Licenciatura em Sistemas e Tecnologia da Informação

Desenvolvimento de uma Framework Real Time Web para HTML5

Projeto Final de Licenciatura

Elaborado por Pedro Manuel da Conceição Fernandes

Aluno nº 20101345

Orientador: Professor Doutor Marcirio Silveira Chaves

Coorientador: Professor Sérgio Rodrigues Nunes

Barcarena

Junho 2013

Universidade Atlântica

Licenciatura em Sistemas e Tecnologia da Informação

Desenvolvimento de uma Framework Real Time Web para HTML5

Projeto Final de Licenciatura

Elaborado por Pedro Manuel da Conceição Fernandes

Aluno nº 20101345

Orientador: Marcirio Silveira Chaves

Coorientador: Professor Sérgio Rodrigues Nunes

Barcarena

Junho 2013

O autor é o único responsável pelas ideias expressas neste relatório

Agradecimentos

Às minhas pessoas queridas.

Desde já agradeço à paciência de quem vai ler este texto, com certeza, será um texto informal e que contem muitos parágrafos e virgulas.

Resumo

Desenvolvimento de uma Framework Real Time Web para HTML5

Este trabalho visa o estudo do estado de arte do paradigma *real time web*, bem como o desenvolvimento de uma *framework* utilizando as principais técnicas existentes atualmente para o efeito. O objetivo essencial do conjunto de tecnologias real time web é a criação de uma web que seja transparente ao utilizador sem que este tenha a necessidade de constantemente atualizar os seus conteúdos. Este tipo de tecnologias oferece uma alternativa às tradicionais propostas de apresentação de dados, melhorando significativamente a performance dos sistemas web, bem como a redução de custos relacionados com a manutenção dos mesmos. Neste trabalho aborda-se técnicas e tecnologias como o paradigma *Publish*/*Subscribe* e tecnologias como o *Comet* ou os *WebSockets*. Para isso discute-se conceitos, requisitos necessários e pesquisas sobre temas e trabalhos relacionados. Foi desenvolvido um inquérito sobre o tema, bem como o desenvolvimento de testes unitários, testes de desempenho e uma prova de conceito que demonstra o desempenho e funcionalidade da tecnologia.

Palavras-chave: Real time, web, publish, subscribe, *AJAX*, *Comet*, *pub/sub*, *publisher*, *subscriber*, *internet* e sistemas distribuídos

Abstract

Development of Real Time Web Framework for HTML 5

This work aims to study the state of the art real time web paradigm, as well as the development of a framework using the main techniques currently exist for this purpose. The main objective of the real time web technology is creating a website that is transparent to the user without him having the need to constantly update their content. This type of technology offers an alternative to traditional proposals of data, greatly improving the performance of web systems, as well as the reduction of costs related to maintenance. In this work, I will address techniques and technologies like a paradigm *Publish/Subscribe* and technologies like *Comet* or *WebSockets*. To that, I will discuss concept, requirements, research issues, and related work. We developed a survey about the topic, as well as the development of unit testing of performance testing and a proof of concept to prove the performance and functionality of the technology.

Key-words: Real time, web, publish, subscribe, AJAX, Comet, pub/sub, publisher, subscriber, internet and distributed systems

Índice

Índice de figuras

Índice de tabelas

Lista de abreviaturas e siglas

AES *Advanced Encryption Standard*

AJAX *Asynchronous JavaScript and XML*

API *Application Programming Interface*

CSS *Cascading Style Sheets*

HTML *HyperText Markup Language*

HTTP *HyperText Transfer Protocol*

IRC *Internet Relay Chat*

OOP *Object-oriented Programming*

PaaS *Plataform as a Service*

SD *Sistema Distribuído ou Distributed System (DS)*

SQL *Structured Query Language*

TCP *Transmission Control Protocol*

TLS *Transport Layer Security*

RDF *Resource Description Framework*

RSS *Really Simple Syndication*

TI *Tecnologias da Informação*

URL *Uniform Resource Locator*

WS *WebSockets*

XML *eXtensible Markup Language*

1. Introdução

Mason, Diretor Executivo da *MuleSoft* e evangelista de tecnologias *Cloud Computing*, afirmou no seu *Blogue* "eu tenho um background financeiro, e para mim, o tempo é medido em milissegundos" (Mason, 2011). Esta citação expressa bem o objetivo da *real time web* e das tecnologias envolvidas para atingir um objetivo principal: entregar conteúdos o mais rápido possível, através de canais fiáveis de comunicação.

A *real time* web é um conjunto de tecnologias e boas práticas que permitem ao cliente (navegador web) receber informações o mais rapidamente possível após a publicação pelo autor, sem que seja requerido um pedido por parte do *software*. Em uma entrevista, (Fromm, 2008) descreveu a *real time web* como: "*It Happens Without Waiting*" acontece sem que se espere.

Empresas estabelecidas no mercado, e.g. a *Google*, gastam partes substanciais do tempo de desenvolvimento das suas tecnologias neste tipo de padrões. Um exemplo prático são os produtos *GTalk* e *Gmail* que assentam sobre tecnologias *real time*, utilizando práticas de programação como o *AJAX* ou o *Cometd*, por vezes denominado *Reverse AJAX*). Essas tecnologias têm um efeito positivo e determinante sobre determinados produtos e serviços, como aplicações financeiras ou o acompanhamento em tempo real da bolsa de valores, ou mesmo as apostas desportivas que funcionam baseadas em "*ticks*". "Este tipo de aplicações precisam hoje utilizar complexos sistemas próprios para atender suas necessidades de curta latência, e por isso seriam altamente beneficiados ao poderem utilizar um protocolo web, padronizado e com implementações livres e facilmente disponíveis, sem abrir mão dos seus requisitos" (Varela & Stanley, Implementação e análise da utilização de websockets em sistemas computacionais, 2012).

Outras aplicações práticas são os *web sites* de notícias onde é importante obter a notícia ao segundo, ou mesmo outras aplicações como os leilões online em tempo real (Rumpe & Wimmel, 1999). Além disso, são vantajosas as aplicações em *Dashboards* ou sistemas de comunicação *online* como o *Facebook* ou o *Twitter* (Gilmour & MG, 2009).

O *real-time* tornou-se viável com o aparecimento do *HTML5*, bem como das implementações de *Web Sockets* disponíveis atualmente através de *JavaScript*. Anteriormente, o real-time era desenvolvido através de técnicas como o *Cometd* ou *reverse AJAX*, sendo "esta a tecnologia na qual se baseia a *web real time"* (Carbou, 2011). O seu funcionamento é simples de perceber, ao contrário do *AJAX* em que cada pedido ao servidor, por parte do cliente, é recebido uma resposta por parte do servidor. No *Cometd* é criada uma conexão que nunca é fechada sendo que o servidor envia periodicamente, ou quando necessário, uma atualização ao cliente sem que este necessite de o pedir.

1.1. Motivação

Uma das motivações para o desenvolvimento deste trabalho é a frase proferida por Kirkpatrick, "ainda estamos a viver os primeiros tempos. Entrega em tempo real de informação deve se tornar omnipresente, um requisito para qualquer *site* ou serviço." (Kirkpatrick, 2012)

Eugster et. al. "defendem que o paradigma *publish/subscribe* é a chave para a construção de aplicações móveis *ad-hoc*." (Eugster, Holzer, & Garbinato, 2005). Já a Microsoft, nos seus documentos de apoio aos programadores, refere um problema que existe nas arquiteturas distribuídas atualmente: "Como uma aplicação que funciona em uma arquitetura integrada consegue enviar mensagens ao recetor, aplicação, apenas do seu interesse, de forma anónima, sem conhecer a sua origem?" (Microsoft patterns & pratices, 2012).

No momento em que vivemos uma mudança bastante profunda nos paradigmas de desenvolvimento aplicacional, com o surgimento das denominadas *Apps*[1] para dispositivos móveis e muito recentemente para os sistemas operativos Windows e *MacOS*, a recepção de mensagens é crucial para manter o utilizador informado e assim ajudar a obter conteúdos numa fracção de segundo.

Neste sentido existe a necessidade de criar protocolos mais rápidos e paradigmas de desenvolvimento que permitam cortar os tempos de latência dos protocolos de rede actuais.

No mercado existem diversas implementações desta tecnologia, a maioria delas fechadas e monolíticas (sem qualquer suporte ao programador) funcionando como *proxies* entre um servidor e um cliente, trabalhando como um distribuidor de mensagens entre sistemas. Entre estas implementações as mais conhecidas são a *APE project*, solução *OpenSource*, que é um servidor de páginas assíncrono, que recebe e distribui mensagens entre o cliente e o servidor.

Outra solução que funciona no modelo de *PaaS*[2] é o *Pubnub.com*. Esta solução tal como o *APE project* é um distribuidor de mensagens, mas atua como um serviço em nuvem em que o aderente paga por uma subscrição que contêm *n* mensagens.

[1] *Apps* - Aplicações inicialmente desenvolvidas para os dispositivos móveis e que são executadas de uma forma virtualizada por parte dos sistemas operativos.

