O Uso de *WebSockets* no Desenvolvimento de Sistemas Baseados em uma Arquitetura *Front-end* com API

Guilherme S. A. Gonçalves, Paulo Henrique O. Bastos, Daniel de Oliveira

Instituto de Computação – Sistemas de Informação – Universidade Federal Fluminense (UFF) – Caixa Postal 24.210-240 – Niterói – Rio de Janeiro – Brasil

galves@id.uff.br, ph_ouverney@id.uff.br,danielcmo@ic.uff.br

Resumo. *Este artigo apresenta um relato de experiências sobre o uso de WebSockets como uma tecnologia capaz de atender um novo tipo de tendência gerada pelas demandas dos sistemas de informação que necessitam respostas em tempo real. Também é discutido nesse artigo como o WebSocket pode ser integrado a uma arquitetura RESTful, criando um meio de comunicação em módulos que são utilizados para prover serviços aos usuários. Assim, nesse artigo, discutimos as vantagens e as desvantagens do uso dessa tecnologia em relação ao padrão de arquitetura atual para o desenvolvimento de aplicações Web.*

1. Introdução

Em muitos dos sistemas de informação atuais a pronta resposta a ações do usuário tem se tornado um requisito quase que obrigatório. Dependendo do tipo de aplicação, não é recomendado que o usuário espere muito tempo até que se tenha um *feedback*. Um exemplo de aplicação com essa necessidade é o TaxiVis (Ferreira *et al.*, 2013), uma aplicação desenvolvida pela NYU (Universidade de Nova York) onde agentes do governo são capazes de mapear dados de movimento de taxis na cidade de Nova Iorque em tempo real. Para que se tomem ações rápidas, não faz sentido que a resposta do sistema seja demorada. Esse problema torna-se ainda mais complexo em aplicações *Web,* dado o tipo de arquitetura das aplicações. Assim, é indispensável nos dias de hoje que o atraso de comunicação entre o servidor e o cliente seja quase nulo e o limite de atraso sempre seja reduzido (Horey e Lagesse, 2011). Exemplos práticos são aplicações de conversas *online* ou um sistema de compra de ações, onde nesses casos queremos que as informações sejam disponibilizadas para nós o mais rápido possível.

Tradicionalmente, as aplicações *Web* tem sido desenvolvidas utilizando arquitetura MVC (*Model-View-Controller*), que é um padrão de projetos amplamente utilizado nas práticas de programação. Este é composto por três módulos: o modelo, onde são implantadas as regras de modelagem e de negócio; a visão, onde estão localizadas as interfaces no qual as interações com os usuários são realizadas; e por último o controlador, no qual são geridas as comunicações e conexões entre o modelo e a visão. (Armeli-Battana, 2012).

O funcionamento desse padrão de programação ocorre de modo que uma requisição é realizada por meio da visão ao controlador que executa um processo, podendo invocar um ou mais serviços para entregar à visão os dados requisitados ou executar uma determinada ação. O controlador é responsável por gerenciar o fluxo das páginas e persistir em sessão os objetos que serão acessados. Uma das grandes vantagens do modelo MVC é que ele é de fácil implementação e aceitação no mercado,

já que é o padrão *de facto* na indústria. Por seu claro entendimento, por prover uma organização do código e facilitar a programação, muitos entusiastas do padrão disponibilizam materiais, explicações, tutoriais e cursos sobre o tema, que acaba difundindo ainda mais o padrão. Além de ter uma compreensão transparente do fluxo, por ele estar bem explícito.

Apesar das vantagens anteriormente apresentadas, o MVC tem algumas desvantagens na sua implementação. As regras de negócio se encontram fixas no código e não podem ser compartilhadas para outras aplicações, como por exemplo aplicações móveis em celulares e *tablets*. Isto torna a aplicação "engessada" para um determinado tipo de arquitetura, inviabilizando reutilização de código e dos serviços. Para resolver esse problema, seria necessário implementar uma API para que esses serviços estivessem disponíveis para outras arquiteturas. Isso faz com que o trabalho seja dobrado. Além disso, a própria definição do MVC já faz com que a troca de mensagens insira um retardo de comunicação que em alguns tipos de aplicação pode não ser aceitável.

