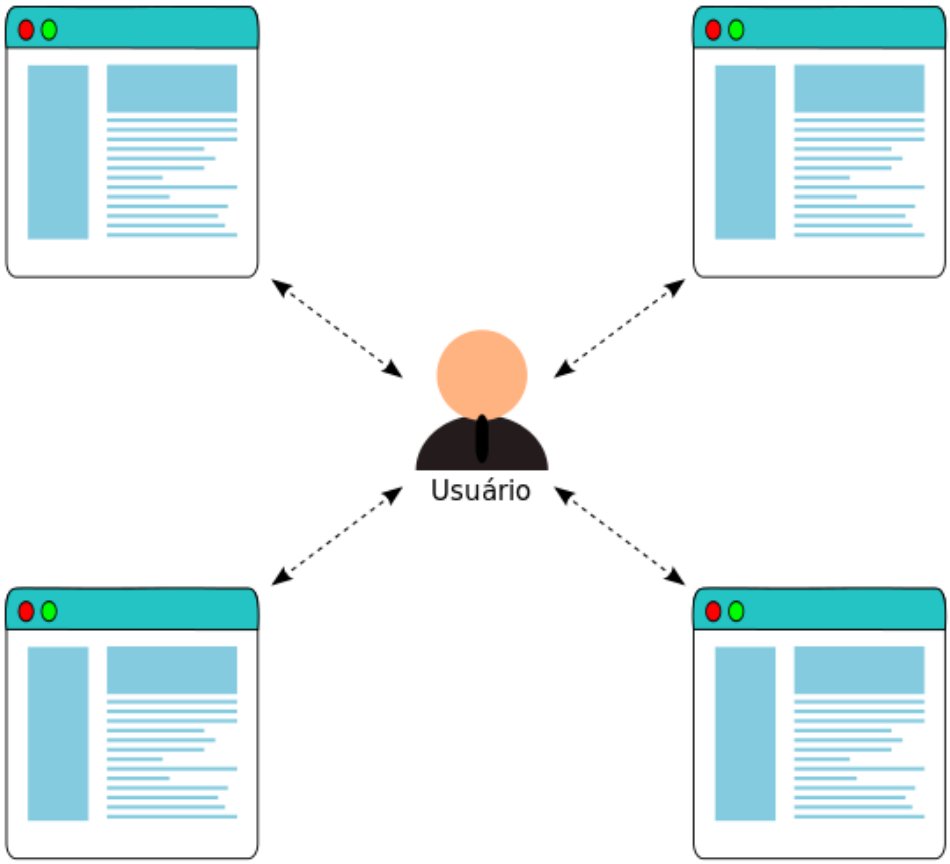

Figura 17 *– Contexto Push: Usuário recebe informação*

Adaptado de (RODEN, 2010)

Roden (2010) aponta que o modelo *push* não é algo novo, mas que já houvera vários diferentes padrões regrando seu modo de funcionamento, diferenciando-se em requerimentos e em implementações dos fabricantes de navegadores *Web*. Muitas aplicações, seguindo o modelo *pull*, para oferecerem atualizações dinâmicas para seus usuários, tiveram que recorrer a tecnologia *AJAX* sondando conteúdo novo a cada tempo predeterminado. Roden (2010) afirma que tal prática aumenta o número de requisições levando a uma experiência menos elegante do que se deveria.

Em contrapartida, argumenta que o método *push* dá uma experiência muito mais engajante, ao passo que utiliza menos recursos no servidor. Em comparação, menos requisições resultam em menos largura de banda utilizada e menos processamento no servidor.

De acordo com ele, as interações mudaram. Agora, com as tecnologias entregando conteúdo em tempo real, não mais as aplicações *Web* modernas devem esperar que seus usuários naveguem para sua *URL*, elas devem contatar o usuário seja lá onde ele estiver. Esta mudança traz o paradigma das interações na *Web* para um modelo centrado ao usuário, ao invés de um modelo centrado no *website*, na qual este era o centro de toda a interação.

3.2.2 Os Benefícios do Protocolo *WebSocket*

Conforme as características do *WebSocket* já estudadas e demonstradas neste trabalho, é possível elencar alguns benefícios que ela pode oferecer para o desenvolvimento de aplicações *Web* em tempo real. Benefícios tais como:

- Redução de latência;
- Economia de recursos utilizados no servidor;
- Implementação simplificada da comunicação entre servidor e cliente na aplicação;
- Extensibilidade para comunicação com outros protocolos;
- Maior performance.

Segundo Hickson (2009):

> Reduzindo kilobytes de dados para 2 bytes... e reduzindo latência de 150ms para 50ms é muito mais do que marginal. De fato, estes dois fatores sozinhos são o bastante para tornar Web Sockets seriamente interessante para a Google. (HICKSON, 2009, tradução nossa)

Lubbers e Greco (s.d.) corroboram salientando que métodos como *polling, long polling* e *HTTP streaming* tentam oferecer dados em tempo real envolvendo cabeçalhos *HTTP* de solicitação e resposta, os quais contém muitas informações adicionais, tais como dados de cabeçalhos, introduzindo latência para a comunicação. Eles explicam que esforços para simular conectividade *full duplex* sobre a natureza *half duplex* do protocolo *HTTP* requerem que tais soluções usem duas conexões, uma para a ida (*upstream*) e outra para volta (downstream). Acrescentam que, dentro deste cenário, a manutenção e organização para estas conexões introduz sobrecarga (*overhead*) de consumo de recursos e adiciona complexidade para a solução. Em resumo, argumentam que o protocolo *HTTP* não foi designado para funcionalidades em tempo real nem comunicação *full duplex*. A Figura 18 exemplifica as complexidades inerentes

a arquitetura do padrão *Comet*.

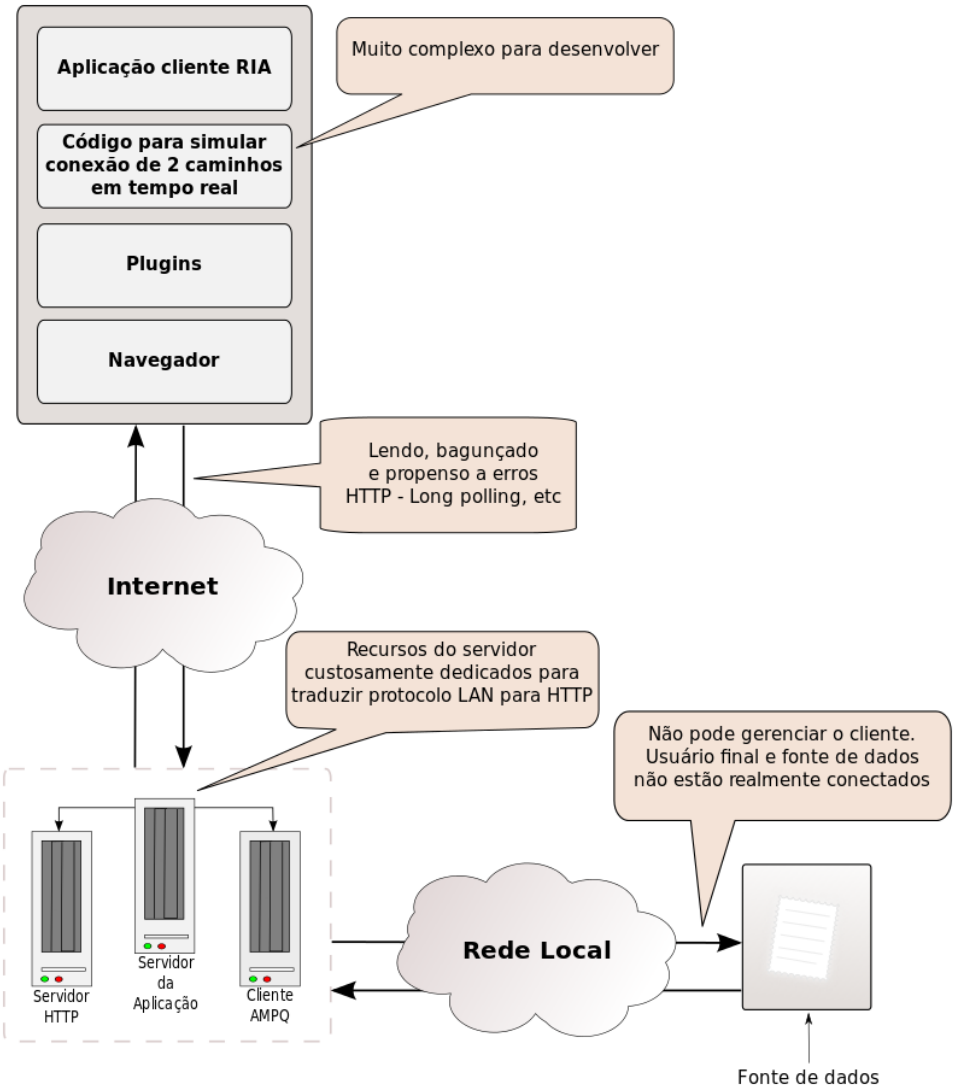

Figura 18 *– Arquitetura Comet*

Adaptado de (LUBBERS; GRECO, s.d.)

Lubbers e Greco (s.d.) relatam que as soluções *Comet* não escalam bem devido que a simulação de comunicação bidirecional sobre o *HTTP* é propenso a erros e complexo. Ainda mais, toda a experiência em tempo real dos usuários é afetada devido aos problemas de latência, tráfego de rede desnecessário e perca de performance de *CPU*. Em contrapartida, com *WebSockets* existe um padrão que torna possível desenvolver verdadeiras aplicações em tempo real e que escalam, pois elimina muito dos problemas os quais o padrão *Comet* está suscetível.

Grigorik (2013) qualifica a *API* do protocolo *WebSocket* como a mais próxima de um *socket* puro de rede, mas no navegador. A conexão com *WebSocket* é, entretanto, mais do que um *socket* de rede, pois o navegador abstrai toda a complexidade com uma simples *API*

e oferece vantagens sobre o *socket* de rede puro, oferecendo alguns serviços adicionais, tais como:

- Negociação de conexão e política de mesma origem forçada;
- Interoperabilidade com a infraestrutura *HTTP* existente;
- Comunicação orientada a mensagem e eficiente enquadramento;
- Negociação de subprotocolo e extensibilidade.

Segundo o sítio Websocket.org (2014a), um dos recursos do protocolo *WebSockets* é a habilidade para transpor *firewalls* e *proxies*, os quais são considerados problemáticos para muitas aplicações. As aplicações que utilizam o padrão *Comet* geralmente empregam técnicas rudimentares de defesa contra *firewalls* e *proxies*, as quais não são adequadas para aplicações que tem latência menor a 500 milisegundos e altos requerimentos de transferência. Tecnologias baseadas em *plugins*, como o *Adobe Flash* também sofrem de problemas em torno de *firewalls* e *proxies*. *WebSockets* resolve este problema por detectar o servidor *proxy* e emitir um *HTTP CONNECT* para ele. O servidor *proxy*, por sua vez, abre uma conexão *TCP/IP* para uma porta e *host* específicos, resultando em um túnel para a comunicação fluir através do servidor *proxy*.

3.2.3 *WebSockets* - Infraestrutura

Em conformidade com Wang et al. (2013), as conexões *WebSocket* surgem a partir de uma típica requisição *HTTP*. A diferença desta requisição para outras está na presença de um cabeçalho específico chamado Upgrade. Como o nome sugere, este cabeçalho indica para o destinatário da mensagem que o remetente gostaria de atualizar a conexão para um protocolo diferente. Parte-se do protocolo *HTTP*, requisitando uma conexão sob o protocolo *WebSocket*. As Figuras 19 e 20 abaixo demonstram os cabeçalhos de requisição do cliente e de resposta do servidor:

```
GET /echo HTTP/1.1
Host: echo.websocket.org
Origin: http://www.websocket.org
Sec-WebSocket-Key: 7+C600xYybOv2zmJ69RQsw==
Sec-WebSocket-Version: 13
Upgrade: websocket
```

Figura 19 *– Cabeçalho de Requisição de Conexão WebSocket*

Fonte: (WANG et al., 2013)

```
101 Switching Protocols
Connection: Upgrade
Date: Wed, 20 Jun 2012 03:39:49 GMT
Sec-WebSocket-Accept: fYoqiH14DgI+5ylEMwM2sOLzOi0=
Server: Kaazing Gateway
Upgrade: WebSocket
```

Figura 20 *– Cabeçalho de Resposta de Conexão WebSocket*

Fonte: (WANG et al., 2013)

Como podemos notar nas figuras acima, o cliente faz uma requisição com sua origem sendo o endereço *http://www.websocket.org* para o servidor *echo.websocket.org*. O cabeçalho *Upgrade* está presente indicando o protocolo desejado, que neste caso é o *WebSocket*. O cabeçalho *Sec-WebSocket-Key* é uma chave gerada pelo cliente para que o servidor possa testar e responder se suporta o protocolo, enquanto que o *Sec-WebSocket-Version* é o número da versão do protocolo que o cliente quer utilizar. Em outra via, a resposta do servidor deve conter uma resposta com o código 101 para confirmar a troca de protocolo, junto com outros parâmetros que confirmam pedidos do cliente, tais como subprotocolos e extensões, através dos parâmetros *Sec-WebSocket-Protocol* e *Sec-WebSocket-Extensions*, respectivamente.

No mínimo, como informa Wang et al. (2013), para uma conexão ser bem sucedida é necessário os seguintes critérios:

- O cliente precisa enviar os parâmetros *Sec-WebSocket-Version* e *Sec-WebSocket-Key*;
- Servidor deve confirmar o protocolo por retornar *Sec-WebSocket-Accept*;
- Cliente pode enviar uma lista de subprotocolos através de *Sec-WebSocket-Protocol*;
- Servidor deve selecionar um dos subprotocolos através de *Sec-WebSocket-Protocol*, e caso não suporte algum, abortar a conexão;
- O cliente pode enviar uma lista de extensões através de *Sec-WebSocket-Extensions*;
- Servidor pode confirmar uma ou mais extensões selecionadas através de *Sec-WebSocket-Extensions*, e caso nenhuma extensão oferecida, a conexão é realizada sem elas.

Após a conexão iniciada, toda a troca de mensagens acontecerá através de *WebSocket frames* e sob o protocolo *WebSocket* em uma conexão de vida longa. A partir de então, podemos mapear um cenário típico do funcionamento de uma aplicação baseada em *WebSocket*

demonstrando sua arquitetura. A Figura 21 demonstra a arquitetura baseada em *WebSocket* com comunicação para um protocolo personalizado e servidores remotos:

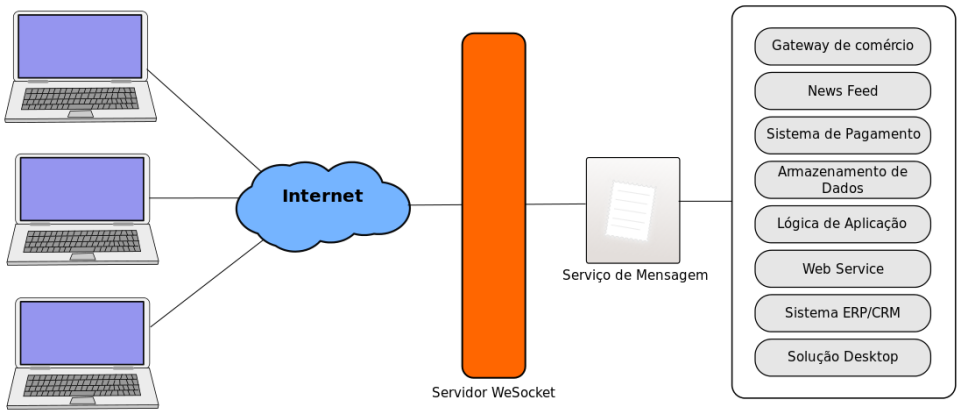

Figura 21 *– Arquitetura WebSocket*

Adaptado de (WANG et al., 2013)

3.2.4 *WebSocket vs AJAX polling* - Um Estudo de Caso

Lubbers e Greco (s.d.) divulgaram um estudo sobre *WebSocket* bastante interessante, no qual baseia-se em uma aplicação demonstrativa, na qual uma página *Web* solicita páginas em tempo real a partir de um corretor de mensagens *RabbitMQ* utilizando um modelo de inscrição e publicação. Este estudo demonstra estatisticamente reais diferenças entre a técnica *AJAX polling* e o *WebSocket*.

A aplicação foi desenvolvida com *Javascript*, sendo um fictício exemplo de uma aplicação de contação de ações que atualiza seus dados em tempo real. A página de cotação de ações da aplicação se inscreve para um canal que envia dados para uma específica cotação de ações de mercado. A Figura 22 exibe a referida página.

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
SANDISK 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	35.58	-0.15		35.73	32.16	39.30

Figura 22 *– Tela da Aplicação AJAX polling*

Fonte: (LUBBERS; GRECO, s.d.)

Nesta página, além de inscrição para um canal de dados específico, é utilizado *AJAX* para sondar por atualizações a cada segundo. Lubbers e Greco (s.d.) optaram pela técnica *polling* neste estudo, devido a grande quantidade de dados geradas pela fonte a cada segundo. Segundo eles, como muitos dados são gerados por segundo, é mais interessante usar *polling* a cada segundo do que utilizar uma técnica *Comet* do tipo *long polling*, pois causaria sondagens contínuas em série.

Lubbers e Greco (s.d.) descobriram que, ao utilizar a abordagem *polling* neste cenário, uma grande quantidade de overhead de cabeçalhos estava presente associado a cada solicitação *HTTP GET* que chega ao servidor a cada segundo. Tomando como exemplo os cabeçalhos *HTTP*, eles levantaram uma quantidade total de 871 *bytes* de informação no cabeçalho *HTTP* para requisição e resposta, ainda que aqueles não contenham nenhum dado relevante do servidor para o cliente ou vice-versa. Tal quantidade de informação do cabeçalho pode ainda variar para um valor maior. Em contramão, o tamanho de cada mensagem para as ações é de somente 20 caracteres, o que evidencia o excesso de informação somente para cabeçalhos.

Neste ponto, Lubbers e Greco (s.d.) fizeram um estudo estatístico para três casos de uso distintos, levando em consideração uma base de usuários grande e variante. O levantamento estatístico é para o processamento de rede associado apenas aos dados de cabeçalho *HTTP* para requisição e resposta. A Figura 23 exibe estes dados.

Número de clientes sondando (*polling*) por segundo	Cálculo para Transferência de Rede	Total em *bytes*	Total em Mbps
1000	871 x 1000	871.000 bytes	6.968.000 bits / s = 6,6 Mbps
10000	871 x 10000	8.710.000 bytes	69.680.000 bits / s = 66 Mbps
100000	871 x 100000	87.100.000 bytes	696.800.000 bits / s = 665 Mbps

Figura 23 *– Dados Estatísticos para caso AJAX polling*

Adaptado de (LUBBERS; GRECO, s.d.)

Podemos perceber através destes números que há uma enorme quantidade de transferência de rede desperdiçada somente com cabeçalhos *HTTP* para requisições e respostas. Adiante, Lubbers e Greco (s.d.) desenvolveram a versão desta aplicação utilizando *Web Sockets*, adicionando um manipulador de eventos para a página Web assincronizadamente escutar por mensagens de ações atualizadas a partir do corretor de mensagens *RabbitMQ*. Os *frames WebSocket* que trafegam pela rede são de apenas 2 bytes de sobrecarga contra os 871 da versão da aplicação com *polling*. A Figura 24 demonstra as estatísticas apuradas.

Número de clientes sondando (*polling*) por segundo	Cálculo para Transferência de Rede	Total em *bytes*	Total em Mbps
1000	2 x 1000	2.000 bytes	16.000 bits / s = 0,015 Mbps
10000	2 x 10000	20.000 bytes	160.000 bits / s = 0,153 Mbps
100000	2 x 100000	200.000 bytes	1.600.000 bits / s = 1,526 Mbps

Figura 24 *– Dados Estatísticos para caso WebSockets*

Adaptado de (LUBBERS; GRECO, s.d.)

O que Lubbers e Greco (s.d.) queriam mostrar com estes dados estatísticos é que, com *WebSockets*, é possível eliminar este grande desperdício por enviar somente informações essenciais através da rede. Pelo total de processamento de rede em *Mbps* exibidos nas duas tabelas, é nitidamente perceptível uma diferença significativa de consumo de tráfego entre *polling* e *WebSockets* e de como este novo protocolo economiza processamento na rede.

3.3 *SSE - Server Side Events*

Server Side Events é uma combinação de uma interface *EventSource* residente no navegador, cujo permite o servidor enviar notificações para o cliente como eventos do *DOM*, e um formato de dado chamado *event stream*, utilizado para entrega de atualizações individuais.

Como explica Grigorik (2013), a combinação da *EventSource API* e o formato de dado de fluxo de evento faz com que *SSE* (*Server Side Events*) seja uma eficiente e indispensável ferramenta para manipulação de dados em tempo real. Grigorik (2013) destaca quatro características:

1. Baixa latência através de uma única conexão de vida longa;
2. Análise eficiente de mensagem sem *buffers* ilimitados;
3. Automático rastreamento da última mensagem vista e auto reconexão;
4. Notificação de mensagens no cliente como eventos do *DOM*.

Wang et al. (2013) corrobora a favor da tecnologia *SSE* ao afirmar que este é uma boa solução para casos em que não se necessita de interatividade do cliente com o servidor, como aplicações de fontes de notícias ou previsão do tempo, na qual a interatividade acontece somente a partir do servidor que envia informações para o cliente em tempo real. Wang et al. (2013) revela que *SSE*, sendo parte da especificação do *HTML5*, é a consolidação de técnicas *Comet*, tais como *HTTP polling, long polling* e *HTTP streaming*, com a adição de recursos exclusivos do *SSE*, como por exemplo a auto reconexão e indentificadores de eventos.