[2] Plataforma como serviço (*PaaS*) é uma categoria de serviços que fornecem uma plataforma de computação e uma pilha de solução como serviço de computação em nuvem.

1.2. Objetivos e Perguntas de Investigação

O objetivo deste trabalho é o desenvolvimento de uma *framework* cliente/servidor que permita a implementação genérica da tecnologia *real time web* para programadores através de uma *Application Programming Interface (API)* intuitiva. Este *framework* será validado através do desenvolvimento de um protótipo funcional.

Este trabalho também visa responder às seguintes perguntas de investigação:

- Será a *real time web* um paradigma a ter em consideração na implementação de soluções de internet?
- Como se podem usar as tecnologias atuais para melhorar a comunicação de aplicações utilizando o paradigma de *publish/subscribe*?
- Qual é a opção de tecnologia mais adequada na utilização de *real time web*, o *reverse-ajax* ou o *WebSocket*?

As respostas a estas perguntas serão sustentadas através da verificação do estado da arte das tecnologias *pub/sub* e *WebSockets*, a análise face a outras tecnologias e técnicas existentes que rivalizem com a mesma, bem como as vantagens e desvantagens de cada uma.

1.3. Metodologia

Esse trabalho é um estudo sobre a *real time web* e, por consequente, o paradigma *Publish/Subscribe* (Eugster, Felber, Guerraoui, & Kermarrec, 2003) e como este pode ser utilizado na plataforma *web* atual.

A abordagem utilizada neste trabalho consiste nos seguintes pontos: i) fazer o levantamento da literatura sobre a real time web ii) pesquisar artigos e trabalhos sobre o paradigma *Publish/Subscribe*; iii) estudar os trabalhos relacionados levantando seus pontos fortes e fracos; iv) levantar os requisitos que devem ser atendidos pelo *Publish/Subscribe* no cenário da *real time web*; v) definir uma *framework* que implemente o paradigma *Publish/Subscribe* para utilizar no desenvolvimento de solução web e plataformas móveis; vi) desenvolver uma aplicação que utilize o paradigma definido e vii) avaliar a sua utilização, através de testes unitários, consultas a programadores e estatísticas de utilização na implementação de soluções.

A metodologia é exploratória e experimental. A metodologia exploratória tem como objetivo a familiarização com o fenómeno que está a ser investigado, de modo que a pesquisa subsequente possa ser concebida com uma maior compreensão e precisão. A metodologia experimental entra como a segunda fase do projeto, em que se verificam as teorias apresentadas durante a investigação exploratória.

1.4. Estrutura do Documento

Este documento encontra-se organizado da seguinte forma: No capítulo 2 são apresentados os conceitos fundamentais para a correta compreensão do problema tratado, os estudos relacionados com as áreas de *real time web*, bem como as tendências e adoções de sistemas existentes. Também se aborda as soluções existentes no mercado e as melhorias que este documento pretende retratar, por forma a preencher algumas lacunas nos sistemas atuais no mercado.

No capítulo 3 descrevem-se a análise dos processos e definem-se os requisitos do sistema a implementar. Ainda neste capítulo aborda-se as tecnologias usadas na implementação do sistema e descreve-se o desenho lógico e físico da solução bem como a sua implementação.

O capítulo 4 verifica a avaliação e validação do modelo apresentado bem como as conclusões que se retiram da avaliação do sistema. No capítulo 5 são retiradas as conclusões finais dos resultados tendo por base a implementação efetuada, comparando-os com a análise prévia efetuada e descrita nos capítulos anteriores deste documento. Em seguida são apresentadas as limitações do trabalho e as propostas para trabalhos futuros sobre o mesmo tema.

2. Conceitos Básicos e Trabalhos Relacionados

Neste capítulo será feita uma revisão bibliográfica da *real time web*, bem como o paradigma *Publish/Subscribe*, modelo adotado na construção da *framework* deste trabalho, apresentando por fim as principais implementações existentes na literatura.

2.1. A *real time web*

De acordo com Marshall Kirkpatrick, "a real time web é um paradigma baseado na busca de informação para o utilizador o mais rapidamente possível - ao contrário do que existe atualmente em que o *software* requer uma verificação periódica por atualizações." (Kirkpatrick, 2012).

Numa entrevista a Paul Buchheit, um dos criadores do *Gmail*, e fundador da *FriendFeed*, ele afirma que "*Real time* oferece uma maneira fácil e eficiente de conversar e esta conversa acontece numa questão de minutos ou segundos. É a similaridade entre a diferença de uma chamada telefónica e uma série de mensagens por voz. – A chamada por voz ocorre em tempo real, logo toda a conversação pode ser concluída muito rapidamente. O que se torna crítico quando o tempo é escasso – por exemplo. No nosso grupo interno no *FriendFeed* eu coloco uma mensagem sobre executar uma atualização no sistema, e é importante que todos vejam essa mensagem."

Kirkpatrick afirma que a *real time web* "está a ser implementada em redes sociais, de procura de notícias, e em outros sítios - tornando essas experiências mais como mensagens instantâneas e facilitar inovações imprevisíveis" (Kirkpatrick, 2012). Sendo que o "facilitar inovações imprevisíveis" que o autor refere, são as potencialidades que se podem desenvolver e que ainda não foram exploradas.

Mason vai mais longe, especificando algumas aplicações que atualmente usam este tipo de tecnologia i) *Salesforce*; ii) *Twitter*, oferece atualizações de estado em tempo real; iii) *Facebook*, permite subscrever as mudanças de estado no seu gráfico social; iv) *Superfeedr*; puxa todos os tipos de *feed* em uma única API; v) *Digg*; todo o tipo de observações e comentários vi) *Instagram*; atualização em tempo real de fotos; (Mason, 2011).

As arquiteturas deste tipo de sistemas baseiam-se na arquitetura de computação distribuída *(ver secção 2.5)* em que "o modelo de comunicação *publish/subscribe (ver secção 2.3)* é baseado na troca assíncrona de mensagens, conhecidas como eventos." (Silvestre, 2005).

A "comunicação baseada em eventos é importante para aplicações sensíveis ao contexto pois permite a deteção de mudanças no meio de execução sem que seja necessária uma

consulta periódica para obtenção do contexto corrente." (Baptista, Endler, Rubinsztejn, & Sacramento, 2005), ao contrário do que acontece hoje em dia com os sistemas web baseados em tecnologia *AJAX*, em que existe uma obrigatoriedade de obter informação do servidor num determinado intervalo de tempo ou a pedido do utilizador, tornando na maioria das vezes a informação recebida obsoleta.

Vale a pena referir ainda que "O modelo de *real time web* é normalmente implementado para leitura de dados. Este é usado para fornecer dados aos consumidores, não para fazer escrever ou excluir" (Mason, 2011). Estes sistemas, *real time web*, são "também o mecanismo de interação mais apropriados para aplicações móveis devido ao desacoplamento entre cliente/servidor" (Baptista, Endler, Rubinsztejn, & Sacramento, 2005), o que o torna ideal para o desenvolvimento de sistemas em que a apresentação seja importante.

Os "benefícios imediatos incluem o envolvimento dos utilizadores aumentando ("fluxo") e diminuindo tráfego entre servidores" (Kirkpatrick, 2012). Outra principal característica "é o desacoplamento espacial (ver a Figura 1) e temporal (ver a Figura 2), dos participantes da comunicação. Como eles interagem através de um intermediário, não necessitam se conhecer, e nem estar ativos simultaneamente para a troca de mensagens, ficando a cargo da infraestrutura *pub/sub* o armazenamento e distribuição das mensagens" (Baptista, Endler, Rubinsztejn, & Sacramento, 2005). (Silvestre, 2005) Refere ainda que este desacoplamento entre os produtores e os consumidores dá "mais flexibilidade e extensibilidade às aplicações que utilizam esse modelo, pois a inclusão de novos produtos e consumidores se torna simples".

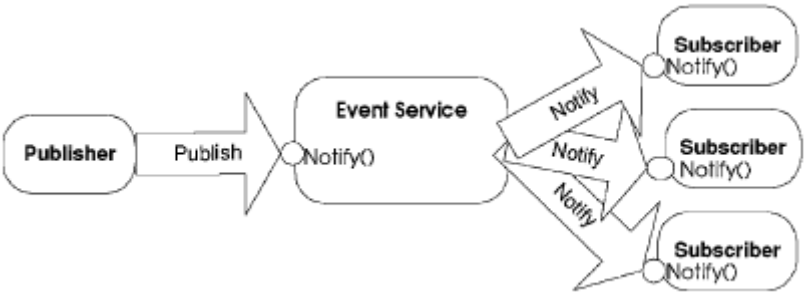


Figura 1: Desacoplamento Espacial (Eugster, Felber, Guerraoui, & Kermarrec, 2003)

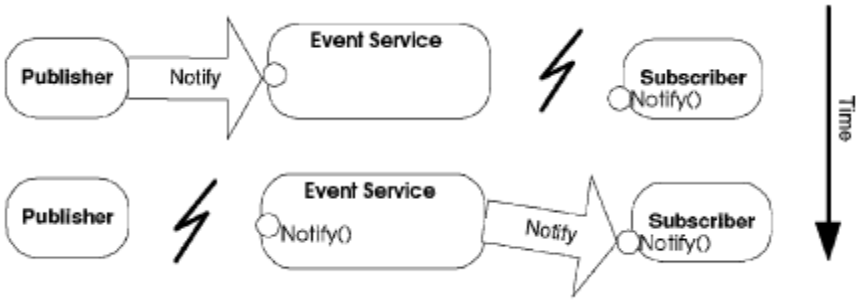


Figura 2: Desacoplamento Temporal (Eugster, Felber, Guerraoui, & Kermarrec, 2003)

Este modelo de desacoplamento em que a informação circula livremente tem alguns problemas. Segundo Chakinala *et al (2007)* "um problema natural neste tipo de cenário é encontrar um desenho de um mecanismo eficiente que permita a distribuição de dados desde a fonte até a origem satisfazendo todos os requisitos da comunicação".

