UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

João Ricardo Mattos e Silva

ESTUDO E AVALIAÇÃO DE IMPLEMENTAÇÃO DA TECNOLOGIA WEBSOCKET

Florianópolis.

2013

João Ricardo Mattos e Silva

ESTUDO E AVALIAÇÃO DE IMPLEMENTAÇÃO DA TECNOLOGIA WEBSOCKET

Trabalho de Conclusão de Curso submetido ao Curso de Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Ricardo Pereira e Silva

Florianópolis.

2013

João Ricardo Mattos e Silva

ESTUDO E AVALIAÇÃO DE IMPLEMENTAÇÃO DA TECNOLOGIA WEBSOCKET

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Curso de Ciências da Computação.

Florianópolis, 1 de outubro de 2013.

_____

Prof. Dr. Vitório Bruno Mazzola
Coordenador do Curso

Banca Examinadora:

_____

Prof. Dr. Ricardo Pereira e Silva
Orientador

_____

Prof. Dr. Frank Augusto Siqueira

_____

Prof. Dr. Mário Antônio Ribeiro Dantas

_____

M. Sc. Roberto Silvino da Cunha

AGRADECIMENTOS

À minha família, por todo o suporte e por deixar o caminho para graduação, algo natural na minha vida.

Ao meu orientador, por me mostrar sempre o melhor caminho possível.

À minha namorada, pelo apoio e carinho constantes, e por toda a formatação deste trabalho.

À Universidade Federal de Santa Catarina e ao curso de Ciência da Computação, por toda a experiência da graduação, e por não me expulsarem mesmo depois de tanto tempo.

RESUMO

A Internet foi criada com um propósito muito mais básico do que a forma como é utilizada atualmente. O protocolo HTTP, utilizado pela World Wide Web, tem uma funcionalidade muito simples, uma resposta para uma pergunta, resumindo de modo grosseiro. As páginas web evoluíram de documentos estáticos, para grandes portais e aplicações web, com funcionalidades das mais variadas.

Atualmente as páginas web vão muito além da entrega de documentos, processando requisições, e, muitas vezes, entregando resultados assincronamente. Tais funcionalidades ainda operam sobre o protocolo HTTP, utilizando recursos extras para criar sessões, e manter a impressão de uma conexão permanente.

O HTML5, a quinta versão da linguagem de marcação utilizada pelo HTTP, trouxe, entre muitas outras funcionalidades, um novo protocolo, ainda baseado no arquitetura do HTTP, porém com a característica de manter um canal bilateral aberto entre servidor e cliente. Tal protocolo é o Websocket.

Esta tecnologia recente, vem sendo adotada aos poucos por grandes empresas em diferentes funções. Porém, como toda tecnologia, tem o seu custo e condições adequadas para utilização. Neste estudo, foi realizado um trabalho experimental, implementando servidores com a mesma funcionalidade, utilizando HTTP e o Websocket, com a finalidade de compará-los em termos de desempenho, fluxo de dados, custo de processamento e memória, em diferentes taxas de uso. Com o resultado desta análise, foi possível perceber que apesar de oferecer uma comunicação bidirecional, esta nova tecnologia apresenta um custo computacional mais elevado que a tecnologia tradicional, em situações com pouca troca de mensagens. Por outro lado, ela oferece uma maior qualidade para a comunicação em aplicações de tempo real.

Palavras Chave: Websocket, HTTP, Web, Internet, Servidor

ABSCTRACT

The internet was created with a purpose simpler than that it is used for nowadays. The HTTP protocol, used by the World Wide Web, has a very simple functionality, an asswer to a question, briefly summarizing. The web pages have evolved from static documents to great web portals and web applications, with a variety of features.
Currently, the web application go far beyond than just delivering static documents, processing requests, and delivering the result asynchronously. Such functionality still operates on the HTTP protocol, using extra resources to create sessions and maintain the impression of a permanent connection.
The HTML5, the fifth version of the markup language used for HTTP, brought, among other features, a new protocol, still based on HTTP, but with the characteristic of keeping a bilateral communication channel between server and client. Such protocol, is the Websocket.
This recent technology, is gradually being adopted by great companies in different functions. However, as all technology, it has its own cost and adequate scenarios of using. In this work, a experiment was made, implementing servers with the same functionality, using HTTP and websockets, comparing them in terms of performance, data flow, memory and processing cost, in different rates of use. With the result of this analysis, it was possible to find out that despites offering a bidirectional communication channel, the new technology presents a higher computational cost in situations with a low message exchange rate. On other side it provides a better quality communication on real time applications.

Keywords: Websocket, HTTP, Web, Internet, Server.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ARPANET	Advanced Research Projects Agency Network
CSS	Cascading Style Sheets
DNS	Domain Name Server
DoD	Department of Defence
DOM	Document Object Model
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IEFT	Internet Engeneering Task Force
IP	Internet Protocol
OSI	Open Systems Interconnection
TCP	Transmission Control Protocol
XML	eXtensible Markup Language
WWW	World Wide Web
W3C	World Wide Web Consortiun

SUMÁRIO

1 INTRODUÇÃO

Como outros grandes avanços tecnológicos, incluindo o próprio computador, a internet foi criada com fins bélicos. Um sistema de distribuição de informações descentralizado, que continuaria disponível, mesmo que uma de suas bases fosse destruída. Foi uma solução perfeita para o compartilhamento de informações no período da guerra fria.
Com o fim do período de tensão, esta rede de comunicação, chamada ARPANET, foi aberta pelo governo dos EUA, para que pesquisadores de universidades também pudessem conectar seus computadores a esta rede. E logo aos seus alunos, e amigos de seus alunos. Em um certo momento, o governo perdeu o controle sobre a quantidade de maquinas conectadas à rede, e decidiu, então, dividir a rede militar desta nova rede de comunicação, com fins acadêmicos. Ao longo do tempo, com investimento em infraestrutura, e a popularização dos computadores pessoais, a internet tomou proporções como meio de comunicação, sem precedentes na história. Atualmente é produzido por dia a mesma quantidade de informação que se tem armazenada desde o início da escrita, até o ano de 1925.

Para essa troca de dados ser possível, foi preciso definir um padrão de comunicação, de modo que todas as máquinas conectadas a rede conversassem na mesma lingua. O protocolo HTTP (Hyper Text Transfer Protocol) foi o padrão criado para este propósito. Um protocolo de comunicação que deu a base para a World Wide Web que conhecemos hoje.
Todos os computadores conectados à internet, tem seu endereço próprio, muito semelhante a um CEP ou número de telefone, o IP. Este endereço, é a base para qualquer requisição web. Mais adiante veremos com muito mais detalhes esse funcionamento, mas, simplificando grosseiramente, o uso básico da internet nada mais é do que uma requisição HTTP a um endereço IP, e uma resposta, como uma imagem, por exemplo. No início da web, nada mais do que isto era necessário. Resultados de pesquisa, textos científicos, e posteriormente, imagens de gatinhos, eram facilmente compartilhados.

Com o amadurecimento da tecnologia, e avanço da capacidade computacional como um todo, a web foi ganhando diversas outras funcionalidades. Grandes portais de notícias, redes sociais, lojas online, transações bancárias, entre outros, apenas mencionando os que funcionam utilizando o protocolo HTTP. Muitos outros protocolos, que fogem do escopo deste trabalho, foram criados para a transmissão de tipos específicos de dados, como, por exemplo, áudio e vídeo.

Outra característica dos serviços na internet é a sua disponibilidade, isto é, está respondendo sempre, não importando a quantidade de requisições que está recebendo. Tal particularidade só é possível com uma computação distribuída, ou seja, vários computadores atuando como apenas um único serviço, para que a demanda computacional para as requisições seja sempre saciada. Utilizar apenas um único computador, com uma capacidade computacional altíssima, além de ser mais caro, terá um limite máximo de expansividade. Esse método de implementação de serviço, utilizando vários computadores simultaneamente, e requer alguns paradigmas de implementação que diferem do tradicional, os quais são abordados neste trabalho.

Mesmo com a limitação do protocolo HTTP, de apenas uma pergunta para uma resposta, os serviços web atualmente mantêm um sistema de sessão aberto com o cliente, como após um login, por exemplo. Para isso, são utilizados recursos adicionais, transmitindo, a cada requisição posterior à identificação, uma chave de sessão, que é vinculada com seu endereço IP no servidor. Fora isso, muitos portais web têm uma atualização de dados constante, dando a impressão ao usuário de uma conexão permanente.

Atualmente, o método mais utilizado para manter esta simulação de um ambiente em tempo real, é o chamado ‘polling’. Será explicado adiante com mais detalhes, mas seu funcionamento é básico, consiste em uma requisição a cada certa quantidade de tempo ao servidor, verificando atualizações. Apesar de funcionar muito bem, este método possui algumas desvantagens muito claras, como a quantidade de recurso que é desperdiçado ao processar cada requisição inútil, e os dados dos cabeçalhos dos protocolos, trafegados em todas as mensagens. Outra desvantagem, para certos tipos de aplicação, é a falta do desempenho em velocidade, no caso de uma troca muito grande de mensagens, como por exemplo, em jogos online com múltiplos jogadores.

Dentro de uma das novas funcionalidades disponibilizadas pelo HTML5, o Websocket foi criado com o intuito de sanar tais desvantagens, simplesmente criando um canal de comunicação bilateral com o servidor. Uma conexão permanente com um servidor não é novidade nenhuma. O grande avanço está nesse recurso ser disponível diretamente pelo HTML, que é a principal caixa de ferramentas de qualquer desenvolvedor web. Até então, para criar este tipo de conexão na web, eram utilizados apenas recursos proprietários, como Flash e Java, tendo a necessidade da instalação de *plugins* de terceiros no computador do usuário, quando possível. Como HTML é o padrão da *World Wide Web*, o Websocket tende a ser aceito em qualquer plataforma existente, aumentando a portabilidade do serviço.

A meta deste trabalho é observar o custo real de implementação do Websocket, criando benchmarks relacionando a quantidade de dados trafegados, custo de processamento e memória, com diferentes taxas de requisições e mensagens por segundo. Espera-se, assim, observar um resultado que demonstre quais classes de aplicações tendem a ganhar performance com a utilização de Websockets, em relação às tecnologias mais utilizadas atualmente.

1.1 MOTIVAÇÃO

A motivação deste trabalho surgiu da grande popularidade que o HTML5 e o Websocket, em particular, tem tomado nos últimos anos na internet. Prometendo otimizar serviços em tempo real e possibilitando o desenvolvimento de novos tipos de aplicações diretamente no navegador, muito tem sido comentado pela web. Porém, esta tecnologia tem um custo de implementação possivelmente diferente das mais utilizadas atualmente.

Este trabalho visa esclarecer este custo, e seu resultado em desempenho, como também apresentar metodologias utilizadas em implementações de servidores voltados para grande demanda de requisições.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo deste trabalho é realizar um estudo sobre a atual metodologia de implementação de servidores web de alta demanda, e analisar o impacto que o protocolo Websocket tem na substituição dos métodos mais utilizados para a criação de sistemas em tempo real.

1.2.2 Objetivos Específicos

- Estudo teórico sobre os protocolos utilizados pela World Wide Web.
- Estudo teórico sobre o protocolo Websocket.
- Estudar conceitos e metodologias utilizados para desenvolver servidores web de alta demanda.
- Estudar ferramentas utilizadas para desenvolvimento de aplicações web.
- Implementar servidores de teste com funcionalidade idêntica, utilizando diferentes tecnologias.
- Testar servidores implementados em diferentes cenários de taxa de usuários simultâneos, quantidade de mensagens por segundo e tamanho da mensagem.
- Observar os resultados, analisando tempo de resposta, custo para o servidor em termos de processamento e memória, dados trafegados e taxa de falha.
- Analisar os resultados e obter uma faixa de aplicações web que têm a ganhar com a implementação de Websockets.

1.3 METODOLOGIA

Este trabalho está organizado em 6 capítulos. O primeiro capítulo apresenta a introdução e a motivação para a realização deste estudo. O capítulo seguinte consiste em um estudo do estado atual da internet e seus protocolos. Também é apresentado o protocolo Websocket, o principal foco de estudo deste trabalho.

O capítulo 3 e 4 apresentam paradigmas e tecnologias utilizadas para o desenvolvimento de servidores web, com foco principalmente em escalabilidade. Estas técnicas foram utilizadas para o desenvolvimento de aplicações de teste, para a realização de comparações entre as tecnologias apresentadas no capítulo anterior.

O capítulo 5 consiste na demonstração dos resultados observados das implementações de teste. São apresentados gráficos que demonstram a diferença na utilização

das tecnologias implementadas. O capítulo final apresenta as conclusões que foram obtidas com este estudo, e na avaliação dos resultados dos testes realizados.

2 WEB E SEUS PROTOCOLOS

O presente capítulo apresenta um breve histórico da Web e descreve suas principais características e padrões. Os protocolos HTTP e Websocket - sendo este último o objeto de investigação do presente trabalho - são descritos, como também os protocolos das camadas inferiores, nas quais eles se suportam. Ademais, é apresentada a linguagem HTML e Javascript, com especial ênfase aos novos recursos apresentados com o HTML 5.

2.1 O INÍCIO

Como foi brevemente mencionado na introdução deste trabalho, a internet foi fruto da necessidade de uma distribuição descentralizada de informações. Não será abordado muito mais sobre a grande história da internet neste trabalho. Por ser muito extensa e cheia de detalhes relevantes, um resumo deixaria muitos fatos importantes de fora. Contudo, é importante apresentar seu início, principalmente do ponto de vista tecnológico.

Antes da utilização do protocolo HTTP, os protocolos TCP e IP já tinham sido desenvolvidos para a ARPANET. A base para a troca de dados que conhecemos hoje foi o modelo de rede TCP/IP, que determina as camadas e protocolos utilizados para a transmissão de pacotes pela internet.

> A ARPANET era uma rede de pesquisa patrocinada pelo Departamento de Defesa dos Estados Unidos (DoD). Pouco a pouco, centenas de universidades e repartições públicas foram conectadas, usando linhas telefônicas dedicadas. Quando foram criadas as redes de rádio e satélite, começaram a surgir problemas com os protocolos existentes, o que forçou a criação de uma nova arquitetura de referência. Desse modo, a habilidade para conectar várias redes de maneira uniforme foi um dos principais objetivos de projeto, desde o início. Mais tarde, essa arquitetura ficou conhecida como Modelo de Referência TCP/IP (TANENBAUM, Andrew S. 2011)

A seguir, são tratados com um pouco mais de detalhamento as camadas de transporte e internet (TCP/IP) e a camada final de aplicação, o HTTP.

2.2 O MODELO CLIENTE SERVIDOR

A característica deste modelo, está na relação entre programas em uma aplicação. Os clientes fazem requisições através de mensagens para o servidor, que são as máquinas que processam este pedido, e enviam resposta de volta. Este é o modelo predominante utilizado pela World Wide Web. Um usuário, usando o seu navegador, envia uma requisição, encapsulada em um protocolo HTTP, até o servidor que responde com os dados processados.

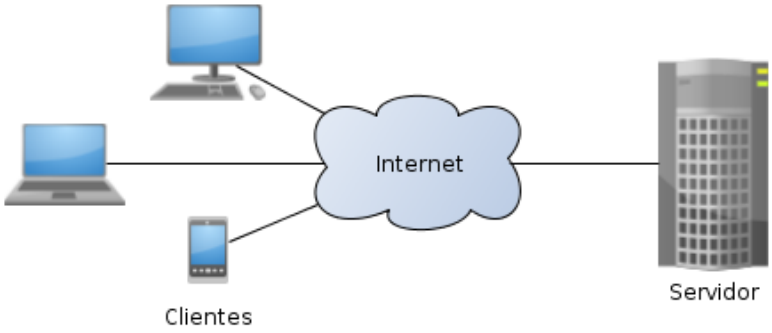

Figura 1 - Modelo cliente-servidor, utilizando a internet
(Wikipédia - <http://pt.wikipedia.org/wiki/Cliente-servidor>)

Uma vantagem desse modelo é o desacoplamento de funcionalidades do cliente e servidor, sendo possível desenvolver as partes de forma independente, padronizando apenas a comunicação dos dados. Outra vantagem clara está na possibilidade de uma funcionalidade ser compartilhada por diversos usuários, que utilizam o mesmo servidor. As tecnologias usadas pela web, para este modelo, são apresentados nas sessões a seguir.