Em contrapartida, Wang et al. (2013) indica algumas limitações do *SSE* em comparação com o *WebSockets*. Por exemplo, com *SSE* não é possível enviar dados do cliente para o servidor, além de suportar somente dados de texto. Outras limitações do *SSE* são levantadas por Grigorik (2013). Uma é que esta tecnologia não trata casos de solicitação de envio de grandes dados para o servidor. Grigorik (2013) também alerta para a ineficiência na transmissão de binários, já que o protocolo de fluxo de eventos é designado especificadamente para transferir dados em *UTF-8*. Outra ponto negativo é que *push* em tempo real pode ter sério impacto na vida da bateria de dispositivos, requerendo cuidados especiais nestes casos. A seguir se encontra a Figura 25 que diferencia o fluxo de informações com *SSE* e *WebSockets*.

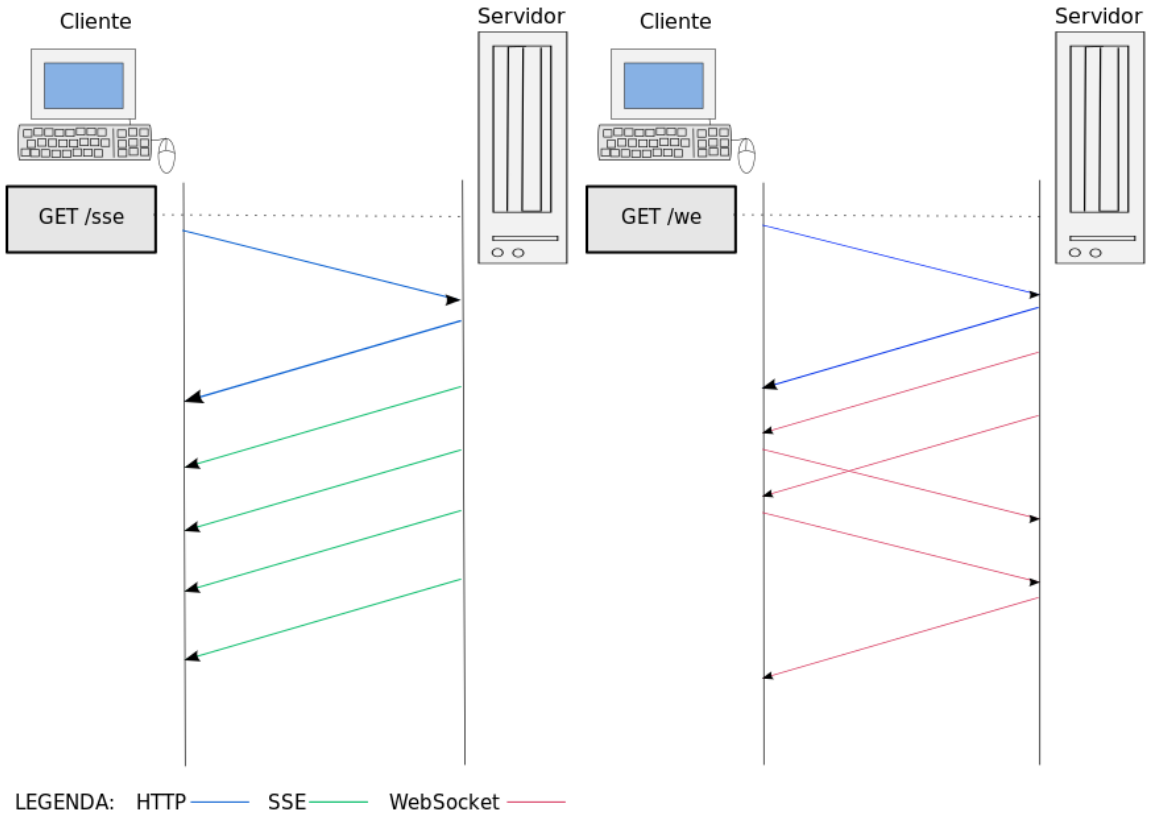

Figura 25 *– SSE vs WebSockets*

Adaptado de (GRIGORIK, 2013)

3.4 *SPDY (speedy)*

Conforme Wang et al. (2013), esta tecnologia é um novo protocolo que está sendo desenvolvido pela *Google* visando melhorar a performance das requisições *HTTP*, com o intuito de melhorar a performance final das páginas *Web*. *SPDY* utilizar abordagem diferenciadas, como por exemplo, comprimir e multiplexar cabeçalhos *HTTP*.

Diferencia-se do protocolo *WebSockets* no que tange sua área de aplicação. Enquanto *WebSockets* procura melhorar a comunicação entre o cliente e o servidor, *SPDY* aplica melhorias para a entrega de conteúdo da aplicação e também de páginas estáticas.

SPDY procura manter-se padronizado arquiteturalmente com o protocolo *HTTP*, tanto em estilo, quanto em semântica, corrigindo problemas que ocorrem com o *HTTP* e adicionando recursos extras como multiplexação, *pipelining*, compressão de cabeçalhos, entre outros. Além disso, este protocolo possui interoperabilidade, sendo compatível com o protocolo *WebSocket*.

Segundo Grigorik (2013), *SPDY* é um protocolo experimental, desenvolvido pela *Google* em meados de 2009. Sua meta principal era a de tentar reduzir a latência de carregamento

das páginas *Web* por resolver algumas das limitações de performance conhecidas do protocolo *HTTP 1.1*. Ele destaca as metas do projeto conforme abaixo:

- Focar em uma redução de tempo de carregamento de página (*PLT*) em 50%;
- Evitar qualquer necessidade de mudança de conteúdo pelos desenvolvedores;
- Minimizar a complexidade na implantação e evitar mudanças na estrutura da rede;
- Desenvolver este protocolo em parceria com comunidade de código fonte aberto;
- Obter performance real de dados para validar ou invalidar o protocolo experimental.

Ainda conforme Grigorik (2013), a redução de 50% da *PLT* por parte deste protocolo consistia em fazer um mais eficiente uso da conexão *TCP* interna pela introdução de uma nova camada de enquadramento binário, com o intuito de permitir multiplexação de requisição e resposta, priorização, e minimizar e eliminar desncessária latência de rede.

3.5 Comunicação em Tempo Real com *WebRTC*

WebRTC (*Web Real Time Communications*) é uma especificação da *W3C* que define padronizadas *APIs*, capacidades internas de áudio e vídeo em tempo real e *codecs* para os navegadores *Web* sem a necessidade de *plugins*. (WEBRTC.ORG, 2014a)

Esta tecnologia oferece aos desenvolvedores de aplicações *Web* a possibilidade de escrever ricas aplicações multimídia em tempo real na *Web* sem requerer instalação de *plugins*. Seu propósito é ajudar a construir uma forte plataforma *RTC* (*Real Time Comunications*) que funcione através de múltiplos navegadores *Web* e múltiplas plataformas. (WEBRTC.ORG, 2014b)

Segundo Loreto e Romano (2014), a *WebRTC* é um novo padrão e esforço industrial para estender o modelo de navegação da *Web*. Esta tecnologia seria a pioneira em possibilitar que os navegadores sejam capazes de trocar mídias em tempo real com outros navegadores em um padrão ponto a ponto. O *design* da *API* do *WebRTC* prevê que um fluxo de dados em tempo real e contínuo é transmitido através da rede entre dois navegadores *Web*, sem nenhum intermediário pelo caminho. Esta abordagem diferencia-se do padrão *Web* cliente-servidor que estamos acostumados e claramente representa uma abordagem revolucionária para a comunicação baseada na *Web*, conforme Loreto e Romano (2014).

Para Sergiienko (2014), esta tecnologia é uma nova *framework* para a *Web*, ainda em desenvolvimento, que permite aplicações de navegador para navegador compartilharem chamadas de áudio e vídeo, videoconferências e troca de arquivos sem a necessidade de qualquer *software*

de terceiros. Tendo seu código fonte aberto pela *Google* em 2011, empresas como o *Google, Mozilla*, e *Opera* suportam e trabalham no processo de desenvolvimento desta tecnologia.

Em conformidade com o sítio oficial da ferramenta, Webrtc.org (2014b), a missão deste projeto é permitir aplicações *RTC* (Comunicação em Tempo Real) ricas e de alta qualidade serem desenvolvidas no navegador via simples *APIs Javascript* e *HTML5*.

3.5.1 *WebRTC* - Infraestrutura

Sergiienko (2014) divide a arquitetura da *WebRTC* em três componentes principais:

- *getUserMedia*: componente que permite o navegador acessar a câmera e o microfone;
- *PeerConnection*: componente para configuração de chamadas de áudio e vídeo;
- *DataChannels*: componente que permite os navegadores compartilharem dados através de conexões ponto a ponto.

A seguir, a Figura 26 descreve mais detalhadamente a arquitetura *WebRTC*:

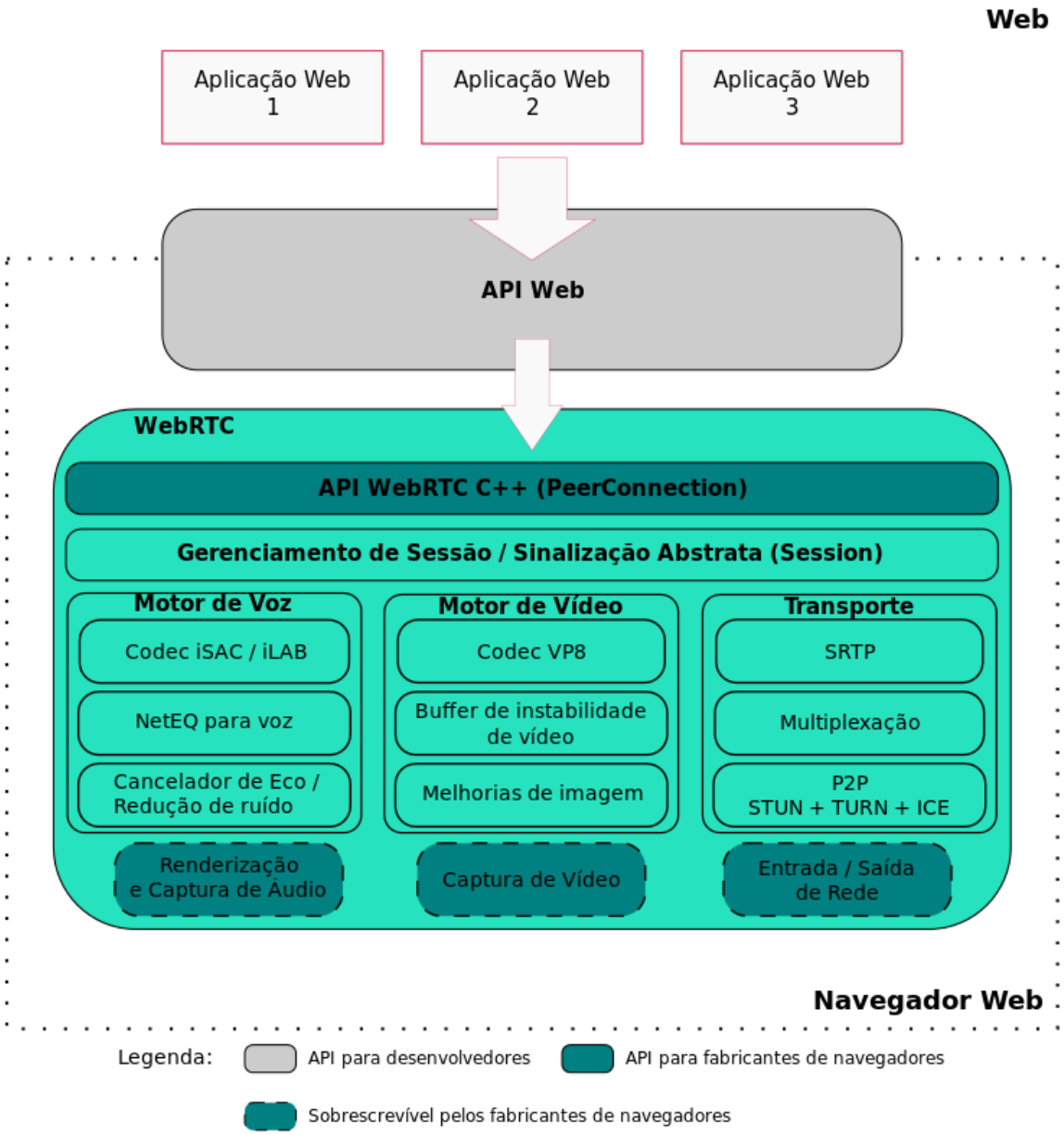

Figura 26 *– Arquitetura WebRTC*

Adaptado de (WEBRTC.ORG, 2014a)

Na Figura 26 podemos notar uma separação em três camadas: uma camada para aplicações *Web*, outra para a *API Web*, e outra camada para *API WebRTC*. A *API Web* será foco dos desenvolvedores *Web*, enquanto a *API WebRTC* foco dos fabricantes de navegadores *Web*. As aplicações *Web*, por sua vez, farão uso da *API Web* para se capacitarem dos recursos de comunicação em tempo real, proporcionadas pela *API WebRTC*.

Dentro do navegador *Web*, a arquitetura funciona sendo dividida em partes que tomam de conta da conexão ponto a ponto (*PeerConnection*), camada de gerenciamento de sessão, motores de áudio e vídeo, camada de transporte e partes sobrescrevíveis pelos fabricantes de

navegadores: renderização e captura de áudio, captura de vídeo, e entrada e saída de rede. Abaixo há uma descrição de cada camada:

- Gerenciamento de Sessão: esta é uma camada de sessão abstrata que permite configurações de chamada e gerenciamento, deixando a implementação de protocolo para o desenvolvedor da aplicação;

- Motor de voz: um conjunto de ferramentas para mídias de áudio, a partir da placa de áudio para a rede;

  Codec iSAC: um *codec* de áudio de banda larga e super banda larga para *VoIP* e fluxo de áudio;

  Codec iLBC: um *codec* de voz de banda estreita para *VoIP* e fluxo de áudio;

  NetEQ para voz: Um *buffer* de instabilidade (tremulação) dinâmico e algoritmo de ocultamento utilizado para esconder efeitos negativos da instabilidade da rede e perca de pacotes, mantendo latência no mais baixo nível possível enquanto permanecendo com alta qualidade de voz;

- Motor de vídeo: um conjunto de ferramentas para mídias de vídeo, a partir da câmera para a rede, e da rede para a tela;

  Codec VP8: um *codec* de vídeo do projeto *WebM* que é bem adequado para comunicações em tempo real por ser designado para baixa latência de rede;

  Buffer de instabilidade de vídeo: ajuda a ocultar os efeitos de tremulação e perca de pacote na qualidade do vídeo no geral;

  Melhorias de imagem: remoções de ruídos no vídeo capturado pela *webcam* e outros ajustes no geral;

- Transporte: camada responsável pela transmissão de dados e está dividido em:

  Pilha *RTP* (*SRTP*): uma pilha de rede para o Protocolo em Tempo Real (*RTP*);

  STUN: Utilitários de Sessão Transversal para *NAT* (*Session Transversal Utilities for Nat*). Este é um protocolo para descoberta do endereço público do cliente e determinação de quaisquer no roteador de rede que podem evitar uma direta conexão com um outro ponto (outro cliente);

  TURN: Transversal utilizando Transmissão em torno do NAT (*Transversal Using Relays around NAT*). Destina-se a ignorar uma restrição conhecida como *NAT* Simétrico (esta restrição é encontrada em alguns roteadores que, ao utilizarem *NAT*, aceitam somente conexões de pontos que o cliente já se conectou antes);

ICE: Estabelecimento de Conectividade Interativa (*Interactive Connectivity Establishment*). Conjunto de ferramentas que permitem o navegador do cliente conectar-se com outros pontos utilizando técnicas como *STUN, TURN, SDP, Offer / Answer*.

3.5.2 Protocolo de Descrição de Sessão (*SDP*)

O *WebRTC* possui um padrão para descrever conteúdos de multimídia que são transmitidos pela rede pelos pares para que cada um entenda um ao outro quando transferindo dados. Este padrão é o *SDP*, Protocolo de Descrição de Sessão (em inglês, *Session Description Protocol*). Apesar destes dados serem transmitidos, eles não são a mídia transmitida de fato, mas metadados, tais como resolução, formatos, *codecs*, encriptação, entre outros.

Segundo Grigorik (2013), *SDP* é um formato de dados utilizado para negociar parâmetros da conexão ponto-a-ponto. Em outras palavras, *SDP* é um protocolo baseado em texto em que descreve-se as propriedades de uma desejada sessão. Conforme Grigorik (2013) explica, as aplicações baseadas em *WebRTC* não precisam se preocupar em lidar diretamente com o *SDP*, pois o Protocolo de Estabelecimento de Sessão *Javascript* já abstrai todos os mecanismos funcionais internos do *SDP* através de alguns métodos de chamadas simples no objeto *RTCPeerConnection*.

Conforme Grigorik (2013) ensina, para haver uma conexão ponto a ponto é necessário que os pares sigam algumas regras simétricas na troca de descrições *SDP* de seus parâmetros, como o tipo de áudio, vídeo, largura de banda, entre outros. A Figura 27 demonstra este funcionamento:

Figura 27 *– WebRTC - Offer e Answer entre pares*

Adaptado de (GRIGORIK, 2013)

O usuário João inicia a tentativa de estabelecimento de conexão, registrando um ou

mais fluxos com seu objeto local *RTCPeerConnection*, em seguida cria uma *Offer* (oferta) e configura a descrição local de sua sessão. João então envia uma oferta de sessão gerada para o outro par. A oferta é, então, recebida por Maria, que por sua vez, configura a descrição da sessão do João como a descrição remota de sua sessão. Em seguida, Maria configura seu próprio fluxo com o objeto *RTCPeerConnection*, gera a *Answer* (resposta) e seta-a como descrição local de sua própria sessão, enviando de volta para João a resposta da sessão gerada. Assim que João receber a *Answer* de Maria, ele seta esta resposta como a descrição remota de sua própria sessão.

Com os passos descritos acima, os pares negociam os tipos de fluxos a serem trocados e suas respectivas configurações. São necessários em seguida checagens de conectividade e transversal de *NAT*. Para conseguir estabelecer uma conexão ponto a ponto, os pares devem ser capazes de rotear pacotes um ao outro, tarefa que se torna difícil atingir na prática devido as númeras camadas de *firewalls* e dispositivos *NAT* entre os pares. (GRIGORIK, 2013)

O canal de sinalização descrito na Figura 27 é o meio pelo qual o *WebRTC* cria conexões entre os pares. Sozinho, o *WebRTC* não consegue criar conexões, dependendo então deste intermediário como sendo uma espécie de canal de comunicação para troca de informações antes de configurar a conexão. A informação trocada é a *Offer* (oferta) e a *Answer* (resposta), que é apenas a *SDP* mencionada acima. (JOUANNEAU et al., 2014)

Os pares, antes de iniciarem a conexão ponto a ponto, precisam passar por um servidor *STUN*. O cliente envia uma requisição para o servidor *STUN* na *Internet*, quem responderá com o endereço público deste cliente e se existe ou não um cliente disponível através do *NAT* do roteador. (JOUANNEAU et al., 2014). A figura abaixo demonstra esta requisição e resposta do servidor *STUN*:

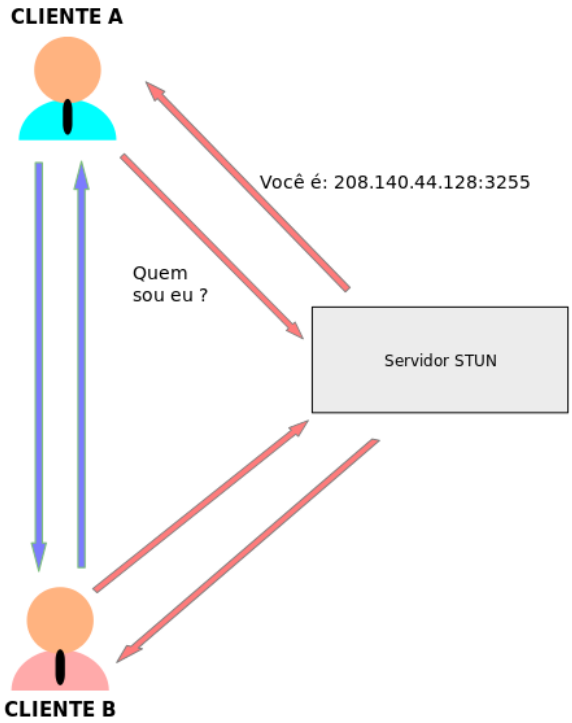

Figura 28 *– WebRTC - Servidor STUN*

Adaptado de (JOUANNEAU et al., 2014)

Algumas vezes se encontram na rede roteadores que utilizam *NAT* com restrições de conexão simétrica, que só aceitam conexões entre pares que já se conectaram anteriormente. Neste cenário se faz necessário então a intervenção do servidor *TURN*. Uma conexão será aberta para este servidor e toda a informação deverá passar por ele. Todos os pares devem então enviar pacotes para este servidor, de modo que o servidor encaminhará toda a informação para o destino. (JOUANNEAU et al., 2014). A figura abaixo exemplifica este caso:

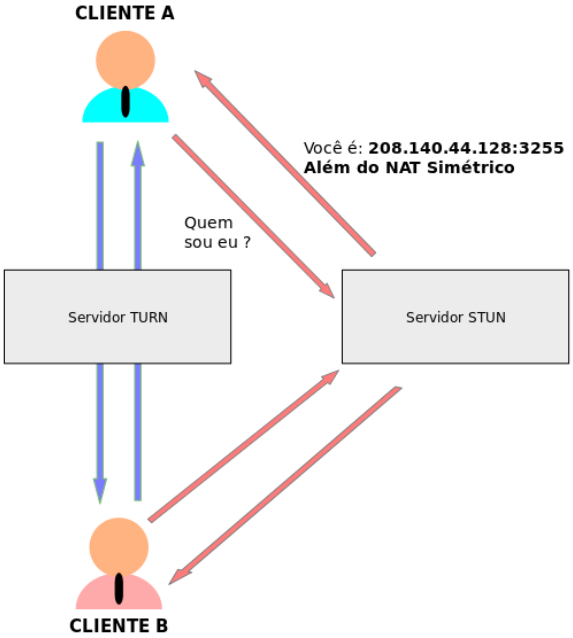

Figura 29 *– WebRTC - Servidor TURN*

Adaptado de (JOUANNEAU et al., 2014)

Apesar dos pares terem disponível um protocolo de descrição como o *SDP*, é necessário informar detalhes sobre a conexão da rede. Esta informação sobre a conexão da rede é conhecida como um candidato *ICE*. Este candidato *ICE* detalha os métodos disponíveis que um par é capaz de comunicar, seja através do servidor *TURN* ou diretamente. (JOUANNEAU et al., 2014). O diagrama de sequência presente na Figura 30 descreve todo fluxo de troca de informações entre os pares:

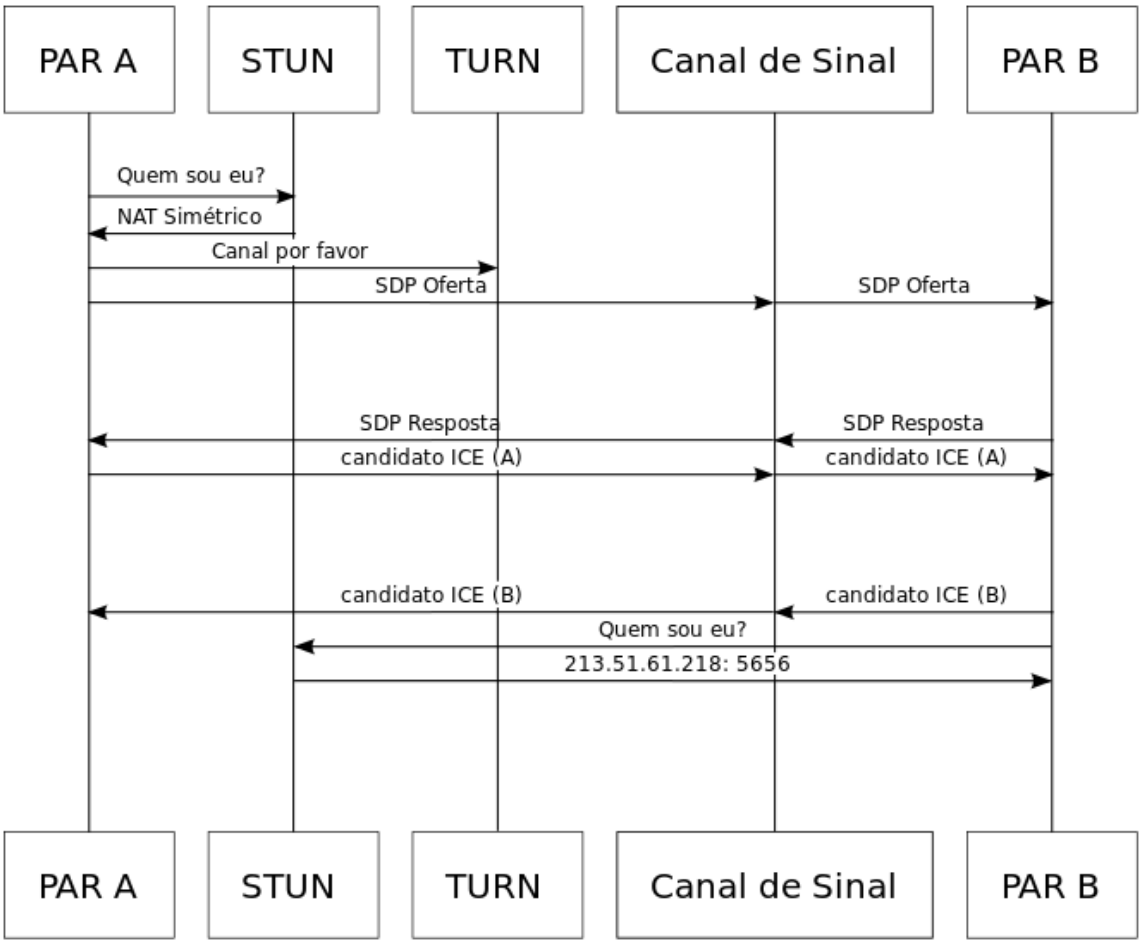

Figura 30 *– WebRTC - Diagrama de Sequência*

Adaptado de (JOUANNEAU et al., 2014)

3.5.3 *WebRTC - TCP vs UDP*

Conforme Grigorik (2013), a comunicação em tempo real é sensitiva ao tempo, ou seja, as variações de tempo exercem bastante fluência para uma comunicação em tempo real. As aplicações de fluxos de áudio e vídeo, devem, em favor da entrega de conteúdo no tempo mais rápido possível, tolerar percas de pacotes dentro da rede durante a transmissão dos fluxos de dados. Em resumo, para comunicação em tempo real, pontualidade e baixa latência são mais importantes do que confiabilidade. Grigorik explica que:

> Fluxo de áudio e vídeo em particular têm que se adaptar para propriedades de nosso cérebro. Acontece que nós somos muito bons em preencher os espaços vazios, mas altamente sensíveis para atrasos de latência. Adicione alguns atrasos em fluxo de áudio, e "isto apenas não parece certo", porém retire algumas amostras entre, e a maioria de nós nem mesmo notará! (GRIGORIK, 2013, tradução nossa)

Ainda segundo Grigorik (2013), a pontualidade sobre a confiança é a razão primária pelo qual o protocolo *UDP* é a transporte preferido para entregas de dados em tempo real. Por

outro lado, o protocolo *TCP* entre confiável e ordenado fluxo de dados. Quando um pacote é perdido na rede, o *TCP* trata de armazenar todos os pacotes após ele, esperar uma retransmissão, e então entregar os pacotes em ordem. Em comparação, o *UDP* não garante entrega de mensagens, ele não tem retransmissões ou *timeouts*. O *UDP* não garante ordem de entrega, não faz reordenamento nem mantém sequência de número de pacote. Ele rastreia estado de conexão, não faz estabelecimento de conexão nem faz controle de congestionamento.

O protocolo *UDP* se torna fundamental para a comunicação em tempo real, pois, apesar de não dar confiabilidade na entrega e na ordem dos dados, ele tenta entregar os pacotes de dados para a aplicação no momento em que eles chegam.

Por apresentar características que se adequam ao modelo de dados em tempo real, o *UDP* é o protocolo de escolha do *WebRTC*. No entanto, como explica Grigorik (2013), é preciso mecanismos para transpor muitas camadas de *firewalls* e *NATs*,negociação de parâmetros para cada fluxo, encriptação de dados de usuário, controle de fluxo e de congestionamento, entre outros serviços. Neste cenário, para atender todos os requisitos do *WebRTC*, um conjunto de protocolos que entreguem estes serviços é necessário estar acima do *UDP*. A Figura 31 demonstra esta arquitetura:

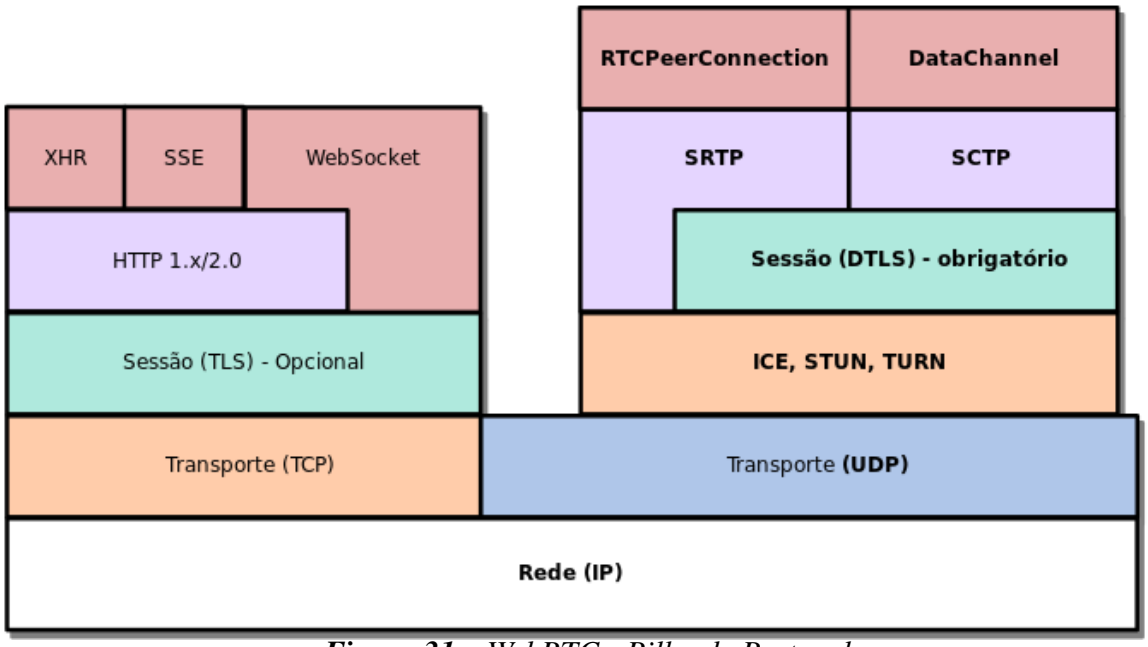

Figura 31 *– WebRTC - Pilha de Protocolos*

Adaptado de (GRIGORIK, 2013)

3.5.4 Benefícios do *WebRTC*

Segundo Sergiienko (2014), o *WebRTC* oferece diversos benefícios para os negócios desenvolvidos na *Web*. Ele classifica-os da seguinte maneira:

- Redução de custos: com o *WebRTC* não é mais necessário pagar por complexas e proprietárias soluções, pois esta tecnologia é gratuita e de código aberto. Há também redução de custos no suporte e na implantação, pois não é mais necessário distribuir software específico para os clientes;
- *Plugins*: Não é necessário *plugins*. Se antes era necessário tecnologias como *Adobe Flash, Java Applets*, complicadas soluções para a construção de aplicações ricas e interativas, obrigatoriedade de instalações de *plugins* para oferta de conteúdo, ou preocupação com diversos sistemas e plataformas, agora com *WebRTC* isto não é mais necessário;
- Comunicação ponto a ponto (*peer-to-peer*): Maioria dos casos a comunicação será estabelecida diretamente entre os clientes sem a necessidade de pontos de intermediação entre eles;
- Facilidade de uso: o *WebRTC* oferece fácil integração de suas funcionalidades para as aplicações *Web* através de *API Javascript* ou *frameworks* específicas, não necessitando de conhecimentos especiais;
- Única solução para todas as plataformas: *WebRTC* é uma tecnologia multiplataforma e universal. Não é necessário desenvolver versões nativas e específicas;
- Livre e gratuito: *WebRTC* possui seu código fonte aberto e é gratuito. Isto leva a certas vantagens sobre tecnologias proprietárias, já que novos bugs podem ser descobertos por qualquer pessoa e são solucionados mais rapidamente. Também possui empresas do porte da *Google, Opera e Mozilla* que participam ativamente do desenvolvimento e suporte desta tecnologia.

Esta tecnologias trás também benefícios e possibilidades mais fáceis de implementação de diversas aplicações *Web* interativas. Grigorik (2013) cita algumas possibilidades:

- Conferências ponto a ponto de áudio e vídeo;
- Transmissão de vídeos pré-gravados;
- Compartilhamento de telas de monitor *Desktop*;

- Transferência e compartilhamento de arquivos ponto a ponto;
- *Webinars*;
- Serviços de entrevista de empregos;
- Serviços de transmissão de rádios;
- Ensino à distância;
- Serviço interativo de encontros, entre outros.

3.6 O Protocolo *HTTP 2.0*

O protocolo *HTTP* vem sofrendo atualizações em seu design conforme a *Web* avança e as demandas mudam. Como a versão 1.1 do *HTTP* apresentava problemas, era preciso evoluir, e com isto, um trabalho na versão 2.0 foi iniciado. Segundo Grigorik (2013), esta versão fará as aplicações mais rápidas, mais simples e mais robustas, na medida que permite-nos desfazer muito das soluções do *HTTP* 1.1 previamente feitas dentro das aplicações e concentrar estas preocupações dentro da camada de transporte, abrindo uma série de novas oportunidades para otimização das aplicações e melhoria de performance.

As metas primárias desta nova versão do *HTTP* são, segundo Grigorik (2013), reduzir a latência por meio de uma completa multiplexação de requisição e resposta, minimizar sobrecarga do protocolo via compressão eficiente dos campos de cabaçalho do *HTTP*, e adicionar suporte para priorização de requisição e *push* no servidor. Esta versão tem o apoio de um conjunto de melhorias de protocolos, como por exemplo, novo controle de fluxo, tratamento de erros e mecanismos de atualização.

A grande diferença desta versão 2.0 para a 1.1 é que ela modifica a maneira como os dados são formatados (enquadrados) e transportados entre o cliente e o servidor. Como Grigorik (2013) explica, a versão 2.0 deixa intacta a semântica que já existia na versão anterior. Conceitos núcleos, como métodos *HTTP*, códigos de *status*, *URIs*, campos de cabeçalhos, entre outros, permanecem inalterados. Tal abordagem de atualização adotada nesta versão produz o benefício de que as aplicações não necessitam serem modificadas para funcionarem com o novo protocolo.

A versão 2.0 do protocolo *HTTP* tem principal influência de outro protocolo já citado neste trabalho: o protocolo *SPDY* da *Google*. Como bem explica Grigorik (2013), no ano de 2012, o experimental protocolo *SPDY* já era suportado pelos navegadores *Google Chrome, Firefox e Opera*, assim como grandes aplicações *Web* como o *Google*, *Twitter* e o *Facebook*. Isto

era uma prova de que o protocolo já oferecia benefícios de grande performance e estava para ser adotado como padrão pela indústria. A partir de então, o Grupo de Trabalho *HTTP* (*Working Group HTTP- HTTP-WG*) deu início ao novo esforço como sendo a nova versão *HTTP 2.0*, aplicando as lições aprendidas com o *SPDY*. O Grupo de Trabalho *HTTP 2.0* utilizou, após discussões iniciais, a especificição do *SPDY* como ponto de partida para os futuros trabalhos no padrão do protocolo, tendo sido feito até hoje, muitas mudanças e melhorias no padrão oficial do *HTTP 2.0*.

Em trecho da carta redigida pela IETF (IETF.ORG, 2012, tradução nossa) destaca-se o escopo do *HTTP 2.0* e os seus critérios chave do *design*:

> É esperado que o HTTP/2.0 fará:
>
> - Substancialmente e mensuravel melhoria da percebida latência do usuário final na maioria dos casos, sobre HTTP 1.1 utilizando TCP;
> - Corrigir o problema "cabeça de linha bloqueante"no HTTP;
> - Não requerer múltiplas conexões para um servidor habilitar paralelismo, portanto melhorando seu uso do TCP, especialmente em relação ao controle de congestionamento;
> - Reter as semânticas do HTTP 1.1, entregando documentação existente, incluindo (mas não limitado a) métodos HTTP, códigos de status, URIs, e onde apropriado, campos de cabeçalhos;
> - Claramente definir como HTTP 2.0 interage com HTTP 1.x, especialmente em intermediários;
> - Claramente definir novos pontos de extensibilidade e política para seu uso adequado.
>
> A resultante especificação (s) é esperada cumprir estas metas comuns implantações existentes do HTTP; em particular, navegação na Web (desktop ou móvel), não-navegadores ("HTTP APIs") serviços na Web (em uma variedade de escalas), e intermediação (por proxies, firewalls corporativas, proxies 'reversos' e Redes de Entrega de Conteúdo). Da mesma forma, atuais e futuras extensões semânticas para o HTTP 1.x (ex: cabeçalhos, métodos, códigos de status, diretivas de cache) devem ser suportados no novo protocolo.
>
> - HTTPbis WG charter
> HTTP 2.0

Conforme Grigorik (2013), o protocolo *SPDY*, ainda que sendo adotado como ponto de partida para o *HTTP 2.0*, continuou a coexistir e evoluir em paralelo. A motivação da continuidade do trabalho sobre o *SPDY* é que ele pode ser alvo de experimentação de novas recursos e propostas para o *HTTP 2.0*, oferecendo testes e avaliação antes de inclusões no padrão oficial do *HTTP 2.0*. Grigorik (2013) argumenta que pelo fato de termos estes dois protocolos coexistindo, há benefícios, como uma especificação mais robusta e extensivamente testada de implementações de cliente e servidor, de modo que, quando o *HTML 2.0* for marcado como "pronto", já existam soluções bem testadas de clientes e servidores, tempo em que o *SPDY* poderá ser descontinuado em favor de uma solução padronizada com o *HTTP 2.0*.

4 Contextualizando a *Web* em Tempo Real

Neste capítulo o trabalho foca em contextualizar e sintetizar casos de uso da indústria na *Web*, e que corroboram para demonstrar os benefícios, mudanças e oportunidades obtidas com a adoção de diversas tecnologias que vêm permitindo a oferta para as demandas atuais, conteúdos e serviços em tempo real, ao passo que, vêm mudando a forma que enxergamos, como usuários finais, as interações com as páginas e aplicações *Web*.

4.1 Casos de Uso

Kirkpatrick (2009c) publicou uma pesquisa com alguns estudos de casos de uso sobre a *Web* em tempo real. Estes casos de uso demonstram os benefícios e mudanças ocorridas nas plataformas, nas quais modelos de fluxos em tempo real mudaram a forma com que seus clientes interagiam. As seções abaixo dividem-se em cada caso individualmente.

4.1.1 O Aplicativo *enjoysthin.gs*

Neste caso de uso, Ted Roden, engenheiro do *New York Times*, construiu um projeto chamado *enjoysthin.gs*, um sítio de favoritos visuais, parecido com o *Delicious*. O sítio oferece recursos para contas premium que dá acesso a visualização de conteúdo em tempo real. Através de uma barra lateral, novos conteúdos, inclusive imagens, são adicionados tão breve o conteúdo é compartilhado dentro do sítio. Kirkpatrick (2009c) relata que, apesar de que *enjoysthin.gs* ser ainda pequeno, implicações da adição de recursos em tempo real para este site provavelmente beneficia sítios de qualquer tamanho.

Kirkpatrick (2009c) relata alguns benefícios detectados neste caso. Ted Roden diz que: “Tempo-em-site tem tido um enorme aumento. É como quando o novo conteúdo vem no *Live Feed* do *Facebook*: se você sabe que ele está para aparecer em cinco segundos, você permanecerá em volta.” [1](informação verbal, tradução nossa)

Outro benefício foi a de diminuição de custos. Ted Roden após implementar a infraestrutura em tempo real disse: "meu site roda muito mais suavemente. Eu provavelmente moverei todo o site para esta tecnologia, porque bem abaixo é muito mais fácil no banco de dados para

[1] Entrevista concedida em (KIRKPATRICK, 2009c)

mim."[2](informação verbal, tradução nossa)

Segundo Brad Fitzpatrick, da *Google*, ao utilizar *PubSubHubhub* para enviar notícias de itens compartilhados no *Google Reader* para *FriendFeed*, mudou de polling para tempo real, o que resultou em redução de tráfego entre os dois sites em 85%. Do mesmo modo, o leitor de notícias *Feedly* disse que a parte do serviço que consome *PubSubHubhub* do *Google Reader* viu uma redução de 72% em largura de banda [3](informação verbal).

Por outro lado, ao introduzir infraestrutura em tempo real, houve complicações de publicidade. Ted Roden constata isto ao comentar:

> Se você nunca clicar em torno de fora da página inicial, então o *Google Analytics* diz que isto é somente uma visualização de página. Agora se você está enviando histórias para o topo da página, então você não sabe quantas histórias as pessoas têm visto, a não ser que você comece a mensurar diferentemente. [4](informação verbal, tradução nossa)

Kirkpatrick (2009c) relata ainda, através de Ted Roden, que fazer publicidade em sites como este, muda totalmente. Ted Roden confirma isto:

> Mensurar engajamento de usuário muda totalmente. Pessoa que usa enjoysthin.gs em uma barra lateral no *Firefox*: eu conto isto como visualização de página completa? Eu a conto como uma, mesmo que pessoas tenham-na aberta por oito horas? Você pode convencer um anunciante que eles estão indo ver uma propaganda 100 vezes enquanto olham na página apenas uma vez, e eles querem isso? Para projetos como enjoysthin.gs, isto está indo ser um mundo assustador lá fora para publicidade por um tempo. [5](informação verbal, tradução nossa)

4.1.2 O Aplicativo *Superfeedr*

Superfeedr é um serviço que traz fontes de notícias a partir da *Web* e então oferece atualizações para estas fontes em formato *XMPP* ou *PubSubHubhub*. Os pontos chaves que Kirkpatrick (2009c) destaca aqui são:

- Oportunidade para adicionar valor através da transformação tecnológica de recursos legados para tempo real;
- A facilidade de levantar, em tempo real, dados normalizados através de uso de serviços como o *Superfeedr*;
- Como os mercados consumidores podem não estar preparados para dados em tempo real como os desenvolvedores.

