Uma Abordagem Baseada em WebSocket para Comunicação em Tempo Real no GeneMaisLab

Eliseu Germano[1], Marcelo Narciso[1]

[1]Embrapa Arroz e Feijão
Santo Antônio de Goiás – GO – Brasil

eliseu.germano1@gmail.com, marcelo.narciso@embrapa.br

Abstract. *There is a growing number of software development projects that use service-oriented architecture with RESTful stateless APIs. However, there are a number of challenges to developing computer systems with these characteristics, especially when it is necessary to ensure some synchronization logic and data consistency between the active client applications. In this article some approaches are presented to deal with this issue, emphasizing the use of the WebSocket protocol in an Embrapa system called GeneMaisLab. For this, a software architecture is presented that represents the way in which the protocol was used and the results obtained from an implementation are discussed.*

Resumo. *É crescente a quantidade de projetos de desenvolvimento de software que utilizam arquitetura orientada a serviços com APIs RESTful stateless. No entanto, há uma série de desafios para desenvolver sistemas computacionais com essas características, principalmente quando é necessário garantir alguma lógica de sincronização e uma consistência de dados entre os clientes ativos. Neste artigo são apresentadas algumas abordagens para lidar com essa questão, dando ênfase ao uso do protocolo WebSocket em um sistema da Embrapa chamado GeneMaisLab. Para isso, é apresentada uma arquitetura de software que representa a forma na qual o protocolo foi empregado e são discutidos os resultados obtidos a partir de uma implementação.*

1. Introdução

As práticas contemporâneas de desenvolvimento de *software* incentivam a construção de sistemas seguindo abordagens baseadas em componentes, ou seja, fomentam a construção de conjuntos de componentes funcionais e lógicos de forma independente e o agrupamento deles de acordo com a arquitetura de cada sistema [Kaur and Mann 2010]. Apesar de alguns benefícios oferecidos por esta estratégia, como a facilidade de reutilização de componentes, a complexidade em relação a integração e implantação do sistema é consideravelmente maior.

Como forma de viabilizar a construção de serviços escaláveis em uma arquitetura cliente/servidor, uma estratégia utilizada consiste no desenvolvimento APIs RESTFul *stateless*. Essa estratégia funciona a partir do desenvolvimento de serviços *web* que não guardam em sessão os estados (informações) das aplicações clientes. Dessa forma, pode ser viabilizado a construção de sistemas com escalabilidade horizontal a partir da replicação em tempo de execução de recursos implementados na forma de serviços. No entanto, a decisão de não guardar essas informações em sessão implica na necessidade de adotar um mecanismo de sincronização para trocas de mensagens entre os clientes e os serviços que executam os recursos no servidor.

Neste artigo são abordados os mecanismos utilizados para viabilizar uma lógica de sincronização e garantir a consistência de dados entre diferentes usuários que utilizam simultaneamente um sistema implementado com APIs RESTFul *stateless*. Esse sistema, chamado de GeneMaisLab, é responsável por automatizar o processo de gerenciamento de ensaios de genotipagem de experimentos realizados na Empresa Brasileira de Pesquisa Agropecuária (Embrapa). Durante o desenvolvimento deste texto são apresentados tanto os procedimentos implementados no GeneMaisLab, que seguem uma abordagem baseada no protocolo WebSocket, quanto os procedimentos alternativos acompanhado das justificativas de não utilizá-los.

O conteúdo deste artigo está organizado da seguinte forma: na Seção 2 são apresentados os conceitos em torno do projeto GeneMais a partir de uma visão arquitetural do GeneMaisLab; A Seção 3 traz alguns requisitos de adaptação do sistema e algumas decisões de projeto; A Seção 4 apresenta o componente de notificação implementado no sistema; e na Seção 5 são apresentadas as considerações finais.