2.3 AS CAMADAS DA WEB

Como vimos, a internet é uma rede mundial de computadores. O sistema de camadas de rede é utilizado para que cada nível seja desenvolvido abstraindo os outros. O protocolo HTTP se encontra no nível mais abstrato, a camada de aplicação, onde todo o transporte de dados já foi tratado pelas camadas inferiores.

Diferente do modelo genérico de 7 camadas de rede do modelo OSI, as camadas utilizadas pelo modelo TCP/IP, utilizado pela internet, são separadas em 4 níveis. A Figura 2 demonstra a equivalência com o modelo OSI.

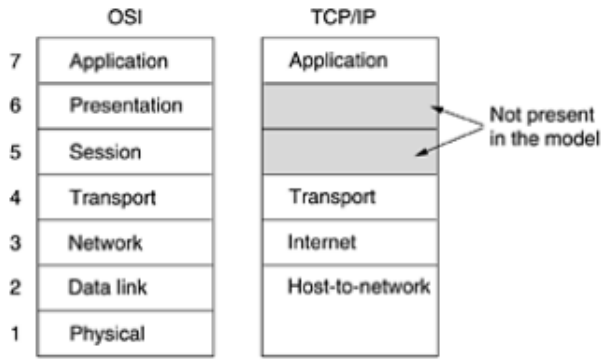

Figura 2 - Camadas OSI em comparação com TCP/IP
(TANENBAUM, Andrew S. 2011)

2.4 IP

A sigla IP vem do Inglês, *Internet Protocol*, Protocolo de Internet. A função deste é a transmissão de pacotes, por uma rede de computadores, de uma origem ao seu destino. O encaminhamento de informações é feito empacotando os dados em um datagrama IP que contém, entre outras informações, o endereço IP do destino e da origem.

O endereço é um campo de 32 bits que funciona como um número de telefone, por exemplo. Este campo de 32 bits, se refere ao modelo IPv4, o qual é utilizado atualmente. Por ser um número binário, a capacidade endereços possíveis é de 2^{32}, um total de 4294967296. Devido ao grande crescimento da internet, e da quantidade de computadores e dispositivos conectados a ela, o modelo está ficando saturado, de forma que o IPv6, com um campo de endereçamento de 128 bits, está sendo usado como substituto, e deverá se tornar o padrão mais utilizado em pouco tempo.

+	0 - 3	4 - 7	8 - 15	16 - 18	19 - 31
0	Versão	Tamanho do cabeçalho	*Tipo de Serviço* (ToS) (agora DiffServ e ECN)	Comprimento (pacote)	
32	Identificador			*Flags*	*Offset*
64	*Tempo de Vida* (TTL)		Protocolo	*Checksum*	
96	Endereço origem				
128	Endereço destino				
160	Opções				
192	*Dados*				

Figura 3 - Datagrama do protocolo IP
(Wikipédia - <http://pt.wikipedia.org/wiki/Protocolo_de_Internet>)

Outra responsabilidade do protocolo IP é o roteamento de pacotes. A internet é formada por diversos nodos interconectados, que ao receber um pacote de dados, tem que encaminha-lo ao destino correto, de preferência, pela caminho mais eficiente possível.

O transporte de dados é feito por este protocolo, porém, não existe garantia na entrega e na integridade dos pacotes transportados. Isto será feito pela próxima camada, a de transporte, pelo protocolo TCP.

2.5 TCP

> O TCP (Transmission Control Protocol — protocolo de controle de transmissão), é um protocolo orientado a conexões confiável que permite a entrega sem erros de um fluxo de bytes originário de uma determinada máquina em qualquer computador da inter-rede. Esse protocolo fragmenta o fluxo de bytes de entrada em mensagens discretas e passa cada uma delas para a camada inter-redes. No destino, o processo TCP receptor volta a montar as mensagens recebidas no fluxo de saída. O TCP também cuida do controle de fluxo, impedindo que um transmissor rápido sobrecarregue um receptor lento com um volume de mensagens maior do que ele pode manipular (TANENBAUM, Andrew S. 2011).

O protocolo TCP utiliza um cabeçalho com informações bem definidas, para todas as mensagens trocadas. A Figura 4 demonstra os campos e as informações deste cabeçalho.

Figura 4 - Cabeçalho do TCP
(Wikipédia - <http://pt.wikipedia.org/wiki/Transmission_Control_Protocol>)

O TCP tem três fases determinadas durante uma conexão: Estabelecimento, transferência e término da ligação.

O estabelecimento da ligação, consiste na troca de sinais entre cliente e servidor, como está mostrado na Figura 5. O servidor abre um *socket*, em modo passivo, esperando conexões em uma determinada porta. O cliente envia um pacote TCP para este servidor, com a flag SYN ativada. O servidor então responde a esta requisição com uma flag SYN e ACK ativadas, sinalizando que a requisição foi tratada. O cliente responde com uma flag ACK, demonstrando que foi recebido o sinal de conexão.

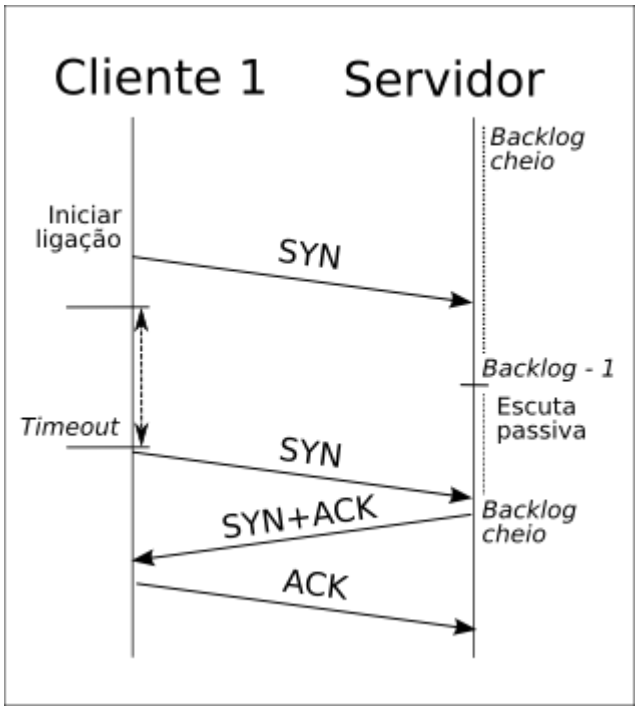

Figura 5 - Estabelecimento de uma conexão TCP entre cliente e servidor (Wikipédia - http://pt.wikipedia.org/wiki/Transmission_Control_Protocol)

Após a conexão ter sido feita, pode ser iniciada a transmissão dos dados. Durante a fase de transferência de dados, o TCP garante a integridade dos pacotes, utilizando o campo *Checksum* do seu cabeçalho. Uma função de checksum, ou soma de verificação, consiste na aplicação de uma função matemática em um bloco de dados, a fim de extrair um número único daquela sequência de bits. A função checksum do TCP está definida no RFC 793, a especificação oficial do protocolo. Aplicando esta função no recebimento, e comparando o resultado obtido com o transmitido no cabeçalho, tem-se a garantia da integridade do pacote.

O protocolo TCP também reordena os pacotes na sequência correta, caso algum atraso na transmissão cause um embaralhamento da informação.

O término da ligação consiste em duas partes de finalização, uma para cada lado da conexão. Um dos lados envia uma mensagem com a flag FIN, e aguarda uma resposta ACK para confirmação. Após isso o outro lado envia a mesma mensagem com a flag FIN, e a conexão é finalizada com a última resposta ACK.

2.6 HTTP

O protocolo de transferência utilizado em toda a World Wide Web é o HTTP (HyperText Transfer Protocol). Ele especifica a sintaxe das mensagens que os clientes podem enviar aos servidores, e das respostas eles receberão.

> Do ponto de vista dos usuários, a Web é uma vasta coleção mundial de documentos, geralmente chamados páginas da Web, ou apenas páginas. Cada página pode conter links (vínculos) para outras páginas em qualquer lugar do mundo. Os usuários podem seguir um link (por exemplo, dando um clique sobre ele), que os levará até a página indicada. Esse processo pode ser repetido indefinidamente. A ideia de fazer uma página apontar para outra, agora chamada hipertexto, foi criada por um professor de engenharia elétrica do MIT, Vannevar Bush, em 1945, bem antes da criação da internet. (TANENBAUM, Andrew S. 2011).

Como foi visto até agora, as camadas inferiores assumem a responsabilidade de conexão, transporte e integridade de dados. O protocolo HTTP dará uso prático a este fluxo de dados, com a transferência do hipertexto. Mais à frente, veremos como o formato do hipertexto compartilhado na web foi também padronizado com o HTML (HyperText Markup Language, Linguagem de Marcação de Hipertexto).

Considerado a fundação de toda a World Wide Web, ele tem sido usada desde 1990. A sua primeira versão, o HTTP/0.9, era um simples protocolo com um único método: GET, pegar, em Inglês. Através deste único método de requisição, eram transferidos dados no formato de texto ASCII.

A segunda versão do protocolo, o HTTP/1.0, supriu a necessidade da transferência de outros tipos de dado, além de textos. Nesta versão, os dados transferidos seguiam o tipo MIME (Extensões Multi função para Mensagens de internet). Utilizado também no protocolo de email SMTP, o MIME dispõe mecanismos para a interpretação de outros tipos de dados com diferentes codificações do padrão ASCII, como imagens, sons, filmes e outros arquivos de computador. Nesta versão, também foram implementados os métodos HEAD e POST. O

método POST é utilizado até os dias atuais, para a transmissão de informações do cliente ao servidor, encapsulando os dados dentro da requisição, e não na url, como no método GET.

Na sua última versão, o HTTP/1.1, que é utilizado atualmente, foram implementadas outras funcionalidades. Novos métodos foram adicionados, como PUT, DELETE, TRACE, OPTIONS e CONNECT. Além disso, foram adicionadas funcionalidades de conexão permanente com o header "Keep-Alive". Apesar de ser permanente, este tipo de conexão difere do Websocket pelo fato de que, apesar da conexão TCP/IP não ter sido encerrada, ele ainda funciona apenas com uma requisição do cliente, e uma resposta do servidor, com os mesmos headers.

Uma requisição HTTP GET bem sucedida, possui os seguintes cabeçalhos:

```
GET/index.html HTTP/1.1
Host: www.exemplo.com
```

E como resposta:

```
HTTP/1.1 200 OK
Date: Fri, 24 May 2013 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

Na resposta, a página HTML, ou algum outro formato de arquivo, está em sequência.

Analisaremos agora uma requisição feita por um usuário na página web do Google, por exemplo. Primeiramente ele entraria com o seguinte endereço na barra de seu navegador: http://www.google.com.br/. O http:// se refere ao protocolo, o www.google.com.br o host, ou endereço do servidor, que será convertido para um endereço IP por um DNS. Subentendido nesta requisição, estão a porta 80, que é o padrão para o protocolo HTTP, e o recurso que está na raiz, definido pela '/' sem acréscimos no final do endereço.

Utilizando o programa para Unix curl, podemos observar todo o resultado:

```
$ Curl -v http://www.ufsc.br
* About to connect () to www.google.com.br port 80 (#0)
* Trying 150.162.2.10...
* connected
* Connected to www.google.com.br (150.162.2.10) port 80 (#0)
> GET / HTTP/1.1
> Host: www.google.com.br
```

Podemos observar nesta requisição, inicialmente uma conexão TCP/IP sendo criada em um endereço IP fornecido pelo DNS, e a seguir o envio do header do HTTP, com a função GET no parâmetro /, seguindo o protocolo HTTP/1.1. Explicando rapidamente, o DNS (Sistema de Domínio de Nomes) é um serviço na web, que transforma nomes, em endereços IPs. É um serviço normalmente pago, que mantém uma base de dados atualizada, com tabelas relacionando domínios e IPs.

Como resposta desta requisição, temos:

```
< HTTP/1.1 200 OK
< Date: Sat, 29 Jun 2013 23:31:21 GMT
< Expires: -1
< Content-Type: text/html; charset=ISO-8859-1
< Set-Cookie:
PREF=ID=92298d1f72097732:FF=0:TM=1372548681:LM=1372548681:S=rwVV1Tc7HjrcGHj
U; expires=Mon, 29-Jun-2015 23:31:21 GMT; path=/; domain=.google.com.br
< Set-Cookie: NID=67RK618vU5nCONZCjwth6ndKE52HHbIK; expires=Sun, 29-Dec-2013
23:31:21 GMT; path=/; domain=.google.com.br; HttpOnly
<
<!doctype html><html itemscope="itemscope"
itemtype="http://schema.org/WebPage"><head><meta itemprop="image"
content="/images/google_favicon_128.png"><title>Google</title>...
```

Na primeira linha temos o protocolo HTTP/1.1 enviando o sinal 200, que significa que tudo correu bem. Alguns outros cabeçalhos também compõem esta resposta, como o *Content-Type*, especificando a resposta sendo do tipo texto/html, e sua codificação. O *Set-Cookie* é uma função do HTTP/1.1 para um sistema de dados chave/valor no computador do cliente, que será transferido posteriormente em todas as outras requisições neste mesmo domínio. O cookie será explicado com mais detalhes adiante.

Após, uma linha vazia, identificando o final do cabeçalho, está o conteúdo da resposta, uma página HTML.

Esta é basicamente a principal funcionalidade do protocolo HTTP. Outros métodos, como *POST*, *PUT*, *DELETE*, servem para o tratamento de dados em serviços web. Estes métodos têm parâmetros adicionais no cabeçalho, indicando variáveis a serem alteradas.

Por ser um protocolo que não guarda nenhum estado, ou seja, todas as requisições são independentes uma das outras, outros recursos são utilizados para a criar um sistema de sessões. O cookie, mencionado anteriormente, é a base para isto. Em uma resposta, utilizando no header "Set-Cookie:", um servidor irá armazenar um valor em uma variável no computador do cliente, que será enviado em todas as próximas requisições.

Uma maneira de utilizar o cookie seria, após uma requisição de *login*, colocar na resposta um "Set-Cookie:user=joao". Com isto, em todas as próximas requisições, o serviço web saberá qual usuário está realizando esta requisição. Está é uma maneira muito ingênua de criar uma sessão, por ser facilmente burlada por qualquer pessoa que consiga alterar o header de uma requisição HTTP, conseguindo assim se passar por outro usuário.

Segurança é um estudo à parte em qualquer área da computação, mas para este trabalho, basta dizer que isto atualmente é bem tratado pelos serviços web, normalmente criptografando a sessão, e associando uma chave ao endereço do usuário no lado do servidor, de forma que não é tão simples se passar por outro usuário.

2.7 WEBSOCKET

> Historicamente, desenvolver aplicações web que necessitam uma comunicação bidirecional entre cliente e servidor, implicava em um abuso do protocolo HTTP, para verificação de updates no servidor, enquanto transmitia novas informações em requisições HTTP distintas. Uma solução mais simples seria usar uma única conexão TCP para o tráfego em ambas as direções. O protocolo Websocket é projetado para criar essa comunicação, utilizando os benefícios da infraestrutura já existente para o protocolo HTTP (FETTE, I.; MELNIKOV, A. RFC6455 - THE WEBSOCKET PROTOCOL.).

O protocolo Websocket é uma tecnologia que disponibiliza um canal de comunicação bidirecional (full-duplex) entre cliente e servidor, empregando uma única conexão TCP/IP. Este protocolo foi padronizado pela IETF[1] em 2011. Esta padronização se encontra no

[1] **IETF** - Internet Engineering Task Force <**http://www.ietf.org/>.** Uma ampla comunidade aberta com a finalidade da evolução da arquitetura da internet.

RFC[2]6455, que está em estado de proposta, ainda não definido como um padrão da internet. Apesar de não ser um padrão oficial, todos os navegadores recentes, já implementam o protocolo, permitindo seu uso sem maiores dificuldades. Com a capacidade dos navegadores de utilizar o Websocket, grandes empresas da web, como *Facebook, Google, Stack Exchange*, entre outras, já estão utilizando em alguns de seus serviços.

Um exemplo simples da utilização do Websocket, pode ser testado em http://www.websocket.org/echo.html. Este exemplo trata de um servidor de eco, talvez o primeiro servidor que qualquer programador desenvolve. A funcionalidade consiste em conectar, enviar uma mensagem, e receber a mesma na resposta do servidor.

Para demonstrar a parte técnica por trás deste eco, foi utilizado o programa WireShark, que monitora todos os pacotes de todos os protocolos trocados por uma interface de rede. O resultado do monitoramento de uma conexão, envio e recebimento de um texto, no serviço indicado acima, pode ser observado na Tabela 1.

Tabela 1 - Resultado do monitoramento de uma conexão Websocket, envio e recebimento de uma mensagem, utilizando o programa WireShark.

No.	Time	Source	Destination	Protocol	Length	Info
51	5.190422000	192.168.0.110	174.129.224.73	TCP	74	46410 > http [SYN] Seq=0 Win=14600
52	5.513094000	174.129.224.73	192.168.0.110	TCP	74	http > 46410 [SYN, ACK] Seq=0 Ack=1
53	5.513155000	192.168.0.110	174.129.224.73	TCP	66	46410 > http [ACK] Seq=1 Ack=1 Win=
54	5.514925000	192.168.0.110	174.129.224.73	HTTP	705	GET /?encoding=text HTTP/1.1
56	6.502714000	174.129.224.73	192.168.0.110	HTTP	608	HTTP/1.1 101 Web Socket Protocol Ha
57	6.502769000	192.168.0.110	174.129.224.73	TCP	66	46410 > http [ACK] Seq=640 Ack=543
60	8.204004000	192.168.0.110	174.129.224.73	WebSocke	100	WebSocket Text [FIN] [MASKED]
61	8.396505000	174.129.224.73	192.168.0.110	WebSocke	96	WebSocket Text [FIN]
62	8.396551000	192.168.0.110	174.129.224.73	TCP	66	46410 > http [ACK] Seq=674 Ack=573

Podemos ver nas 3 primeiras mensagens (#51-53), o handshake da conexão TCP/IP, explicado anteriormente. Seguinte ao estabelecimento da conexão, existe uma requisição HTTP (#54) pedindo o estabelecimento de uma conexão Websocket, que foi respondida na mensagem seguinte (#56). Iremos verificar os headers destas requisições HTTP a seguir.

Após uma mensagem TCP (#57), confirmando o recebimento da mensagem HTTP anterior, podemos ver a mensagem sendo enviada ao servidor (#60), utilizando o protocolo Websocket, e, em seguida a resposta (#61), com o mesmo conteúdo, afinal, é um servidor de eco.

Analisando superficialmente este resultado, já podemos observar algo interessante: o tamanho do pacote, em bytes, das mensagens trocadas. Repare que mesmo uma mensagem

[2] **RFC** - Request For Comments, é um documento que descreve os padrões de cada protocolo utilizado na internet, previamente de se tornar um padrão. Todos os protocolos descritos neste trabalho, tem seu RFC publicado em <**http://tools.ietf.org**>

HTTP (#54), sem conteúdo nenhum, apenas com uma requisição GET, tem o tamanho de 705 bytes, enquanto as mensagens do protocolo Websocket tem um tamanho muito reduzido. Isto se deve ao cabeçalho mínimo que o protocolo utiliza, após uma conexão ter sido formada.

Examinando o cabeçalho da primeira requisição HTTP podemos observar detalhes do início da conexão Websocket:

```
> GET /?encoding=text HTTP/1.1
> Upgrade: websocket
> Connection: Upgrade
> Host: echo.websocket.org
> Origin: http://www.websocket.org
> Sec-WebSocket-Key: 9BTWLAIUIXGRRvBWXMaI1g==
> Sec-WebSocket-Version: 13
> Sec-WebSocket-Extensions: x-webkit-deflate-frame
> User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like > Gecko) Chrome/27.0.1453.110 Safari/537.36
```

Os cabeçalhos “Upgrade: Websocket” e “Connection: Upgrade” indicam ao servidor, o pedido de transformar esta conexão HTTP, em uma conexão Websocket, aplicando a mesma conexão TCP/IP. Os campos “Sec-WebSocket” indicam parâmetros para serem utilizados no handshake, como a versão, e uma chave para ser usada no servidor, para preparar uma resposta. Na resposta do servidor para esta requisição, temos:

```
< HTTP/1.1 101 Web Socket Protocol Handshake
< Upgrade: WebSocket
< Connection: Upgrade
< Sec-WebSocket-Accept: DkJnmWDc7RPF8gvKBwm4tqFWIW4=
< Access-Control-Allow-Origin: http://www.websocket.org
< Access-Control-Allow-Credentials: true
< Access-Control-Allow-Headers: content-type
< Access-Control-Allow-Headers: authorization
< Access-Control-Allow-Headers: x-websocket-extensions
< Access-Control-Allow-Headers: x-websocket-version
< Access-Control-Allow-Headers: x-websocket-protocol
```

A primeira linha, indica uma conexão bem sucedida. A partir de agora, o cliente e o servidor podem trocar mensagens com o padrão Websocket, não mais HTTP.

O datagrama do cabeçalho Websocket pode ser observado na Figura 6.

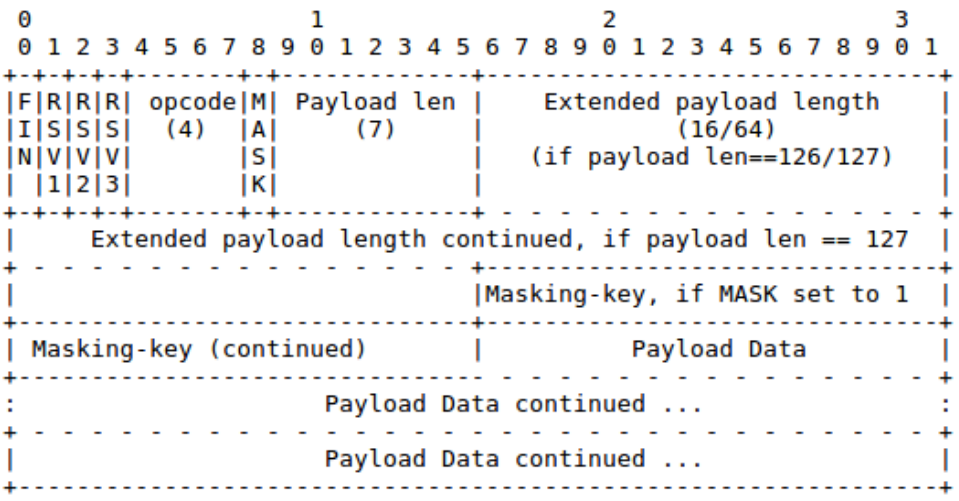

Figura 6 - Datagrama de um bloco de mensagem Websocket.
(FETTE, I.; MELNIKOV, A. *RFC6455 - The WebSocket Protocol*.)

Por ter um cabeçalho binário, não faria sentido demonstrar o pacote trocado aqui, como foi feito com o HTTP. A explicação completa para cada parâmetro demonstrado na Figura 6, pode ser encontrada no RFC6455. Descreveremos alguns destes parametros, que são relevantes para este trabalho. O primeiro bit da mensagem, forma a flag FIN, que aponta se este pacote indica o final de uma mensagem. No caso de mensagens maiores, onde o limite é definido na implementação do cliente/servidor, estas são particionadas e enviadas em vários blocos. Caso seja menor que este limite, o bloco único terá a flag FIN ativada.

O campo opcode, indica como deverá ser a interpretação do payload. Os principais valores utilizados são 0, para um frame particionado, 1 denota um texto e 2 um frame binário. A flag MASK apontam se o payload está ou não mascarado, com a chave no campo Masking-key. Payload-len é um campo, de tamanho elástico, onde o tamanho do payload é mostrado.

2.8 COMPARAÇÕES ENTRE HTTP E WEBSOCKET

Foi visto até agora o funcionamento do protocolo HTTP e Websocket. Mesmo sem nenhuma implementação, já podemos tirar algumas conclusões comparando alguns parametros destes protocolos.

Digamos que um servidor tenha 10 mil usuários, fazendo pooling a cada segundo requisitando por atualizações. Cada requisição possui um header HTTP de tamanho 700 bytes, sendo esta uma aproximação bem realista, fazendo uma simples média observando websites populares. Este número pode crescer muito, caso a aplicação em questão tenha uma grande quantidade de cookies para serem trafegados em cada mensagem. A cada segundo teriamos:

10.000 * 700 * 8 = 56.000.000 bits/s

Teríamos um fluxo de dados supérfluos de 53 Mbps. Fazendo o mesmo cálculo, utilizando o tamanho do header do Websocket, de 2bytes, o fluxo dos cabeçalhos tem uma queda para 0.153 Mbps. Uma economia gigantesca em tráfego de dados.

Este é apenas um dos pontos a serem analisados no custo do Websocket. Durante a avaliação da implementação prática dos servidores, será também levado em consideração o custo do estabelecimento de conexão TCP/IP para cada requisição HTTP, como também o custo para o servidor para manter aberta todas as conexões Websocket.

2.9 PROGRAMAÇÃO DO LADO CLIENTE

Foi abordado até agora, o padrão da internet para conexões e a troca de mensagens. Para finalizar este capítulo, são abordados os padrões para a programação no lado do cliente. Serão exibidas a linguagem de marcação HTML e a linguagem de programação Javascript. Ambas são o padrão para apresentação das informações trafegadas, da forma mais humanamente amigável.

2.9.1 HTML

Como mencionado anteriormente, o HyperText Markup Language, ou linguagem de marcação de hipertexto, é o padrão para a montagem de páginas web. Uma página simples, possui a seguinte sintaxe:

```
<html>
        <head>
                <title>Titulo da pagina</title>
        </head>
        <body>
                <p>Ola mundo!</p>
        </body>
</html>
```

Uma página da Web consiste em um cabeçalho e um corpo entre as tags (comandos de formatação) <html> e </html>, embora a maioria dos navegadores não reclame se essas tags não estiverem presentes. O cabeçalho começa e termina com as tags <head> e </head> respectivamente, enquanto com o corpo é delimitado pelas tags <body> e </body>. Os strings entre as tags são chamados diretivas. A maioria das tags de HTML tem esse formato, ou seja, <algo> para marcar o início de alguma coisa e </algo> para marcar seu fim. (TANENBAUM, Andrew S. 2003).

Uma página HTML completa é formada por uma série de *tags* com diferentes diretivas, interpretadas pelo navegador, gerando uma interface gráfica para o usuário, quando renderizado graficamente pelo navegador utilizado. Por ser um padrão, todos os navegadores, independente de plataforma, tendem a gerar o mesmo resultado final.

Também podem fazer parte de uma página HTML, o CSS e o Javascript. O CSS é uma linguagem de estilo, utilizada para personalizar, e definir a apresentação de documentos HTML. O Javascript é uma linguagem de programação interpretada, utilizada em documentos HTML para criar funcionalidades e interagir com o usuário, sem a necessidade de interação com o servidor. Mais adiante, a utilização do Javascript será mostrada com mais detalhes. O CSS não será tratado, pois esse é utilizado principalmente para design, assunto não abordado neste trabalho.

2.9.2 HTML5

A primeira versão do HTML se tornou disponível em 1991 e continha apenas 18 tipos de elementos. Em 1993, com esforço do IETF, foi publicada a primeira proposta para um padrão formal. Apenas em 1995, o IETF criou um grupo de trabalho, focado na criação e padronização do HTML, que deu origem ao primeiro padrão da internet aceito, o HTML 2.0.

Muito se avançou desde então, com adição de um a grande quantidade de tags e a adoção do XHTML como um padrão. O XHTML especifica que um documento HTML seja também aceito nas normas do XML. As normas de XML definem, entre outras, que todas as

tags abertas, tem obrigatoriamente que serem fechadas. Todos os elementos também tem que ter um único elemento pai, no caso do HTML, a tag <html></html>.

O HTML5 é um padrão, ainda no estado de proposta, para substituir o HTML4.1 como o novo padrão oficial da internet. Ele vem sendo desenvolvido pela W3C, o World Wide Web Consortiun, a principal organização para padrões da WWW. Apesar de estar em desenvolvimento, todos os navegadores modernos já implementam a maioria de seus novos recursos.

Pela primeira vez, é possível a reprodução de áudio e vídeo, em diversos formatos, diretamente pela interpretação de um texto HTML pelo navegador, com a utilização das tags <audio> e <video>. Entre outras funcionalidades, estão a implementação de um *canvas*. Um objeto canvas, presente em diversas linguagens de programação, é utilizado para desenhos e animações.

As novas APIs para desenvolvimento gráfico, eliminam a necessidade de plugins de terceiros, como o Adobe Flash, Silverlight da Microsoft entre outros. A Netflix, uma grande empresa de streaming de filmes e séries, já está migrando seu serviço web, para um player totalmente em HTML5, deixando de lado a utilização de plugins de terceiros, e tornando o sistema independente de plataforma.

Outras alterações do HTML estão na utilização de armazenamento local, e recursos para que aplicações possam trabalhar offline. Novas tags também foram adicionadas, e antigas tags ganharam novos recursos, como por exemplo, a utilização de uma tag de <input type=’datetime’>, que renderiza um formulário de inclusão de Data, mostrando um pequeno calendário para melhor visualização do usuário, o que até então era feito ‘manualmente’ pelo desenvolvedor.

Vários padrões utilizados pela Internet, ganharam formalização com o HTML5, facilitando o desenvolvimento de páginas web. A tag , utilizada para a delimitação de áreas de uma página, foi substituída por diversas outras, como <header> e <bottom>, que apesar de não ter nenhuma mudança na renderização da página, auxiliam o entendimento do código, tanto para o desenvolvedor, como também para máquinas interpretarem o conteúdo mais facilmente, que podem entre outras funções, adaptar conteúdo para pessoas com deficiência visual.

Outra novidade da caixa de ferramentas do HTML5, é claro, é o Websocket. Porém como HTML é apenas uma linguagem de marcação, e é estática, depende da linguagem Javascript para utilizar e dar ação a seus elementos.

2.10 JAVASCRIPT

O Javascript é uma linguagem de script, o que significa que ela é utilizada dentro de um outro programa para estender as funcionalidades deste. No caso, ela é executada dentro de um navegador. Atualmente, é a principal linguagem para programação do lado do cliente, em aplicações web. Foi criada com conceitos de orientação a objetos, baseada em protótipos, tipagem fraca e dinâmica e funções de primeira classe.

A sua principal função é interagir com a página web, através do DOM (Document Model Object, Modelo de Objeto de Documento), uma especificação criada pela W3C, para interação com elementos de um documento web. Com o Javascript é possível criar, editar e remover elementos HTML, e utilizar seus recursos de maneira mais dinâmica, sem a necessidade da interação com o servidor.

Ademais, o Javascript permite fazer requisições, assincronamente em relação ao carregamento da página em que está contido. Isto deu origem ao AJAX. Asynchronous Javascript and XML, ou Javascript Assíncrono e XML, é utilizado para pequenas interações com o servidor, sem a necessidade de se trocar de página. Apesar do nome, XML não é um padrão para esta troca assíncrona de informações. Na verdade pode ser feita uma troca de qualquer tipos de dados.

Atualmente, na utilização do javascript, é utilizado um framework muito popular, o jQuery. Uma biblioteca de código aberto, amplamente utilizada na web, que facilita a implementação em diferentes browsers, adaptando o seu código às pequenas diferenças entre diferentes navegadores e suas versões. Claro que tudo feito com jQuery, pode ser feito com Javascript puro, será somente um pouco mais complicado, e muito menos portável.

Um pequeno exemplo de um código que realiza polling, utilizando jQuery, tem a seguinte sintaxe:

```
$.ajax({
  url: "http://www.teste.com.br/ajax",
  success: function(data){
     // trata os dados recebidos
     console.log(data);
  },
  timeout: 3000 // intervalo entre cada requisição em 3 segundos
});
```

Com o Javascript, que é possível a criação de um objeto Websocket no lado do cliente, utilizando a API disponibilizada pelo HTML5. A utilização básica do Websocket, com Javascript, possui a seguinte sintaxe:

```
var ws = new WebSocket("ws://localhost:9998/echo");

// Metodos passivos o objeto websocket, executados pelos eventos de conexão, recebimento de mensagem e perda de conexão.
ws.onopen = function() {
// Executado ao estabelecer uma conexao
  alert("Websocket conectado!");
};
ws.onmessage = function(msg) {
  // Executado ao receber mensagens do servidor.
  var mensagem_recebida = msg.data;
  alert("Mensagem redebida: "+ mensagem_recebida);
};
ws.onclose = function() {
  // Executado na perda de conexão.
  alert("Conexão perdida...");
};
// Metodo para enviar mensagens, utilizando o objeto
ws.send("Enviando mensagem ao servidor!")
```

Por implementar o conceito de threads, e por ser aplicado na API do Websocket, o Javascript mantém uma conexão aberta escutando por mensagens, não interrompendo o fluxo de execução das demais funções. Podemos notar que está sintaxe é bastante simples, pois escrever um código do lado do cliente, para envio e recebimento de mensagens, é trivial, o que tem atraído diversos desenvolvedores.

3 COMPUTAÇÃO CONCORRENTE E DISTRIBUÍDA

Apesar de distintos, os paradigmas de programação concorrente e distribuída são semelhantes e complementares, sendo de vital importância na implementação de servidores web de alta demanda. Estes dois conceitos são sucintamente apresentados, sem qualquer aprofundamento, visando apenas criar um embasamento para a abstração na utilização prática em implementação de servidores.

3.1 PROGRAMAÇÃO CONCORRENTE

"A área de programação concorrente abrange todas as técnicas relacionadas a programação de entidades que concorrem paralelamente para terminar um trabalho, sendo a análise de regiões críticas do código o maior desafio nessa área" (TANENBAUM, Andrew S. 2010).

Programação concorrente consiste na execução simultânea de tarefas, por um programa. Esta execução pode ser de fato simultânea, na existência de mais de um processador na máquina ou, interrompendo e alternando o contexto do único processador entre a as tarefas, executando um pouco de cada tarefa por vez. Esta concorrência é importante para que um longo processo não interrompa a execução de todo um serviço na espera de resultado de funções, podendo utilizar o tempo ocioso de uma tarefa, na execução de outra. Outro fator importante é o compartilhamento de recursos, feito igualmente entre diferentes usuários, como no caso de um servidor web que trata diversas requisições simultaneamente.

A programação tradicional, chamada de sequencial, consiste somente em um fluxo de execução. Utilizando um exemplo, podemos imaginar uma lavanderia como um processo. Primeiro o cesto de roupa suja vai para a máquina, depois para a secadora, e após isto ele é passado e está pronto para entrega, sendo este o processo completo. Na programação tradicional, os cestos seriam processados um por vez, somente iniciando o próximo, quando o anterior já foi entregue. Como podemos imaginar, está não é uma abordagem muito eficiente, pois enquanto um cesto de roupa está secando, outra já poderia estar sendo tratada. A programação concorrente, visa utilizar o tempo ocioso entre o início e término de funções de

um fluxo, no processamento de outras requisições, simultaneamente. Desta forma, em um determinado momento em um sistema concorrente, teremos vários fluxos de execução simultâneos.

Em um programa de computador, tarefas que demorem para terminar, são chamadas de funções bloqueantes. Uma escrita em disco, um acesso a banco de dados ou um longo cálculo não podem parar toda a execução de um programa, esperando que seu resultado esteja pronto. Em um serviço web, por exemplo, a requisição de um cliente a uma simples página, não deve esperar que a consulta de outros 3 clientes ao banco de dados seja concluída, para somente após receber a sua resposta.

3.1.1 Processos e Threads

Um processo consiste em um programa, que foi iniciado e está em execução em um sistema operacional. É possível executar o mesmo programa várias vezes, tendo diferentes processos. Inicialmente, um processo possui apenas um fluxo de execução, chamado de thread. Um sistema concorrente, inicia inúmeras threads, dentro do mesmo processo, tendo vários fluxos de execução.

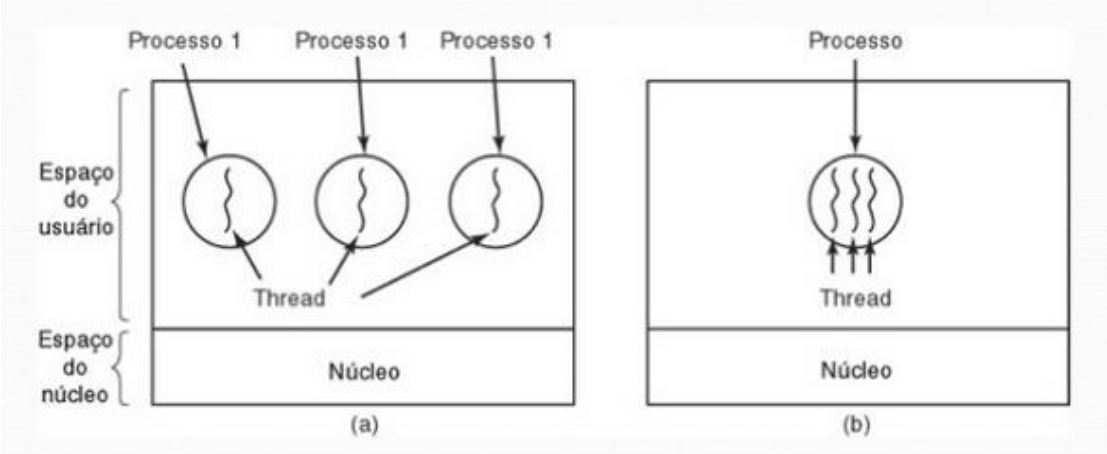

Figura 7 - (a) Três processos, com uma única thread cada. (b) Um processo com três threads. (TANENBAUM, Andrew S. 2010)

O suporte a threads, é fornecido pelo sistema operacional, que isola certa parte do contexto do programa, como o estado e as variáveis do fluxo de execução, e abstrai a troca de contexto no nível do processador. No caso de processadores com múltiplos núcleos, a

programação com várias threads, otimiza a execução de um processo, utilizando toda a capacidade do hardware. Atualmente, praticamente todos os sistemas operacionais oferecem o suporte a threads, deixando que isto seja facilmente abstraído pelo desenvolvedor, utilizando uma linguagem de programação, que implemente o conceito de thread.

3.1.2 Programação Orientada a Eventos

Uma grande dificuldade no desenvolvimento de um sistema concorrente, está na quebra dos procedimentos, para que possam ser executados paralelamente, e na ordenação e sincronização destas atividades. Uma maneira de desenvolver um sistema concorrente, é implementar de forma que tudo gire em torno de eventos, e que esses sejam esperados e tratados de um modo adequado. Com isso, se um processo requisita uma função, ela é iniciada, e, ao final de sua execução, um evento será disparado, passando seu resultado como parâmetro de alguma forma. Outra função, que espere este tipo de evento, é executada com esta entrada, e assim por diante.

> Um evento é uma ocorrência dentro de um sistema ou um domínio particular, é algo que aconteceu, ou é contemplado como tendo acontecido neste domínio. A palavra evento também é utilizada para significar uma entidade de programação que representa uma tal ocorrência em um sistema de computação. Processamento de eventos é a computação que executa operações em eventos. Operações de processamento de eventos comuns, incluem leitura, criação, transformação, e excluir eventos. (ETZION, Niblett. 2010).

Neste modelo, teremos elementos produtores e consumidores de evento. Estes serão conectados geralmente através de um outro elemento, chamado de *event loop*, ou loop de eventos. Este event loop, é um ciclo infinito, que está em rodando o tempo todo durante a execução do programa, onde sua função é escutar estes eventos e encaminhar aos consumidores devidos. Esta é uma abordagem bastante comum em programação no lado do cliente, como vimos no exemplo do Websocket em Javascript, e suas funções que são executadas por algum evento. Todo sistema que reage quando um botão é clicado, por exemplo, também serve como exemplo.

"Utilizar eventos em sistemas computacionais, não é novidade. Nos primórdios da computação, os eventos apareceram na forma de exceções, cujo papel era o de interromper o fluxo normal de execução, e causar a inicialização de processos alternativos." (ETXION, Niblett)

Implementando este conceito em servidores web, cada requisição irá gerar um evento, que será tratado sem interromper o processo principal, que continuará tratando outras requisições. Somente quando a função bloqueante terminar, gerando um evento, é que a resposta será encaminhada de volta ao cliente. Uma maneira de implementar isso utilizando threads é com um *thread pool*, que consiste em um número limitado de threads já iniciadas, com a mesma função, que escutam e processam um tipo determinado de evento, que normalmente se encontra em uma fila. Por exemplo, um servidor web, pode ter uma thread pool de tamanho diferente para várias funcionalidades, dependendo da frequência de uso da mesma. Desta forma, o sistema economiza tempo, não tendo que instanciar uma thread para cada nova requisição daquela funcionalidade, com o custo de gastar recurso computacional para manter a mesma aberta, mesmo em períodos ociosos. Sendo assim, é possível controlar o uso de processamento em diferentes funções, com distintas prioridades.

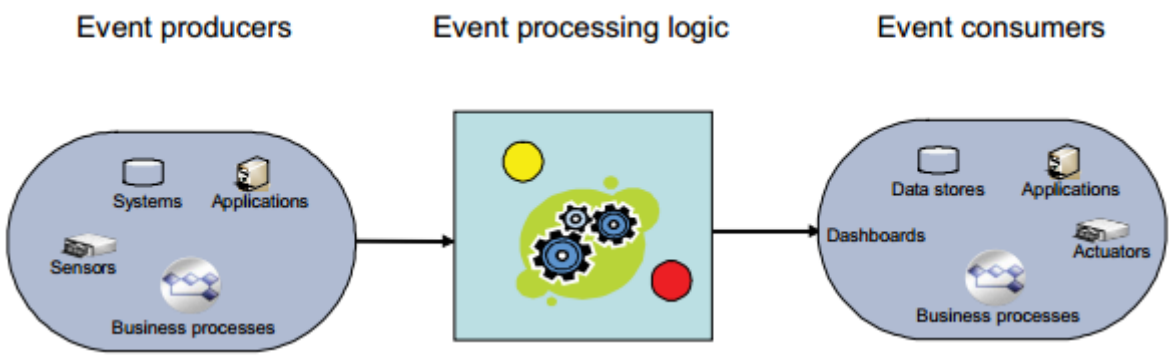

Figura 8 - Produtores e consumidores de evento, isolados da lógica de processamento do sistema (ETZION, Niblett. 2010)

Por ser apenas um paradigma, e não uma tecnologia, existem inúmeras formas de implementar estes conceitos.

3.2 PROGRAMAÇÃO DISTRIBUÍDA

"Um sistema distribuído, é aquele onde os componentes estão em computadores conectados por uma rede e coordenam suas ações apenas pela troca de mensagens."
(COULOURIS et al, 2011).

Os computadores, chamados de hosts, trabalham em conjunto como uma única máquina, do ponto de vista dos usuários do serviço. Este conglomerado de computadores

conectados, são chamados de *clusters*, caso estejam geograficamente próximos, conectados por uma rede interna por exemplo, ou *grids*, quando estão espalhados por uma grande área, visando atender requisições locais.

O modelo cliente-servidor, descrito no Capítulo 2, é um sistema distribuído afinal, temos programas no lado do cliente, interagindo com os servidores. Porém, nesta sessão será tratada apenas o ponto de vista de distribuição de serviço no lado do servidor.

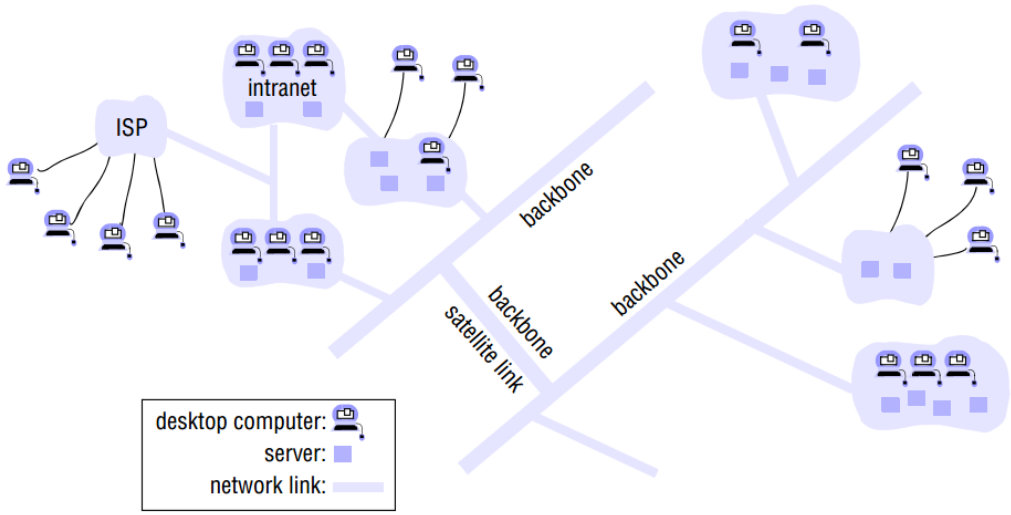

Figura 9 - Uma porção da internet, que mostra clientes, servidores e a rede que os interconecta (COULOURIS et al, 2011).

A principal razão para se distribuir um sistema entre várias máquinas, está no poder de distribuição de carga de trabalho, podendo aumentar o desempenho e a capacidade de um sistema, em relação ao incremento da quantidade de máquinas utilizadas. Outro motivo para está no compartilhamento de recursos, como por exemplo, um banco de dados.

> No sentido contrário à centralização dos serviços e algoritmos, uma das principais metas dos sistemas distribuídos é a escalabilidade. Um sistema é descrito como escalável se permanece eficiente quando há um aumento significativo no número de recursos e no número de usuários a ele conectados (COULOURIS et al, 2000).

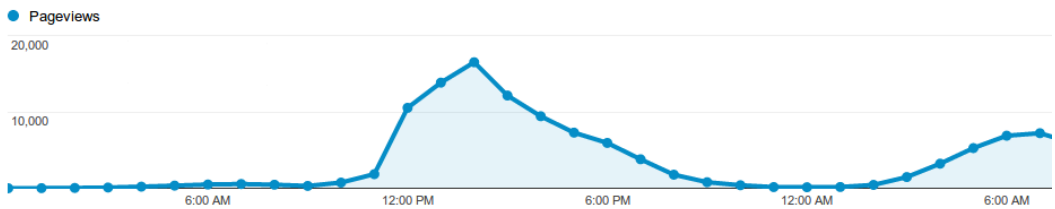

Figura 10 - Oscilação na quantidade de acessos em um site de comércio online real, de acordo ao horário.

A escalabilidade é um fator crucial para serviços web, por diversos fatores. O primeiro deles, é a estabilidade. Um sistema online, de qualquer natureza, que ao ter um aumento na quantidade de acesso saia do ar, ou não funcione apropriadamente, está deixando de atender clientes. Outro fator é a economia de recursos, a escalabilidade permite que, em períodos com poucos usuários, seja possível desabilitar porções dos servidores, gerando uma economia de recursos. Como pode ser observado na Figura 10, que mostra a quantidade de acessos, de acordo ao horário, em um site de comércio online real, em horários de pico, os acessos chegam a quase 20mil por hora, enquanto durante a madrugada, caem para menos de 500.

Ao se distribuir o processamento de um serviço, novos problemas têm que ser levados em consideração. Em alguns casos, os sistemas interconectados, são heterogêneos, com sistemas operacionais e linguagens diferentes. Desta forma, a comunicação entre eles tem de seguir algum padrão. A monitoração também se torna um trabalho mais complexo, afinal são diversas máquinas, conectadas por uma rede, e a própria rede tem de ser monitorada, podendo ela causar atraso e perda de dados.

Uma grande dificuldade da distribuição, é também um desafio quando se desenvolve um sistema concorrente. Particionar um fluxo de processamento, em diversas funções, que precisam ser executadas em uma ordem correta. O que já não era tão trivial quando feito em um sistema concorrente, fica ainda mais complexo, quando os processos estão em máquinas distintas.

3.2.1 Modelo Publish-Subscribe

Uma maneira para auxiliar a distribuição de um sistema, é a definição de um padrão de mensagens, a fim de que os computadores conectados à rede consigam se comunicar, independente da linguagem ou sistema operacional que utilizam. O protocolo HTTP, por exemplo, é o padrão de mensagem utilizado para a troca de informação entre cliente e servidores na web. Para a comunicação entre os hosts de um sistema distribuído, podem ser utilizados quaisquer outros padrões, desde que todos os computadores saibam como interpretar a mensagem transmitida.

Uma modo de implementar o sistema de mensagem, é empregando o modelo publish-subscribe (publicador-assinante), ou somente pub-sub.

> “Um sistema publish-subscribe é um sistema onde publicadores publicam eventos para um serviço de eventos e assinantes indicam interesses em eventos particulares através de assinaturas, que podem seguir uma regra sobre padrões arbitrários dos eventos.” (COULOURIS et al, 2011)

Desta forma, os processos podem tanto escutar quanto publicar mensagens em canais, classificados em tópicos e filas. Ao publicar uma mensagem em um tópico, todos os processos que escutam o mesmo, irão receber esta mensagem, já ao publicar em uma fila, apenas um processo, dos muitos que podem estar assinando a mesma, irá receber esta mensagem. Este comportamento pode ser observado na Figura 11.

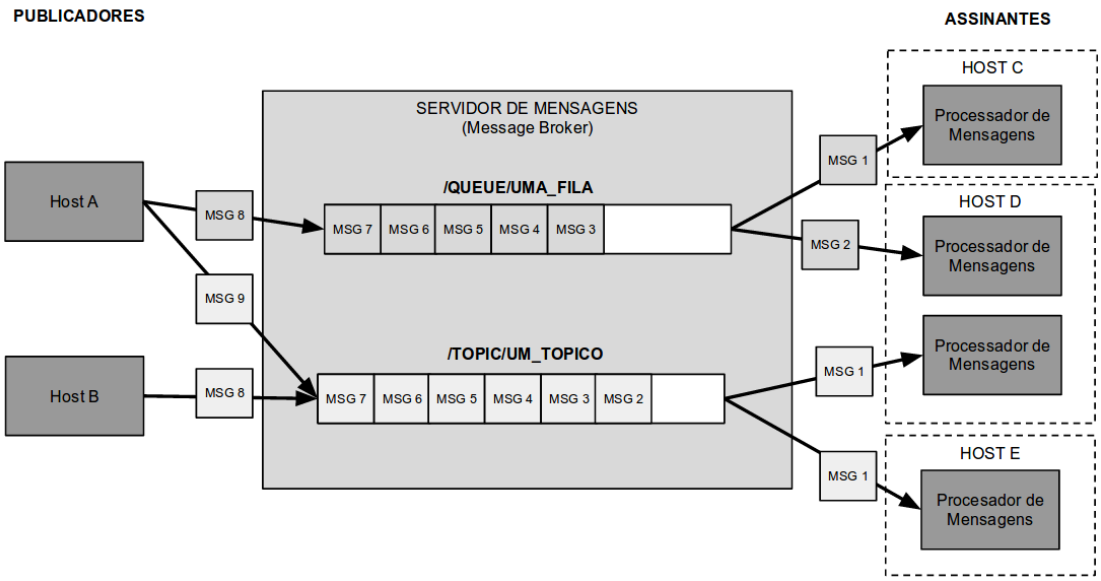

Figura 11 - Publicadores e assinantes de fila e tópico, em diferentes hosts.

Assim, é possível intercomunicar os diversos processos de um sistema distribuído, sem se preocuparmos com o conhecimento mutuo dos computadores envolvidos, desde que todos conheçam o serviço de mensagem, chamado também de *Message Broker*. Um Message Broker é um programa em que a sua única função é transmitir mensagens de maneira apropriada entre os processos envolvidos em um sistema distribuído.

Esta metodologia é bastante semelhante ao sistemas distribuídos orientados a eventos, e com paradigmas de implementação muito parecidos, no que se diz ao particionamento de funcionalidades. Um evento ocorrido em um sistema, pode enviar uma mensagem para o Message Broker, e irá gerar um evento de mensagem recebida em outro processo, possivelmente em outro host.

A grande vantagem da utilização desta metodologia, está na simplificação em se resolver gargalos no sistema. Podemos ter quantidades diferentes de processos para cada tipo de função, e não necessariamente na mesma máquina. Desta forma, tendo desmembrado todas as funções de um fluxo completo, é possível alocar mais recursos para as funções mais requisitadas e com custo computacional maior.

3.3 CONSIDERAÇÕES NA IMPLEMENTAÇÃO

Muitos frameworks atualmente, implementam servidores web, abstraindo a concorrência, e auxiliando o desenvolvedor a se preocupar apenas com a funcionalidade, sem ter que programar diretamente um modelo concorrente, para atender requisições. Outras ferramentas auxiliam na distribuição do sistema entre diversas máquinas, até mesmo por demanda.

Neste capítulo foi apresentado, conceitos que auxiliam a capacidade de escalabilidade em um sistema. Em um servidor web, escalabilidade é um pré-requisito essencial, para qualquer serviço que tenha o intuito de se manter estável, mesmo em diferentes condições de utilização.

Como veremos no capítulo 4, aplicando tecnologias amplamente utilizadas pela web atualmente, é possível criar sistemas escaláveis de forma relativamente simples, desde que se leve em consideração os paradigmas apresentados neste capítulo, durante o desenvolvimento.

4 TECNOLOGIAS E ARQUITETURA DE UM SERVIDOR ESCALÁVEL

Neste capítulo são tratadas técnicas e ferramentas para a implementação de um servidor web. Boa parte do conteúdo do capítulo anterior será abstraída pela utilização de aplicações e frameworks, que implementam esses paradigmas, e são amplamente utilizadas, na prática, por grandes empresas web.

4.1 PYTHON

Python é um linguagem de programação de alto nível, interpretada, orientada a objetos, funcional e de tipagem dinâmica e forte. Lançada em 1991, foi desenvolvida com a filosofia de dar importância ao esforço do programador, sendo assim sua sintaxe é limpa, priorizando a legibilidade do código. Por estes fatores, atraiu diversos desenvolvedores ao redor do mundo, se destacando atualmente pela enorme quantidade de bibliotecas disponíveis, para as mais diferentes áreas de programação.

"Python permite que seja escrito o código necessário, rapidamente. E graças a um compilador altamente otimizado em código binário, o código Python roda mais rápido que o necessário para a maioria das aplicações." (Python.org, tradução nossa).

A cultura da linguagem, segue a filosofia do *"Zen of Python"*, um poema que pode ser lido, ao executar o comando *"import this"*.

The Zen of Python, by Tim Peters[3]

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.

[3] **O Zen de Python, por Tim Peters -** Bonito é melhor que feio. Explícito é melhor que implícito. Simples é melhor que complexo. Complexo é melhor que complicado. Plano é melhor que aninhado. Esparso é melhor que denso. Legibilidade conta. Casos especiais não são especiais o bastante para se quebrar as regras. Embora a simplicidade supere o purismo. Erros nunca deveriam passar silenciosamente. A menos que explicitamente silenciados. Ao encarar a ambiguidade, recuse a tentação de adivinhar. Deveria haver uma – e preferencialmente apenar uma- maneira óbvia de se fazer isto. Embora aquela maneira possa não ser óbvia à primeira vista se você não for holandês. Agora é melhor que nunca. Embora nunca, seja muitas vezes melhor que pra já. Se a implementação é difícil de explicar, é uma má ideia. Se a implementação é fácil de explicar, pode ser uma boa ideia. Namespaces são uma ideia estupenda – vamos fazer mais deles.

Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Por seguir estas características, é uma linguagem de fácil aprendizado, e ainda assim muito poderosa. Um pequeno exemplo da sintaxe, pode ser observado no trecho de código a seguir:

```
class Pessoa():

        def __init__(self, nome):
                self.nome = nome
                self.conhecidos = []

        def adicionarConhecido(self, pessoa):
                self.conhecidos.append(pessoa)

        def imprimirConhecidos(self):
                for pessoa in self.conhecidos:
                        print pessoa.nome

# Isto é um comentario

joao = Pessoa("Joao")
joao.adicionarConhecido( Pessoa("Jose") )
joao.adicionarConhecido( Pessoa("Maria") )
joao.imprimirConhecidos()

output:
> Jose
> Maria
```

Em python, os blocos de código são delimitados pela identação. A função *__init__*, é o construtor da classe, e a variável *self* representa a instância, equivalente ao *this*, em outras linguagens de programação. O código python, por muitas vezes se assemelha a uma

metalinguagem, sendo a leitura e compreensão intuitiva para qualquer pessoa familiarizado com linguagens de programação.

4.1.1 Framework Twisted

" Twisted é um framework dirigido a eventos. Isto significa que ao invés de ter as funções chamadas em uma ordem específica, definida pela lógica do programa, elas são chamadas em resposta a ações externas, ou eventos." (FETTING, Abe. 2005)

O Twisted é um framework escrito em python, que segue o padrão de orientação a eventos, e implementa o event loop em um objeto, chamado *reactor*, que pode ser utilizado para a chamada de funções em modo não bloqueantes. É uma boa opção tanto para grandes processamentos, como também para implementar serviços web. Um simples servidor "eco", que devolva o que foi recebido pelo protocolo TCP, na porta 8080, pode ser escrito da seguinte maneira:

```
from twisted.internet import protocol, reactor

class Echo(protocol.Protocol):
  def dataReceived(self, data):
     self.transport.write(data)

class EchoFactory(protocol.Factory):
  def buildProtocol(self, addr):
     return Echo()

def stop()
   reactor.stop()

reactor.listenTCP(8080, EchoFactory()) #Esperando conexões na porta 8080
reactor.callLater(60, stop) #Agendando a execucao da funcao stop para 60 segundos
reactor.run() #Iniciando a execucao do event loop
# Só chegará nesta linha, quando a funcao stop, terminar o event loop
print "Servidor terminado"
```

Neste exemplo, analisando o objecto reactor, pode-se observar que, após executar o comando run(), o loop de eventos é iniciado. A chamada *callLater*, invocará a função *stop()* após 60 segundos, que interromperá o ciclo do reactor, e finalizará o programa.

As classes *Factory* e *Protocol* são a base para o desenvolvimento de aplicações de rede. A classe Factory, está ligada diretamente com a porta que o serviço escuta, decidindo o que fará com cada conexão. Ela possui métodos para iniciar e interromper o tratamento de requisições, que podem ser estendidos. Além disso, possui o método “buildProtocol”, que define a classe Protocol, que será instanciada para cada nova conexão. Na classe Protocol, toda a transmissão de dados já foi abstraída, dando a liberdade de se preocupar somente com a lógica do dados. Analisando o trecho de código acima, podemos utilizar outra facilidade do twisted, e trocar apenas a chamada “reactor**.**listenTCP” por “reactor**.**listenHTTP”, ou até mesmo “listenWS”, sem alterar o resto do código, para termos o mesmo serviço, usando outros protocolos. Porém, é possível criar qualquer outro método, para um protocolo próprio, seguindo as diretivas do framework, sem ter que alterar as camadas superiores.

Neste trabalho, esse framework será utilizado apenas para o desenvolvimento de servidores de teste, com o intuito de comparar o desempenho do protocolo Websocket, em relação ao protocolo HTTP em diferentes cenários.

4.2 APACHE APOLLO

O Apache Apollo, é o um servidor de mensagens, com código aberto, muito utilizado atualmente. É mantido pela Apache Software Foundation, também conhecida pelo amplamente utilizado Apache HTTP Server.

Entre suas características, está em destaque o seu desempenho e capacidade de escalabilidade, podendo se replicar em um cluster com pequenas configurações, caso se torne um gargalo no sistema. Nas configurações, após a sua instalação, é possível escolher qual protocolo de mensagem será utilizado e se as mensagens trocadas serão armazenadas de forma persistente ou não. Há diversas formas de configurar este serviço, podendo ser aplicado a variados cenários, entretanto, para o desenvolvimento deste trabalho, ele é utilizado somente como um Message Broker.

4.2.1 STOMP

O protocolo STOMP, do inglês *Simple* (or Streaming) *Text Oriented Message Protocol*, Simples Protocolo de Texto Orientado a Mensagens, é um protocolo baseado em texto, como o HTTP. Sua principal característica é oferecer interoperabilidade entre diferentes sistemas, facilitando a transmissão e compreensão das mensagens trafegadas. Possui uma estrutura muito simples, de fácil usabilidade. É possível criar um cliente em qualquer linguagem de programação, sem muita dificuldade, aplicando socket e manipulando strings.