Essas limitações do MVC vem sendo suplantadas nos últimos anos graças às novas tecnologias que surgem para prover novos tipos de serviços para áreas que demandam cada vez mais processamento e transferência de dados com atrasos mínimos e garantias de segurança. Um exemplo é a arquitetura *Front-end* que consome serviços implementados por uma API. Nessa arquitetura, o *Front-end* pode utilizar o modelo MVVM - *Model-View-ViewModel*, que pode ser visto na Figura 1, implementado, por exemplo, pelo AngularJS. A API que vai prover os serviços pode ser desenvolvida em qualquer linguagem ou até em múltiplas linguagens, sendo o mais importante a forma como será realizada a comunicação entre elas e os módulos interligados. Elas podem utilizar o formato REST ao invés do SOAP para a comunicação, utilizando o JSON (Richardson e Ruby, 2008) no formato de mensagens trocadas. Na Figura 2 é exibido um exemplo de mensagem no formato JSON que pode ser trocada pelas aplicações.

Assim como o MVC, a arquitetura *Front-end* com API possui vantagens e desvantagens associadas. A maior vantagem dessa arquitetura *Front-end* com API é a separação das obrigações, onde o desenvolvedor define que as regras de negócio ficam na API, o que possibilita seu compartilhamento por diversas aplicações e viabiliza as práticas de reutilização de código, evitando o retrabalho.

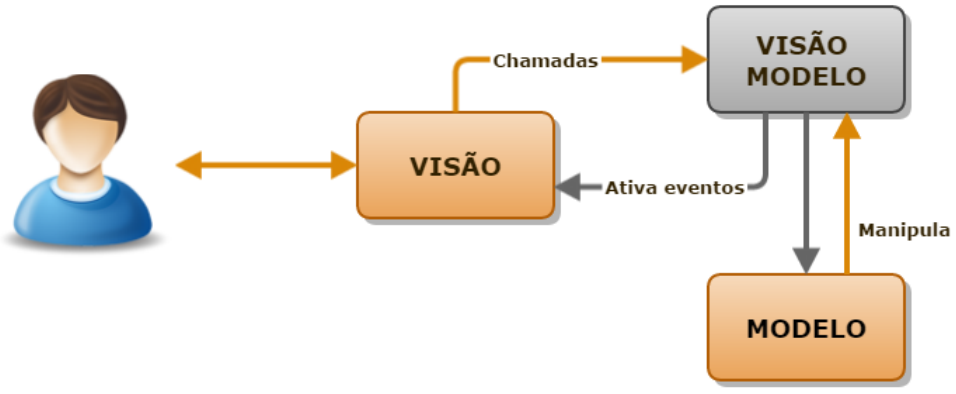

Figura 1. Um esquema conceitual da arquitetura Model-View-ViewModel.

Já a maior desvantagem dessa arquitetura é o fato de ser ainda pouco conhecida e adotada, pois há pouca divulgação e oferta de aprendizado. Outra desvantagem conhecida está relacionada ao esforço de se implantar um serviço de segurança, onde o tráfego de mensagens JSON deve ser criptografado antes de ser enviado, pois há o risco das mensagens serem interceptadas durante a comunicação. Esse requisito de segurança é um ponto crucial na aplicação dessa arquitetura. Além disso, ainda existe o problema de retardo de comunicação que deve ser tratado para que possa ser utilizado para desenvolver aplicações que demandem respostas em tempo real.

Esse artigo apresenta algumas discussões e soluções para a implantação e o uso da arquitetura *Front-end* com API por meio do uso de *WebSockets* de forma que ela possa ser mais difundida como uma alternativa ao padrão MVC.

```
{
    id:2,
    nome: Daniel de Oliveira,
    endereco: {
                    id: 2,
                    estado: Rio de Janeiro,
                    cidade: Niterói,
                    rua: Rua Passo da Pátria,
                    numero: 156
                }
}
```

Figura 2. Exemplo de uma mensagem no formato JSON.

2. Princípios da Arquitetura *Front-end* com API com uso de *WebSockets*

Para que possamos suplantar os problemas apresentados anteriormente da arquitetura *Front-end* com API, temos que adaptar alguns pontos no seu uso. Em relação aos atrasos de comunicação e de forma a obter latências de conexões quase nulas ou em tempo real entre clientes e servidores é necessário que utilizemos mecanismos os quais oferecem mais recursos do que o serviço oferecido pelo protocolo HTTP.

Os *WebSockets* surgiram para prover esse tipo de serviço com retardo reduzido, o qual o HTTP não oferece apoio atualmente. O *WebSocket* é um padrão ainda em desenvolvimento que está sob os cuidados da IETF (*Internet Enginnering Task Force*) e uma API padrão para implementação do protocolo está em processo de formalização pela W3C (*World Wide Web Consortium*) para que os navegadores ofereçam apoio ao protocolo. O suporte dos navegadores ao protocolo já se encontra em desenvolvimento e disponibilização ao usuário final, entretanto utilizar as versões mais recentes dos navegadores será indispensável para quem quiser desenvolver sistemas de informação utilizando essa arquitetura e usufruir do serviço que está em constante aprimoramento. (Cutting Edge, 2014).