2. Visão Arquitetural do GeneMaisLab

O GeneMaisLab, referenciado como Gene+ em [Germano et al. 2016], é um sistema desenvolvido pela Embrapa Arroz e Feijão, cujo o propósito é gerenciar os dados de ensaios de genotipagem desde o planejamento até as etapas em que são feitas as considerações finais. A partir da centralização desses dados, o sistema permite o registro de eventos durante a execução dos ensaios, viabiliza o rastreamento das informações registradas e possibilita consultas aos resultados obtidos durante um processo de genotipagem.

Do ponto de vista computacional, o GeneMaisLab é composto por um conjunto de componentes de *software* implementados na forma de serviços *web stateless* e um conjunto de componentes de interfaces que compõem aplicações *web* que realizam chamadas aos serviços. A Figura 1 apresenta, de forma simplificada, a arquitetura do sistema dando destaque a duas regiões, uma região à esquerda representada por dispositivos que podem acessar o sistema a partir de *web browsers* e uma região à direita contendo os componentes de *software* que recebem, processam e respondem a requisições das aplicações.

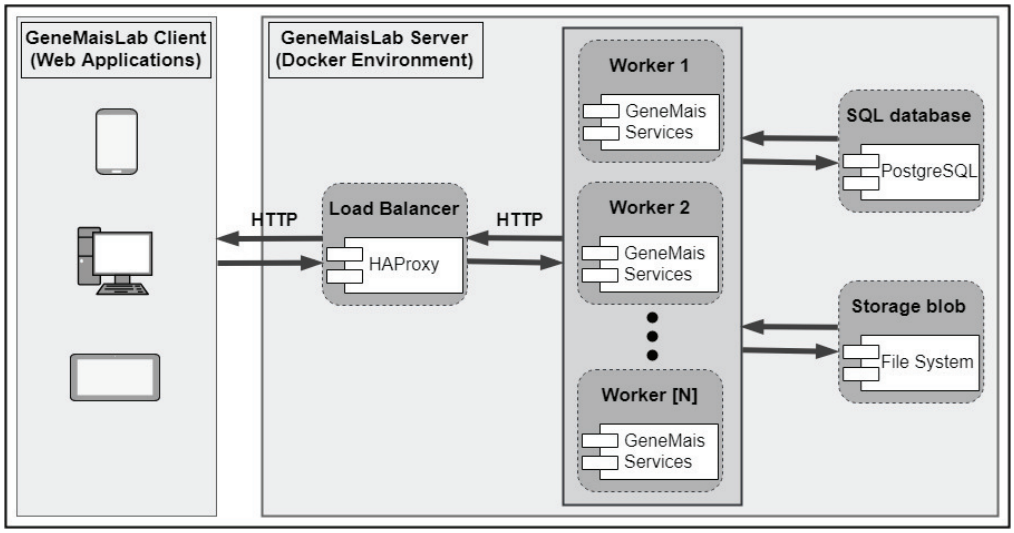

Figura 1: Arquitetura do GeneMaisLab.

Os componentes que recebem as requisições das aplicações são executados em um ambiente composto por *containers* da plataforma Docker. Os *containers* são uma forma de virtualização em nível de sistema operacional que viabiliza a execução de múltiplos componentes de *software* em um mesmo *host* com um nível de isolamento [Bernstein 2014]. Na Figura 1 há um *container* que recebe as requisições (*Load Balancer*) e, de acordo com um algoritmo de balanceamento de carga, realiza o redirecionamento de cada uma delas para um serviço que possa processá-la (*Worker*). Além disso, há dois outros *containers*, sendo um deles para persistência de dados (*SQL database*) e um outro para gerenciamento de arquivos estáticos (*Storage blob*).