```
SEND
destination:/queue/a
content-type:text/plain

hello queue a
^@
```

Figura 12 - A sintaxe de uma mensagem Stomp, enviada para a fila /queue/a. (STOMP <http://stomp.github.io/>)

O protocolo STOMP é utilizado neste trabalho, por implementar o padrão Publish-Subscribe, explicado anteriormente. Para implementar as funções desse padrão, são usados os seguintes comandos: Connect, Send, Subscribe e Unsubscribe. O Apache Apollo, utilizado na implementação deste trabalho, foi configurado para aplicar o STOMP como protocolo de mensagens. Sendo assim, ele é responsável por transportar as requisições e respostas de processamentos, entre os hosts.

4.3 REDIS NOSQL

Como todo serviço web, precisamos de um sistema de armazenamento de dados. O termo NoSQL (Not Only SQL, ou não apenas SQL), consiste em uma classe de bancos de dados não necessariamente relacionais, e não obrigatoriamente implementado as propriedades ACID (Atomicidade, Consistencia, Isolamento e Durabilidade). Ao abrir mão destas características, possuem um desempenho muito mais alto, em relação a operações de consulta e alteração de dados, se comparados a bancos de dados relacionais.

O Redis é um NoSQL do tipo chave/valor. Ou seja, é possível acessar um dado, somente pela sua chave, sem uma pesquisa de conteúdo. Funciona basicamente como uma memória de objetos, compartilhada entre processos. Outra característica, é a possibilidade de manter estas chaves na memória do computador onde foi instalado, agilizando as operações consideravelmente. Este serviço possui aspectos de persistência periódica, sendo as alterações salvas em disco, conforme alguma regra configurável, relacionando tempo e operações. Pode ser utilizado em conjunto com banco de dados tradicional, para evitar consultas recorrentes, armazenando diretamente o resultados, relacionado a uma chave, que pode ser por exemplo, o hash de uma consulta SQL.

A utilização do Redis, neste trabalho, é para armazenamento de notificações pendentes de clientes. Além disso, é responsável também, por armazenar resultados de operações bloqueantes.

4.4 HOSPEDAGEM E FERRAMENTAS NA NUVEM

A pouco tempo, a única forma de se criar e hospedar um serviço web, era comprando e mantendo grandes máquinas físicas. Com isso, a responsabilidade pela manutenção tanto das máquinas quanto da rede interna era de quem as comprava. Ademais, também existia a preocupação com a estabilidade e capacidade do ponto de internet do local onde as máquinas estavam.

Com a popularização do Cloud Computing (Computação nas Nuvens), é possível agora alugar máquinas, de configurações customizáveis, pagando por hora de utilização. E não somente máquinas, mas também serviços inteiros podem ser alugados, diminuindo o custo de infraestrutura e manutenção consideravelmente. Umas das empresas que fornece este serviço é a Amazon.

4.4.1 AWS - Amazon Web Services

A Amazon, grande empresa americana, é a líder em serviços de Computação nas Nuvens mundialmente. A AWS é um conjunto de serviços oferecido, que constituem uma plataforma completa para a hospedagem de aplicações web, sendo possível contratar diversos

recursos relacionados a servidores. O serviço mais utilizado é o Amazon EC2 (Amazon Elastic Compute Cloud). Com o EC2, é possível alugar máquinas virtuais, de configurações simples, até as mais potentes, com poucos cliques, ou até mesmo utilizando uma API. Outro serviço fornecido pelo AWS é o Elastic Load Balancing (Balanceamento de Carga Elastico), que distribui automaticamente requisições entre mais de uma instancia. Diversos outros serviços, como bancos de dados, armazenamento de arquivos estáticos, implementação de redes, também são oferecidos a preços equivalentes à utilização.

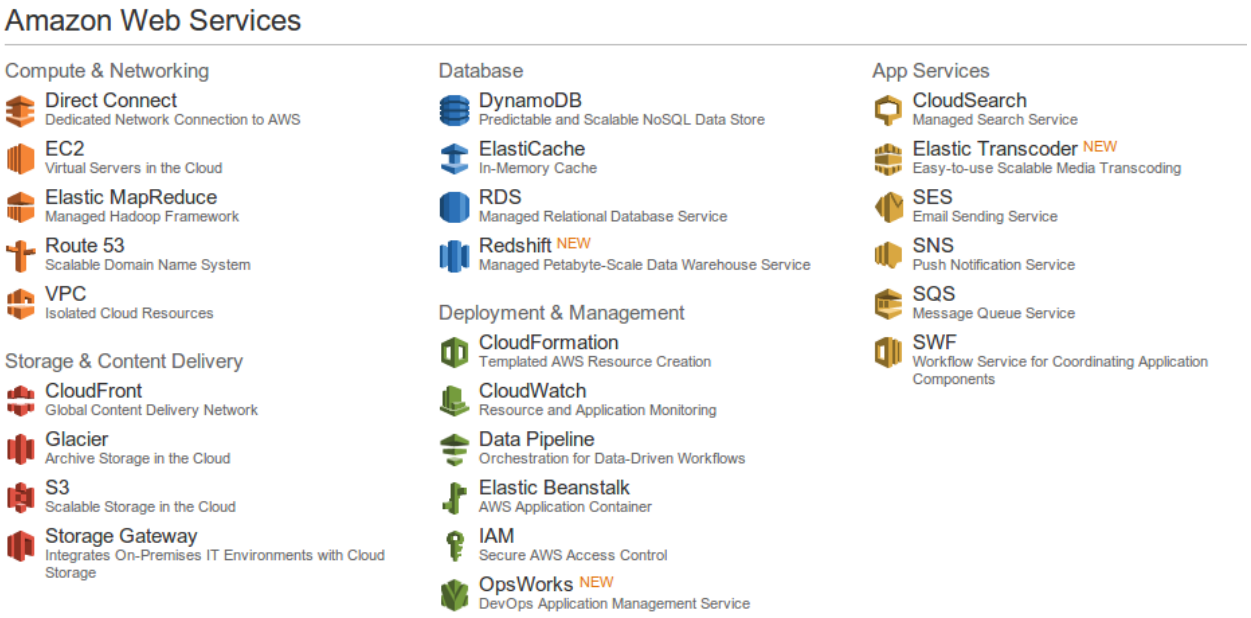

Figura 13 - Serviços web oferecidos pela Amazon AWS. (Amazon <http://www.amazon.com>)

Para desenvolvimento deste trabalho, o servidor foi hospedado em máquinas na EC2. Ademais, também foi empregado o serviço de Elastic Load Balancing, sendo utilizado para demonstrar a escalabilidade do servidor, ao distribuir a carga entre várias máquinas.

4.5 ARQUITETURA DE SERVIDORES ESCALÁVEIS

A presentadas as soluções tecnológicas adotadas, é discutido como elas se encaixarão em um serviço web completo. Fique claro que todas as ferramentas e linguagens aqui apresentadas, são facilmente substituídas por outras semelhantes, sendo a escolha destas aplicações, tão importante quanto a marca dos tijolos, na construção de um prédio.

4.5.1 Requisições HTTP Síncronas e Assíncronas

Podemos dividir as requisições web em dois grandes grupos, as síncronas e Assíncronas. Uma requisição síncrona, é aquela que obtém uma resposta imediatamente. Basicamente todas as requisições feitas pelo navegador, após entrar com uma URL, são síncronas, pois retornam imediatamente o resultado da consulta. Requisições assíncronas, são aquela que verificam se existe algo de novo no servidor, mesmo sem uma nova ação do usuário. Por exemplo, ao acompanhar um jogo de futebol em portal qualquer, o usuário, mesmo sem trocar de página, recebe atualizações constantemente, com textos dos novos lances. Outro exemplo de requisições assíncronas, são aquelas onde a operação requisitada, irá demorar algum tempo para ser computada. Ao submeter uma transação bancária, por exemplo, imediatamente após a requisição, o servidor irá responder com uma chave única, chamada de chave de transação, então o usuário, mesmo sem consciência, irá enviar requisições posteriormente, para verificar se a transação com aquela chave já possui uma resposta pronta.

As requisições assíncronas serão o foco da implementação deste trabalho. Afinal, são o principal caso de uso do protocolo Websocket, pois para cada nova requisição - seja para atualização de conteúdo, ou consulta do resultado de uma operação - será aberta uma nova conexão TCP, enviando uma mensagem HTTP, sendo esta processada e respondida, mesmo que não haja uma resposta. Com Websocket, o servidor simplesmente envia as atualizações quando disponíveis, e responde com o resultado da operação, sem a necessidade de uma nova requisição.

Os termos síncrono e assíncrono, usados nesta sessão, não dizem nada em relação à implementação do servidor, sendo apenas para diferenciar requisições do lado do usuário. Uma requisição dita síncrona, pode ser executada de maneira assíncrona, em termos da implementação da função.

4.5.2 Montando as Peças

Agora que temos as peças utilizadas, podemos montar uma arquitetura para o servidor. Os casos simulados, são descritos no Capítulo 5, mas as partes são as mesmas para todos, alterando apenas a lógica de operação.

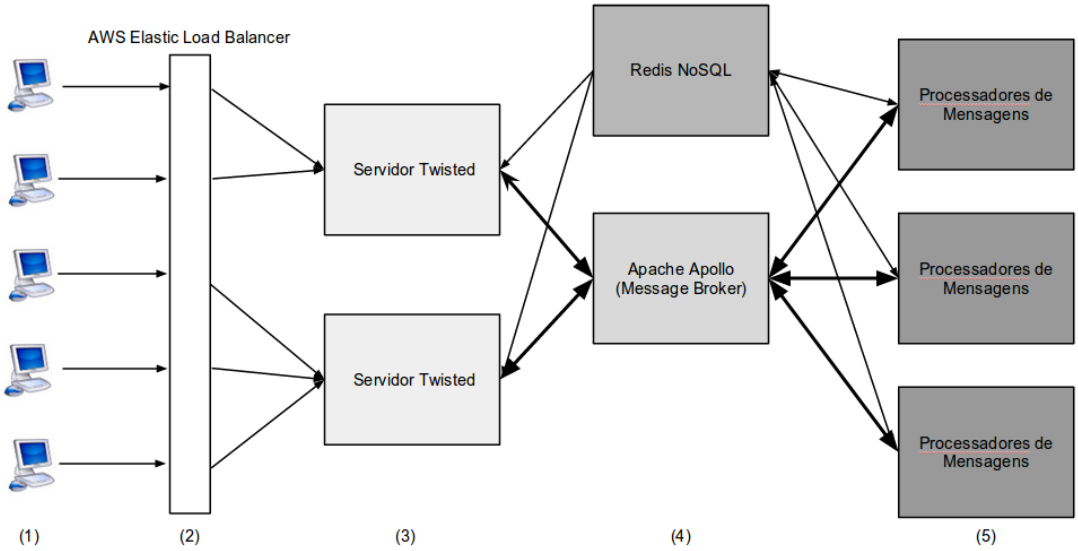

Figura 14 - Arquitetura de um servidor web, utilizando um modelo de distribuição.

Analisando a Figura 14, vemos que na borda entre os clientes (1) e o servidor (3), estará o Load Balancer (2), que apenas encaminhará as requisições adiante, para os servidores Twisted.

Os servidores Twisted, são cópias exatamente iguais, instanciados separadamente em máquinas da EC2. A carga destas máquinas, são o ponto principal de análise deste trabalho, afinal, são elas que se comunicarão com os clientes, sendo por HTTP ou Websocket. A vantagem de combinar o Load Balancer com instancias de servidores identicos, está na capacidade de distribuir as requisições sobre um mesmo endereço IP.

Comunicando-se com os servidores Twisted, está o Apache Apollo (4), que funciona como o Message Broker e faz a ligação entre as requisições dos clientes, e os processadores (6), instanciados em outras máquinas.

Os processadores de mensagens, estão em outras máquinas, assinando canais do Apollo, e respondendo para o mesmo. Esses processadores, podem exercer as mais variadas funções, sendo estes os responsáveis pela parte lógica da aplicação, processando as funcionalidades do sistema.

Com este formato, os servidores que tratam das conexões com os clientes, estão isolados da parte lógica da aplicação. Desta forma, se a demanda por conexões crescer, podemos replicar as máquinas em (3), e caso a demanda pelo serviço se tornar um gargalo,

replicamos máquinas responsáveis pela parte lógica (5). Sendo assim, temos um servidor com uma capacidade elástica, que mantém a estabilidade do serviço, para diferentes quantidades de clientes.

5 PROCEDIMENTO EXPERIMENTAL

Neste capítulo, todos os conhecimentos tratados nos capítulos anteriores, tomarão um caráter prático, na implementação de servidores de teste, para a observação do comportamento do Websocket, em relação ao desempenho em diversos fatores, comparando com o HTTP.

5.1 CENÁRIOS DE TESTE

Para a experimentação deste trabalho, foi desenvolvido um servidor web com as ferramentas e técnicas abordadas nos capítulos anteriores. A implementação foi feita de forma mais simples possível, para evitar que qualquer pedaço de código desnecessário, influenciasse o resultado de alguma forma. No total, foram três situações implementadas e avaliadas, todas desenvolvidas com o protocolo HTTP e Websocket, como forma de comunicação cliente-servidor.

O primeiro caso, é um simples servidor de eco, onde é avaliado o tempo de envio e resposta de uma série de mensagens, entre cliente e servidor. Apenas um cliente envia uma diferente quantidade de mensagens, e é medido o tempo total de envio e recebimento de todas, nos diferentes casos. O propósito principal deste teste é avaliar a diferença de performance, principalmente no lado do cliente, e também a diferenca de utilização de um canal de comunicação permanente, comparado a simples requisições unitárias.

O segundo teste implementa um caso mais prático, a de streaming de conteúdo. Neste teste, um processador transmite uma cópia do livro “O guia do Mochileiro das Galáxias”, em pequenos trechos de cada vez, em um tópico no Apollo. Uma diferente quantidade de clientes simultâneos acompanham estas mensagens. Este é um cenário muito comum em serviços web, e é observado principalmente a diferença na carga sofrida pelo servidor, com as diferentes implementações.

O terceiro e último caso avalia um servidor de bate-papo, onde cada mensagem enviada por um cliente, é propagada a todos os outros conectados. Este teste visa criar um ambiente semelhante ao de aplicações online, que utilizam uma grande troca de mensagens entre os clientes.

5.2 AMBIENTE DE TESTE E MONITORAMENTO

Para os testes, foi implementado um serviço no servidor, que escuta requisições para monitoramento. Ao receber uma requisição, com uma identificação do começo de um teste, é iniciado um processo que monitora a cada 200ms a porcentagem de uso de processamento do serviço web em questão, e armazena estes dados em um arquivo de log. Estes dados são combinados com diversas repetições do mesmo caso testado, e são utilizados para gerar médias e gráficos. Vale também ressaltar que entre cada teste o servidor foi reiniciado, para evitar qualquer contaminação entre os mesmos.

Para criar quantidades diferentes de requisições, foram utilizados scripts, também escritos em python, para simular usuários de teste. Estes scripts foram executados, dependendo do caso, em múltiplas máquinas, para que fosse possível criar uma demanda no servidor, a fim de causar algum stress. Os clientes executados, também geram dados, que são armazenados, para a avaliação dos resultados.

O servidor que lida diretamente com a conexão com os clientes, como dito anteriormente, foi desenvolvido utilizando o framework Twisted. Ele foi instanciado em máquinas na EC2 que rodam com o sistema operacional Ubuntu Server 12.04 32bit. Sua funcionalidade é basicamente a de encaminhar as mensagens entre os clientes e o Message Broker.

5.3 COLETA DE DADOS

No servidor, o processo de monitoramento gera um arquivo com o seguinte formato:

```
…
Tempo:110.74    CPU:11.1000    Mem:3.2644
Tempo:111.44    CPU:10.0000    Mem:3.2644
Tempo:112.14    CPU:20.0000    Mem:3.2644
Tempo:112.84    CPU:10.0000    Mem:3.2644
…
```

Este arquivo foi analisado por um script, e os dados combinados, a fim de criar gráficos que apresentem os resultados de forma mais clara.

Os clientes, após o teste, geram o seguinte relatório:

```
[ff586822-d303-4a08-8375-65d41a40f7d0] Requests:500 Sucess:500 Inter-
val:0.20 Elapsed_Time:120.5052068233 Network_Time:20.5052068233
```

Estes dados são combinados, para exibir resultados médios dos clientes. A variável ‘Network_Time” representa o tempo gasto apenas pelas requisições, excluindo o tempo gasto no intervalo (Network_Time = Elapsed_Time - Requests*Interval).

Todos os testes realizados, seguem o seguinte padrão para nome:

(teste)-(protocolo)-(clientes)x(requisicoes)x(intervalo)ms

Exemplos: echo-ws-100x100x200ms, echo-http-10x100x0ms

5.4 RESULTADO DAS OBSERVAÇÕES

Em todos os casos avaliados, o resultado apresentado demonstra a média da realização de 15 repetições do mesmo teste. A duração de cada teste foi de 2 minutos.

5.4.1 Servidor Eco

O Primeiro teste, com o servidor eco, consiste no envio e recebimento de uma quantidade de mensagens, por um único cliente. Este teste foi realizado, utilizando um servidor Twisted, com um único cliente, nas seguintes circunstancias:

- **Situação:**
  - Total de Mensagens: 1, 2, 5, 10,100
- **Observar:**
  - Tempo total de realização

Os resultados obtidos estão representados na Figura 15.

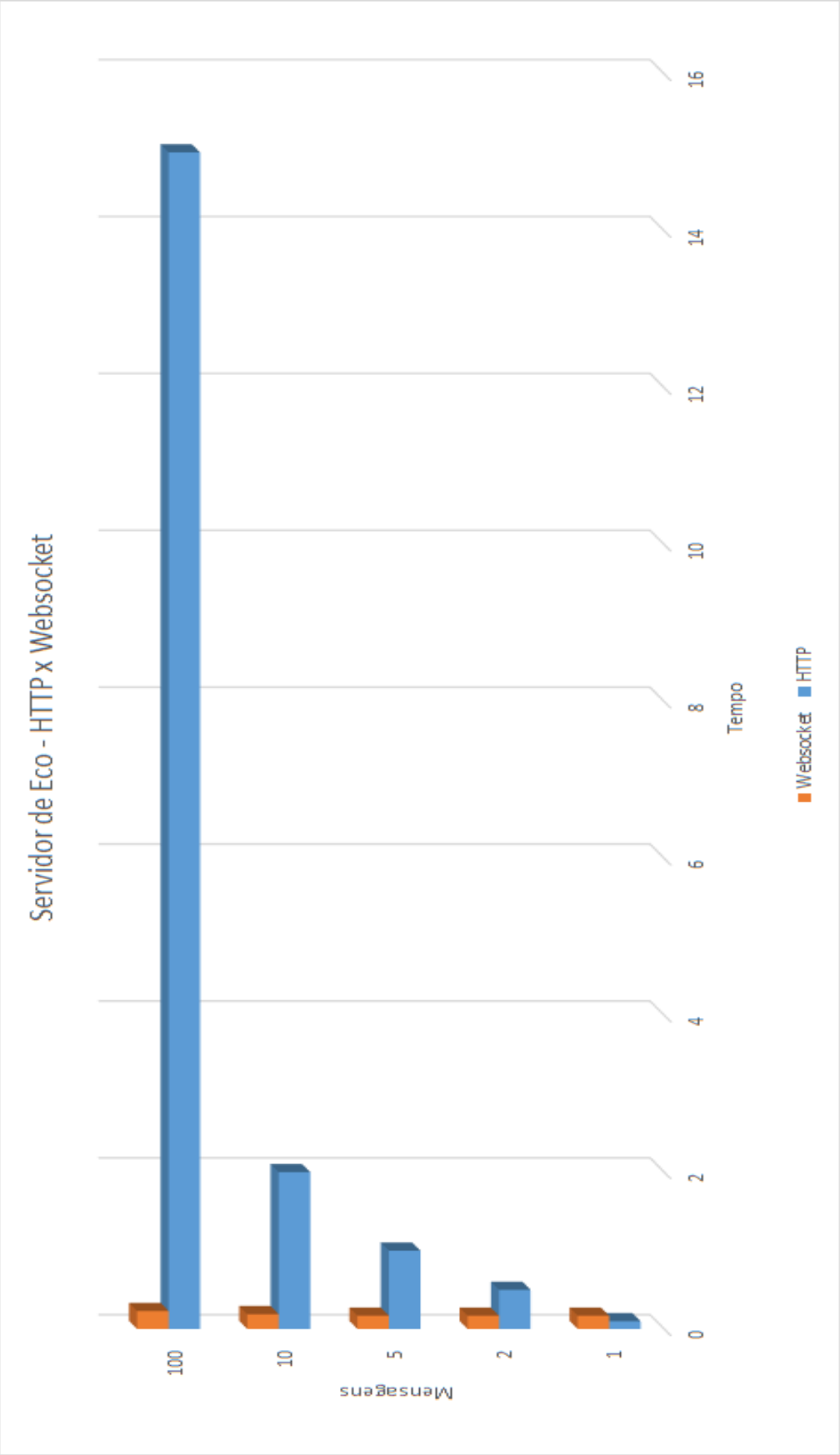

Figura 15 - Teste do servidor de eco, com um cliente, em ambos os protocolos Websocket e HTTP.

Podemos observar claramente a vantagem do envio e recebimento assíncrono de mensagens, garantido pelo Websocket. Enquanto o protocolo HTTP tem um crescimento de tempo linear, em relação a quantidade de mensagens, o protocolo Websocket tem um aumento menor do que 40%, em relação ao envio e recebimento de 1 e 100 mensagem.

A única vantagem do protocolo HTTP foi observada no envio e recebimento de uma única mensagem, por uma diferença de 0.11s contra 0.17s, certamente devido as 2 mensagens trocadas para o estabelecimento de uma conexão Websocket, como visto anteriormente. Todos os resultados observados neste teste, eram bastante esperados, afinal ao estabelecer uma conexão, utilizando o protocolo Websocket, é possível enviar as mensagens sem ter de aguardar a sua resposta, para o envio da mensagem seguinte.

5.4.2 Servidor de Stream

Neste teste, um processo em um host, envia mensagens em um tópico para o Message Broker, com trechos de 1KB do livro "O Guia do Mochileiro das Galáxias" a cada intervalo randômico de 1 a 15 segundos. Utilizando o Websocket, a cada nova mensagem, o servidor repassará a todos os clientes conectados imediatamente. No caso do HTTP, o servidor mantém em memória uma lista com as mensagens, e os clientes, através de um polling de 1s, irão requisitar por atualizações, mandando o índice da última mensagem recebida, através de cookies.

- **Situações:**
  - Clientes simultâneos: 10, 100, 250
  - Tamanho da mensagem: 1KB
  - Intervalo entre atualizações (s): 1-15 randomicamente
  - Polling HTTP de 1 segundo
- **Observar:**
  - Carga do servidor

Após uma média de 15 testes, podemos verificar, através da Figura 16 seguinte comportamento do servidor.

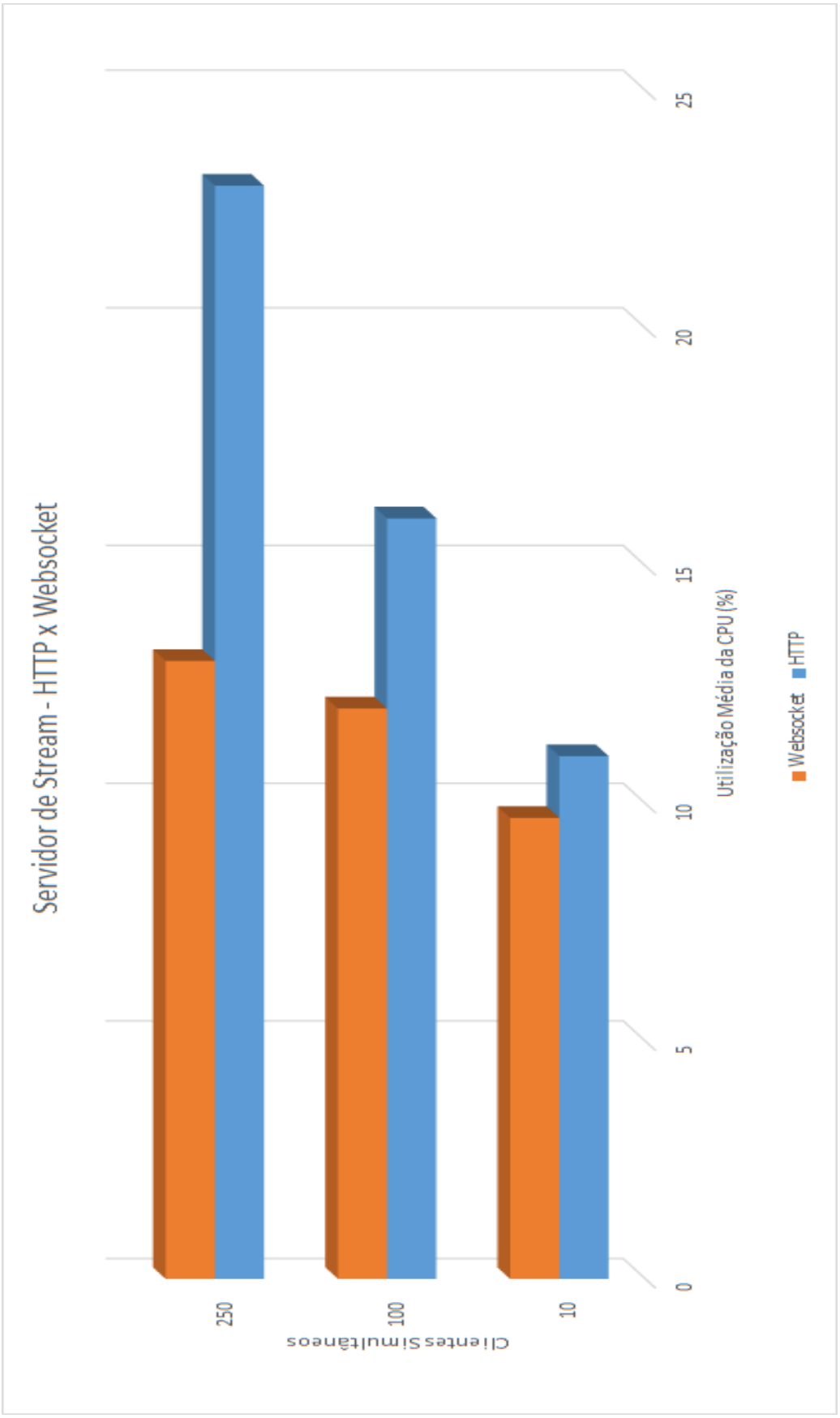

Figura 16 - Teste do servidor de Stream, em ambos os protocolos Websocket e HTTP.

Comparando as duas implementações, podemos observar a vantagem da utilização de uma conexão permanente, fornecida pelo Websocket. Nesta situação, manter as conexões abertas, mesmo nos períodos ociosos, tem um custo computacional médio menor do que utilizando um polling HTTP de um segundo. Sem a necessidade de processar centenas de requisições desnecessárias a cada segundo, houve uma grande economia de recursos.

Outro teste foi realizado, ainda com o servidor de stream, para observar o comportamento do servidor, com diferentes tempos de polling HTTP. No caso anterior, o polling HTTP foi de 1 segundo, e os trechos são enviadas em um intervalo randomico entre 1 e 15 segundos. O polling de 1 segundo é utilizado para garantir um atraso mínimo entre a publicação de uma nova atualização, e a requisição de um cliente. Neste outro teste, foi testado o mesmo sistema, com um número constante de 250 clientes, com diferentes tempos para o intervalo polling. O resultado obtido está representado na Figura 17.

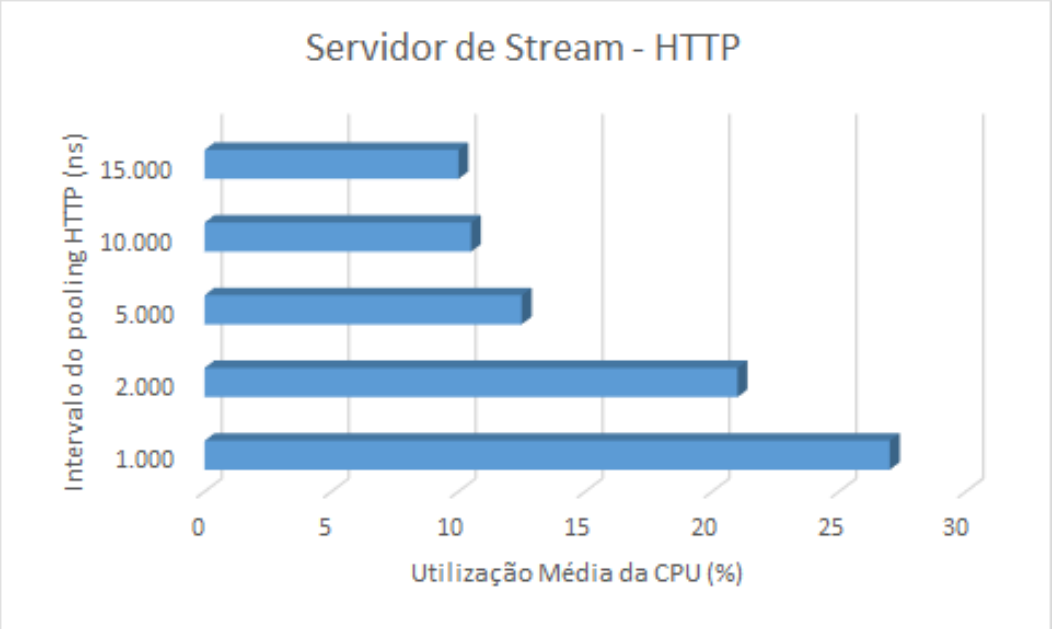

Figura 17 - Resultado do teste de servidor Stream utilizando polling HTTP com diferentes intervalos.

Podemos observar que ao diminuir o intervalo entre as requisições, a taxa de utilização do servidor cai vertiginosamente, onde 250 clientes fazendo polling de 5s, tem um custo equivalente ao de 100 clientes conectados via Websocket.

Neste contexto, a utilização do HTTP passa a ser mais vantajosa que a implementação com Websocket. Se o sistema a ser desenvolvido não perde a sua qualidade,

ou até mesmo a sua funcionalidade, caso as mensagens sejam entregues aos clientes algum segundos após a sua real publicação, ao utilizar o Websocket se está pagando um preço desnecessário, sendo o polling a alternativa com menor custo computacional.

5.4.3 Servidor de Chat

Neste caso, clientes enviam mensagens, que devem ser encaminhadas a todos os outros clientes conectados. Utilizando o Websocket, o envio e o recebimento, é feito utilizando a mesma conexão, e a mensagem é retransmitida aos outros clientes imediatamente ao recebimento. Com o HTTP, uma requisição do tipo POST é utilizada para enviar a mensagem, e um polling com uma requisição GET, consulta o servidor sobre novas mensagens.

- **Situações:**
  - Clientes simultaneos: 10, 100, 250
  - Tamanho da mensagem: 100B
  - Intervalo entre envio de mensagem pelos clientes (s): 1-5s randomicamente
  - Polling HTTP de 1 segundo
- **Observar:**
  - Carga do servidor

Os resultados obtidos estão representados nas Figuras 18 e 19.

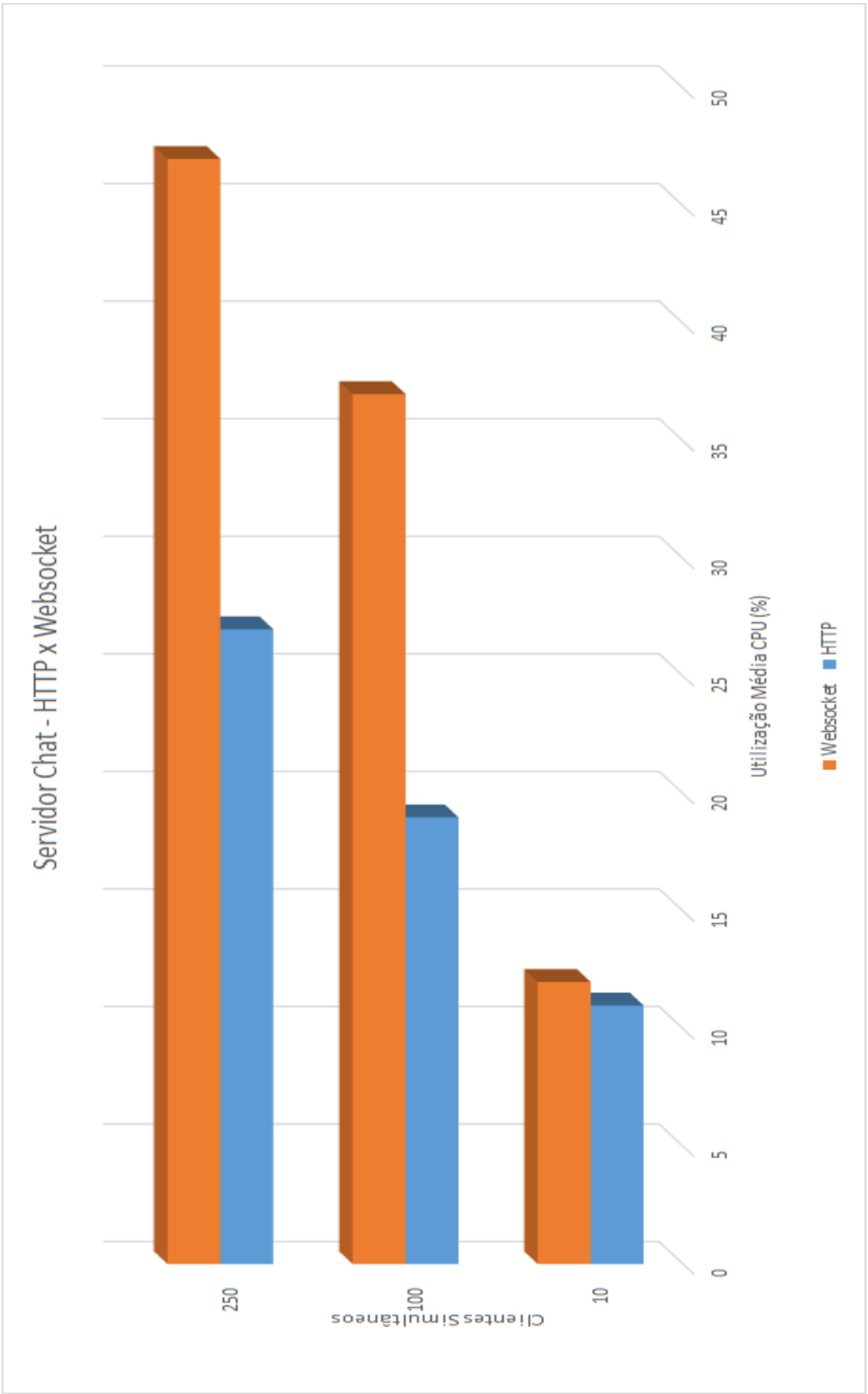

Figura 18 - Teste do servidor de chat, em ambos os protocolos Websocket e HTTP.

Analisando as Figura 18, pode se observar um custo maior quando utilizada a implementação com o protocolo Websocket. Isto se deve a quantidade de mensagens trocadas neste caso, chegando a 100 mensagens por segundo com 250 clientes simultâneos, o que gera o envio de 25000 mensagens por segundo do servidor aos clientes. Quando utilizado o polling HTTP, apenas uma mensagem por segundo, para consulta de todas as novas mensagens, é realizada.

Como no teste anterior, o polling HTTP obteve um custo mais baixo. Entretanto, em uma implementação real, até mesmo um intervalo de 1 segundo, seria inaceitavel para a funcionalidade de um sistema em tempo real, como por exemplo um jogo online. Observando este teste, e o anterior, podemos concluir que a utilização do Websocket tem uma grande vantagem na qualidade de um sistema de tempo real, com envio e recebimento assincrono de mensagens, porém paga-se esta funcionalidade com um custo computacional mais elevado.

Outro teste, visando demonstrar a escalabilidade do serviço, foi realizado nas mesmas circunstâncias, porém com o Load Balancer encaminhando as requisições para dois servidores Twisted, idênticos ao do teste anterior. Os resultados obtidos estão na Figura 19.

Como era esperado, a utilização média dos servidores caiu. Apesar da quantidade de mensagens enviadas ainda seja a mesma, a concorrência entre clientes por servidor é dividida em duas. Independende da implementação utilizada, aplicando esta metodologia, podemos intercomunicar clientes mesmo que não estejam conectados ao mesmo host diretamente, com o custo de se manter um serviço de mensagens em execução. Desta forma, o limíte máximo de utilização de uma aplicação, está na quantidade de computadores que podem ser utilizados como hosts do sistema, independente do protocolo utilizado.