Por implementar um canal de transmissão bidirecional e por ter um cabeçalho menor, o *WebSocket* supera o HTTP em relação ao número de requisições feitas e no tamanho das mensagens. Podemos ver na Figura 3, que o *handshake* do *WebSocket* é mais simples, pois é feito apenas um por conexão, enquanto no HTTP é realizado por

requisição. Após a conexão ser estabelecida, somente são trocadas mensagens e sem necessidades de um cabeçalho extenso.

Em relação ao requisito de segurança, podemos utilizar o *WebSocket secure* (WSS) como uma alternativa de meio de transmissão seguro em relação ao *WebSocket* convencional (Richardson e Ruby, 2008).

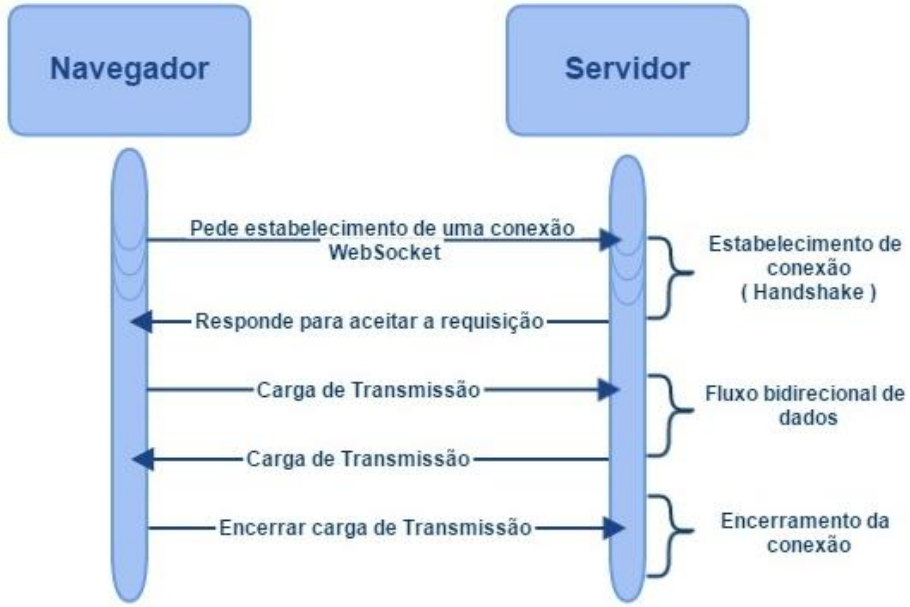

Figura 3 Handshake no WebSocket.

A Figura 4 e a Tabela 1 apresentam resultados comparativos entre um sistema de informação *Web* utilizando o REST por HTTP e a outra utilizando *WebSocket*. Os sistemas que utilizam *WebSocket* são capazes de processar mais mensagens em uma determinada faixa de tempo em comparação aos sistemas que são baseados no protocolo HTTP, dessa forma aumentando a vazão de informação que o sistema é capaz de processar.