Uma outra observação que deve ser considerada nessa arquitetura é que os componentes que implementam os serviços do GeneMaisLab são acessados a partir do protocolo *Hypertext Transfer Protocol* (HTTP). Além disso, o mecanismo de acesso a esses serviços segue uma abordagem baseada em uma arquitetura RESTFul, que é derivada de um estilo arquitetural conhecido como *Representational State Transfer* (REST) [Fielding 2000]. Uma característica comum das implementações de serviços *web* realizadas a partir de arquiteturas RESTFul é que geralmente são utilizadas APIs *stateless*, ou seja, os recursos executados no servidor não guardam em sessão o estado de aplicações cliente. Assim, toda requisição HTTP acontece em completo isolamento, de forma que qualquer requisição feita por uma aplicação cliente deve conter todas as informações necessárias para o servidor possa atendê-la. Uma das vantagens de utilizar APIs *stateless* está relacionada a possibilidade escalar os serviços de acordo com a demanda de acesso em um determinado momento.

Por outro lado, utilizar APIs *stateless* em serviços implementados a partir do protocolo HTTP pode implicar na necessidade de tratar requisitos de sincronização e consistência de dados em aplicações clientes, principalmente quando há acesso simultâneo a recursos compartilhados (eg., diferentes navegadores *web* acessando uma mesma página). Um cenário comum para esse tipo de situação ocorre em interfaces de aplicações que apresentam dados que são registrados em um banco de dados e que são compartilhados com vários usuários de um sistema. Nesse cenário, sempre que um dado é alterado no banco, as interfaces que estejam apresentando esses dados também precisam ser atualizadas para evitar inconsistências entre os valores registrados e os valores apresentados.

Uma vez que serviços *web* implementados com o protocolo HTTP são agentes passivos em uma comunicação cliente/servidor, é necessário que as aplicações clientes acessem o(s) servidor(es) para realizar qualquer tipo de sincronização de dados. Dessa forma, é necessário que sejam utilizados mecanismos de comunicação que geram eventos nas aplicações para que elas busquem algo em um servidor. Os detalhes em relação a esses mecanismos são apresentados na próxima seção.

3. Adaptação de Componentes de Interface em Tempo Real

Uma das principais motivações para realizar o desenvolvimento de um sistema gerenciador de ensaios de genotipagem na Embrapa está relacionada a falta de padronização em relação a forma em que os dados de um ensaio são estruturados. Essa falta de padronização se deve ao fato de que os tipos de dados que podem compor um ensaio cresce e modificam constantemente. Os marcadores moleculares, por exemplo, que são regiões sequenciais no *Deoxyribonucleic acid* (DNA) geralmente utilizadas em laboratórios para distinguir células, individuos, populações ou espécies, é um tipo de dado que deve ser usado na composição de um ensaio. No entanto, constantemente são

descobertos novos marcadores moleculares e algumas informações em relação aos marcadores existentes são frequentemente modificadas (ex., custo de utilizá-lo em uma análise).

A solução encontrada no GeneMaisLab para representar os dados de um ensaio foi identificar padrões entre os dados que podem compor um ensaio e agrupá-los em um conjunto de listas. Dessa forma, além dos marcadores moleculares, outros tipos de dados como, métodos de extração de DNA, plataformas de genotipagem, natureza das amostras, entre outros, foram padronizados e agrupados em listas de padrões. Diante disso, foram criados mecanismos para cadastrar/modificar essas listas no sistema.

As listas de padrões do GeneMaisLab contém apenas dados que são comuns a diferentes tipos de ensaios. Por exemplo, a cultura arroz é um tipo de dado que é comum a todos ensaios em que são analisadas amostras de arroz, logo ela é um padrão no sistema. Os dados que possuem uma alta variabilidade entre os ensaios são importados de planilhas preenchidas pelo usuário ou de outros sistemas. Assim, o cadastro de um ensaio no GeneMaisLab consiste de um formulário dinâmico, constituído de vários campos cujo os possíveis valores são determinados pelas listas de padrões. Dessa forma, no campo de marcadores moleculares, por exemplo, deve conter todos os marcadores registrados no sistema. Caso um novo marcador seja adicionado ou modificado na lista de padrões de marcadores, o campo que contém os marcadores no formulário de cadastro deve ser automaticamente modificado na *interface*. Essa ideia se aplica a todos os outros componentes que possuem elementos dinâmicos na *interface* do sistema.