Diferente de fazer sondagem nos anunciantes de notícias repetidamente para checar por atualizações, Kirkpatrick (2009c) explica que o serviço de consumo de notícias precisa apenas esperar que o *Superfeedr* entrega atualizações automaticamente assim que eles estejam

[2] Entrevista concedida em (KIRKPATRICK, 2009c)
[3] Entrevista concedida em (KIRKPATRICK, 2009c)

disponíveis. Neste caso, o anunciante não necessita nem mesmo publicar notícias em tempo real, pois o *Superfeedr* já toma de conta disto.

Julien Genestoux afirma que: "Para cada fonte, nós tentamos determinar qual o mais apropriado modo de pegar as atualizações: *PubSubHubhub, RSSCloud, SUP, APIs* específicas (*Twitter, stream*, entre outros). Nós fazemos *polling* como plano secundário."[6](informação verbal, tradução nossa)

Além disso, cita que há uma interdependência entre os serviços de feed dentro da *Web* em tempo real e que tempo real pode ser um diferencial. Genestoux revela: "O mercado é enorme. No fim, todo mundo está indo necessitar de tempo real. Isto está indo ser o diferencial entre os serviços."Por fim, Genestoux acredita que a *Web* em tempo real mudará como as aplicações *Web* são construídas e interagem junto. [7](informação verbal, tradução nossa)

Por outro lado, (KIRKPATRICK, 2009c) revela que Genestoux admite que nem todas as companhias se sentem confortáveis em confiar em serviços de terceiros para tal tipo de funcionalidade. (KIRKPATRICK, 2009c) cita alguns outros serviços do tipo serviço-como-um-tempo-real, do estilo do *Superfeedr*, como o *Notfiy.me* e o *Kaazing*.

4.1.3 *Trigger*: A Tecnologia de Análise de Notícias da Companhia *Evri*

A companhia *Evri* observa fontes de notícias para ver quando um tópico de notícias é a tendência do momento, o que inclui artigos na *Wikipedia* que publicamente exibem dados disponíveis que tem saltado em visualizações de página. Ela visita base de dados como *Wikipedia* e *FreeBase* para checar por atualizações para as relacionadas entidades. Sua infraestrutura então cria ou atualiza uma página de tópico com novos *links*, imagens e resultados de busca no *Twitter*.

Kirkpatrick (2009c) resume as contribuições que a companhia *Evri* fez para o entendimento da *Web* em tempo real:

- Criativas maneiras movendo fontes de dados em tempo real e mais lentas podem ser usadas juntas para criar valor;
- *Wikipedia* como uma fonte de dados em tempo real além do *Twitter* e *Facebook*;
- Exemplo de análise de texto como uma parte muito importante de fornecedor de serviço trabalhando em entrega de conteúdo sensitivo ao tempo;
- Lutas vividas por empresas iniciantes futuras buscando trazer serviços em tempo real para empresas mais antigas.

[6] Entrevista concedida em (KIRKPATRICK, 2009c)
[7] Entrevista concedida em (KIRKPATRICK, 2009c)

4.1.4 *Web* em Tempo Real nos Negócios de Música

Neste estudo de caso, Kirkpatrick (2009c) relata uma aplicação de *dashboard* (painel, em português), desenvolvida por Ethan Kaplan, que funciona em tempo real para exibir o número de pessoas quem visitam cada website de um artista da *Warner Bros* a qualquer tempo. Quando um site aparece no painel, a equipe ver em um gráfico dados estatísticos sobre resultados de pesquisas de *blogs*, *Twitter* e outros lugares para determinar o que causou o aumento no gráfico e responder imediatamente. Kaplan relata os benefícios:

> Nós costumávamos sermo orientados em volta de obtenção de dados somente uma vez por semana, por causa que era assim nós erámos alimentados de *SoundScan*, Mediabase, etc. Teríamos então que conciliar os dados com base no nosso plano para a semana.
> Agora nós temos um completo *back-end* que expõe dados em quase e em tempo real: compras dentro do sistema, visitas ao site, visitantes logados, comentários deixados. A cultura deste ambiente em tempo real tem impactado em como as bandas estão sendo divulgadas e produtos criados. Pessoas querem mais e mais em tempo real.
> Um dia, por exemplo, eu vi um site com tráfego marginal que, de repente, teve 7.000 pessoas nele. Nós fizemos uma pesquisa no *Twitter*, checamos o *WeSmith.com* [*blog* agregador de celebridade] e encontramos que este artista estava tendo um bebê. Ninguém nos contou! Nós imediatamente começamos a planejar mudar a propaganda no site, talvez um chá de bebê; nós adicionamos uma enquente perguntando às pessoas se eles pensavam que seria um menino ou menina; todos os passos para tomar vantagem do tráfego que esta vindo para o site naquele momento.
> Alguma coisa como esta acontece todo o dia. Um dos sites pode estar tendente mais do que o comum, porque o artista acaba de lançar uma gravação. Nós podemos correlatar e reagir da correta maneira. Omniture é um bom dado, mas ele não é tão rápido quanto temos aqui. [8](informação verbal, tradução nossa)

Novamente, Kirkpatrick (2009c) extrai algumas lições a partir deste estudo:

- Uma legada indústria capaz de tomar novas formas de ação baseada em substanciais reduções em atrasos na entrega de informações;
- O valor de ter seus próprios dados em tempo real, ao invés de confiar em serviços de terceiros;
- Oportunidades que surgem por ser capaz de criar interfaces para exibir dados em tempo real internamente;
- Oportunidades ainda inexploradas quando dados em tempo real são analisados em massa.

4.1.5 A Cruz Vermelha Salvando Vidas Com a *Web* em Tempo Real

A *Web* em tempo real não está apenas mudando nossas vidas *online*; ela está começando a fazer grande diferença *offline* também. Esforços de socorro na Cruz Vermelha Americana tem sido transformadas pela tecnologia em tempo real. *Walmart* pode ser famoso por seu poderoso sistema de controle de inventário, mas algumas pessoas dizem que a Cruz Vermelha está

tornando-se outro principal exemplo de uma altamente efetiva organização de larga escala coordenando atividades ao redor do mundo em tempo real. (KIRKPATRICK, 2009c)

Kirkpatrick (2009c) divulgou em seu estudo de casos, alguma declarações de Michael Spencer, líder da tecnologia *SharePoint*, na *American Red Cross National Headquarters* (Sede Nacional da Cruz Vermelha Americana):

> A Cruz Vermelha tem estado em volta de 100 anos. Eu tenho estado aqui por 12 anos, e com o que eu tenho visto no último ano em termos de informação em tempo real, coordenação e nosso painel supervisionando tudo, eu acho que nós temos feito avanço de 50 anos em um ano ou dois por causa das tecnologias em tempo real. Na Cruz Vermelha, a Web em tempo real salva vidas. [9](informação verbal, tradução nossa)

Kirkpatrick (2009c) comenta que a Cruz Vermelha mantem latência e tempo de inatividade no site "*Safe and Well*"o mais baixo possível. Spencer também comenta que uma coisa que a organização tem que fazer quando respondendo para disastres é verificar as reivindicações da perca da casa que as pessoas arquivam e que isto costumava tomar muito tempo, mas não mais. [10](informação verbal)

Michael Spencer corrobora:

> Neste último ano, nós temos visto voluntários saírem com *PDAs*. Nós costumávamos anda ao redor com um carro e folha de papel para verificar danos. Agora nós temos dispositivos portáteis que permitem tomar uma foto da casa – ele tem *GPS* nele – enviá-lo para um satélite, e então nós podemos fazer monitoramente em tempo real a partir de um painel.
> Aquela visão de painel das casas danificadas? Isso deveria tomar semanas antes. Agora nós podemos fazê-lo imediatamente. O governo pode também fazer sobrevôos que alimentam estimativas aproximadas de dano a partir do avião para dentro do nosso portal, então nós podemos pegar uma visão geral dentro de poucas horas, e então nossos voluntários saem com seus dispositivos. Isso costumava me tomar uma semana e meia ou duas semanas, até mais. Eu nunca poderia pegar um sobrevôo pelo governo ou ter meus voluntários dentro. Agora é alimentado automaticamente para meu painel. Eu não tenho que chamar pessoas e relatar novos números. Nós até costumávamos fazer números abrigos pela mão para encomendar refeição. Agora isto é tudo feito através da *Web*. [11](informação verbal, tradução nossa)

Os pontos principais destacados por Kirkpatrick (2009c) são:

- As consequências no mundo real da tecnologia em tempo real;
- Transformação de uma legada instituição utilizando tecnologia em tempo real;
- Estratégica dependência em *software* de terceiros em um contexto em tempo real;
- A importância do planejamento, relativa para a implementação da tecnologia.

[10] Entrevista concedida em (KIRKPATRICK, 2009c)

5 Estudo Prático

Este capítulo abordará as implementações e estudos práticos feitos sobre o desenvolvimento de aplicações *Web* em tempo real. Foi desenvolvido duas aplicações: a primeira aplicação foi um *chat* em tempo real com a plataforma *Node.js*, que foi a base para um teste comparativo, descrito no capítulo seguinte, e a segunda uma rede social de perguntas e respostas com a *framework MeteorJS.js*. Nas seções subsequentes será detalhada cada uma das ferramentas utilizadas, bem como as tecnologias que fazem parte de cada uma destas plataformas.

5.1 A Plataforma *Node.js*

Na JSConf Europeu em 2009, o jovem programador Ryan Dahl apresentou um projeto na qual estava trabalhando. Conforme Rai (2013), Ryan Dahl estava interessado em construir um pequeno servidor *Web* em *Javascript* e se inclinava por esta linguagem devido a ela ser independente de sistema operacional e vir sem qualquer *API* de *I/O* (entrada e saída). O projeto que apresentou na JSConf Europeu, nomeado *Node.js*, era uma plataforma que combinava com o motor *Javascript V8* do *Google*, um *event loop*, e uma *API I/O* de baixo nível, resultado de seu esforço e conhecimentos adquiridos. No entanto, antes de alcançar este resultado, Ryan Dahl desenvolveu outros projetos, como o *Ebb*, servidor *Web* baseado em *Ruby* e *C*, porém percebeu que ele não estava funcionando tão rápido quanto ele queria. Além deste, fez vários outros experimentos no desenvolvimento de um número de pequenos servidores *Web*. (RAI, 2013)

O diferencial deste projeto era que ele não era como outras plataformas *Javascript* do lado servidor onde todas as primitivas de entrada e saída eram orientados a evento. *Node* oferece uma infraestrutura puramente orientada a eventos, porém não-bloqueante para o desenvolvimento de programas altamente concorrentes. (TEIXEIRA, 2013)

Segundo o sítio oficial da ferramenta, *Node.js* é uma plataforma construída sobre o *Javascript runtime* do *Chrome* para facilmente desenvolver rápidas, escaláveis aplicações de rede. *Node.js* utiliza um modelo orientado a eventos, não-bloqueante *I/O* que o faz leve e eficiente, perfeito para aplicações em tempo real de dados intensivos que executam em multi dispositivos distribuídos. (NODEJS.ORG, 2014)

Segundo Teixeira (2013), *Node* tem recebido atenção de grandes entidades na indústria, utilizando-o para implantar serviços em rede que são rápidos e escaláveis. Ele cita algumas razões do porquê utilizar *Node*:

- A linguagem *Javascript: Node.js* utiliza a linguagem *Javascript*. Ela é a linguagem mais largamente utilizada no mundo. Maioria dos desenvolvedores *Web* estão acostumados a utilizar *Javascript* no navegador *Web* e o servidor é uma extensão natural para isto;

- Simplicidade: *Node.js* é simples. As funcionalidades núcleo de *Node.js* são reduzidas ao mínimo e todas as *APIs* existentes são elegantes, expondo o mínimo de complexidade possível aos desenvolvedores. Quando necessário desenvolver algo mais complexo, basta pegar, instalar e utilizar vários módulos de terceiros;

- Fácil de iniciar: Outra razão que Teixeira aborda é a facilidade de se iniciar com *Node.js*. Você pode baixar e instalá-lo facilmente e em questão de minutos estar com tudo configurado.

Somado a estas razões, Rai (2013) corrobora ao explicar mais uma vantagem de *Node.js*. Segundo ele, a primeira e principal vantagem desta plataforma é a linguagem *Javascript*. Se o desenvolvedor conhece e codifica em *Javascript* regularmente, ele já sabe a maior parte de *Node.js*, faltando aprender somente *APIs* e melhores práticas. Esta plataforma permitiu aos desenvolvedores escrever inteiras aplicações com *Javascript*, utilizando-a, além do *frontend*, no *backend* da aplicação, economizando o desenvolvedor de *frontend* de ter que aprender uma nova linguagem ou confiar na entrega de serviços de outro desenvolvedor para ser consumido pela sua aplicação.

5.1.1 O Modelo Orientado a Eventos Assíncrono e o *Event Loop*

Para Rai (2013), o que diferencia *Node.js* dos outros servidores e aplicações tradicionais é o seu modelo de desenvolvimento orientado a eventos assíncrono. *Node.js*, conforme sendo uma tecnologia construída com *Javascript*, se beneficia da característica de *callbacks* e eventos da linguagem. A tecnologia incorpora esta filosofia e todo e qualquer aspecto da plataforma, seja em tratamento de requisições de servidor, *I/O*, ou interações com base de dados. *Node.js* idealmente será manipulado por um *callback* acomplado a um evento por um ouvinte (*listener*, em inglês).

Este modelo orientado a eventos assíncrono é obtido aplicado através dos *callbacks* acomplados aos eventos. Neste contexto, Pereira (2014a) diz que:

> O *Event-Loop* é o agente responsável por escutar e emitir eventos no sistema. Na prática ele é um loop infinito que a cada iteração verifca em sua fila de eventos se um determinado evento foi emitido. Quando ocorre, é emitido um evento. Ele o executa e envia para fila de executados. Quando um evento está em execução, nós podemos programar qualquer lógica dentro dele e isso tudo acontece graças ao mecanismo de função *callback* do *Javascript*.

Ele acrescenta que o modelo orientado a eventos do *Node.js* foi inspirado pelas *frameworks Event Machine*, do *Ruby*, e *Twisted* do *Python*, diferenciando-se por ser mais performático, devido a sua natureza não-bloqueante.

Rai (2013), por sua vez, nos oferece uma analogia bem interessante para explicar o modelo orientado a eventos assíncrono e o *Event Loop*:

> Considere um restaurante onde você vai para o caixa, coloca o seu pedido, e espera até que sua comida esteja pronta. Neste caso, o caixa não pode servir os outros clientes até que você tenha o seu pedido, e a fila fica bloqueada. Se o restaurante tem um grande fluxo de clientes e precisa ampliar, eles vão ter que investir na contratação de maior número de caixas, criando mais contadores, e assim por diante. Isto é semelhante ao modelo de multithreading tradicional.
> Por outro lado, vamos ver o modelo de muitos outros restaurantes usam. Neste caso, você vai para o caixa e faz seu pedido (que ele/ela entrega para a cozinha); ele/ela então aceita o seu pagamento e lhe dá uma senha. Você, então, se afasta, e o caixa passa para o próximo cliente. Quando o pedido está pronto, o servidor da cozinha anuncia isso chamando o seu nome ou pisca o seu número de senha e você caminha e busca a sua encomenda. Esta abordagem orientada a eventos otimiza o trabalho do caixa e permite que você aguarde na lateral, liberando os recursos relevantes para servir os outros até que o seu trabalho esteja feito.
> Em Node.js, o servidor é o caixa, e todos os manipuladores são a equipe da cozinha. O servidor aceita o pedido e passa-o para um manipulador. Ele então se move para aceitar outros pedidos. Quando o pedido é processado e os resultados estão no lugar, a resposta é enfileirada no servidor e enviado de volta para o cliente quando ele atinge a dianteira da fila. (RAI, 2013, tradução nossa)

Rai (2013) explica que oposto a abordagem tradicional de lançamento de *threads* ou processos do servidor (similar a adicionar mais caixas, no exemplo dado), o método adotado pelo *Node.js* é mais eficiente, conforme os trabalhadores tem dedicadas responsabilidades, o que torna muito mais leve e barato do que replicar todo o servidor para escalar a aplicação. Em outras palavras, ele também explica que a vantagem se dá devido ao design não-bloqueante, que libera recursos do servidor que, ao contrário, estariam sendo desperdiçados caso o servidor permanecesse na espera pela requisição, de modo que assim, os recursos podem ser utilizados para outras tarefas na fila.

5.1.2 A Biblioteca *Socket.IO*

Conforme Rai (2013), esta biblioteca é um módulo disponível através do *NPM* (*Node Package Manager*) que oferece um servidor e cliente fácil para fazer envio de atualizações em tempo real entre o servidor e o cliente. Ela é, conforme Grigorik (2013), uma popular biblioteca que oferece uma implementação do objeto *WebSocket*, mas vai um passo além por oferecer um servidor personalizado que implementa suporte para *WebSocket* e uma variedade de transportes alternativos. Grigorik (2013) afirma que ela se destaca de algumas outras bibliotecas de mesma natureza por implementar recursos extras, como *hearbeats, timeouts*, suporte para reconexões automáticas, além de funcionalidade multitransporte de fallback.

Rai (2013) corrobora ainda acrescentando que, a biblioteca *Socket.IO* oferece uma *API* muito simples e fácil de se utilizar, que expõe muitas funcionalidades. Além disto, ela funciona

multi-navegador e uniforme sobre os vários mecanismos de transporte oferecidos.

Apesar de já existir um protocolo padrão para comunicação em tempo real, o já estudado *HTML5 WebSockets*, é interessante o uso e existência de bibliotecas como o *Socket.IO*, que são construídas no topo de *WebSocket*, mas ainda oferecem compatibilidade multi-navegador. Rai (2013) explica que o protocolo *WebSocket* ainda não é suportada por todos os navegadores, e, somado a isto, problemas com *firewall* e servidores de *proxy* bloqueando comunicação e não permitindo conexão *WebSocket* ser estabelecida, pode fazer com que muitas pessoas não sejam capaz de utilizar a aplicação. É neste cenário que a biblioteca *Socket.IO* auxilia proporcionando mecanismo de transporte de segunda via, caso *WebSocket* não esteja disponível. Rai apresenta a seguinte ordem de tentativa de uso tecnologias por parte do *Socket.IO*:

- *WebSocket*;
- *FlashSocket*;
- *XHR long polling*;
- *XHR multipart streaming*;
- *XHR polling*;
- *JSONP polling*;
- *iframe*.

5.1.3 A *framework Express*

Segundo Pereira (2014a), a ferramenta *Express* é uma *framework* com o objetivo de facilitar o desenvolvimento de aplicações *Web* de larga escala. Contribui na medida em que facilita trabalhar com *APIs* como a *HTTP*, diminuindo a complexidade do projeto e facilitando a manutenção. Sua filosofia é inspirada em outra *framework* conhecida como *Sinatra*. Com ela, é possível criar serviços *REST*, aplicações *Web* tanto em padrão *MVC* (Modelo-Visão-Controlador) quanto em *MVR* (Modelo-Visão-Rotas), ou até mesmo sem padrão algum, cabendo ao desenvolvedor aplicar o que for mais é adequado ao projeto em questão.

Conforme o sítio oficial da ferramenta, *Express* é uma *framework* de aplicações *Web Node.js* flexível e mínima, ainda oferecendo um robusto conjunto de recursos para desenvolver aplicações de única e multi-página e aplicações híbridas.

5.2 A *Framework MeteorJS*

Após o surgimento da plataforma *Node.js*, um mundo se abriu para desenvolvedores *Javascript* pelo lado do *backend* do servidor. Se antes os desenvolvedores *Web* eram acostumados a trabalhar com *Javascript* somente do lado do cliente, manipulando o *DOM*, seja através de *Javascript* puro ou através de bibliotecas como o *JQuery*, agora se fazia capaz, através de uma plataforma que continua crescendo em adoção, escrever aplicações completas somente com *Javascript*, utilizando-o tanto do lado cliente quanto do lado servidor.

Conforme Pereira (2014b), graças ao *Node.js*, diversos *frameworks* surgiram com o passar do tempo, e, diferente de outras linguagens, existem mais de 30 *frameworks* para *Node.js*. Dentre tantas, Pereira (2014b) destaca as *frameworks Express* e *MeteorJS* (também conhecido como *MeteorJS*).

Pereira (2014b) ressalta que, com um bom conhecimento sobre *Express* e outras *frameworks*, o desenvolvedor é capaz de ter velocidade suficiente para desenvolver uma aplicação de forma ágil. Neste cenário entra a *framework MeteorJS*, enquanto ela preocupa-se em entregar componentes prontos que são facilmente customizáveis e com configurações complexas simplificadas em alto nível, aplicando os princípios de convenção sobre configuração (PEREIRA, 2014b). Pereira argumenta que esta abordagem deu certo com a *framework Rails* e que agora temos uma inovação semelhante para a plataforma *Node.js*. Pereira (2014b) define a ferramenta:

> O *MeteorJS* é um *framework Web full-stack* 100% *Javascript*, ou seja, com ele você vai construir aplicações programando em todas as camadas: cliente, servidor e banco de dados, usando *Javascript*, *Node.js* e *MongoDB*, tudo isso utilizando apenas uma única linguagem de programação, o *Javascript*. (PEREIRA, 2014b)

Para Coleman e Greif (2013), *MeteorJS* é uma plataforma construída no topo de *Node.js* para desenvolvimento de aplicações *Web* em tempo real. Ele é o que vai entre a base de dados da aplicação e a interface do usuário e é o que assegura que ambos se mantenham me sincronia. *MeteorJS* também é capaz de fazer uso de *Javascript* tanto no servidor quanto no cliente, porém com o diferencial de compartilhar código entre os dois ambientes. Coleman e Greif (2013) explica que o resultado disto é uma plataforma que consegue ser muito poderosa e muito simples pela abstração de muito dos habituais problemas e armadilhas do desenvolvimento de aplicações *Web*.