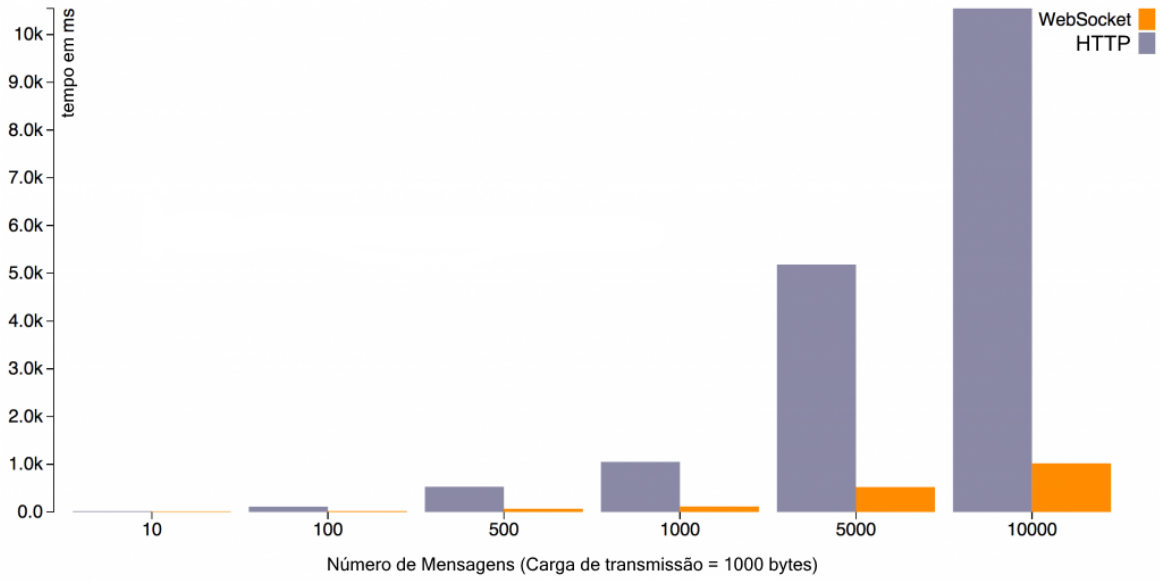

Figura 4. Gráfico comparativo entre o *WebSocket* e a arquitetura REST utilizando o protocolo HTTP em relação ao números de mensagens enviadas adaptado de (Planet JBoss, 2014).

Tabela 1. Tabela comparativa entre o *WebSocket* e a arquitetura REST utilizando o protocolo HTTP em relação ao números de mensagens enviadas (Planet JBoss, 2014).

Mensagens	HTTP (em ms)	*WebSocket* (em ms)	Proporção
10	17	13	1,31
100	112	20	5,60
500	529	68	7,78
1000	1.050	115	9,13
5000	5.183	522	9,93
10000	10.547	1.019	10,35

Outro problema do uso de *WebSockets* é que o seu cabeçalho não é extenso e somente disponibiliza os métodos "*OnOpen*", "*OnMessage*" e "*OnClose*", não é possível identificar que tipo de requisição é realizada através de uma URI, como é feito em aplicações *RESTful* com HTTP (Fette e Melnikov, 2014). Devido a essa limitação, a solução proposta nesse artigo é que o tipo de requisição realizada esteja na mensagem e não no cabeçalho do protocolo. Assim, na Figura 5 é exemplificada uma mensagem JSON sendo enviada para uma conexão já previamente aberta. Nota-se que o campo "*method*" é responsável por informar à API que tipo de requisição é realizada, podendo aceitar os valores: "*get*", "*create*", "*update*", "delete" e "*findAll*".

```
{
        "callback_id": "1",
        "method": "get",
        "data": {
                "type": "user",
                "id": "9"
                }
}
```

Figura 5. Exemplo de uma mensagem no formato JSON utilizando a metodologia para criar uma arquitetura RESTful.

É importante abrir uma conexão por módulo/domínio e cada conexão aberta deve ser responsável por receber os tipos de requisições para o seu módulo/domínio. Na Figura 6, é apresentada a arquitetura da solução proposta, onde os módulos/domínios

são usuário e empresa, e os métodos abaixo do módulo/domínio serão acessados por meio de sua conexão responsável. A conexão somente irá se fechar quando não for mais necessário acessar qualquer serviço de usuário ou empresa.