Mas será isto, apenas mais uma "*buzzword* [3]" da indústria? A empresa *AYR Consulting*, numa *Research Note* editada em 2011, diz que "é uma verdade dizer que a web tornou-se um padrão de interação do utilizador … ao mesmo tempo os programadores demonstram o "reino de possibilidades" para aos utilizadores, e eles estão a exigir semelhantes recursos nos seus portais e *sites*" (AYR Consulting, 2011).

Tais possibilidades contam com as mais recentes novidades como: a *Cloud Computing* que têm vindo a ter um crescimento exponencial (ver a Figura 3) ao longo dos últimos anos e que caminha para ser a plataforma de eleição para suportar a web no futuro.

[3] Buzzword – É uma palavra ou frase usada para impressionar, ou uma expressão que está na moda.

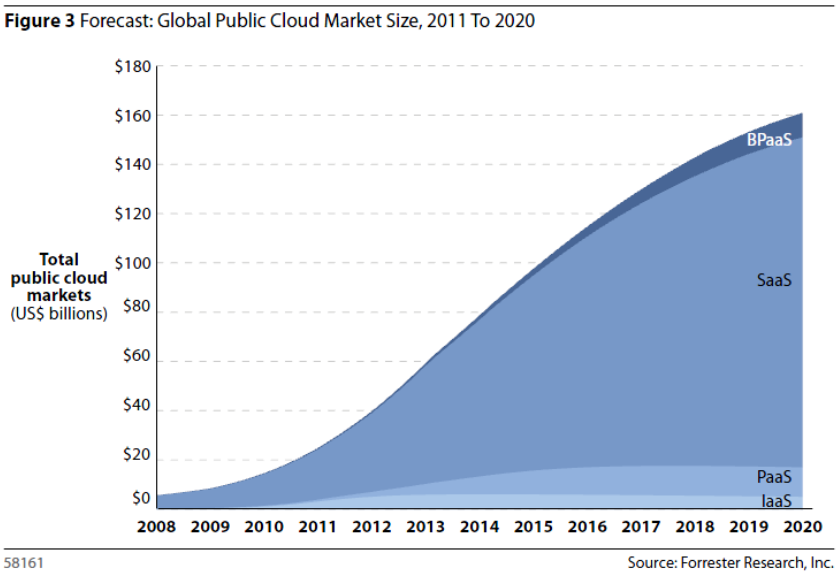


Figura 3: Crescimento da Cloud Computing ao longo dos anos, (Dignan, 2011)

Outro fator tecnológico que tem vindo a aumentar a aceitação de tecnologias como *real time* são as novidades da web semântica como a recente especificação *HTML 5.0 (Hypertext Markup Language)*, o *RDF (Resource Description Framework)* e o *RSS (Really Simple Syndication)*. A adoção do *HTML 5.0* e do *CSS 3.0 (Cascading Style Sheets)* trouxe novos produtos ao desenvolvimento de aplicações. Pois com esta nova implementação surgiu uma enorme oferta de *frameworks* web entre elas, as mais conhecidas o *jQuery* e o *Underscore*, que permitiram aos fornecedores transportarem os seus aplicativos para web mantendo a mesma experiência de utilização que tinham nas suas aplicações *desktops*.

Estas migrações aplicacionais foram alavancadas pelo crescimento de tecnologias como o *AJAX* em 2000 e pelo melhoramento do fornecimento de acesso à Internet por parte dos provedores com o aparecimento do cabo e da *ADSL (Asymmetric digital subscriber line)*.

Mas o atual modelo contem imperfeições, os utilizadores navegam na Internet pensando que estão a navegar em tempo real e que obtém os seus conteúdos instantaneamente. Isto é falso na maioria dos casos, devido à atual estrutura da *WWW* e do *HTTP*[4]. A Figura 4 apresenta uma ilustração de como a disponibilização do serviço funciona pelo protocolo *HTTP*:

[4] HTTP – (Protocolo de Transferência de Hipertexto). Protocolo utilizado para transferências de páginas Web de hipertexto.

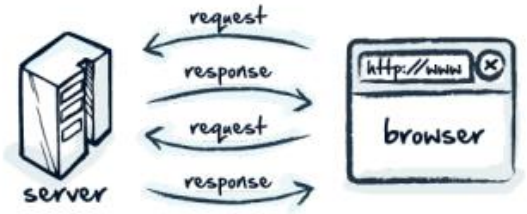


Figura 4: Modelo de comunicação de um serviço pelo protocolo HTTP, (AYR Consulting, 2011)

O navegador ("*browser*") do utilizador contacta o servidor web, enviando um pedido de «*request*» por conteúdos; o servidor web retorna o pedido de informação com um «response» com a informação pedida pelo navegador do cliente, na maioria dos casos este funcionamento é através dos denominados *postbacks* ou de *html forms*. No modelo apresentado na Figura 4, é o navegador que controla o modelo, sendo que o servidor tem um lugar passivo no processo visto que apenas envia a informação pedida pelo cliente.

Como foi referido este não é um processo em tempo real, visto que é necessária interação por parte do cliente para que alguma ação possa acontecer. A Internet atual, com as ligações de alta velocidade, torna este processo mais parecido ao *real time* tornando-o uma espécie de *semi real-time*.

A empresa *AYR Consulting* refere que uma alteração ao modelo atual da forma como a web é transmitida teria grande impacto na performance, o desenvolvimento utilizando plataformas de desenvolvimento atuais como o *HTML 5.0*, *frameworks JavaScript*, bem como o acesso a servidores de mensagens tornaria a web mais assíncrona e mais fluida aos utilizadores.

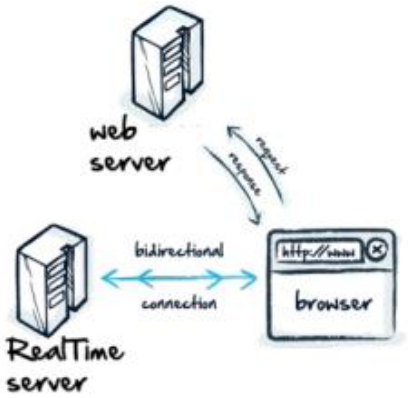


Figura 5: Comunicação de um serviço através de um relay real time, (AYR Consulting, 2011)

A alteração proposta pelo modelo, na Figura 5, tem como objetivo introduzir um serviço externo de mensagens que permite um alívio na carga dos servidores de internet, facilitando assim a comunicação com os diversos intervenientes, além de permitir uma comunicação em tempo real e tornar as aplicações mais responsáveis.

Esta proposta de alteração ao modelo atual de *web 2.0* torna-o mais parecido a um sistema distribuído, como foi referenciado no início deste texto, o que claramente tem as suas vantagens (ver secção 2.5), visto ser mais tolerante a falhas e não ter quaisquer dependências entre componentes.

2.2. Os três modelos de valor: Contexto, Automatização e Emergência

O desenvolvimento de soluções que tirem partido da componente "em tempo real", não devem ser implementados em todos os casos, é necessário como em todo desenvolvimento analisar e procurar a melhor solução para cada caso.

Um exemplo é uma simples *Wiki* ou um outro *site* do género em que a informação não é crítica o suficiente para chegar ao segundo. Cada solução deve ser analisada ao pormenor, não faz sentido uma solução como a *Wikipedia*, por exemplo, usar uma solução de *AJAX*, pois os conteúdos não são atualizados ao minuto, nem de hora a hora, a estaticidade neste caso torna-o um sucesso, pois torna a página simples de usar e mais fácil leitura, visto não ter atualizações periódicas que dificultam a leitura.

Por isso, é necessário ter em conta três modelos que são importantes nesta análise.

i) O **contexto** em que a solução e o utilizador estão inseridos;

Segundo o dicionário da Língua Portuguesa, o contexto é um conjunto de características que definem um lugar, estado, uma qualidade, um tom, ou uma particularidade (Porto Editora, 2012). Este é um bom termo para perceber o conceito de contexto em aplicações web pois a mesma é um ambiente de informação, interligado entre si, como define a (Carvalho, 2008): "A web é composta por páginas ligadas entre si... Muitas dessas páginas são assinaturas sociais, como *blogues*, *bookmarks*, *tweets* ou outras ligações que não são pesquisáveis".