5.2.1 Vantagens da Plataforma *MeteorJS*

Coleman e Greif (2013) citam algumas vantagens que podem levar os desenvolvedores a escolher esta *framework* como ferramenta. Eles acreditam que uma primeira razão é a facilidade de aprendizagem de para trabalhar com a ferramenta. Outro motivo é a rapidez com

que é possível desenvolver aplicações Web em tempo real, sendo possível prototipar uma real aplicação e implantá-la em questão de horas. Além disso, há a vantagem de ser uma ferramenta que utiliza uma única linguagem: *Javascript*.

Pereira (2014b) corrobora com a assertiva sobre a facilidade de aprendizado com *MeteorJS*: "Trabalhar com *MeteorJS* é trabalhar com *Javascript*, e isso faz com que muitos desenvolvedores tenham uma curva de aprendizado rápida, e em poucos dias de estudo você terá dominado os principais pontos e aspectos desta *framework*."(PEREIRA, 2014b)

O sítio oficial do *MeteorJS* (METEOR.COM, 2014) classifica alguns pontos chave que podem ser vantajosos para quem utilizá-la:

1. Puro *Javascript*: escreva sua aplicação inteiramente em puro *Javascript*, com todas as *APIs* disponíveis no lado cliente e no lado servidor, de modo que o mesmo código por ser executado facilmente em qualquer ambiente;

2. Atualizações de página ao vivo: qualquer *template* será atualizado automaticamente conforme os dados na base de dados muda, não sendo mais necessário qualquer código extra para atualizar as páginas. Suporte para qualquer linguagem de *template*;

3. Limpa e poderosa sincronização de dados: escreva seu código como se eles estivessem rodando no servidor e tenha acesso direto a base de dados, não sendo mais necessário carregar dados a partir de terminais *REST*;

4. Compensação de latência: quando o usuário faz uma mudança, a tela atualiza imediatamente, sem espera de confirmação do servidor. Se o servidor rejeitar as mudanças ou executá-la de maneira diferente, a tela do cliente é atualizada com o que realmente mudou;

5. Envio de atualizações no código: atualize sua aplicação enquanto seus usuários estão conectados sem perturbá-los. Quando uma nova versão de código é enviada, este código é perfeitamente injetado dentro de cada quadro do navegador em que a aplicação está aberta;

6. Código sensível executa em ambiente privilegiado: escreva seu código totalmente em *Javascript*, caso queira, com a interface de usuário rodando no navegador e funções sensíveis são executadas em ambiente privilegiado no servidor;

7. Pacotes de aplicativos totalmente auto-suficientes: basta um comando para compilar toda a aplicação em um arquivo *tarball* (arquivo compactado). Basta descompactar em qualquer lugar que possua um servidor *Node.js* executando e a aplicação está no ar;

8. Interoperabilidade: é possível conectar qualquer coisa para *MeteorJS*, desde aplicativos móveis nativos para bancos de dados legados para *Arduinos*. Basta implementar um protocolo simples *DDP* (*Data Distribution Protocol*);

9. Pacotes inteligentes: os pacotes inteligentes do *MeteorJS* são apenas pequenos programas que podem injetar código dentro do cliente ou do servidor, ou até mesmo atrelar para o empacotador para pré processar sua fonte.

5.2.2 *Framework* de *Frameworks*

Pereira (2014a) ressalta que o *MeteorJS* é *framework* de muitos recursos, sendo constituído por vários outras bibliotecas que fazem desta ferramenta uma plataforma muito modular. Ele divide a arquitetura em algumas frameworks principais:

- *SockJS*: *framework* emulador de *WebSockets*, sendo responsável pelo funcionamento do protocolo *DDP* (*Data Distribution Protocol*);
- *MongoDB*: banco de dados *NoSQL* padrão do *MeteorJS*;
- *Handlebars*: um *template engine* para gerar *HTML*. Nas versões mais recentes do *MeteorJS*, o *template engine* padrão agora é o *Spacebars*, que nada mais é do que uma extensão ao *Handlebars*;
- *PubSub*: biblioteca de emissão e escuta de eventos via padrão *Publisher/Subscriber*;
- *MiniMongo*: *API client-side* que interpreta a maioria das funcionalidades do *MongoDB*;
- *Connect*: módulo *Node.js* que possui funcionalidades para interoperabilidade com o protocolo *HTTP*.

5.2.3 Os Princípios do *MeteorJS*

Conforme Pereira (2014b) e o próprio sítio oficial da ferramenta (METEOR.COM, 2014), a ferramenta apresenta sete princípios interessantes de se conhecer:

1. *Data On the Wire*: não envie *HTML* pela rede, mas sim apenas dados, deixando que o cliente decida como apresentá-los;

2. Uma Linguagem: escreva em *Javascript* em ambas as partes para o cliente e servidor;

3. Base de dados em todo lugar: use uma mesma e transparente *API* para acessar sua base de dados tanto do cliente quanto do servidor;

4. Compensação de Latência: no cliente, utilize pré-busca e simulação de modelo para fazer parecer que você tem uma conexão com a base de dados de latência zero;

5. *Full Stack Reactivity*: faça tempo real o padrão. Todas as camadas, desde a base de dados até o modelo, deve fazer uma interface orientada a eventos disponível;

6. Abrace o Ecossistema: a *framework MeteorJS* é de código fonte aberto e integra, ao invés de substituir, ferramentas de código fonte aberto e *frameworks*;

7. Simplicidade igual Produtividade: desenvolva mais recursos de forma rápida e simplificada. O *MeteorJS* mantém um conjunto de *APIs* fáceis de implementar e a comunidade *MeteorJS* está sempre colaborando para evolução do *framework*. A melhor maneira de se fazer algo parecer simples é ele ser realmente simples.

5.2.4 O Banco de Dados *MongoDB*

Segundo o sítio oficial da ferramenta (MONGODB.ORG, 2014), o *MongoDB* é um banco de dados de documentos que oferece alta performance, alta disponibilidade, e fácil escalabilidade. O sítio também destaca alguns recursos chaves da ferramenta:

- Flexibilidade: *MongoDB* armazena dados em documentos do tipo *JSON*. Este tipo oferece um modelo de dados rico que mapeia para nativos tipos de linguagem de programação. *MongoDB* possui um esquema dinâmico que faz mais fácil evoluir o modelo de dados do que os esquemas rígidos dos tradicionais modelos de bancos de dados relacionais;

- Poder: *MongoDB* oferece muitos recursos de um tradicional modelo de bancos de dados relacionais como índices secundários, consultas dinâmicas, ordenação, atualizações ricas, *upserts*, entre outros;

- Velocidade e Dimensionamento: permite manter dados relacionados juntos em documentos, o que permite que as consultas possam ser muito mais rápidas do que num banco de dados relacional na qual os dados estão separados em múltiplas tabelas. *MongoDB* também torna fácil o dimensionamento do banco de dados, permitindo aumentar a capacidade sem qualquer tempo fora do ar, que é muito importante na *Web* quando uma carga pode aumentar subitamente e tirar do ar um website para manutenção prolongada, o que pode custar grande quantia de receita perdida;

- Facilidade de uso: *MongoDB* é fácil de instalar, configurar, manter e usar. Ela oferece algumas configurações de opção e tenta fazer automaticamente fazer a coisa certa sempre que possível. Isto faz a ferramenta funcionar fora-da-caixa, permitindo ao desenvolvedor focar na aplicação ao invés de gastar tempo com tunagem do banco de dados através de configurações obscuras.

Uma instalação *MongoDB* hospeda uma série de banco de dados, na qual os bancos de dados mantém um conjunto de coleções. As coleções, por sua vez, possuem um conjunto de documentos e estes são um conjunto de pares chave-valor. Os documentos possuem esquema dinâmico, ou seja, os documentos na mesma coleção não necessitam ter os mesmos campos ou estrutura e campos em comum podem armazenar diferentes tipos de dados (MONGODB.ORG, 2014).

5.2.5 A Biblioteca *SockJS*

Conforme Grigorik (2013), a biblioteca *SockJS*, assim como seu semelhante, *Socket.IO*, é uma biblioteca para simplificar a distribuição multi-navegador das funcionalidades do protocolo *WebSocket*, implementando um servidor personalizado que o estende com funcionalidades extras. Esta biblioteca é a base para o protocolo *DDP* que o *MeteorJS* utiliza para transporte de dados sobre a rede.

5.2.6 O Protocolo de Dados Distribuídos (*Distributed Data Protocol*)

O protocolo que lida com transporte de dados no *MeteorJS* é o *DDP*. Segundo Pereira (2014b), ele é um servidor inteiramente dedicado para lidar com a biblioteca *PubSub*, as funções *MeteorJS.methods* e as funções do *MongoDB*, funcionando graças a implementação personalizada do protocolo *WebSocket*, a framework *SockJS*. O *DDP* é o encarregado de ser o transporte para a biblioteca *PubSub* enviar e receber dados através de objeto *JSON* em tempo real.

Pereira (2014b) também explica que o *MeteorJS* trabalha com servidor *HTTP*. Este servidor é o responsável por servir conteúdo estático e tratar requisições do cliente. Internamente utiliza o módulo *Connect* do *Node.js* para este fim.

5.3 Uma Aplicação de Exemplo

Neste seção será demonstrada um passo a passo de algumas técnicas utilizadas no desenvolvimento de uma aplicação de exemplo, a qual traz interações em tempo real com o usuário. Não será demonstrado recursos como estilização das páginas, autenticação, autorização, ou comandos de execução *scripts* do *MeteorJS*, desde que estas funcionalidades se encontram

fora do escopo deste trabalho e esta seção também não tenta ser um tutorial completo sobre desenvolvimento com a *framework MeteorJS*. No entanto, a aplicação está disponível, em sua integridade, no repositório: *https://github.com/aislanmaia/ask-us* e está aberto para qualquer pessoa visualizar, modificar ou distribuir sem qualquer aviso prévio.

Para o desenvolvimento da aplicação, foi utilizado a *framework MeteorJS*, que utiliza, como parte de sua arquitetura, a tecnologia *Node.js*, a linguagem de marcação *HTML 5*, a linguagem de programação *Javascript* e a folha de estilos *CSS*. O editor de código fonte utilizado foi o *Vim 7.4* e o Sistema Operacional foi o *Elementary OS Luna*, que é uma distribuição *Linux* derivada do *Ubuntu 12.04*.

A aplicação desenvolvida é uma rede social de perguntas e respostas. Nesta rede é possível cadastrar ou pesquisar por tópicos com alguma pergunta e interagir em tempo real com outros usuários por meio de respostas. Cada pergunta poderá ser seguida e, ou, respondida por outro usuário, enquanto a aplicação trata de notificar os autores das perguntas em tempo real do que está ocorrendo. Também é possível aprovar uma resposta.

5.3.1 Diagrama de Entidades

Na aplicação temos as entidades: *Users* (Usuários), *Questions* (Questões), *Replies* (Respostas), *QuestionsFollowed* (QuestõesSeguidas), *Notifications* (Notificações). Os usuários podem ter zero ou muitas perguntas, enquanto uma pergunta deve obrigatoriamente pertencer a um usuário. Um usuário pode ou não seguir muitas perguntas, mas uma pergunta seguida deve se referenciar a um usuário. O mesmo se aplica no relacionamento entre usuários e respostas.

Dentro do sistema temos três tipos de notificações: as notificações para um usuário que uma pergunta sua foi seguida por outro usuário, as notificações de novas respostas de outros usuários para uma determinada pergunta pertencente a um usuário e as notificações de aprovações nas respectivas respostas de um determinado usuário. A Figura 32 a seguir demonstra o relacionamento entre as entidades:

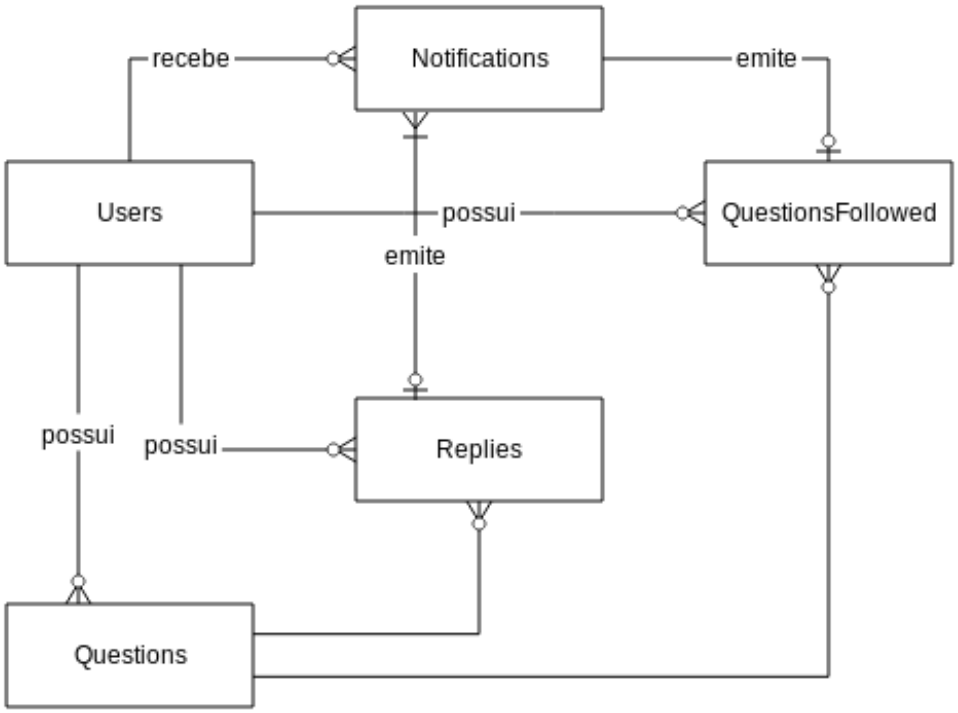

Figura 32 *– Diagrama Entidade-Relacionamento*

5.3.2 Convenções do Projeto

No projeto de desenvolvimento deste *software* foi utilizado algumas convenções de modo a padronizar e seguir algumas boas práticas de programação.

A primeira convenção utilizada neste projeto é a de escrita de código usando somente o inglês. Como o inglês é um idioma mundial, ter toda a base de código neste idioma, incluindo além do código em si, comentários e documentação, facilita a contribuição por qualquer pessoa do mundo.

A segunda convenção está relacionada a *framework MeteorJS* e incide na organização da estrutura de diretórios do projeto. Esta *framework* não impõe ao projeto nenhuma estrutura específica de diretórios e arquivos, porém é de boa praxe organizar o código em uma estrutura que faz sentido e facilita a manutenção. Abaixo há uma figura demonstrando a estrutura do projeto:

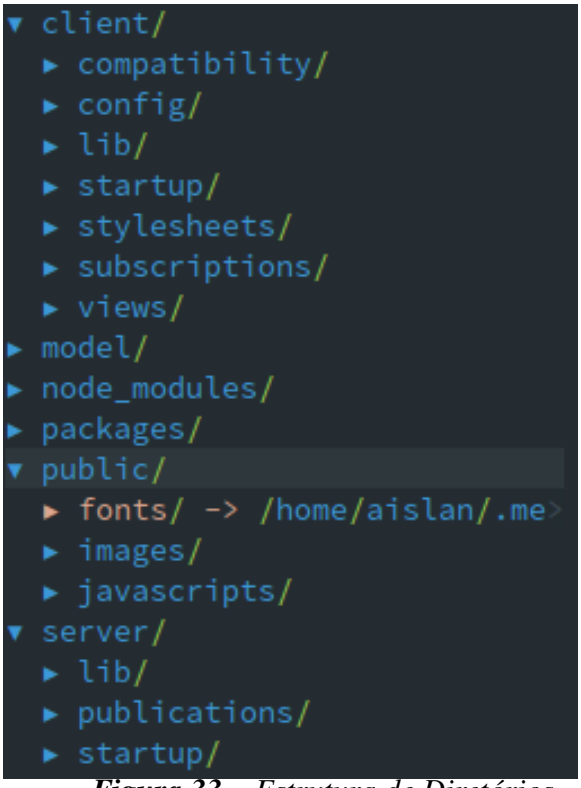

Figura 33 *– Estrutura de Diretórios*

O projeto está dividido em quatro diretórios principais: *client, model, public, server*. A *framework MeteorJS* em seu sítio oficial revela algumas vantagens de organizar o código em diretórios específicos.

No diretório *client* reside todo o código que é executado somente no lado cliente (navegador *Web*). No diretório *server*, por outro lado, fica todo o código que é executado somente no lado servidor. Com exceção destes dois diretórios (e o diretório *tests* que não está presente neste projeto), todos os outros diretórios conterão arquivos que poderão ser acessados tanto no lado cliente quanto no servidor. No diretório *public* vão os arquivos que são carregados estaticamente, como fontes e imagens. E por fim, no diretório *model* reside código que instancia as coleções no *MongoDB* e código específico para cada modelo (entidade).

Dentro do diretório *client* temos os subdiretórios: *compatibility, config, lib, startup, stylesheets, subscriptions, views*. É importante frisar aqui que a estrutura demonstrada acima não é obrigatória, mas ao organizar arquivos em diretórios específicos, como estes por exemplo, podemos usufruir da ordem natural que a *framework MeteorJS* carrega os arquivos. Abaixo é especificado cada importância desta estrutura:

- *compatibility*: neste diretório reside bibliotecas *Javascript* que somente funcionarão quando colocados aqui. Por convenção do *MeteorJS*, estes arquivos serão executados antes de outros arquivos *Javascript* do lado cliente;

- *config*: diretório que guarda código relacionado a algum tipo de configuração no lado cliente. Está presente nesta estrutura somente por questões de organização de código;
- *lib*: código residente neste diretório será movido pelo *MeteorJS* antes de qualquer outro durante o carregamento de código no lado cliente, mantendo sua ordem alfabética dos nomes dos arquivos;
- *startup*: diretório para guardar código relacionado a inicializações de alguma parte específica no lado cliente. Está presente nesta estrutura somente por questões de organização de código;
- *stylesheets*: diretório em que reside os arquivos de folha de estilo. Arquivos *CSS* ou pré-processadores como *LESS, SASS*, entre outros. Vale ressaltar que organizar folhas de estilo dentro de um diretório como este não é obrigatório;
- *subscriptions*: neste diretório reside qualquer código que faça *subscription* (incrição, em português) para alguma *publication* (publicação, em português) de forma global;
- *views*: neste diretório reside todo o código relacionado aos *templates*, seja em *HTML* ou *Javascript*. Está presente nesta estrutura somente por questões de organização de código.

Dentro do diretório *server* temos os subdiretórios: *lib, publications, startup*. Abaixo é especificado a importância de cada um deles:

- *lib*: código residente neste diretório será movido pelo *MeteorJS* antes de qualquer outro durante o carregamento de código no lado servidor, mantendo sua ordem alfabética dos nomes dos arquivos;
- *publications*: dentro deste diretório reside código que cria publicações de dados que serão inscritos pelo lado cliente;
- *startup*: diretório para guardar código relacionado a inicializações de alguma parte específica no lado servidor.

O *MeteorJS* também mantém uma ordem de carregamento implícita para qualquer projeto (METEOR.COM, 2014), conforme observado abaixo:

- Arquivos em subdiretórios são carregados antes dos arquivos nos diretórios pai, de modo que arquivos no mais profundo subdiretório são carregados primeiro, e arquivos no diretório raiz são carregas por último;

- Dentro de um diretório, os arquivos são carregados em uma ordem alfabética pelo nome do arquivo;
- Após a ordenação alfabética, todos os arquivos sob diretórios com o nome *lib* são movidos antes de quaisquer outros, porém preservando suas ordens;
- Todos os arquivos que correspondem para *main.** são movidos após quaisquer outros, preservando suas ordens.

5.3.3 Os *Templates HTML* da aplicação

Ao todo no projeto são 26 *templates HTML*. Dentro deste conjunto, vários são para rotinas como autenticação e autorização, cadastro de usuário, cadastro de perguntas e respostas, e em sua maioria, *templates* para *design* da aplicação, distribuídos para a parte da barra de menu superior e para a barra lateral. A Figura 34 exibe os *templates* para *design* da aplicação:

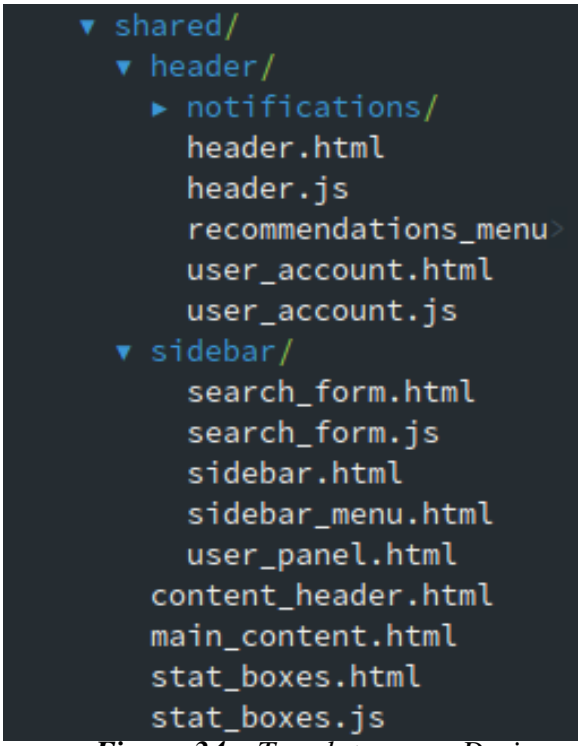

Figura 34 *– Templates para Design*

Para cada template *HTML*, há um arquivo *Javascript* de mesmo *template* nome, onde está incluso funções *helpers* e funções para captura de eventos que ocorram nos seus respectivos *templates*. O *template* inicial é o *layout* que está contido no arquivo *home.html* e é exibido logo

abaixo:

```
1  <head>
2      <meta charset="UTF-8">
3      <title>Ask Us</title>
4      <meta content='width=device-width, initial-scale=1, maximum-scale=1, user-scalable=yes' name='viewport'>
5      <!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements and media queries -->
6      <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
7      <!--[if lt IE 9]>
8        <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
9        <script src="https://oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js"></script>
10     <![endif]-->
11   </head>
12 <body>
13 </body>
14
15 <template name="layout">
16   {{> header }}
17   <div class="wrapper row-offcanvas row-offcanvas-left">
18
19     <!-- Left side column. contains the logo and sidebar -->
20     <aside class="left-side sidebar-offcanvas">
21       {{> sidebar }}
22     </aside>
23
24     <!-- Right side column. Contains the navbar and content of the page -->
25     <aside class="right-side">
26       {{> yield }}
27     </aside><!-- /.right-side -->
28   </div><!-- ./wrapper -->
29 </template>
```

Figura 35 *– Template Layout*

Na Figura 35 codificamos o *template* global da aplicação. É este *template* que envolverá todos os outros *templates* que serão renderizados de acordo com as ações do usuário. Na linha 16 temos uma inclusão de *template*, na qual solicitamos que naquele ponto seja renderizado um outro *template* chamado *header*. Mais abaixo, na linha 21, renderizamos o *template sidebar* e em seguida, na linha 26, um *include* especial chamado *yield*, cujo renderizará dinamicamente os *templates* que responderão às ações do usuário na aplicação.