Considerando que o GeneMaisLab é um sistema *web* onde múltiplos usuários podem utilizá-lo simultaneamente, sempre que um dado é adicionado ou modificado em uma das listas de padrões, as interfaces *web* precisam ser adaptadas para todos os usuários que estejam utilizando o sistema naquele momento. Caso contrário, o sistema não poderia garantir a consistência de dados entre esses usuários. Diante disso, um das preocupações durante o desenvolvimento desse sistema consistiu em analisar o estado da arte em relação às técnicas de comunicação de em tempo real para lidar com o problema de sincronização de dados entre aplicações que consomem serviços a partir de APIs *stateless*.

Atualmente, considerando os métodos de conexão entre aplicações clientes e servidoras na *web*, os que são mais conhecidos são: *Polling* (ou *Short Polling*), *Long Polling* e WebSocket [Zhang and Shen 2013]. A técnica conhecida como *Polling* consiste em um processo pelo qual o cliente solicita regularmente novos dados ao servidor. De modo geral, a técnica pode ser realizada de duas formas, sendo conhecidas como *Short Polling* e *Long Polling*.

O *Short polling* é um procedimento que conta com um temporizador utilizado no cliente que determina a periodicidade em que são realizadas chamadas ao servidor. Assim, caso essa periodicidade seja baixa, o cliente pode ficar desatualizado por um longo tempo. Por outro lado, caso a periodicidade seja alta, deve haver uma sobrecarga de requisições ao servidor e um aumento de tráfego na rede. Diante disso, o *Short polling* não se mostra como um bom método para lidar com sistemas que precisam de algum tipo de sincronização em tempo real.

Como alternativa ao *Short polling*, o *Long polling* consiste em um procedimento em que o cliente inicia e mantém uma conexão aberta por um determinado tempo. Usando esse método o servidor responde uma requisição de um cliente apenas quando ocorre um evento ou quando o tempo limite conexão é excedido (*timeout*). Portanto, esse mecanismo minimiza a latência e o uso de recursos de processamento no servidor e

de rede se comparado ao *Short polling*. No entanto, alguns problemas ainda persistem, como a necessidade do cliente ficar bloqueado aguardando uma resposta e de reenviar requisições "desnecessárias" ao servidor (que embora sejam menor que no *Short polling* elas ocorrem até que a informação requisitada ao servidor seja atualizada).

Há outros mecanismos de comunicação que não possuem os problemas mencionados a técnica de *polling*. A opção adotada pelo GeneMaisLab foi por meio de um protocolo chamado WebSocket. A partir desse protocolo é possível criar uma conexão bidirecional por canais *full-duplex* utilizando um *socket* baseado no protocolo TCP. Isso permite que cliente e servidor possam trocar dados em ambas direções a qualquer momento em que conexão estiver ativa. Dessa forma, diferentemente da técnica de *polling*, com o WebSocket o cliente não precisa realizar múltiplas requisições ao servidor para obter uma informação, pois, uma vez que é estabelecida a conexão entre eles é criado um canal de comunicação que pode permanecer ativo até que o mesmo seja explicitamente fechado [Sparkes et al. 2016][Maia and Silva 2017].

A Figura 2 apresenta uma análise comparativa entre as abordagens que lidam com comunicação em tempo real discutidas nesta seção. A partir dela, percebe-se a ocorrência de dois eventos dentro de um determinado intervalo de tempo em um servidor, assim como o comportamento de aplicações clientes usando *Polling*, *Long Polling* e WebSocket. No caso da comunicação usando *polling*, são realizados acessos de forma periódica e contínua ao servidor antes e após a ocorrência dos eventos. No caso da comunicação usando *Long Polling* o procedimento é parecido, exceto pelo fato da periodicidade em que são feitas as requisições ser menor. Dessa forma, o cliente com *Long Polling* faz uma requisição e fica aguardando até que ocorra um evento no servidor (ex., Evento 1) ou que ocorra um *timeout* (ex., Evento 2). Por fim, na abordagem baseada em WebSocket é realizado uma única requisição do cliente ao servidor. Após essa requisição, é criado um canal de comunicação persistente entre eles e sempre que ocorre um evento no servidor o cliente pode ser notificado.