Cada conteúdo tem o seu contexto e público-alvo, a informação deve ser apresentada ao utilizador conforme o seu contexto ou este tenha a necessidade de visualizar essa informação.

ii) A **automatização** necessária que obtenha uma melhor vantagem competitiva;

No dicionário da Língua Portuguesa a automatização é descrita como: "É o uso de meios automáticos para se realizar uma determinada atividade" (Porto Editora, 2012).

No contexto de apresentação de dados em tempo real a automatização é um dos elementos mais importantes e deve ser primeiramente tomado em consideração. No contexto de um projeto web em que a informação está constantemente a fluir para o cliente a automatização torna-se um fator importante, pois permite libertar recursos humanos durante a disseminação da informação.

iii) A **emergência** de apresentar informação ao utilizador, tornando-a mais valiosa.

A emergência de apresentação da informação é o valor mais importante na implementação de soluções em tempo real. A necessidade crescente que o utilizador tem em obter a informação, por forma a poder tomar decisões mais rapidamente, aumenta o valor acrescentado desta tecnologia.

2.3. Paradigma *Publish/Subscribe*

O modelo *pub/sub* é usado para situações em que muitos clientes (*Subscribers*) têm que escutar informações publicadas por um ou mais servidores (*Publishers*). O *Publisher* não consegue distinguir entre os assinantes que estão em escuta sobre ele, portanto, envia a informação para todos os assinantes.

Chockler *et al* (2006) definem que "*Publish/subscribe (pub/sub)* é um popular paradigma para o suporte em redes distribuídas, em que existe uma comunicação de muitos para muitos" (Chockler, Melamed, Tock, & Vitenberg, 2006).

O paradigma baseia-se na ideia que "os utilizadores interessados em receber mensagens publicadas em certos tópicos subscrevem os pedidos a esses tópicos de interesse" (Chockler, Melamed, Tock, & Vitenberg, 2006). Outros autores sublinham que "o principal conceito inerente ao paradigma *publish/subscribe* é ver a interação de dois tipos de entidades diferentes: a primeira entidade, o *publisher* consiste na geração de eventos e conteúdos, enquanto a segunda entidade, o *subscriber*, consiste na subscrição de eventos que sejam do seu particular interesse" (Eugster, Holzer, & Garbinato, 2005).

A forma simples de descrever os conceitos em cima referidos é transpô-los para uma tabela de forças *(ver Tabela 1)*, em que o componente é uma parte do paradigma, sendo a responsabilidade a sua ação no sistema e a colaboração o objetivo, a que cada componente se propõe a trabalhar.

Componentes	**Responsabilidades**	**Colaboração**
Infraestrutura de comunicação	Mantém as assinaturas dos subscritores. Inspeciona as informações relacionadas ao tópico ou as informações de conteúdo que estão incluídas em cada mensagem publicada. Transporta a mensagem para os aplicativos subscritos.	O editor publica mensagens. O Assinante assina tópicos e recebe mensagens.
Publisher	Insere informações relacionadas com o tópico ou informações de conteúdo em cada mensagem. Publica a mensagem para a infraestrutura de comunicação.	A infraestrutura de comunicação transporta mensagens para os assinantes.
Subscriber	Se inscreve em um ou mais tópicos ou tipos de conteúdo de mensagem. Consome mensagens publicadas para os tópicos subscritos.	Os transportes de infraestrutura de comunicação publicaram mensagens do editor.

Tabela 1: Comparativo entre Componentes e Responsabilidades/Colaboração

Como é evidente pela *Tabela 1* todos os componentes trabalham de uma forma uniformizada, sendo que cada componente tem uma tarefa atribuída e bem definida. Os principais elementos deste tipo de sistemas são o *Publisher*, *Subscriber* e a Infraestrutura de comunicação responsável pela ligação entre os restantes componentes do sistema.

A Infraestrutura mantém a base de dados de todos os utilizadores ligados à rede, é também a responsável por filtrar as mensagens provenientes do *Publisher* e distribui-las de forma correta ao *Subscriber*, para que este possa usufruir da informação. A comunicação é sempre feita na Infraestrutura, não havendo qualquer conectividade

direta entre os elementos *Publisher* e o *Subscriber*. Desta forma é mantido o anonimato na rede.

Outro elemento importante dentro do sistema em análise é o *Publisher*, que define quais conteúdos estão disponíveis na rede bem como o canal de comunicação em que os mesmos estão disponíveis. O *Publisher* é o componente responsável pela disponibilização da informação, e é quem define os conteúdos a serem partilhados na rede.

Já o *Subscriber* é o componente responsável pela subscrição de conteúdos, através do pedido à infraestrutura dos canais que deseja assinar. Estes canais podem ser variados não estando dependente apenas da subscrição de um único canal. Nas implementações mais atuais o subscriber tem funcionalidades como por exemplo o *Content-Based Subscription* que permite ao subscritor assinar apenas mensagens que circulem num determinado canal e que obedeçam a regras de conteúdos como títulos, parágrafos ou outros elementos que contêm uma mensagem.

2.4. Arquitetura do modelo *pub/sub*

Na arquitetura do modelo, conforme representada na Figura 6, o *pub/sub* assenta sobre quatro camadas lógicas, denominadas: *Network Infrastructure*, *Overlay Infrastructure*, *Event Routing* e *Matching*.

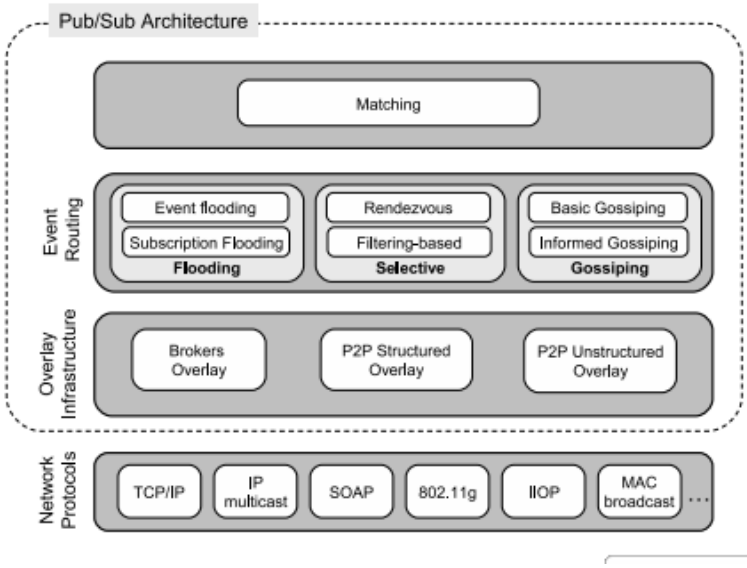


Figura 6: Modelo da arquitetura Publish/Subscribe (Baldoni, Querzoni, & Virgillito, 2009)

A camada de ***Network Infrastructure*** *é* a camada responsável pela comunicação de rede que herda o seu modelo da arquitetura *TCP/IP* de redes, que é responsável por toda a camada de comunicação em qualquer dispositivo que utilize o *TCP/IP*.

No ***Overlay Infrastructure***, podem existir um de três tipos de modelos possíveis. Esta camada é a camada principal da arquitetura *pub/sub*, pois é a que indica qual o tipo de conexão que é feita entre *o publisher* e o *subscriber*. Nas implementações que utilizam o *Broker Overlay*, a distribuição de mensagens é feita por um sistema intermediário, geralmente um serviço que recebe e reenvia mensagens. Neste tipo de estrutura este serviço é o responsável por assegurar a segurança da mensagem, verificando a mensagem através de métodos de criptografia, e enviando ao recetor a mensagem corretamente verificada. Neste tipo de configuração acontecem outras validações que asseguram o correto envio de mensagens, fazendo com que o sistema apenas envie as mensagens que o recetor quer receber, evitando tráfego em excesso para o recetor.

O modelo da arquitetura *Publish/Subscribe* possibilita a utilização de redes *P2P* com *Overlay* para o correto funcionamento do sistema. Segundo (Morimoto, 2008), uma rede *Overlay* é uma rede virtual que é construída sobre outra rede, onde os Nós podem ser conectados através de ligações lógicas ou virtuais, diferenciando apenas o nível na camada física. Por exemplo, tecnologias como a *Cloud Computing*, redes *P2P* ou mesmo a arquitetura cliente/servidor, são denominadas redes *Overlay* pois os seus Nós são executados no topo da *Internet*. Por forma a construir as ligações *Overlay*, utiliza-se *IP Tunneling*, esses túneis de *IP* são ligações ponto a ponto virtuais dando a ilusão de criarmos e dotarmos de uma ligação direta aos *Nós*. A localização dos conteúdos é distribuída na rede Overlay, através de algoritmos que encaminham a informação geridos pela camada do *P2P*.