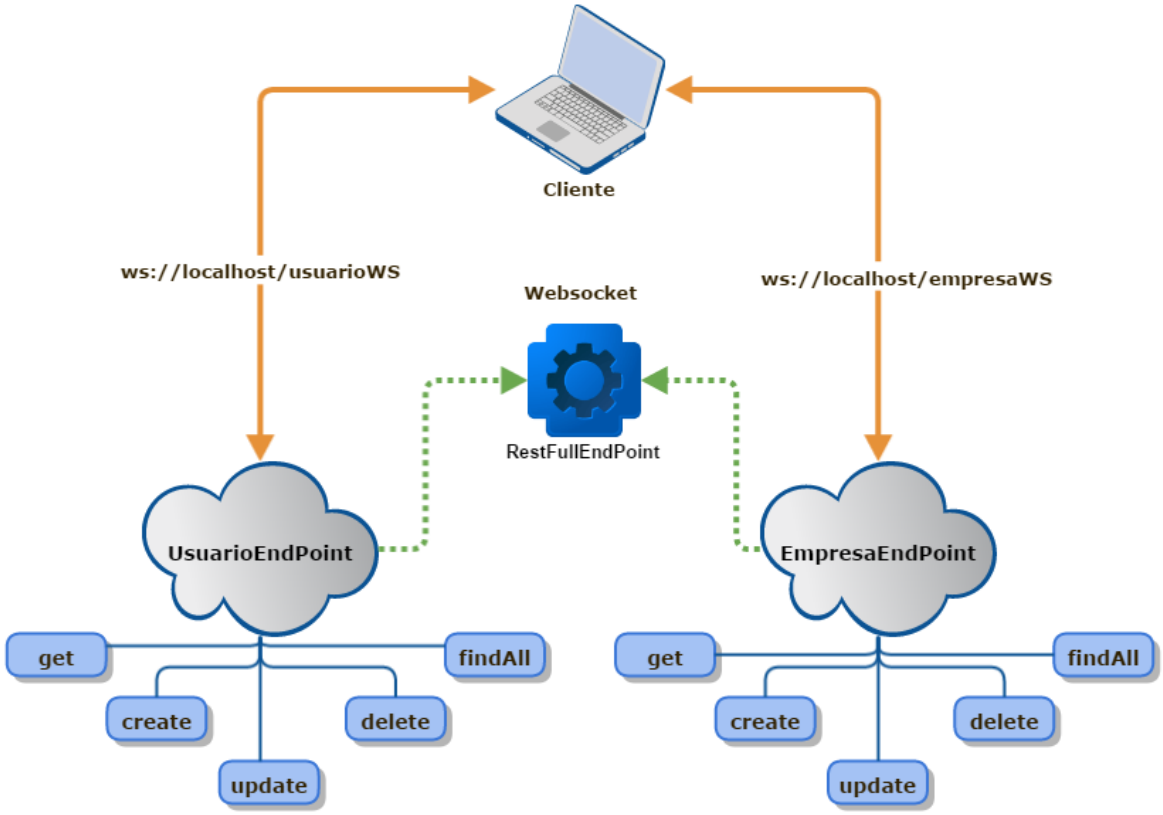

Figura 6. Exemplo de conexão do cliente com dois módulos utilizando uma comunicação *WebSocket*.

3. Conclusões

Esse artigo tem como objetivo apresentar uma nova opção de arquitetura de sistemas de informação em relação ao tradicional modelo MVC. Apesar de ainda possuir uma série de limitações e problemas envolvidos, essa arquitetura se mostra promissora para o desenvolvimento de sistemas de informação que necessitem ser multiplataforma e que necessitem de respostas em tempo real. A utilização de *WebSockets* se mostra uma solução promissora para suplantar os problemas da arquitetura *Front-end* com API e certamente deve ser alvo de investigação tanto por parte da academia quando por parte da indústria nos próximos anos.

4. Referências Bibliográficas

Nivan Ferreira, Jorge Poco, Huy T. Vo, Juliana Freire, Cláudio T. Silva. (2013): Visual Exploration of Big Spatio-Temporal Urban Data: A Study of New York City Taxi Trips. IEEE Trans. Vis. Comput. Graph. 19(12): 2149-2158

James Horey, Brent Lagesse. (2011) Latency Minimizing Tasking for Information Processing Systems. ICDM Workshops., p. 167-173

Sebastiano Armeli-Battana. (2012), MVC Essentials [Kindle Edition], developer press.

Padrões de Projeto - O modelo MVC - Model View Controller ([S.d.]). http://www.macoratti.net/vbn_mvc.htm, [acessado em Setembro de 2014].

.NET - Apresentando o padrão Model-View-ViewModel ([S.d.]). http://www.macoratti.net/11/06/pp_mvvm1.htm, [acessado em Setembro de 2014].

Richardson, L. e Ruby, S. (2008), *RESTful Web Services*. O'Reilly Media, Inc.

Kaazing Developer Network ([S.d.]). http://developer.kaazing.com/documentation/jms/4.0/security/c_sec_https_wss.html, [acessado em Setembro de 2014].

Planet JBoss, REST vs WebSocket Comparison and Benchmarks | Planet JBoss Developer ([S.d.]). https://planet.jboss.org/post/rest_vs_websocket_comparison_and_benchmarks, [acessado em Setembro de 2014].

Fette, I. e Melnikov, A. ([S.d.]). The WebSocket Protocol. http://tools.ietf.org/html/rfc6455, [acessado em Setembro de 2014].

Castillo, I. e Pascual, V. ([S.d.]). The WebSocket Protocol as a Transport for the Session Initiation Protocol (SIP). http://tools.ietf.org/html/rfc7118, [acessado em Setembro de 2014].

Cutting Edge - Compreendendo o poder dos WebSockets ([S.d.]). http://msdn.microsoft.com/pt-br/magazine/hh975342.aspx, [acessado em Setembro de 2014].

De Col, A. and Nesello, F. A. (2014). Aplicativo web com JSF 2.0 e PrimeFaces para gerenciamento de requisitos de software.