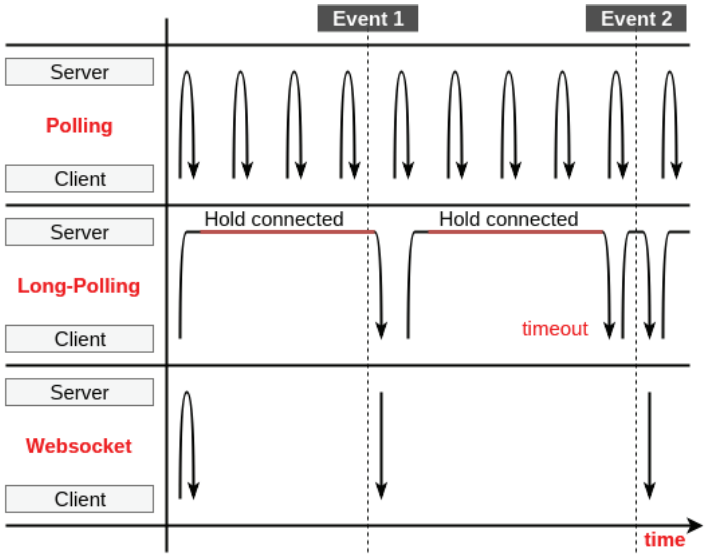

Figura 2: Comparação entre as Abordagens de Comunicação em Tempo Real.

Devido a conexão bidirecional viabilizada a partir do protocolo WebSocket, não há necessidade de um cliente fazer requisições a um servidor apenas para saber se há uma nova informação disponível. Dessa forma, isso pode intervir no desempenho de um sistema em virtude da redução do processamento e da latência proveniente da

diminuição do número de requisições [Jiang and Duan 2012]. Isso influenciou na escolha dessa abordagem durante o desenvolvimento do GeneMaisLab.

4. Usando WebSocket para Controle de Notificações no GeneMaisLab

Diante da decisão em utilizar WebSocket como um protocolo para viabilizar em tempo real a consistência dos dados entre diferentes usuários do GeneMaisLab, foram analisados os conceitos, técnicas e tecnologias em torno do protocolo. Em [Wirasingha and Dissanayake 2016] são apresentados os estilos arquiteturais que podem ser utilizados, além das APIs e frameworks que oferecem suporte a essa tecnologia.

No GeneMaisLab o WebSocket é utilizado como um mecanismo de *server push*, ou seja, uma vez que é criado um canal de comunicação entre um cliente e um servidor *web* do sistema, alguns serviços podem enviar notificações para os clientes. Como o GeneMaisLab foi desenvolvido em Java utilizando o Spring Framework[1] e Google Web Tookit (GWT)[2], não foi necessário adicionar nenhuma biblioteca ou *framework* novo no sistema para que fossem utilizadas as APIs de WebSocket, pois tanto o Spring quanto o GWT oferecem suporte a essa tecnologia. Em [Zhang and Shen 2013] pode ser verificado *frameworks* de outras linguagens de programação com suporte a WebSocket.

A Figura 3 apresenta a arquitetura do sistema após a adição de um módulo para lidar com as notificações. Esse módulo está encapsulado no componente *Notifications* e que é diretamente acessado pela aplicação cliente a partir do protocolo WebSocket. Assim, enquanto a transmissão de dados entre o cliente e um serviço do sistema (por intermédio de um balanceador de carga) ocorre de forma *half-duplex*, a transmissão de dados entre o cliente e o módulo de notificação ocorre de forma *full-duplex*. Isso permite que a partir do uso do protocolo WebSocket os dados sejam enviados de forma bidirecional entre o cliente e o módulo que foi concebido para realizar as notificações.