O *P2P Overlay* é dividido em dois subconjuntos, o *Structured* e o *Unstrutured*. No caso do *Structured* as mensagens são validadas segundo um contrato que é feito entre o *subscriber* e o *publisher*. Este é um contrato ao nível do programador, em que é criada uma interface de mensagens que apenas aceita e envia determinados tipos de conjuntos de informação, evitando assim que o *subscriber* receba informação em excesso ou que possa ser prejudicial ao sistema em causa.

A camada ***Event Routing*** permite ao serviço reconhecer quem são as partes interessadas (clientes) em receber determinada informação, reenviando a informação para o cliente que deseja obter o conteúdo. Assim é assegurado que todos os clientes interessados recebem a informação correta.

O ***Matching*** é uma camada opcional no modelo de *pub/sub* que é implementado em modelos de *Content-Based Publish/Subscribe*. No modelo comum um assinante subscreve um canal (*channel*) por interesse como se de uma conversa de *IRC* se tratasse,

em que o cliente subscreve uma sala de conversação e recebe todas as mensagens que passam por essa sala de conversação. No modelo *Content-Based Publish/Subscribe* é um pouco diferente, pois o subscritor ao contrário de subscrever um canal está antes a subscrever um tópico (*topic*), um tópico não é mais que um título da mensagem ou parte dela, ou seja, o subscritor recebe todos os conteúdos em que o tema seja um determinado tópico.

2.5. Sistemas Distribuídos

Um sistema distribuído (SD) "consiste em vários dispositivos que se comunicam e conseguem compartilhar recursos de sistema: compartilham informação (dados) ou até mesmo recurso de *hardware* ou de *software*" (Tanenbaum, 1995). (Silvestre, 2005) Refere que "um modelo distribuído para o serviço consiste em um conjunto de servidores interconectados, cada um responsável por prover serviços para uma parte dos clientes".

O paradigma *pub/sub* e toda a *real time web* que abordamos neste documento, assentam sobre este mesmo conceito de sistema distribuído, originalmente criado nos laboratórios da ARPA net e na qual se baseia toda a estrutura da Internet moderna.

Segundo (Cao, 2011) este tipo de sistemas têm diversas vantagens, tais como: custos, tolerância a falhas, redundância, segurança, velocidade, confiabilidade, escalabilidade, distribuição inerente e condicional. O mesmo autor define ainda que a segurança é problemática, pois se um subsistema é comprometido pode pôr em causa todos os subsistemas seguintes, causando uma reação em cadeia que por vezes pode ser catastrófica.

Estes mesmos princípios estão presentes nas arquiteturas de *real time web*, precisamente nos modelos mais conhecidos como o *pub/sub* e o *Cometd*. Eles por si só são sistemas distribuídos na sua base genérica, conforme a Figura 7:

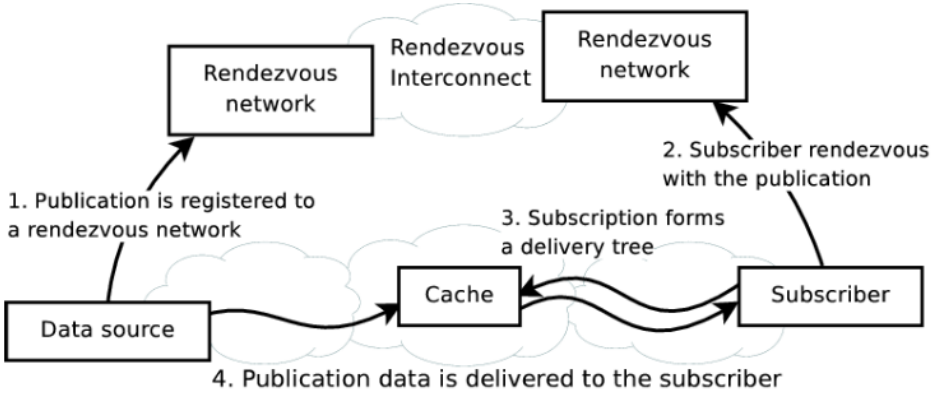


Figura 7: Esquema de uma comunicação pub/sub, segundo (Lagutin, 2011).

A Figura 7 demonstra o conceito de sistema distribuído num sistema de comunicação *pub/sub* através de uma rede *Rendezvous*[5] o subscritor comunica com a *cache* para obter uma resposta de um repositório de dados *(data source)* remoto.

O funcionamento de uma comunicação *pub/sub* é idêntico ao de um sistema distribuído na componente de *software* assentando sobre a mesma definição de conceitos como é demostrado pela lista seguinte (Google Code University, Google Inc., 2011):

- Programa: o código que se escreve;
- Processo: é o que se obtém quando se executa;
- Mensagem: é o que é trocado entre os processos;
- Pacote: é um fragmento de uma mensagem que é trocado na ligação;
- Protocolo: é o formato universal descritivo de como a mensagem deve ser composta;
- Rede: a infrastrutura de máquina que transportam o conteúdo;
- Componente: pode ser qualquer parte do sistema distribuído, mensagem, rede, processo;

Por fim o "sistema distribuído é o todo dos componentes acima indicados que são completamente coordenados por um conjunto de ações em que cooperam de forma sincronizada entre si" (Google Code University, Google Inc., 2011).

Para um sistema distribuído "funcionar corretamente, são necessários mecanismos para uniformizar a comunicação" (Silvestre, 2005), caso contrário seria impossível compartilhar recursos entre processos e/ou máquinas diferentes.

2.6. *HTTP Long Pooling / Comet*

O *HTTP* é o protocolo no qual se baseia toda a web. É um protocolo de camada de aplicação *TCP*, desenvolvido pela *W3C* em 1990. "É um protocolo sem estado (*stateless*), baseado no modelo de *POST/GET*" (Varela, 2012). O seu modo de funcionamento é simples, um cliente estabelece uma conexão com o servidor e envia um pedido (*POST*), que consiste num método, um *URL*, e a versão do protocolo, seguidos opcionalmente por cabeçalhos de requisição e informações sobre o cliente. "O servidor responde com uma linha de estado, incluindo a versão do protocolo e um código de erro (40x) ou sucesso (20x), seguidos por uma mensagem contendo informações sobre o servidor e o corpo de dados do objeto requisitado (*GET*)" (Berners-Lee, 1996).

[5] *Rendezvous network* – é um protocolo de comunicação que permite a recursos numa rede P2P encontrarem outros elementos. Exemplos de redes *Rendezvous* incluem SIP, JXTA ou projeto *Freenet*.

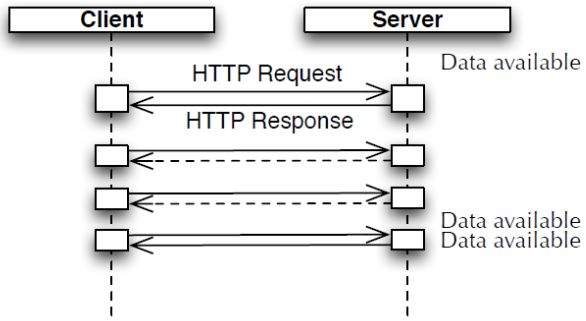


Figura 8: Esquema de uma conexão HTTP (Hämäläinen, 2012)

A Figura 8 demonstra a transferência que é executada a quando de um pedido *HTTP* normal. O cliente envia um pedido ao servidor (*POST*) que retorna com uma resposta (*GET*) podendo esta ser formatada conforme o seu estado que compreende valores entre 100 e 500. Sendo que os típicos valores encontrados são o 200, para resultados bem-sucedidos, o 300 para redireccionamentos, o 400 para erros e por fim o 500 para erros internos de servidor ou erros aplicacionais, também conhecidos como erros do programador.

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (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
```

Figura 9: Cabeçalho de uma mensagem HTTP com Status 200 OK

Na Figura 9, é mostrado um resultado de um pedido (*GET*) de *HTTP* recebido pelo servidor de *web Apache*. O conteúdo de um cabeçalho na maioria dos casos é simples de entender como mostra a figura. O cabeçalho de um pedido é sempre constituido por um Dicionário Chave Valor serializado e separado por : (dois pontos) e um separador de linha \n (*enter*).

O *HTTP Long Pooling* surge da ideia que as aplicações devem ser mais responsáveis e tem como objectivo "manter um canal de comunicação aberto com um servidor, algumas tecnologias Web têm utilizado abordagens baseadas em múltiplos pedidos e na manutenção da conexão *HTTP* convencional aberta pelo maior tempo possível" (Lopes, 2012).

A *Long Pooling* é "uma variação do técnica tradicional de *Pooling*" (Wikimedia Foundation, Inc., 2013). Contudo a grande diferença é quando o servidor não têm infomação de retorno disponivel, no *Long Pooling* o servidor fica com a ligação em espera e retorna a ligação quando têm dados para devolver. (Wikimedia Foundation, Inc., 2013). Esta técnica também é denominado *Comet*. O esquema é demonstrado na Figura 10:

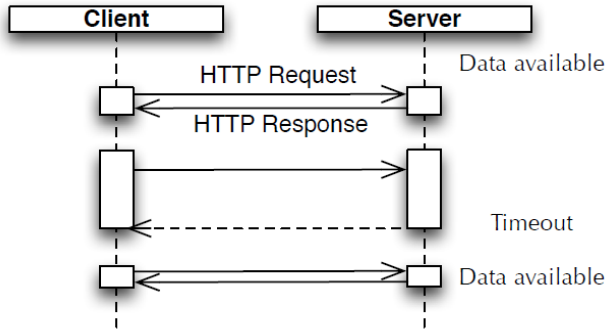


Figura 10: Esquema de uma conexão HTTP Long Pooling (Hämäläinen, 2012)

A comunicação usada nesta técnica é efetuada através dos protocolos *HTTP 1.0/1.1* e com recurso ao *AJAX*. O *AJAX* "é a arte de trocar informação com o servidor, ou actualizar parte de uma página de internet sem recarregar completamente a página" (w3schools, 2012).