O *template header*, por sua vez, está localizado dentro do diretório *header*, o qual contém arquivos com outros *templates* para a área de notificações, menu de usuário e barra

lateral. Abaixo a figura exibe o código completo deste *template*:

```
1  <template name="header">
2    <header class="header">
3      <a href="index.html" class="logo">
4        <!-- Add the class icon to your logo image or logo icon to add the margining -->
5        Ask Us
6      </a>
7      <!-- Header Navbar: style can be found in header.less -->
8      <nav class="navbar navbar-static-top" role="navigation">
9        <!-- Sidebar toggle button-->
10       <a href="#" class="navbar-btn sidebar-toggle" data-toggle="offcanvas" role="button">
11         <span class="sr-only">Toggle navigation</span>
12         <span class="icon-bar"></span>
13         <span class="icon-bar"></span>
14         <span class="icon-bar"></span>
15       </a>
16       <div class="navbar-left">
17         <ul class="nav navbar-nav">
18           <li class="new_question">
19           <a class="btn btn-primary btn-block btn-social" href="{{pathFor 'new_question'}}">
20               <i class="fa fa-question"></i>_ Fazer Pergunta
21             </a>
22           </li>
23         </ul>
24       </div>
25       <div class="navbar-right">
26         <ul class="nav navbar-nav">
27           {{> notifications_menu type='reply'}}
28           {{> notifications_menu type='follow_question'}}
29           {{> notifications_menu type='approbation'}}
30           {{> user_account }}
31         </ul>
32       </div>
33     </nav>
34   </header>
35 </template>
```

Figura 36 *– Template header*

Este *template* não contém muita complexidade. Além da marcação *HTML* para uma barra superior no topo, a única coisa especial aqui são os *includes* que ocorrem da linha 27 até a 30 na imagem. Nestes *includes*, temos a renderização de dois *templates* diferentes: o *template notifications_menu* e o *user_account*. Na inclusão do *template notifications_menu*, ele é referenciado três vezes, porém com um parâmetro passado com valor diferente. O parâmetro *type* serve como uma *flag* para identificar qual tipo de *template* renderizar dentro de *notifications_menu*.

No *template sidebar*, incluso pelo *template layout*, é bastante simples e faz somente a renderização de outros três *templates*: *user_panel, search_form, sidebar_menu*. A figura abaixo

exibe o *sidebar*:

```
1  <template name="sidebar">
2
3    <!-- sidebar: style can be found in sidebar.less -->
4    <section class="sidebar">
5      {{> user_panel }}
6      {{> search_form }}
7      {{> sidebar_menu }}
8    </section>
9
10 </template>
```

Figura 37 *– Template sidebar*

Dentre os *includes* no *template* exibido pela Figura 37, é útil mostrar o *template sidebar_menu*:

```
1  <template name="sidebar_menu">
2
3    <!-- sidebar menu: : style can be found in sidebar.less -->
4    <ul class="sidebar-menu">
5
6      <li class="active home">
7        <a href="{{pathFor 'home'}}">
8          <i class="fa fa-dashboard"></i> <span>Início</span>
9        </a>
10     </li>
11
12     <li class="my-questions">
13       <a href="{{pathFor 'questions_list'}}">
14         <i class="fa fa-question"></i> <span>Minhas Perguntas</span>
15         <small class="badge pull-right bg-blue">{{currentUser.questions.count}}</small>
16       </a>
17     </li>
18
19     <li class="my-replies">
20       <a href="{{pathFor 'replies_list'}}">
21         <i class="fa fa-reply-all"></i> <span>Minhas respostas</span>
22         <small class="badge pull-right bg-green">{{currentUser.replies.count}}</small>
23       </a>
24     </li>
25
26     <!--<li class="my-messages">-->
27       <!--<a href="#">-->
28         <!--<i class="fa fa-envelope"></i> <span>Mensagens</span>-->
29         <!--<small class="badge pull-right bg-yellow">12</small>-->
30       <!--</a>-->
31     <!--</li>-->
32
33   </ul> <!-- /.sidebar-menu -->
34 </template>
```

Figura 38 *– Template sidebar menu*

No *template* da Figura 38 temos um menu com uma lista de ações. É relevante perceber que aqui entra em ação um dos recursos da *framework MeteorJS*, que é a reatividade. Nas linhas 15 e 22 temos uma renderização de atributos através do *template engine Spacebars*. O atributo renderizado aqui é o *count*, variável que está contida no objeto *questions* e *replies* de uma instância da coleção *Users* no banco de dados. Toda vez que o usuário cadastrar uma nova questão ou resposta, ou então um outro usuário cadastrar uma resposta para uma das

perguntas daquele usuário, o *MeteorJS* enviará esta atualização em tempo real para o cliente e a interface atualizará, reativamente. Nas linhas 13 e 20 temos uma chamada para a função *pathFor* passando uma *string* como parâmetro. Esta *string* é o nome da rota e a função *pathFor* trata de montar a *URL* para este nome.

De volta ao template *layout*, no *include* especial *yield*, o *iron-router* (gerenciador de rotas da aplicação) observará qual o caminho presente na *URL* do navegador e renderizará o *template* associado a este caminho. As rotas estão definidas dentro do arquivo *routes.js*, que está dentro do diretório *client/lib/*. Por padrão, a rota inicial está associada ao endereço raiz da aplicação (neste caso representado pelo caractere /). A Figura 39 a seguir demonstra a configuração da rota inicial:

```
65  this.route('home', {
66    path: '/',
67    waitOn: function () {
68      return [
69        Meteor.subscribe('user_questions', Meteor.userId()),
70        Meteor.subscribe('all_follows_question_from_others', Questions.questionIds()),
71        Meteor.subscribe('replies_in_questions', Questions.questionIds(), Meteor.userId()),
72        Meteor.subscribe('user_replies', Meteor.userId())
73      ];
74    }
75  });
```

Figura 39 *– Rota home*

Na linha 65, através do primeiro parâmetro (’*home*’) da função *route*, é definido o nome do *template* que queremos renderizar, e através do primeiro atributo (*path*) do objeto passado como segundo parâmetro da função (linha 66), definimos o caminho *URL* que queremos associar ao *template*. É com esta configuração que a aplicação renderiza, dinamicamente, o primeiro *template* através do include especial *yield*, que neste caso é o *template home*. A Figura 40 demonstra a tela inicial da aplicação com as renderizações de *templates* explicados acima:

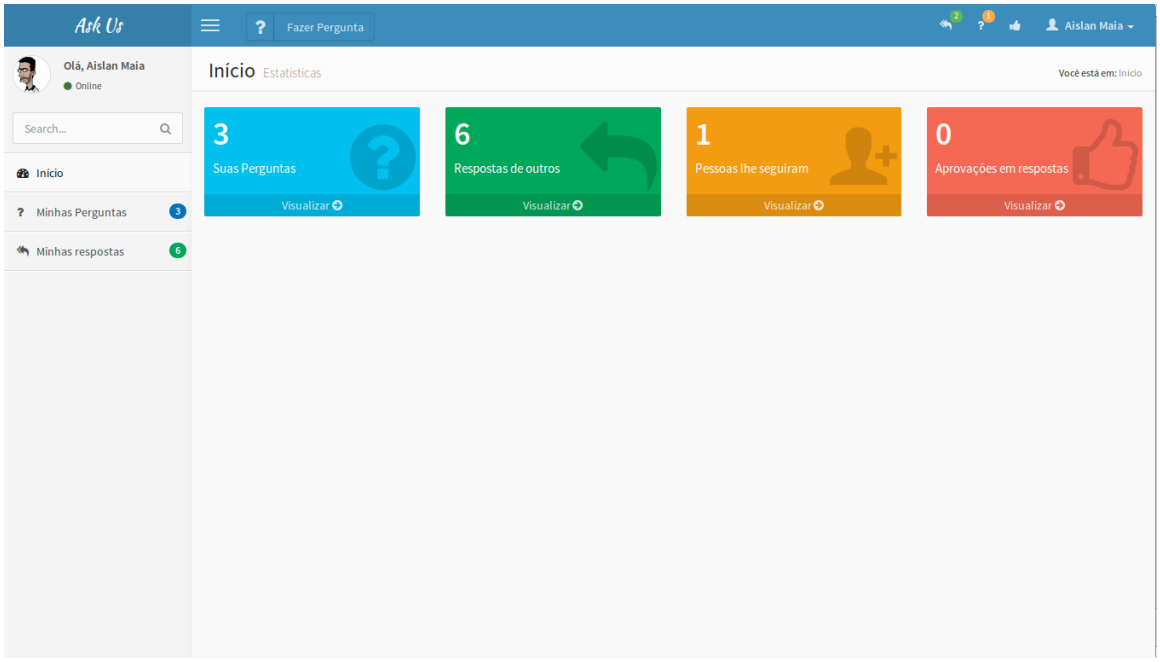

Figura 40 *– Tela Inicial*

Os outros *templates* da aplicação são exibidos nas Figuras 41, 42 e 43 a seguir:

```
▼ views/
  ▼ authentication/
      login.html
      login.js
  ▼ home/
      home.html
      home.js
  ▼ questions/
      new_question.html
      new_question.js
      question_item.html
      question_item.js
      question_page.html
      question_page.js
      questions_list.html
      questions_list.js
      questions_search.html
      questions_search.js
  ▼ replies/
      replies_list.html
      replies_list.js
      reply.html
      reply.js
      reply_submit.html
      reply_submit.js
```

Figura 41 *– Outros templates*

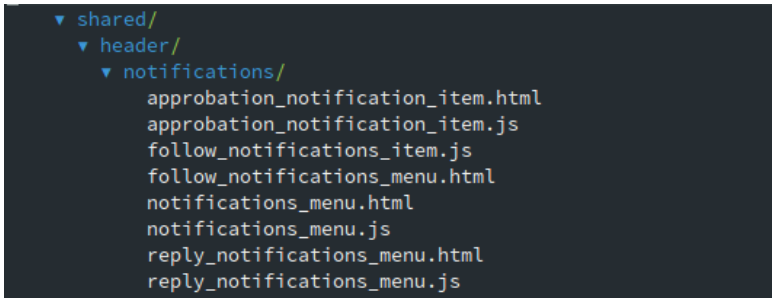

Figura 42 *– Outros templates - notifications*

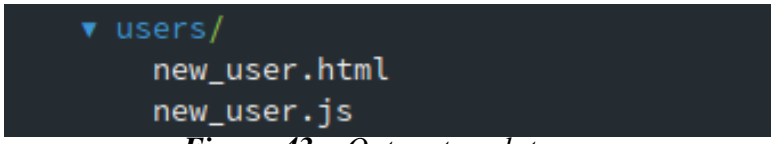

Figura 43 *– Outros templates - users*

5.3.4 Publicações e Inscrições

Conforme Coleman e Greif (2013), as publicações e inscrições (*subscriptions*, em inglês) são um dos mais fundamentais conceitos no *MeteorJS*. Embora fundamentais, eles alertam para como este conceito pode ser um dos mais difíceis de se entender:

> Nós estamos acostumados a definir e a pensar sobre nossas próprias APIs por passar dados entre cliente e servidor, através de protocolos explicitamente designados e utilizados, porém em MeteorJS os dados são sincronizados para nós. Nós não necessitamos pensar diretamente sobre como os dados fazem isso do cliente para o servidor. Ao invés, nós utilizamos publicações para controlar quais dados são sincronizados. (COLEMAN; GREIF, 2013, tradução nossa)

Coleman e Greif (2013) esclarece-nos que uma publicação pode ser conectada utilizando uma inscrição, e que uma publicação é um método de transferir dados a partir de uma coleção do lado servidor (fonte) para uma coleção do lado cliente (alvo). Esta inscrição seria como um funil conectando-se ao armazém de dados original, este que seria a coleção fonte, a qual se comunica com a base de dados no *MongoDB*, e o *cache* do lado cliente, o qual seria a coleção alvo, que representa uma cópia ou subconjunto daqueles dados. A Figura 44 descreve

este cenário:

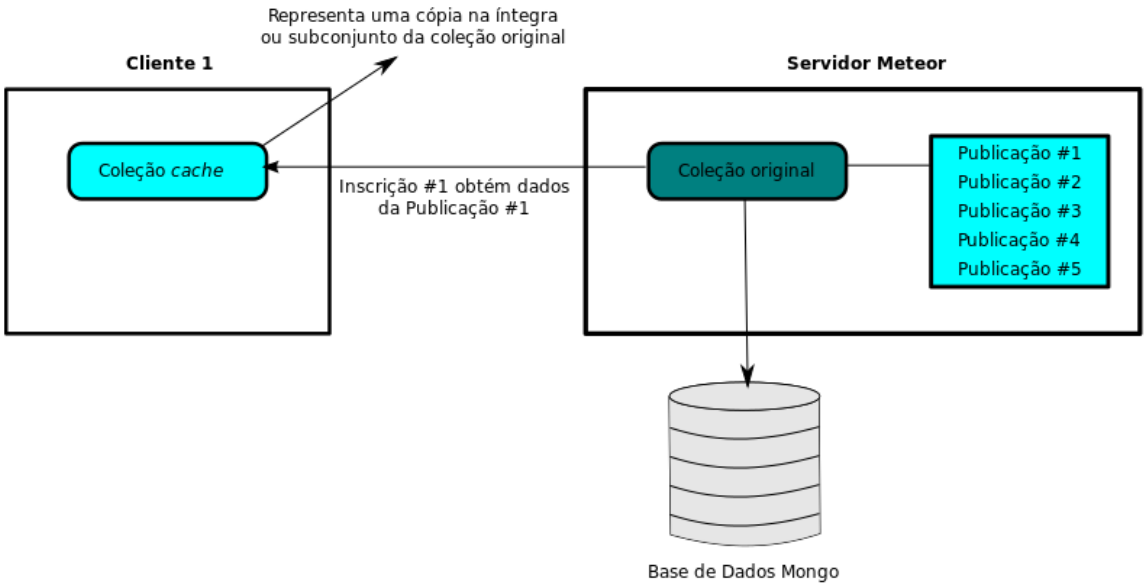

Figura 44 *– PubSub - Publicação e Inscrição*

Pereira (2014b) diz que: "O *PubSub* (*Publications and Subscriptions*), para quem não conhece, é um *pattern* cujo conceito é realizar mensageria através de dois personagens: um *publisher* (o publicador) e um *subscriber* (o assinante)."(PEREIRA, 2014b, p. 91).

No diretório *server/publications* reside código com todas as publicações do sistema, separados por arquivos classificados pelo nome dos modelos. A Figura 45 exibe esta estrutura:

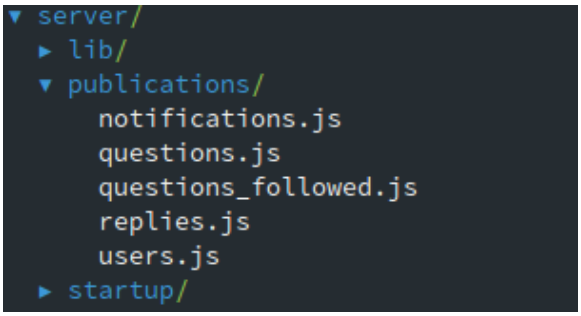

Figura 45 *– Templates - Publications*

No arquivo *notifications.js* as publicações que enviam um conjunto de dados da coleção *Notifications* de acordo com o método *publish* do *MeteorJS*. A Figura 46 a seguir exibe todo o código deste arquivo:

```
1 Meteor.publish('notifications', function (user_id) {
2   return Notifications.find({user_id: user_id, read: false});
3 });
```

Figura 46 *– Pubicações de Notificações*

Para o método *publish* do *MeteorJS* é passado dois parâmetros. O primeiro é o nome da publicação, que neste caso é notifications, e a outra uma função de *callback* que recebe uma variável como parâmetro, neste caso um indentificador de usuário. Dentro desta função, colocamos código que retorna um cursor sobre um conjunto de dados do *MongoDB*. No código acima, o cursor é retornado pela função *find* do *MongoDB*, que por sua vez retornará todas as notificações que correspondam ao filtro aplicado, que neste caso é o *id* do usuário e as notificações com o atributo *read* com o valor *false*, como pode ser observado na Figura 46.

No arquivo *questions.js* temos o código para as publicações de questões, como exibido na Figura 47:

```
1  Meteor.publish('question', function (id) {
2    check(id, String);
3
4    return Questions.find(id);
5  });
6
7  Meteor.publish('user_questions', function (user_id) {
8    return Questions.find({
9      "author._id": user_id
10   });
11 });
12
13 Meteor.publish('searched_questions', function (query) {
14   return Questions.find({
15     $or: [
16       { title: { $regex: query, $options: 'i' } },
17       { text: { $regex: query, $options: 'i' } }
18     ]
19   });
20 });
21
22 Meteor.publish('questions_followed', function (question_ids) {
23   return Questions.find({_id: { $in: question_ids }});
24 });
```

Figura 47 *– Publicações de Questões*

Neste código temos quatro publicações: *question, user_questions, searched_questions* e *questions_followed*. O nome das publicações são bem sugestíveis para seus casos de uso. Na publicação *question*, na linha 2, é verificada se a variável *id* é uma *String* e logo em seguida é retornado um cursor com a questão que corresponde àquele *id*. As demais publicações seguem o mesmo padrão, modificando somente o filtro de retorno de cada publicação.

No arquivo *questions_followed.js* estão separadas as publicações para a coleção *QuestionsFollowed*. O código segue o mesmo padrão que o do arquivo *questions.js*, porém uma publicação dentre as quatro criadas para a coleção *QuestionsFollowed* se difere por uma publicação que contém uma chamada a uma função dentro do modelo *QuestionsFollowed*, presente

no arquivo *models/questions_followed.js*. A Figura 48 exibe o código para estas publicações:

```
1 Meteor.publish('user_questions_followed', function (user_id) {
2   return QuestionsFollowed.find({user_id: user_id});
3 });
4
5 Meteor.publish('user_is_following_question', function (question_id, user_id) {
6   return QuestionsFollowed.isFollowing(question_id, user_id);
7 });
8
9 Meteor.publish('all_followed_from_question', function (question_id) {
10   return QuestionsFollowed.find({question_id: question_id});
11 });
12
13 Meteor.publish('all_follows_question_from_others', function (questions_id) {
14   return QuestionsFollowed.find({question_id: {$in: questions_id}});
15 });
```

Figura 48 *– Publicações de Questões Seguidas*

Na linha 5 temos a publicação *user_is_following_question* que retorna se o usuário está seguindo determinada questão. Diferente das outras, a função que é chamada dentro do *callback* da função *publish* é a *isFollowing*. Em resumo ela é somente uma encapsulamento para a função *find* dentro do modelo *QuestionsFollowed*. A Figura 49 exibe a implementação desta função contida no arquivo *models/questions_followed.js*, da linha 12 até a 19:

```
1 QuestionsFollowed = new Meteor.Collection('questions_followed');
2
3 QuestionsFollowed.questionIds = function (user_id) {
4   var questionIds = this.find({
5     user_id: user_id}).map(function (q) {
6       return q.question_id;
7     });
8
9   return questionIds;
10 };
11
12 QuestionsFollowed.isFollowing = function (question_id, user_id) {
13   return this.find({
14     $and: [
15       { question_id: question_id },
16       { user_id: user_id }
17     ]
18   });
19 };
20
```

Figura 49 *– Função isFollowing*

Todas as outras publicações referentes às coleções *Replies* e *Users* seguem o mesmo padrão de código das exibidas. Estas publicações ficam do lado servidor, e por isso estão contidas sob o diretório *server*.

A contraparte das publicações são as inscrições. Estas permanecem somente do lado cliente. Na estrutura de subdiretórios do diretório *client*, o lugar definido para manter códigos

de inscrição é o diretório *subscriptions*. Dentro deste diretório são definidas inscrições que serão aplicadas por toda a aplicação, ou seja, são globais. Por outro lado, através do módulo *iron-router*, é possível definir rotas para a aplicação com inscrições específicas que funcionam somente nas rotas associadas. Dessa maneira, economiza-se tráfego de rede, pois não será necessário associar canais de inscrição e publicação por toda a aplicação, requisitando dados somente em seções da aplicação em que realmente são necessários.