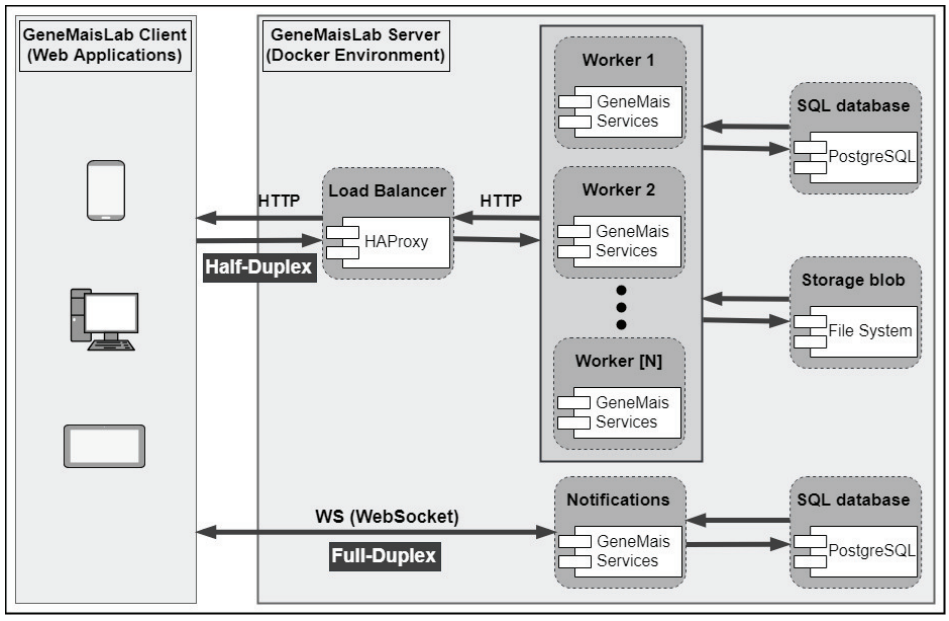

Figura 3: Arquitetura do GeneMaisLab Após a Adição do Módulo de Notificação.

[1] https://spring.io/
[2] http://www.gwtproject.org/

De modo geral, esse módulo de notificação funciona como um sinalizador para lidar com a consistência de dados entre diferentes usuários ativos simultaneamente no sistema. Dessa forma, sempre que um usuário realiza um novo acesso ao sistema (*login*) é criado um canal de comunicação entre a sua aplicação e o módulo de notificação. Logo, esse módulo mantém uma lista de usuários ativos no sistema, de forma que quando o usuário sai (*logout*) ele é removido dessa lista. Diante disso, sempre que é adicionado, removido ou modificado um dado de uma das listas de padrões do sistema, o módulo de notificações informa a um conjunto de usuários ativos (*multicast*) as alterações que devem ser realizadas. Assim, cada aplicação ao receber uma notificação, pode então consultar os serviços do GeneMaisLab para obter as atualizações.