```
$(document).ready(function() {
    function hook() {
        $.ajax({
            type: 'GET',
            async: true,
            cache: false,
            dataType: 'json',
            url: 'http://clx01.cloudapp.net/pooling.aspx',
            data: [],
            success: function(data, status, xhr) {
                alert('Data received');
                hook(); // Repeat LONG POOLING
            }
        });
    }

    hook();
}
```

Figura 11: Exemplo de um pedido HTTP Long Pooling usando AJAX

O exemplo na Figura 11 demonstra uma chamada ao serviço através da técnica de *Long Pooling* utilizando o protocolo *HTTP* normal.

```php
<?php
    if(rand(1,3) == 1){
        /* Fake an error */
        header("HTTP/1.0 404 Not Found");
        die();
    }

    /* Send a string after a random number of seconds (2-10) */
    sleep(rand(2,10));
    echo("Hi! Have a random number: " . rand(1,10));
?>
```

Figura 12: Exemplo de uma espera do Servidor, simulando um Long Pooling

O código da Figura 12 demonstra uma ação no servidor a quando de um pedido *AJAX* através de uma chamada *Long Pooling*. O Servidor neste caso fica em espera entre um tempo de 2 segundos a 10 segundos.

Para alguns autores, "estas estratégias apresentam características que aumentam a complexidade, reduzem a estabilidade e agregam sobrecarga à conexão entre o servidor e o cliente" (Lopes, 2012). Outras desvantagens encontradas por outros autores é o tempo máximo que esta solução pode aguardar, "um servidor *Comet* pode aguardar por uma resposta no máximo 30 segundos, isto prende-se com o facto de ser o limite imposto pelo *Session Timeout* do *HTTP*" (Furukawa, 2011).

2.7. WebSockets

O *WebSocket* "permite uma maior interatividade entre um navegador e um *Web Site*, viabilizando a disponibilização de conteúdos em tempo real. Isso se torna possível pelo provimento de um meio para o servidor enviar conteúdo para o navegador sem que tenha havido uma requisição prévia pelo mesmo, bem como pela possibilidade de envio/recebimento de mensagens mantendo a conexão aberta" (Lopes, 2012). Outros autores indicam que "a especificação *HTML5 WebSocket* define uma conexão *full-duplex single-socket* (ou bidirecional) para o envio e receção de informações entre o cliente e o servidor (ver a Figura 13). Assim, evita-se os problemas de conexão e portabilidade do paradigma *Comet* e fornece uma solução mais eficiente do que o *Ajax Pooling*" (Zhangling & Mao, 2012).

"O protocolo *WebSockets* especifica a interface de comunicação entre a aplicação web e o servidor" (Hämäläinen, 2012). Enquanto "a interface de *WebSocket API* é a responsável pela comunicação entre o navegador e a aplicação *web*" (Hämäläinen, 2012)

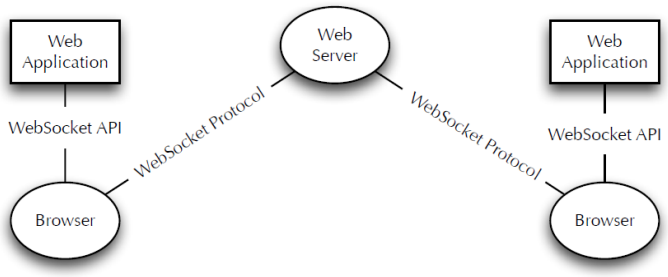


Figura 13: Demonstração da comunicação de um Componente *WebSocket*

O *IETF*[6] *RFC*[7] *6455* é o documento oficial atual que define o protocolo, ele identifica o *WebSocket* como sendo "um protocolo que permite a comunicação bidirecional entre um cliente a execução de código não confiável em um ambiente controlado e um *host* remoto que aceitou a comunicação com esse código" (Fette & Melnikov, 2011). Outros autores dizem que os "*WebSockets* definem um canal de comunicação full-duplex, que opera por meio de um único Socket na web. Os *WebSockets* não são apenas mais uma melhoria incremental para a comunicação *HTTP* convencional, mas representam um avanço colossal, especialmente em tempo real, para aplicações orientadas a eventos web" (Lubbers & Greco, 2012).

O funcionamento do "protocolo consiste na abertura de um *handshake* seguido de uma mensagem simples, sobre a camada *TCP*[8]" (Fette & Melnikov, 2011). O objetivo desta tecnologia é "providenciar um mecanismo que permita a aplicações de comunicação ponto a ponto não usarem múltiplas conexões *HTTP* (e.g. *XMLHttpRequest*, *iframe* ou *Long Pooling*) " (Fette & Melnikov, 2011), como é demonstrado na Figura 14:

[6] *The Internet Engineering Task Force* – Grupo de especialistas responsáveis por desenvolver os protocolos que suportam a Internet

[7] *Request for Comment*, é um documento que descreve os padrões de cada protocolo da Internet previamente a serem considerados um padrão

[8] *Transmission Control Protocol* – É um dos protocolos no qual assenta a base da Internet

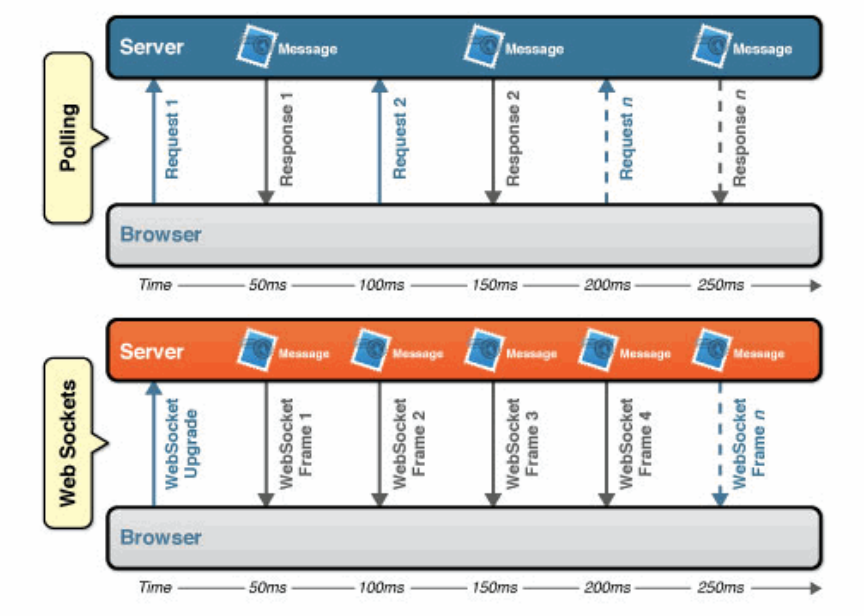


Figura 14: Diferença entre *Long Pooling* e *Web Sockets* (Lubbers & Greco, 2012)

O protocolo é composto por duas partes o *handshake* e a transferência de dados. O *handshake* é o cabeçalho enviado ao cliente que informa que a transferência de dados vai começar (Fette & Melnikov, 2011). Esta transmissão de dados é denominada de *FRAME*, Figura 15. "Sendo uma comunicação bidirecional cada lado da comunicação pode enviar dados a qualquer momento, deferentemente do *HTTP*, onde o servidor não consegue enviar dados para o cliente sempre o desejar" (Lopes, 2012).

A informação é transferida em formato de texto em *UTF-8* ou *UNICODE*, sendo eu a última revisão do protocolo permite a escolha entre um formato *String*, *Blob* ou *ArrayBuffer*. Cada *FRAME* inicia com o *byte 0x00* e termina com o *byte 0xFF*, indicando que o *FRAME* vai ser encerrado. Por fim o "fecho da ligação é efetuado através do envio de um 0xFF seguido de um 0x00 para o destinatário que envia o mesmo pedido para que a ligação seja encerrada" (Varela, 2012).

Outra funcionalidade do protocolo são os campos adicionais que são usados para expandir funcionalidades do protocolo em si. Na versão atual, os campos suportados são o '*Cookie'* que podem ser usados para o envio de informação persistente entre o cliente e o servidor. Outro campo usado na expansão do protocolo é o '*Sec-WebSocket-Protocol*' que contem uma *String* arbitrária que descreve o subprotocolo, ou o protocolo próprio da aplicação como é demonstrado na Figura 16 e na Figura 17.

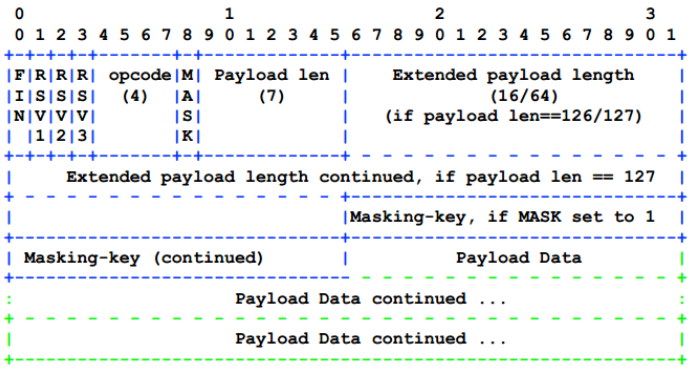


Figura 15: Descrição de uma *FRAME WS* segundo o *RFC 6455*

Os demais campos são relativos à segurança. Sendo que o campo de *Host* serve como proteção conta a alteração indevida de *DNS*, outro campo obrigatório no protocolo é o *Origin* este também relacionado com a segurança, permite que o servidor identifique acessos cruzados não autorizados de pedidos, como é visível na Figura 16.

Na definição do protocolo [*RFC 6455*] são definidos *handshake* cliente/servidores distintos, tal como é demonstrado nas figuras seguintes:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Figura 16: *Handshake* do Cliente para o Servidor

Na Figura 16, as primeiras quatro linhas servem para manter a compatibilidade com o protocolo *HTTP*. A linha *Origin* identifica a origem da ligação (o cliente). As linhas *Sec-* são as responsáveis por manter a conexão segura.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Figura 17: *Handshake* to Servidor para o Cliente

Na Figura 17, as primeiras três linhas permitem a compatibilidade com o protocolo *HTTP* como acontece com a figura anterior. Sendo que o '*Sec-WebSocket-Accept'* é o responsável pelo envio da confirmação de aceitação do pedido efetuado.

Uma vez estabelecida a conexão, não é necessário utilizar estes cabeçalhos a cada pacote enviado, apenas o esquema de enquadramento descrito anteriormente, constituído simplesmente por um *byte* 0x00 no início e um *byte* 0xFF ao final do pacote.

Desta forma, "observa-se que o *WebSocket* visa disponibilizar uma estratégia para comunicação bidirecional em tempo real, provendo um método de conexão permanente que permite o tráfego em duas vias (*full duplex*), iniciado a partir de uma conexão *HTTP*, sendo assim capaz de funcionar na estrutura existente, como *routers* e *proxies*" (Lopes, 2012).

Outras vantagens encontradas são que "uma aplicação web pode ser construída utilizando o protocolo *WebSocket*" (Furukawa, 2011). Tornando-a, uma aplicação distribuída na sua essência. Deve-se ainda referir que "não são necessários addons para a execução de aplicações *WebSocket*, bastando apenas um navegador com compatibilidade para o *HTML5*" (Furukawa, 2011). Por fim, (Furukawa, 2011) refere que "as aplicações *HTML5* oferecem uma liberdade em plataformas" nunca antes alcançadas.

2.7.1. WebSocket API

Para percebermos o funcionamento dos *WebSockets* é essencial demonstrar o seu funcionamento básico. A comunicação é efetuada usando uma *API*[9] de *JavaScript* homologada pela *W3C*.

[9] Application Programming Interface