Dentro do arquivo *client/lib/routes.js*, temos a definição de todas as rotas da aplicação, junto com filtros para cada conjunto de rotas em específico. Para a publicação de notificações descrita anteriormente, temos uma inscrição declarada dentro de uma função de filtro de rotas. Inscrições no *MeteorJS* são declaradas dentro da função *subscribe*, na qual passa-se como primeiro parâmetro o nome da publicação a qual queremos inscrever, e como parâmetros adicionais variáveis ou funções para a função *callback* da publicação pretendida. A Figura 50 exibe a implementação da inscrição de notificações:

```
1  Router.configure({
2    layoutTemplate: 'layout'
3  });
4
5  var BeforeHooks = {
6    isLoggedIn: function (pause) {
7      if (!(Meteor.loggingIn() || Meteor.user())) {
8        this.render('login');
9        pause();
10     }
11   },
12   guestOnly: function (pause) {
13     if(Meteor.user()){
14       var route = Session.get('route_for_back');
15       if(route){
16         return Router.go(route);
17       } else {
18         if(Router.current().route.name === 'login') {
19           return Router.go('home');
20         }
21       }
22     }
23   },
24   notifications: function () {
25     // add notifications here
26     Meteor.subscribe('notifications', Meteor.userId());
27   },
```

Figura 50 *– Inscrições - notificações*

Na linha 26 temos a declaração da inscrição para a publicação *notifications* e o função *MeteorJS.userId()* como segundo parâmetro, o qual retorna o *id* do usuário logado. A Figura 51 a seguir exibe a aplicação da função *notifications* dentro do filtro *onBeforeAction* do *iron-router*:

```
52 Router.onBeforeAction(BeforeHooks.notifications, {except: ['new_user', 'login']});
```

Figura 51 *– Filtro de rotas de notificações*

O código acima aplica a inscrição de notificações para todas as rotas da aplicação, com exceção das rotas *new_user* e *login*. As outras inscrições da aplicação variam de acordo com a rota. Mais abaixo, no mesmo arquivo *routes.js*, estão definidos as rotas com suas respectivas inscrições. Nas linhas 69 à 72 da figura da rota *home* estão declaradas quatro inscrições diferentes. Na linha 69 temos uma inscrição para a publicação *user_questions*. Esta publicação retorna um subconjunto da coleção *Questions* que pertençam ao usuário logado. Na linha seguinte temos uma inscrição para a publicação *all_follows_question_from_others*, a qual retorna todas as seguidas que outros usuários fizeram para as questões do usuário atualmente logado. Na linha 71 temos uma inscrição para a publicação *replies_in_questions*, a qual retorna as respostas para as questões do usuário logado. Por fim, na linha seguinte, temos uma inscrição para a publicação *user_replies*, que retorna as respostas do usuário logado.

Na rota *questions_list*, temos uma inscrição para a publicação *user_questions*, que retorna as questões cadastradas pelo usuário atualmente logado, conforme a Figura 52.

```
81  this.route('questions_list', {
82    path: '/questions/',
83    waitOn: function () {
84      return [
85        Meteor.subscribe('user_questions', Meteor.userId()),
86      ];
87    },
88    data: function () {
89      return {
90        questions: Questions.find({"author._id": Meteor.userId()})
91      };
92    }
93  });
```

Figura 52 *– Rota quetions list*

As Figuras 53, 54, 55, 56 e 57 a seguir exibem as outras rotas e suas respectivas inscrições:

```
95  this.route('questions_list_followed', {
96    path: '/questions/followed/',
97    template: 'questions_list',
98    waitOn: function () {
99      return [
100       Meteor.subscribe('user_questions_followed', Meteor.userId()),
101       Meteor.subscribe('questions_followed', QuestionsFollowed.questionIds(Meteor.userId()))
102     ];
103   },
104   data: function () {
105     return {
106       user_questions_followed: QuestionsFollowed.find({user_id: Meteor.userId()}),
107       questions: Questions.find()
108     };
109   }
110 });
```

Figura 53 *– Rota quetions followed*

```
112  this.route('questions_search', {
113    path: '/search',
114    waitOn: function () {
115      return [
116        Meteor.subscribe('searched_questions', Session.get('query')),
117      ];
118    },
119    data: function () {
120      return {
121        questions: Questions.find({}, {sort:{_id: -1}}),
122      };
123    }
124  });
```

Figura 54 *– Rota quetions search*

```
126  this.route('replies_list', {
127    path: '/replies',
128    waitOn: function () {
129      return [
130        Meteor.subscribe('user_replies', Meteor.userId())
131      ];
132    },
133    data: function () {
134      return {
135        replies: Replies.find({}, {sort: {submitted: -1}})
136      };
137    }
138  });
```

Figura 55 *– Rota replies list*

```
140  this.route('replies_list_from_others', {
141    path: '/replies/others',
142    template: 'replies_list',
143    waitOn: function () {
144      var user_id = Meteor.userId();
145      return [
146        Meteor.subscribe('user_questions', user_id),
147        Meteor.subscribe('replies_in_questions', Questions.questionIds(), user_id)
148      ];
149    },
150    data: function () {
151      return {
152        replies: Replies.find({}, {sort: { submitted: -1 }})
153      };
154    }
155  });
```

Figura 56 *– Rota replies list from others.eps*

```
162  this.route('question_page', {
163    path: '/questions/:_id',
164    waitOn: function () {
165      var question_id = this.params._id;
166      return [
167        Meteor.subscribe('question', question_id),
168        Meteor.subscribe('user_is_following_question', question_id, Meteor.userId()),
169        Meteor.subscribe('all_followed_from_question', question_id),
170        Meteor.subscribe('question_replies', question_id)
171      ];
172    },
173    data: function () {
174      return {
175        question: Questions.findOne(this.params._id),
176        replies: Replies.find(),
177        numberFollowed: QuestionsFollowed.find().count()
178      };
179    }
180  });
```

Figura 57 *– Rota question page*

5.3.5 Modelos de Coleções (*models*)

As entidades descritas no diagrama de entidades da Figura 32 são representadas dentro da aplicação como modelos, residentes no diretório *model*. Estes modelos, por sua vez, representam na realidade as coleções no *MeteorJS*, que nada mais são do que os documentos dentro da base de dados *MongoDB*. Na aplicação temos os modelos: *Notifications, Questions, QuestionsFollowed, Replies* e *Users*.

Dentro do arquivo *model/notifications.js* reside o código para o modelo *Notifications*. Dentro deste modelo há três funções. A primeira função é a *validateUniqueFollowQuestion*, o qual pode ser executada tanto do lado cliente, quanto do lado servidor, e faz um teste para verificar se uma notificação para uma questão seguida em específico já foi emitida. As outras duas funções são atributos de um objeto parâmetro da função *method* do *MeteorJS*. As funções

passadas para a função *method* são executadas do lado servidor. A Figura 58 a seguir exibe estas duas funções:

```
13 Meteor.methods({
14   createNotification: function (attributes, submitted_at) {
15     var notification = _.extend(_.pick(attributes, 'type', 'user_id', 'content', 'submitted'), {
16       read: false
17     });
18
19     var notificiationId;
20
21     if (Notifications.validateUniqueFollowQuestion(notification)) {
22       notificiationId = Notifications.insert(notification);
23     }
24
25     return notificiationId;
26   },
27   setReadNotification: function (id) {
28     Notifications.update({_id: id}, {$set: {read: true}});
29   }
30 });
```

Figura 58 *– Modelo Notifications*

A função *createNotification*, declarada na linha 14, cria um objeto *notification* e o passa para o método *insert* da coleção *Notifications*. Enquanto a função *setReadNotification*, na linha 27, executa um update na base de dados dentro do documento *Notifications* no *MongoDB*.

No modelo *QuestionsFollowed*, dentro do arquivo *model/questions_followed.js*, temos 5 funções: *questionIds, isFollowing, followNotification, question_follow, question_unfollow*. Dentre estas funções, as mais interessantes para ressaltar aqui são as funções *followNotification* e a *question_follow*. A primeira notifica, em tempo real, o usuário autor da questão sendo seguida de que alguém acabou de segui-la. A segunda cria um novo registro na coleção *Questi-*

onsFollowed

```
30 QuestionsFollowed.followNotification = function (follow) {
31   var attributes = {
32     type: "follow_question",
33     content: {
34       _id: follow._id,
35       author_id: follow.user_id,
36       author_avatar_url: Meteor.user().profile.avatar_url,
37       author_name: Meteor.user().profile.name,
38       question_id: follow.question._id,
39       question_title: follow.question.title
40     },
41     submitted: follow.submitted,
42     user_id: follow.question.author._id
43   };
44   Meteor.call('createNotification', attributes);
45 };
46
47 Meteor.methods({
48   question_follow: function (attributes) {
49     var follow = _.extend(_.pick(attributes, 'user_id', 'question_id'), {
50       submitted: new Date().getTime()
51     });
52
53     var follow_id = QuestionsFollowed.insert(follow);
54
55     if (follow_id) {
56       attributes._id = follow_id;
57       attributes.submitted = follow.submitted;
58       QuestionsFollowed.followNotification(attributes);
59     }
60
61     return follow_id;
62   },
```

Figura 59 *– Modelo QuestionsFollowed*

No código acima da Figura 59, na linha 30, é implementada a função *followNotification*. Interessante notar que esta função segue um padrão existente também no modelo *Replies*. Este padrão declara que o objeto instância do modelo *Notifications* terá os atributos *type, content, submitted* e *user_id*, e no último trecho de código da função, uma chamada para o método *call* do *MeteorJS*, que, por sua vez, invoca o método *createNotification* do modelo *Notifications* passando um objeto com os atributos anteriormente citados.

Dentro da função *question_follow* existe um outro padrão também existente no modelo *Replies*, mais especificadamente nas suas funções: *insertReply* e *approbation*. Este padrão consiste em realizar a rotina primária da função, (que no caso da função *question_follow* é criar um novo registro em *QuestionsFollowed*, como explicado anteriormente) e em seguida invocar uma função que tratará de preparar uma nova notificação. As figuras abaixo exibem as funções

insertReply e approbation do modelo *Replies*:

```
37  insertReply: function (attributes) {
38    var user = Meteor.user();
39    var reply = _.extend(_.pick(attributes, 'question_id', 'question_title', 'text'), {
40      submitted: new Date().getTime(),
41      updated_at: undefined,
42      author: {
43        _id: user._id,
44        name: user.profile.name,
45        avatar_url: user.profile.avatar_url
46      },
47      approvers: [],
48      approbations: 0
49    });
50
51    replyId = Replies.insert(reply);
52
53    if (replyId) {
54      reply._id = replyId;
55      Replies.replyNotification(reply, attributes.receiver_id);
56    }
57
58    return replyId;
59  },
```

Figura 60 *– Modelo Replies - função insertReply*

```
74  approbation: function (attributes) {
75    var user = Meteor.user();
76
77    var replyId = Replies.update({
78      _id: attributes._id,
79      approvers: { $ne: user._id },
80      "author._id": { $ne: user._id }
81    }, {
82      $addToSet: { approvers: user._id },
83      $inc: { approbations: 1 }
84    });
85    if (replyId) {
86      Replies.approbationNotification(attributes, user);
87    }
88    return replyId;
89  }
```

Figura 61 *– Modelo Replies - função approbation*

No modelo *Users*, temos funções que incrementam e decrementam, em tempo real, os contadores de quantidade das questões e respostas do usuário. As funções são bem semelhantes entre si, e em comum, invocam o método update para atualizar os respectivos atributos contadores na base de dados. A figura abaixo exibe as funções do modelo *Users*:

```
Meteor.methods({
  increment_count_questions: function (user_id) {
    Meteor.users.update(
      { _id: user_id },
      { $inc: { "questions.count": 1 }
    });
  },
  decrement_count_questions: function (user_id) {
    Meteor.users.update(
      { _id: user_id },
      { $inc: { "questions.count": -1 } }
    );
  },
  increment_count_replies: function (users_id) {
    console.log(users_id);
    for (var i = 0, l = users_id.length; i < l; i ++) {
      Meteor.users.update(
        { _id: users_id[i] },
        { $inc: { "replies.count": 1 }
      });
    }
  },
  decrement_count_replies: function (users_id) {
    console.log(users_id);
    for (var i = 0, l = users_id.length; i < l; i ++) {
      Meteor.users.update(
        { _id: users_id[i] },
        { $inc: { "replies.count": -1 }
      });
  }
  }
```

Figura 62 *– Modelo Users - funções*

5.3.6 *Helpers* e Eventos

Para todo e qualquer *template* no *MeteorJS* é possível associarmos funções *helpers* (ajudantes, em português) e funções tratadoras de eventos que podem acontecer no *template*, como por exemplo cliques do mouse, pressionamento de teclas, submissão de formulários, entre outros. As funções *helpers* são funções que podem ser executadas dentro do template graças ao *template engine Spacebars*.

Como tudo por padrão no *MeteorJS* é reativo, qualquer parte da página será atualizada automaticamente conforme alguma mudança de estado ou de dados ocorra (o princípio *Full Stack Reactivity*).

O princípio *Full Stack Reactivity* do MeteorJS está muito associado ao padrão *PubSub* que ele adota. Pereira (2014b) explica sobre o *PubSub*:

> Esse conceito é utilizado em vários tipos de aplicações, como por exemplo: *newsletters, chats* e leitores de *RSS*. Ele funciona da seguinte maneira: o *publisher* é o responsável por emitir dados para todos os seus *subscribers*, e também um *subscriber* pode enviar mensagens para o *publisher*, fazendo com que o *publisher* trabalhe de forma mais específica com esse *subscriber*. Por exemplo, em um sistema *newsletter*, você, usuário, é o *subscriber* e, para receber atualizações desse serviço (neste caso receber atualizações de um *publisher*), você precisa informar seu *e-mail* para que ele saiba quem você é — ou seja, enviar alguns dados para o *publisher*, para que ele passe a enviar atualizações para seu *e-mail* . (PEREIRA, 2014b, p. 91)

Nesta aplicação temos vários *helpers* que se beneficiam do *PubSub* para retornar dados filtrados sobre as coleções. Como explicado anteriormente, em cada rota temos algumas inscrições específicas. Estas inscrições obterão um subconjunto dados que estarão disponíveis para os *templates HTML*, os quais o *MeteorJS* aos *helpers* e eventos.

Desta maneira, para a página inicial da aplicação (a rota *home*), temos implementado o *template stat_boxes*, que exibe os painéis com alguns contadores, conforme na Figura 40. Estes painéis têm seus dados atualizados em tempo real, graças ao padrão a biblioteca *PubSub*, que envia publicações de dados, e o protocolo *DDP*, que faz com que esta transmissão ocorra em tempo real. Este padrão reativo de envio de dados ocorrerá em todos os *templates* que estejam associados as rotas que possuam inscrições para publicações. A Figura 63 exibe os *helpers* para o *template stat_boxes*:

```
Template.stat_boxes.helpers({
  questionsCount: function () {
    var questions_count = Questions.find().count();
    var questions_follow_count = QuestionsFollowed.find().count();

    var total = questions_count + questions_follow_count;
    return total;
  },
  numberFollowsFromOthers: function () {
    return QuestionsFollowed.find({user_id: { $ne: Meteor.userId() }}).count();
  },
  numberReplies: function () {
    return Replies.find({"author._id": { $ne: Meteor.userId() }}).count();
  },
  numberApprobations: function () {
    var approbations = Replies.find({"author._id": Meteor.userId()}).map(function (r) {
      return r.approbations;
    });
    return approbations.sum();
  }
});
```

Figura 63 *– Template stat boxes - helpers*

No código acima, todas as funções *helpers* estão associadas e são invocadas dentro do respectivo *template HTML*. Cada coleção sendo utilizada nestes *helpers* têm seus dados vindos através das publicações inscritas a partir da rota *home*. A figura (número da figura referente a rota *home*) exibe estas inscrições.

Toda vez que quaisquer destas coleções (que estejam vinculadas a uma inscrição e publicação) sofrer alguma alteração no lado servidor, a biblioteca *PubSub* enviará o dado alterado,

em tempo real, através de *JSON* para os clientes inscritos, reexecutando os *helpers* que iteram sobre dados reativamente. As Figuras 64 e 65 exibem os *helpers* sendo utilizados dentro *template stat_boxes*:

```
<template name="stat_boxes">
  <!-- Small boxes (Stat box) -->
  <div class="row">
    <div class="col-lg-3 col-xs-6">
      <!-- small box -->
      <div class="small-box bg-aqua">
        <div class="inner">
          <h3>
            {{questionsCount}}
          </h3>
          <p>
            Suas Perguntas
          </p>
        </div>
        <div class="icon">
          <i class="fa fa-question-circle"></i>
        </div>
        <a href="#" class="small-box-footer">
          Visualizar <i class="fa fa-arrow-circle-right"></i>
        </a>
      </div>
    </div><!-- ./col -->
    <div class="col-lg-3 col-xs-6">
      <!-- small box -->
      <div class="small-box bg-green">
        <div class="inner">
          <h3>
            {{numberReplies}}<sup style="font-size: 20px"></sup>
          </h3>
          <p>
            Respostas de outros
          </p>
        </div>
        <div class="icon">
          <i class="fa fa-reply"></i>
```

Figura 64 *– Template stat boxes - utilizando os helpers*

```
36          </div>
37          <a href="#" class="small-box-footer">
38            Visualizar <i class="fa fa-arrow-circle-right"></i>
39          </a>
40        </div>
41      </div><!-- ./col -->
42      <div class="col-lg-3 col-xs-6">
43        <!-- small box -->
44        <div class="small-box bg-yellow">
45          <div class="inner">
46            <h3>
47              {{numberFollowsFromOthers}}
48            </h3>
49            <p>
50            Pessoas lhe seguiram
51            </p>
52          </div>
53          <div class="icon">
54            <i class="ion ion-person-add"></i>
55          </div>
56          <a href="#" class="small-box-footer">
57            Visualizar <i class="fa fa-arrow-circle-right"></i>
58          </a>
```

Figura 65 *– Template stat boxes - utilizando os helpers*

Todos os outros *templates* da aplicação seguem o mesmo padrão reativo dos *helpers* do *template stat_boxes*.

6 Um Teste Comparativo

Neste capítulo será detalhado um teste comparativo realizado entre a aplicação de *chat* em tempo real, desenvolvida com *Node.js* e uma outra aplicação de chat desenvolvida com a técnica *AJAX polling* por um tutorial do sítio *CSS Tricks* (*http://css-tricks.com/jquery-php-chat/*). O teste tem o intuito de verificar a carga das aplicações através da quantidade de requisições que ocorre nas duas aplicações, bem como as suas consequências, refletindo sobre a luz dos conceitos aprendidos durante a pesquisa deste trabalho.

O teste realizado neste trabalho não possui o intuito de ser um teste ou afirmação em definitivo a cerca dos resultados levantados, mas sim de por em análise e evidência as diferenças pesquisadas e estudadas entre *AJAX* e *WebSocket* durante este trabalho.

As duas aplicações foram testadas sobre a mesma plataforma de Sistema Operacional. O ambiente de programação consiste de um servidor *Apache* vesão 2.2.22, um servidor *Node*, a linguagem *PHP* versão 5.3.10, a plataforma *Node.js* versão 0.10.6 e a linguagem *Javascript*. As ferramentas para teste utilizadas foram as ferramentas de teste de carga como um serviço na nuvem e de monitoramento de recursos: *BlazeMeter* e *New Relic*, respectivamente.

6.1 *BlazeMeter*

A ferramenta *BlazeMeter* extende uma ferramenta de teste de carga de código fonte aberto conhecida como *Apache JMeter*, uma aplicação *Desktop* feita em *Java* e designada para teste de comportamento funcional de carga e mensuração de performance de aplicações *Web*. Em relação ao *Apache JMeter*, *BlazeMeter* oferece testes automatizados sem expor complexidade ao usuário.

Através do *JMeter*, é possível testar carga e performance de muitos diferentes servidores e protocolos:

- *Web - HTTP, HTTPS*;
- *SOAP*;
- *FTP*;
- Bancos de dados via *JDBC*;
- *LDAP*;

- *Middlewares* orientado a mensagens;
- Servidores de *e-mail - SMTP(S), POP3(S)* e *IMAP(S)*;
- *MongoDB*;
- Comandos nativos ou *shell scripts*;
- *TCP*.

6.2 *New Relic*

Assim como o *BlazeMeter*, esta ferramenta é um serviço na nuvem que faz um monitoramento de recursos e gerenciamento de performance mais completo, apresentando muitos gráficos diferenciados.

A ferramenta obtém dados das aplicações através de uma aplicação agente instalada no ambiente de desenvolvimento. Este agente trata de:

- Visualizar onde a aplicação está gastando tempo;
- Identificar requisições lentas;
- Agrupar métricas;
- Exibir quais porções da aplicação estão, por exemplo, sofrendo de performance lenta de bancos de dados;
- Entre outros recursos.

6.3 A Aplicação *chat* com *AJAX polling*

A aplicação consiste de somente três arquivos: *application.js, index.php* e *process.php*. Esta aplicação roda no servidor *Apache* com o *backend* da aplicação desenvolvido através da linguagem *PHP* e o lado cliente através de *Javascript* e *HTML*.

Esta aplicação não utiliza banco de dados, mas somente um arquivo de texto para guardar as mensagens escritas no *chat*. No lado servidor, as funcionalidades consistem em escrever novas mensagens para o arquivo de texto, ler novas mensagens a partir dele e obter o estado

atual do texto. No lado cliente, a funcionalidade principal consiste no *polling*: periodicamente pergunta ao servidor se novas mensagens têm sido escritas.