A Figura 4 apresenta um trecho de código contendo os principais métodos do componente de notificações do GeneMaisLab. Desses métodos, dois deles são herdados da classe TextWebSocketHandler do Spring Framework, sendo eles o método 1 e o método 2, representado na figura. O método 1 é chamado sempre que um novo cliente solicita uma conexão. Sua função é de adicionar os dados do cliente em uma lista de sessões. Quando algum dos clientes que estão nessa lista enviam uma mensagem a partir do protocolo WebSocket, esta é recebida a partir do método 2. No método 2 as mensagens recebidas são recuperadas e colocadas em um objeto de notificação que é instanciado. Posteriormente é chamado o método 3, que recebe a mensagem já em um formato serializado (em JSON) e encaminha para todos os clientes com conexão ativa, com exceção do cliente que enviou a mensagem. A interpretação e o processamento de cada mensagem ocorrem nos clientes.

```
public class SocketHandler extends TextWebSocketHandler {
    final List<WebSocketSession> sessions = new ArrayList<WebSocketSession>();

    @Override
    protected void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {
        System.out.println("received a new message " + message.getPayload());
        NotificationService service = new NotificationService();
        service.addNotification(message.getPayload());
        multicastMessage(service.getNotification(), session);
    }

    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws Exception {
        sessions.add(session);
    }

    public void multicastMessage(String message, WebSocketSession current){
        sessions.forEach((s) -> {
            try {
                if (s.isOpen() && !s.getId().equals(current.getId()))
                    s.sendMessage(new TextMessage(message));
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
    }

    ... → Other Methods
}
```

2

1

3

Figura 4: Implementação das Notificações.

5. Considerações Finais

Diante da necessidade construir componentes de *software* para o GeneMaisLab que possam ser escaláveis em tempo de execução, foi implementada a partir de uma arquitetura cliente/servidor, um conjunto de APIs RESTful *stateless*. Com base nessa implementação, as aplicações do sistema acessam seus recursos em um servidor a partir do protocolo HTTP. Apesar dessa abordagem de implementação se mostrar adequada às características do sistema, houve a necessidade de desenvolver um mecanismo de sincronização e gerenciamento de dados trocados entre as entidades do sistema.

Neste artigo foram apresentadas as decisões de projeto em relação a coordenação do fluxo de dados entre os clientes do GeneMaisLab. Para isso, foi analisado alguns requisitos do sistema, como a necessidade de garantir a consistência de dados entre um conjunto de usuários ativos. Além disso, foi apresentada uma análise comparativa que levou a escolha do protocolo WebSocket como uma alternativa para lidar com as questões de sincronização em tempo real. Diante disso, foi apresentada a modificação realizada no sistema a partir da implementação de um novo componente que notifica as aplicações clientes sempre que uma nova atualização precisa ser realizada.

Em suma, a abordagem utilizada na implementação do componente de notificação do GeneMaisLab pode ser reproduzida em outros sistemas que utilizam serviços a partir de APIs RESTful *stateless* e que dependem de algum tipo sincronização em tempo real entre os usuários dessas APIs. Algumas escolhas como, o uso do protocolo e das APIs de WebSocket, podem propiciar um ganho de desempenho nos servidores a partir da redução do processamento e da latência proveniente da diminuição do número de requisições.

Referências

Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, (3):81–84.

Fielding, R. (2000). Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85.

Germano, E., Oliveira, H., Narciso, M. G., and Santos, E. H. (2016). Gene+: uma solução computacional distribuída para gerenciar ensaios de genotipagem e marcadores moleculares. *Comunicado Técnico Embrapa Informática Agropecuária*.

Jiang, F.-y. and Duan, H.-c. (2012). Application research of websocket technology on web tree component. In *Information Technology in Medicine and Education (ITME), 2012 International Symposium on*, volume 2, pages 889–892. IEEE.

Kaur, A. and Mann, K. S. (2010). Component based software engineering. *International Journal of Computer Applications*, 2(1):105–108.

Maia, A. d. O. and Silva, D. A. (2017). Proposal to use of the websocket protocol for web device control. In *Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web*, pages 253–260. ACM.

Sparkes, D., Schmidlin, K., and Hsu, M. (2016). Virtual learning environment for entrepreneurship: a conceptual model.

Wirasingha, T. and Dissanayake, N. (2016). A survey of websocket development techniques and technologies.

Zhang, L. and Shen, X. (2013). Research and development of real-time monitoring system based on websocket technology. In *Mechatronic Sciences, Electric Engineering and Computer (MEC), Proc. 2013 International Conf. on*, pages 1955–1958. IEEE.