```
[Constructor(in DOMString url, in optional DOMString protocols)]
[Constructor(in DOMString url, in optional DOMString[] protocols)]
interface WebSocket {
  readonly attribute DOMString url;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSING = 2;
  const unsigned short CLOSED = 3;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long bufferedAmount;

  // networking
            attribute Function onopen;
            attribute Function onmessage;
            attribute Function onerror;
            attribute Function onclose;
  readonly attribute DOMString protocol;
  void send(in DOMString data);
  void close();
};
WebSocket implements EventTarget;
```

Figura 18: *Interface API* do *WebSocket* (Hickson, 2011)

A Figura 18 mostra a interface homologada pela *W3C*. Detalhadamente a *API* é composta por:

- ***Construtor*:** o construtor é a função que em desenvolvimento *OOP* é executada quando um objeto é criado, este construtor em especifico é executado mediante a utilização de 2 parâmetros *String* ***url*** (servidor, caminho e porta) e ***protocol*** (subprotocolo), opcionalmente são executadas funções de *Callback* que são subscritas por eventos:
  - ***onopen:*** executado quando a conexão for estabelecida. Recebe como parâmetro o objeto *WebSocket* criado, usando o próprio objeto *this*. Internamente pode-se executar código customizado;
  - ***onmessage***: executado quando uma mensagem é recebida. É devolvido no parâmetro *msg* o conteúdo do corpo da mensagem formatado em *UTF-8*;
  - ***onerror:*** executado quando ocorre um erro no envio/receção de mensagens, esta função pode executar código customizado para que sejam tomadas medidas necessárias para a correta implementação da solução;
  - ***onclose:*** executado quando a conexão é fechada por *timeout* ou a pedido do programa. Este evento não recebe qualquer tipo de parâmetros.

Além das funções que constituem a subscrição dos eventos, a *API* implementa outros tipos de funções essenciais para a correta verificação do estado e do envio de meta dados e fecho de ligações:

- Método ***send***: utilizado para o envio de informação entre o cliente e o servidor. Esta é a função ideal para o envio de metadados, importante para a correta parametrização da receção de informação. O envio é processado usando um parâmetro do tipo *String* em formato *UTF-8* que contém a mensagem.
- Método ***close***: força o encerramento de uma ligação *WebSocket*.
- Propriedade ***readyState***: informa o estado corrente da ligação através de um enumerador, constituído por:
    - 0 – A conexão ainda não foi estabelecida;
    - 1 – A conexão foi estabelecida com sucesso;
    - 2 – A conexão está a ser encerrada;
    - 3 - Indica que a conexão foi encerrada e não pode ser reestabelecida;

Ao nivel da camada de rede o *WebSocket* é um protocolo homologado que obedece a regras de implementação protocolar e de desenvolvimento de soluções que tirem partido desta tecnologia.

A sua implementação baseada em *TCP* permite manter uma lógica standardizada para uma comunicação em computação distribuida entre diversos componentes, bem como a sua semelhança com o protocolo *HTTP* que permite de forma rápida desenvolver produtos que possam comunicar entre si de forma standard e rápida.

2.8. Soluções existentes com tecnologias Real Time Web

No mercado existem várias soluções que permitem a criação e envio de mensagens através do método de *pub/sub*, todas elas têm as suas vantagens e desvantagens.

Apesar de existirem trabalhos académicos interessantes, na maioria dos casos estão incompletos ou sem qualquer tipo de aplicação prática, por isso, neste documento optou-se por escolher três soluções que existem no mercado empresarial, pois são soluções implementadas e com uma utilização diária.

MS Mobile Services

MS Mobile Services é um serviço disponível na *cloud Azure*, mediante a utilização de uma assinatura de serviços *cloud* da *Microsoft*, este serviço não é gratuito e é pago conforme a utilização baseando-se no modelo de *PaaS*. A infrastrutura é alocada nos *data centers* da *Microsoft*, sendo que a mesma apenas disponibiliza o serviço de cliente e de publicador (através de operações *CRUD* em base de dados *SQL SERVER*).

Pubnub.com

*Pubnub.co*m é um serviço que facilita a criação de aplicativos que atualizam constantemente dados. Além de oferecer infraestrutura na *cloud* o serviço oferece um

cliente JavaScript que utiliza todas capacidades possíveis para entregar os dados rapidamente e com menor *overhead*, tirando partido se possível de tecnologias atuais como os *WebSockets* [10].

APE project

O *APE project* é um projeto *OpenSource* baseado no paradigma *Comet* ou *reverse-ajax*. O seu funcionamento é bastante abrangível pois como foi referido funciona através das tecnologias *AJAX* e *XML*. À data da criação deste documento o sistema não continha versão binária (apenas código-fonte), além de que a sua instalação é bastante complexa pois é através de linhas de comandos e alterações aos ficheiros de configuração de *APACHE*. Outra desvantagem é o suporte apenas para servidores *LINUX*

Não é o objetivo neste trabalho atingir os níveis de complexidade destas soluções, mas sim apresentar um protótipo que implemente as funcionalidades que se desejam comparar.

A proposta que apresento neste trabalho tem o nome de *RTML*, o nome usado varia das palavras *(R)eal(t)ime* e *Hypertext (M)arkup (L)anguage* por ser um nome simples de memorizar. O nome da solução permite no futuro desenvolver uma linguagem de marcação baseada em *XML* que permita a comunicação direta entre o *Real time* e o *HTML* através de etiquetas. A *Tabela 2* mostra as suas principais características em comparação com a solução que proponho neste documento:

	MS Mobile Services	**Pubnub.com**	**APE Project**	**RTML**
Broker	Sim	Sim	Sim	Sim
Peer-to-Peer	Não	Não	Não	Sim
On-premise	Não	Não	Sim	Sim
PaaS	Sim	Sim	Não	Possível
Trigger	Sim	Não	Não	Sim
Tecnologia Servidor	C++	PHP	C++	C#, NET MVC

[10] Para mais informações consultar as especificações oficiais: http://www.w3.org/TR/2009/WD-websockets-20091222

	MS Mobile Services	**Pubnub.com**	**APE Project**	**RTML**
Tecnologia Cliente	JS e NET	JS	JS	JS
Servidor	IIS	Apache	Apache	IIS ou Apache (Mono mod)