Os testes de carga foram feitos simulando 50 usuários simultâneos através do *BlazeMeter*. Esta ferramenta processou as informações e as enviou para o *New Relic* (o *New Relic* possui um *plugin* de integração com o *BlazeMeter*). O New Relic, por sua vez, analisou os resultados e construiu relatórios para eles. As Figuras 66 e 67 a seguir demonstram os gráficos da carga de requisições por minuto e do tempo médio de resposta para cada requisição:

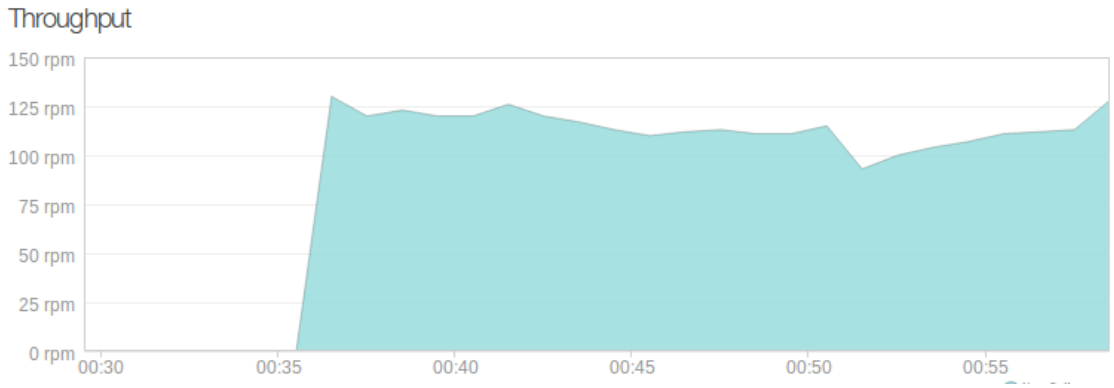

Figura 66 *– Vazão de Rede - Requisições Por Minuto*

Figura 67 *– Tempo Médio de Resposta das Requisições*

No primeiro relatório, temos a vazão (transferência) de rede, ou, em outras palavras, o número de requisições por minuto que o cliente faz ao servidor. Conforme a Figura 66 a quantidade exibida teve picos de 125 requisições por minuto (*rpm*). No segundo relatório, temos o tempo médio de resposta de 0,0085 segundos, com picos de 12,6 milissegundos.

O *New Relic* possui um agendamento de análises semanais, na qual ela mensura e agrupa parâmetros e oferece relatórios com resultados acumulados. As Figuras 68 e 69 exibem dois

relatórios, uma em relação ao cliente e outra ao servidor:

PHP Application

Weekly end user Apdex: **1.00$_{7.0}$* (Excellent)** SLA report

	4/20	4/27	5/04	5/11	5/18	5/25	6/01	6/08	6/15	6/22	6/29	7/06
END USER												
Page views	-	-	-	-	-	-	-	-	-	-	-	4.0
Load time sec	-	-	-	-	-	-	-	-	-	-	-	2.07
Apdex	-	-	-	-	-	-	-	-	-	-	-	1.0
% Satisfied	-	-	-	-	-	-	-	-	-	-	-	100.0
% Tolerating	-	-	-	-	-	-	-	-	-	-	-	0.0
% Frustrated	-	-	-	-	-	-	-	-	-	-	-	0.0

Figura 68 *– Teste Acumulativo Semanal - lado cliente*

APP SERVER												
Requests thousands	-	-	-	-	-	-	-	-	-	-	-	13.3
Error rate	-	-	-	-	-	-	-	-	-	-	-	0
Resp. time ms	-	-	-	-	-	-	-	-	-	-	-	1
Apdex	-	-	-	-	-	-	-	-	-	-	-	1.0
% Satisfied	-	-	-	-	-	-	-	-	-	-	-	100.0
% Tolerating	-	-	-	-	-	-	-	-	-	-	-	0.0
% Frustrated	-	-	-	-	-	-	-	-	-	-	-	0.0

Figura 69 *– Teste Acumulativo Semanal - lado servidor*

6.4 A Aplicação *chat* com *Node.js*

Esta aplicação está disponível no repositório: *https://github.com/aislanmaia/chat_node*. A estrutura de diretórios está de acordo com a Figura 70:

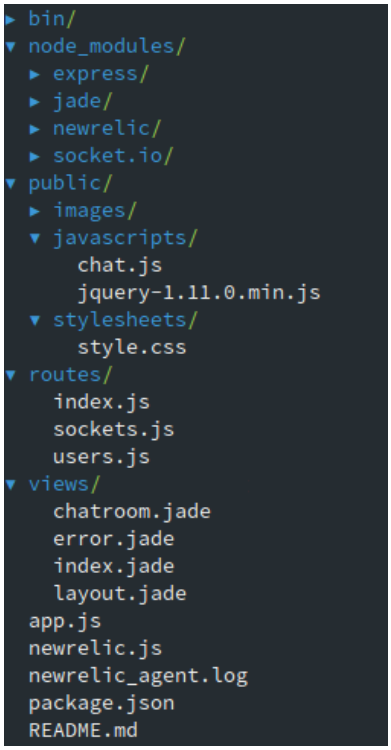

Figura 70 *– Estrutura de Diretórios*

Os arquivos principais da aplicação estão divididos pelos diretórios: *public/javascripts, routes, views*. Os arquivos *Javascript* do lado cliente residem no diretório *public/javascripts*. Dentro deste diretório temos em destaque o arquivo *chat.js*, o qual responderá eventos emitidos transmitidos pelo servidor e emitirá eventos a ele. Os arquivos de interface de usuário residem dentro do diretório *views* e são implementados com o *template engine Jade*. Enquanto isso, no diretório *routes* temos implementada as rotas da aplicação, pelo qual renderizamos *templates* e respondemos a emissões de eventos. Os módulos *Node.js* que esta aplicação utiliza estão sob o diretório *node_modules* e são: a *framework Express*, o *template engine Jade* e a biblioteca *Socket.IO*.

Por fim, temos o arquivo principal na qual é configurada as rotas e dependências da aplicação, inicializado o servidor *Node* utilizando o *Socket.IO* e requerido as bibliotecas *newrelic, express, http* e *path*.

Os testes de carga foram feitos com os mesmos parâmetros que a aplicação anterior, com um diferencial de que fora utilizado uma extensão para o navegador *Google Chrome* chamado *BlazeMeter - The Load Testing Cloud* para gravar a navegação pela aplicação e configurar o

teste automaticamente no *BlazeMeter*. As Figuras 71 e 72 a seguir demonstram os gráficos da carga de requisições por minuto e do tempo médio de resposta para cada requisição:

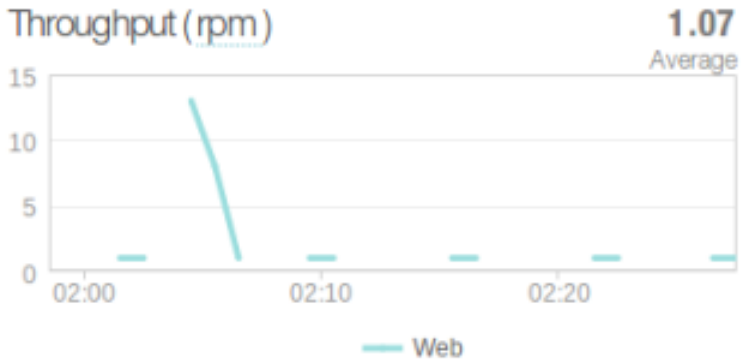

Figura 71 *– Vazão de Rede - Requisições Por Minuto*

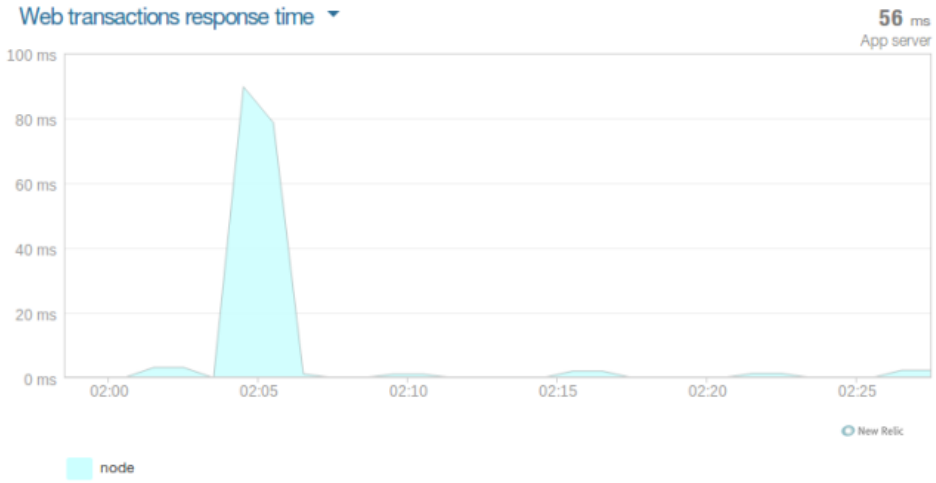

Figura 72 *– Tempo Médio de Resposta das Requisições*

O primeiro relatório demonstra que a vazão de rede fica numa média de 1.07 requisições por minuto (*rpm*), alcançando picos de 13. Já para o segundo relatório tivemos um tempo médio de 56 *ms* (milesegundos) de tempo médio de resposta.

O *New Relic* também forneceu dados agrupados por semana para esta aplicação, mas diferentemente dos relatórios semanais para a primeira aplicação, ela forneceu somente um relatório, diferenciando o primeiro parâmetro que ao invés de ser a quantidade de *page views* (visualizações de página, em português), foi a quantidade de requisições. A Figura 73 exibe este resultado:

Node Chat

Weekly Apdex: **0.94$_{0.1}$* (Excellent)** SLA report

	4/20	4/27	5/04	5/11	5/18	5/25	6/01	6/08	6/15	6/22	6/29	7/06
Requests	-	-	-	-	-	-	-	-	-	-	-	99.0
Error rate	-	-	-	-	-	-	-	-	-	-	-	0
Resp. time ms	-	-	-	-	-	-	-	-	-	-	-	56.7
Apdex	-	-	-	-	-	-	-	-	-	-	-	0.94
% Satisfied	-	-	-	-	-	-	-	-	-	-	-	93.9
% Tolerating	-	-	-	-	-	-	-	-	-	-	-	0.0
% Frustrated	-	-	-	-	-	-	-	-	-	-	-	6.1

Figura 73 *– Teste Acumulativo Semanal - lado servidor*

6.5 Reflexões Sobre Rfesultados

Ambas as aplicações apresentaram vários parâmetros semelhantes. Porém o que ficou em evidência aqui foi a grande diferença entre o número de requisições. Para a aplicação feita com *AJAX*, em seu teste de vazão de rede, podemos perceber pelo gráfico da Figura 66 que o número de requisições esteve em média acima de 100 requisições por minuto, para um número de 50 usuários simultâneos, durante o minuto 35 até o 60. Em comparação com o teste de vazão de rede da aplicação feita com *Node.js*, este número médio foi de 1.07 requisições por minuto, bem abaixo do registrado pela aplicação com *AJAX*.

Esta discrepância se torna maior quando revelados os dados acumulados por análises semanais. No resultado exibido pela Figura 69 referente a dados colhidos do servidor para a aplicação *AJAX*, o número total de requisições fica na casa dos milhares com a quantidade de 13,3 mil requisições. Em comparação com o resultado na Figura 73, o número total de requisições fica em somente 99.

Levando em consideração e analisando o estudo de caso de Lubbers e Greco (s.d.) demonstrado neste trabalho, estes resultados comportam-se da mesma maneira, demonstrando uma grande discrepância em termos de vazão de rede entre a aplicação *AJAX* (utilizando a técnica *polling*) e a aplicação *Node.js*.

Dentro dos resultados exibidos, podemos constatar, de acordo com Lubbers e Greco (s.d.), que aplicações com grande troca de informações, como por exemplo um *chat*, utilizando *AJAX polling*, causa muito tráfego na rede, gerando um desperdício em comparação com o tráfego de rede pertencente a aplicação utilizando *WebSocket*.

7 Conclusões e Trabalhos Futuros

Este trabalho de pesquisa e aprendizagem sobre a *Web* em tempo real reuniu diversas fontes de informação, contextualizando fatores históricos, analisando sua evolução pelo surgimento de diversas tecnologias *Web*, seus casos de uso e aplicação.

Durante a pesquisa, fora estudado a tecnologia *AJAX*, que surgira para atender uma demanda de mercado crescente na *Web*, pela qual transformou seu propósito de utilização, alcançando meios mais interativos e acessíveis, a *Web 2.0*. A tecnologia *AJAX* foi, então, dividida em técnicas, conhecidas como *Comet*. Técnicas como *polling, long pooling, reverse ajax, ajax push, HTTP Streaming* e *HTTP server push* estão entre elas. As mais utilizadas foram explicadas e levantados seus problemas.

Foi alvo de pesquisa também o avanço em direção a uma *Web* mais ágil, novamente em busca de atender novas demandas advindas uma *Web* mais social. Neste cenário surgiu uma tecnologia que está atualmente vem modificando a forma de desenvolver para a *Web*, o protocolo *WebSocket*. Deste protocolo foi levantado suas características, seus benefícios, seus casos de uso e em como ele se sobressai sobre as técnicas *AJAX Comet*. Ademais, outras tecnologias também entraram em foco como o protocolo *HTTP* e sua evolução, o protocolo *SPDY*, e a mais recente e surpreendente *WebRTC*, que quebra o paradigma cliente-servidor e torna possível aplicações cliente-a-cliente (*peer-to-peer*) em tempo real na *Web*.

Neste contexto, fora elencado uma série de opiniões e conceitos a cerca da *Web* em tempo real, sua justificativa e demandas, além de casos de uso na *Web* de empreendedores que ousaram testar novas tecnologias em seus negócios e perceberam ganhos e satisfação aumentada.

Em seguida, como parte e benefício de aprendizagem, foram estudadas algumas ferramentas para desenvolvimento de aplicações em tempo real. Dentre as ferramentas pesquisadas, as plataformas *Node.js* e *MeteorJS* foram selecionadas e utilizadas para desenvolvimento de aplicações e teste comparativo. Em primeira instância, foi desenvolvido uma rede social de perguntas e respostas com a *framework MeteorJS* e explicado a implementação de suas funcionalidades em tempo real da aplicação. Esta aplicação demonstrou o uso do padrão *PubSub* muito utilizado no modelo *push*, na qual temos um serviço que publica informação e agente que se inscreve para receber esta publicação. Além do *PubSub*, esta aplicação demonstra um conceito advindo da *framework MeteorJS* bastante interessante: a reatividade por todo o lugar da aplicação. Em segunda instância, foi desenvolvido uma aplicação de *chat* em tempo real com a plataforma *Node.js* e então realizado um teste comparativo com uma outra aplicação de

chat desenvolvido com técnica *AJAX polling*. Os testes apontaram para uma reafirmação dos benefícios do protocolo *WebSocket* apresentados neste trabalho.

Ainda assim, uma problemática enfrentada durante as pesquisas foi a falta de referências para pesquisa a cerca de estudos sobre a relação de mercado e a *Web* em tempo real com dados mais detalhados e estatísticos.

Em resumo, este trabalho teve o propósito de trazer um estudo e aprendizagem a cerca do tema Web em tempo real, colaborando bibliograficamente para a comunidade e abrindo espaço para trabalhos futuros na área, como por exemplo, desenvolvimento de aplicações e serviços em tempo real na *Web*. Dentre estes trabalhos futuros, pode-se destacar tecnologias que tragam interatividade em tempo real no âmbito acadêmico e social, como aplicações que apoiem o ensino à distância, *webinars*, compartilhamento e edição de documentos em tempo real, serviços de entrevistas com transmissão de áudio e vídeo, aplicações com serviço *peer-to-peer* para *Web*, ou até mesmo uma rede social ou fórum em tempo real. Tomando a rede social desenvolvida para este trabalho, um trabalho futuro pode extendê-la para ter mais funcionalidades como: seguir e encontrar usuários pela rede, compartilhamento de vídeos e tutoriais, criação de salas de *chat* e de escritório colaborativo, entre outras ideias.

Referências

COLEMAN, Tom; GREIF, Sacha. *Discover Meteor*: *Building Real-Time Javascript Web Apps*. [s.n.], 2013. 237 p. Disponível em: <http://book.discovermeteor.com>. Acesso em: 10 jan. 2014.

FROMM, Ken. *The Real-Time Web*: *A Primer, Part 1*. *ReadWriteWeb*, ago. 2009. Disponível em: <http://readwrite.com/2009/08/29/the_real-time_web_a_primer_part_1>. Acesso em: 10 mar. 2013.

GRIGORIK, Ilya. *High Performance Browser Networking*. 1. ed. [S.l.]: O'Reilly Media, 2013. 404 p.

HICKSON, Ian. *web socket protocol in "last call"?* out. 2009. Disponível em: <http://www.ietf.org/mail-archive/web/hybi/current/msg00784.html>.

HOLDENER, Anthony T. *Ajax: The Definitive Guide*. 1. ed. [S.l.]: O'Reilly Media, 2008. 982 p.

IETF.ORG. *Hypertext Transfer Protocol (httpbis)*: *Charter for Working Group*. [S.l.], out. 2012. Tradução nossa. Disponível em: <http://datatracker.ietf.org/doc/charter-ietf-httpbis>. Acesso em: 14 jul. 2014.

INGEBRIGTSEN, Einar. *SignalR: Real Time Application Development*. [S.l.]: Packt Publishing Ltd, 2013. 124 p.

ISKOLD, Alex. *Faster Why Constant Stress is Part of Our Future*. *ReadWriteWeb*, abr. 2008. Disponível em: <http://www.readwriteweb.com/2008/04/24/faster_constant_stress_future>. Acesso em: 15 mar. 2013.

JOUANNEAU, Laurent et al. *Introduction to WebRTC architecture*. Mozilla.org, jun. 2014. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC/WebRTC_-architecture>. Acesso em: 14 jul. 2014.

KIRKPATRICK, Marshall. *3 Models of Value in the Real Time Web*. *ReadWriteWeb*, 2009a. Disponível em: <http://readwrite.com/2009/05/07/three_models_of_value_in_the_real_time_-web>. Acesso em: 15 mar. 2013.

KIRKPATRICK, Marshall. *The Real-Time Web and its Future*. *ReadWriteWeb*, dez. 2009c. Disponível em: <http://readwrite.com/2009/12/03/introduction_to_the_real-time_web_and_-its_future>. Acesso em: 15 mar. 2013.

LENGSTORF, Jason; LEGGETTER, Phill. *Realtime Web Apps*. [S.l.]: Apress, 2013. 312 p.

LORETO, Salvatore; ROMANO, Simon Pietro. *Real-Time Communication with WebRTC*. 1. ed. [S.l.]: O'Reilly Media, 2014. 149 p.

LUBBERS, Peter; GRECO, Frank. *HTML5 Web Sockets*: *A Quantum Leap in Scalability for the Web*. *Websocket.org*, s.d. Disponível em: <http://www.websocket.org/quantum.html>. Acesso em: 10 jun. 2014.

METEOR.COM. *Framework MeteorJS*. 2014. Disponível em: <https://www.meteor.com>. Acesso em: 5 jun. 2014.

MONGODB.ORG. *MongoDB*. 2014. Disponível em: <http://www.mongodb.org>. Acesso em: 5 jun. 2014.

NODEJS.ORG. *Nodejs*. 2014. Disponível em: <http://nodejs.org>. Acesso em: 15 jul. 2014.

PEREIRA, Caio Ribeiro. *Node.js*: *Aplicações web real-time com Node.js*. Rua Vergueiro, 3185 - 8o andar: Casa do Código, 2014a. 145 p.

PEREIRA, Caio Ribeiro. *Meteor*: *Criando aplicações web realtime com JavaScript*. Rua Vergueiro, 3185 - 8o andar: Casa do Código, 2014b. 143 p.

RAI, Rohit. *Socket.IO Real-time Web Application Development*. Packt Publishing Ltd, 2013. 140 p. Disponível em: <http://www.google.com.br>. Acesso em: 10 jan. 2014.

RIORDAN, Rebecca M. *Head First Ajax*. [S.l.]: O'Reilly Media, 2008. 528 p.

RODEN, Ted. *Building the Realtime User Experience*. [S.l.]: O'Reilly Media, 2010. 321 p.

SERGIIENKO, Andrii. *WebRTC Blueprints*. [S.l.]: Packt Publishing Ltd, 2014. 176 p.

SYNODINOS, Dio. HTML 5 Web Sockets vs. Comet and Ajax. *InfoQ*, http://www.infoq.com/news/2008/12/websockets-vs-comet-ajax, dez. 2008. Acesso em: 10 mar. 2013.

TANENBAUM, Andrew S. *Redes de Computadores*. 4. ed. [S.l.]: Campus (Elsevier), 2003. 632 p.

TEIXEIRA, Pedro. *Professional Node.js*: *Building Javascript Based Scalable Software*. [S.l.]: John Wiley & Sons, 2013. 376 p.

UBL, Malte; KITAMURA, Eiji. Apresentando WebSockets: trazendo soquetes para a web. *html5rocks*, http://www.html5rocks.com/pt/tutorials/websockets/basics/, out. 2010. Acesso em: 10 mar. 2013.

WANG, Vanessa; SALIM, Frank; MOSKOVITS, Peter. *The Definitive Guide to HTML5 WebSocket*. [S.l.]: Apress, 2013. 208 p.

WEBRTC.ORG. *General Overview*. *WebRTC.org*, 2014a. Disponível em: <http://www.webrtc.org/reference/architecture>. Acesso em: 16 jul. 2014.

WEBRTC.ORG. *WebRTC*. *WebRTC.org*, 2014b. Disponível em: <http://www.webrtc.org>. Acesso em: 16 jul. 2014.

WEBSOCKET.ORG. *What is WebSocket?* 2014a. Disponível em: <http://www.websocket.org>. Acesso em: 11 jun. 2014.