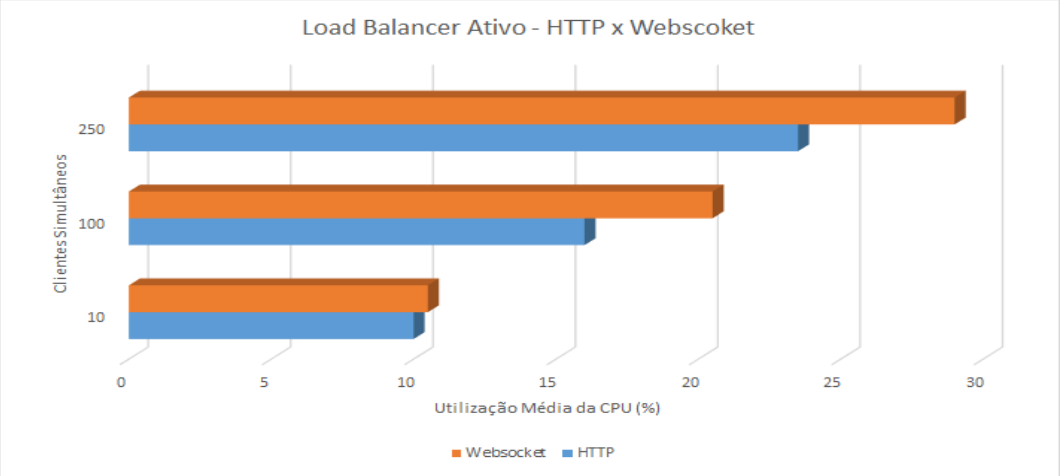

Figura 19 - Resultado do monitoramento de um dos dois servidores ativos em cada teste, com o Load Balancer ativado.

5.5 CONSIDERAÇÕES SOBRE OS TESTES

Todos os testes foram realizados, utilizando instancias do tipo micro, da Amazom EC2. Estas máquinas têm uma configuração bastante simples, com 600MB de memória RAM, e pouco poder de processamento. Esta configuração foi escolhida, primeiramente pelo preço acessível, e por facilitar que uma quantidade relativamente pequena de clientes, causasse um stress no processamento.

Durante os testes, a memória utilizada também foi monitorada. A sua variação acompanhou uniformemente a variação do tamanho da mensagem, igualmente para ambos os protocolos, motivo pelo qual estes dados foram omitidos das comparações finais.

Outro fator não monitorado explicitamente, mas com relevância para comparação das implementações, é a quantidade de dados trafegada.

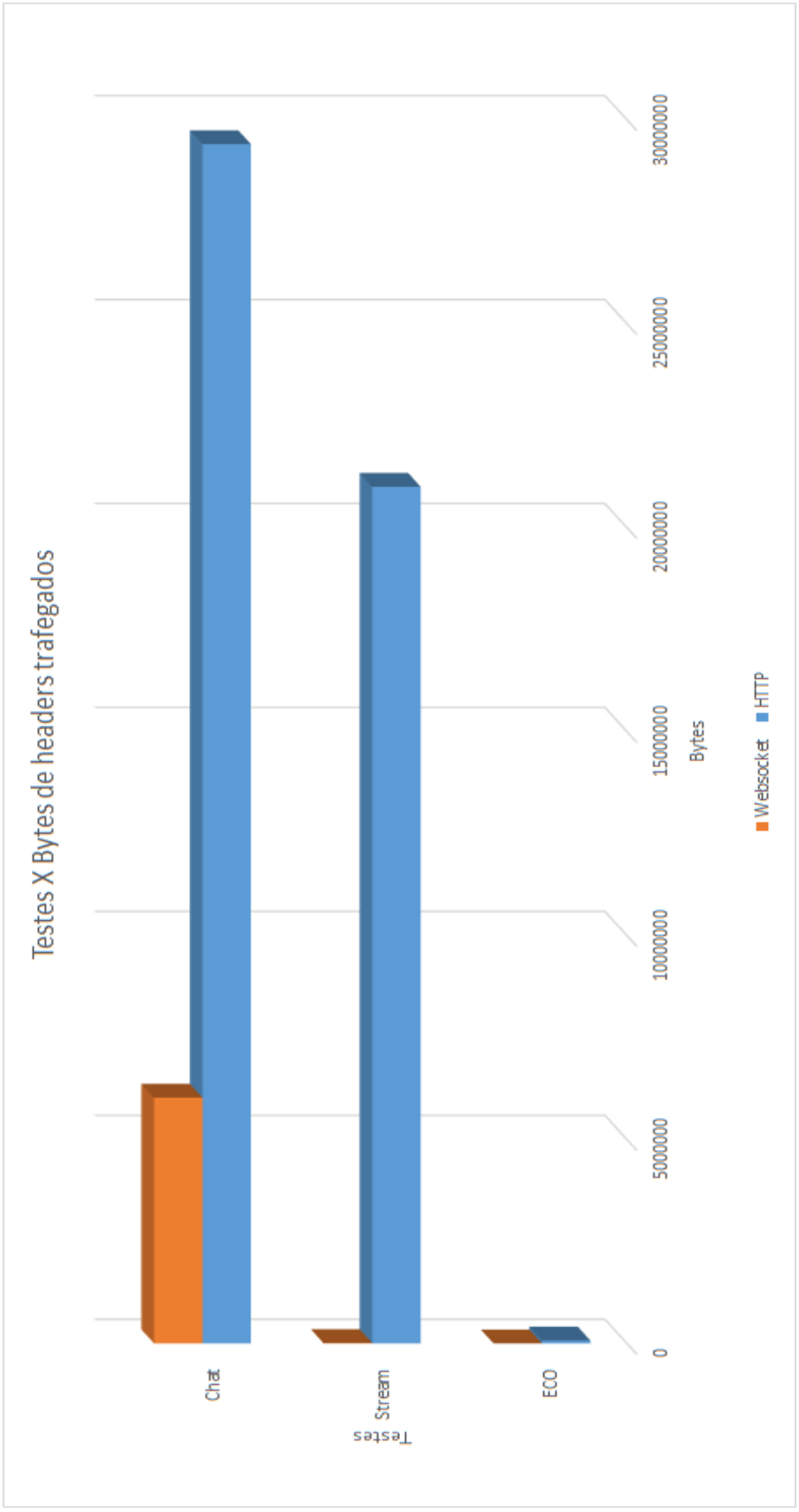

Figura 20 - Tamanho total em bytes dos cabeçalhos trafegados em cada teste, com os diferentes protocolos.

A Figura 20 mostra a quantidade, em bytes, de dados trafegados em cada teste, apenas nos headers das mensagens, nas duas implementações. No caso dos testes Stream e Chat, o resultado representa a situação com 250 clientes e polling de 1 segundo, durante os 2min de teste. Este resultado foi obtido fazendo a média de envios e recebimentos de mensagens, tomando um header HTTP de 700B e um Websocket de 2B, o mesmo tamanho utilizado no Capítulo 2 deste trabalho.

Como pode ser observado, o header binário, de tamanho mínimo, utilizado pelo protocolo websocket, ofereçe uma grande economia na transmissão de dados, mesmo em situações com maior troca de mensagens, como no caso do servidor de Chat. A implementação com websocket teve um total de 3 milhões de mensagens trocadas, enquanto a que utiliza o polling HTTP, apenas 42 mil. Mesmo assim o tamanho total em bytes dos cabeçalhos trafegados, teve uma redução de 28MB, na implementação com polling, para 5MB na implementação que utiliza websocket.

6 CONCLUSÕES

Na aplicação dos diferentes protocolos, observam-se vantagens em ambos, quando diferentes critérios são considerados. A utilização de um canal de comunicação bilateral entre cliente e servidor, apresenta a vantagem do envio de atualizações do servidor ao cliente, sem a necessidade de uma requisição de consulta, o que resulta em atualizações em tempo real mais precisas, do que as obtidas com a técnica de polling HTTP, como pôde ser observado com o resultado da análise do servidor de eco, na figura 15. Por outro lado, a concorrência gerada por manter diversas conexões abertas simultaneamente, causada pela implementação com o protocolo Websocket, tem um custo maior para o servidor, como demonstrado na figura 18.

Atualmente, em aplicações web onde existe uma grande troca de mensagens, e o intervalo entre atualizações influencia diretamente na qualidade do serviço, como por exemplo em jogos com vários jogadores online, são utilizados plugins para criar o canal de comunicação. O Websocket substitui essas ferramentas, ou pelo menos irá substituir definitivamente, quando vir a se tornar de fato um padrão oficial da internet, com a vantagem de ser independente de plataforma, e sem a necessidade de instalação de nenhum software.

A utilização indiscriminada do Websocket para qualquer caso, onde exista comunicação cliente-servidor, pode causar um custo desnecessário para o sistema, diminuindo a quantidade de clientes simultâneos por host que a aplicação consegue atender, como pôde ser observado no teste de stream, nas figuras 16 e 17 . Uma avaliação sobre as características e requisitos do sistema a ser desenvolvido, deve ser levada em consideração, antes de decidir o modo de comunicação.

Com isto, podemos concluir que a utilização do protocolo Websocket é uma solução mais adequada, quando a performance de tempo real é o principal critério a ser levado em consideração. Em contrapartida, a utilização do polling HTTP, apresenta uma economia de recursos computacionais, e ainda é a opção com menor custo, quando o intervalo entre atualizações pode ser ampliado sem comprometer o funcionamento da aplicação.

6.1 TRABALHOS FUTUROS

Apesar deste trabalho mostrar casos de aplicações que tendem a ganhar, e a perder, com a utilização do protocolo Websocket, como forma de comunicação cliente-servidor, falta ainda exatidão neste fator. Por ser abstrair a interpretação dos protocolos e a implementação do modelo concorrente, ainda restam dúvidas se pode existir alguma implementação para otimizar a utilização do Websocket, com o custo computacional como principal critério.

Outros testes, implementados em diferentes linguagens de programação e em diferentes arquiteturas computacionais, devem ser feitos, para observar se existe alguma diferença de performance. Do ponto de vista da análise de desempenho, seria interessante a construção de um serviço web, desenvolvido inteiramente em baixo nível, retirando ao máximo a abstração na implementação do sistema.

Um estudo aprofundado sobre sistemas operacionais, especificamente sobre a abstração do hardware de rede, e suporte a sockets, pode também resultar em características que influenciem no desempenho de determinados tipos de aplicações.

REFERÊNCIAS BIBLIOGRÁFICAS

About HTML5 Websockets. Disponível em: <http://www.websocket.org/aboutwebsocket.html>. Acesso em: 29 jun. 2013.

Amazon Web Service. Disponível em <http://aws.amazon.com>. Acesso em: 25 jun. 2013.

COULOURIS, George. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2000

ETZION, O. e NIBLETT, P. *Event Processing in Action*. Manning Publications, 2010.

FETTING, A. *Twisted Network Programming Essentials: Developing With Python's Event-driven Framework*. Editora O'Reilly Media, 2005

FETTE, I.; MELNIKOV, A. *RFC6455 - The WebSocket Protocol*. Disponível em: <http://tools.ietf.org/html/rfc6455> Acesso em: 20 jun. 2013.

HICKSON, Ian. *The WebSocket API - W3C Candidate Recommendation*. Disponível em: <http://www.w3.org/TR/websockets/>. Acesso em: 20 set. 2013.

HTTP - Hypertext Transfer Protocol Overview. Disponível em: <http://www.w3.org/Protocols/>. Acesso em: 29 jun. 2013.

LEINER, Barry et al. *A Brief History of the Internet. The Internet Society.* Versão 3.32. Última revisão: 10 dez. 2003. Disponível em: <http://www.isoc.org/internet/history/brief.shtml>. Acesso em: 29 jun. 2013.

LUBBERS, P. e GRECO, F. *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web*. Disponível em: <http://soa.sys-con.com/node1315473>. Acesso em: 29. Jun. 2013.

POSTEL, Jon. *RFC793 - TRANSMISSION CONTROL PROTOCOL.* Disponível em:< http://www.ietf.org/rfc/rfc793.txt>. Acesso em: 29 jun. 2013.

STOMP. Disponível em: <http://stomp.github.io/>. Acesso em: 29 jun. 2013

SOCOLOFSKY, T.; KALE, C. *RFC1180 - A TCP/IP Tutorial*. Disponível em: <http://tools.ietf.org/html/rfc1180>. Acesso em: 20 jun. 2013.

TANENBAUM, Andrew S. *Redes de Computadores*. 5ª edição. Editora: Person Prentice Hall, 2011.

TANENBAUM, Andrew S. *Sistemas Operacionais Modernos*. 3ª edição. Editora: Person Prentice Hall, 2010.

TOSCANI, S.S.; OLIVEIRA R.S. e CARISSIMI A.S. *Sistemas Operacionais e Programação Concorrente*. Editora Sagra-Luzzatto, 2003

University of Southern California. *RFC791 - INTERNET PROTOCOL*.
Disponível em: <http://www.ietf.org/rfc/rfc791.txt>. Acesso em: 20 jun. 2013.