Mensagens	XML / JSON	XML	XML	XML / JSON
Escalabilidade	Sim	Não	Não	Sim
Free	3 Meses	Não	Sim	Sim
Open-source	Não	Não	Sim	Sim

Tabela 2: Tabela comparativa entre as soluções existentes com tecnologia real time web.

A "*RTML*" permite que a solução funcione sobre duas metodologias diferentes a *P2P* e o *Broker (ver 2.4)*. Esta solução é uma solução escalável, pois funciona em *On-premise* (no escritório), visto que poucas *frameworks Real Time* funcionam deste modo, em geral as versões comerciais assentam unicamente sobre solução *Plataform-as-a-Service (PaaS)*, suportadas no pagamento de pacotes de mensagens. A possibilidade de migração para um serviço *PaaS* está contemplada.

Ao nível das funcionalidades, o suporte a *triggers* é uma funcionalidade interessante, neste modelo que proponho, as *triggers* são funcionalidades implementadas em base de dados que publicam diretamente sobre o servidor *real time*. Por este meio a informação circula mais facilmente até ao seu destinatário eliminado intermediários no meio do processo.

A tecnologia envolvida será uma tecnologia que possa ser usada pela maioria dos programadores. A opção da escolha do *C#*, foi a premissa que esta solução será disponibilizada como *open source*. Ao escolher este tipo de linguagem bem como a escolha do *JavaSript* permite que futuramente os programadores interessados possam expandir a *framework* por forma a criar novas funcionalidades.

As mensagens serão distribuídas de duas formas distintas, ao contrário das soluções existentes que funcionam sobre o formato de ficheiros *XML (eXtensible Markup Language)*, *ATOM (Atom Syndication Format)* e *Odata*.

O "*RTML*" utiliza o *JSON (JavaScript Serializable Object Notification)*, por ser uma referência atual no desenvolvimento web. Ao possibilitar a escolha de assinar mensagens em *JSON*, obtém-se um corte drástico no número de bytes que circulam durante a conversação do cliente/servidor.

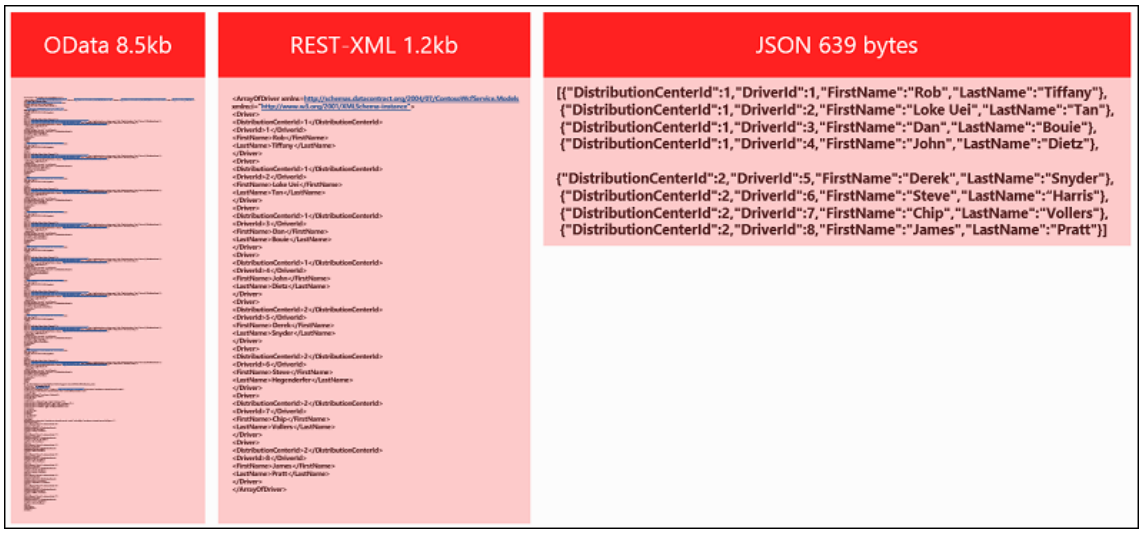


Figura 19: Comparativo entre os diversos formatos de ficheiros: Odata, XML e JSON (Wegner, 2012)

A Figura 19 demonstra que no conteúdo de uma mensagem em diversos formatos pode-se constatar que em *Odata* ocupa 8.5kb, em *XML* 1.2kb e em *JSON* somente 639 bytes. Ou seja, em termos estatísticos um ficheiro *XML* permite uma poupança de 85%, enquanto o ficheiro *JSON* pode chegar aos 92% de poupança em tráfego de rede e processamento de dados.

A escolha do servidor está inerente, pois a escolha da linguagem não permite muitas alternativas, poderia nesta solução avançar com um servidor próprio, mas a escolha parte por um servidor já existente pois assim permite uma maior quantidade de alternativas e melhor integração com os sistemas existentes. A solução estará disponível para *Apache*, e migração para o *Linux* através do módulo *mod_mono* instalável do servidor web *Apache 2.0*.

Todo o desenvolvimento e comunicação entre o cliente/servidor funcionarão sobre o paradigma *publisher*/*subscriber* pois é o que apresenta melhor desempenho computacional. As funcionalidades como o *Matching* e o *Event Routing* permitem apenas entregar as mensagens que interessam ao assinante, melhorando assim o tráfego e a qualidade da rede.

3. Desenvolvimento de uma Framework *Real Time*

Este capítulo apresenta a modelagem da arquitetura proposta para manutenção da ordenação causal e total de mensagens em plataformas *Publish/Subscribe*. Os principais algoritmos são apresentados e discutidos.

3.1. Aspetos estratégicos

A proposta apresentada baseia-se nas quatro forças que a Microsoft apresenta no seu documento *Publish/Subscribe* (Microsoft patterns & pratices, 2012), bem como na ideia que para suportar a escalabilidade de clientes, um serviço de notificação de eventos deve ser capaz de lidar com a desconexão temporária e a mobilidade de clientes, *i.e.* a entrega confiável de notificações independente do endereço *IP* corrente dos mesmos.

Aplicações integradas devem apenas receber mensagens que são do seu interesse, balanceadas através das forças seguintes:

- As aplicações em uma arquitetura de integração consomem diferentes tipos de mensagens. Por exemplo, aplicativos que gerem as informações dos clientes estão interessados em atualizações de informações de clientes. Aplicações comerciais estão interessadas em operações de compra e venda. Aplicações que participem de transações de duas fases estão interessadas em mensagens de fecho.
- Uma aplicação em uma arquitetura de integração pode enviar vários tipos de mensagens. Por exemplo, o aplicativo pode enviar mensagens de informações de clientes e mensagens operacionais sobre o seu estado. (*Status* é também referida como a saúde neste contexto). Da mesma forma, uma aplicação em uma arquitetura de integração está normalmente interessada em apenas um subconjunto das mensagens que são enviadas pelas outras aplicações. Por exemplo, um gerente de carteira está interessado apenas em operações financeiras que afetam as ações que ele gerencia.
- A forma como as aplicações permitem adicionar informações às suas mensagens varia muito. Mensagens binárias fixas normalmente não fornecem nenhuma flexibilidade ou, quando fornecem, é uma flexibilidade limitada nesta área. Em contraste, geralmente é fácil estender mensagens *SOAP*[11] através de elementos do envelope.
- A maioria das arquiteturas de integração integra aplicativos proprietários. Essas aplicações costumam fazer fortes suposições sobre as mensagens que eles usam

[11] *SOAP* é um protocolo baseado em *XML* que permite a aplicações trocar informações através de canais *HTTP*.

para se comunicar com outras aplicações no ambiente. Mesmo com um formato de mensagem flexível, ela pode ser difícil de inserir ou de processar os elementos de mensagem que a aplicação não tem conhecimento.

Com o intuito de tratar destas questões, propõe-se uma *API* que oferece suporte à comunicação *pub/sub* para aplicações distribuídas entre clientes servidor como por exemplo *desktops*, *web* e dispositivos móveis sensíveis ao toque, que trate das questões de escalabilidade para as aplicações *pub/sub* sem requerer nenhuma infraestrutura adicional.

3.2. Modelo Conceptual

Neste tópico será apresentada a arquitetura dos componentes, os diagramas e a explicação da composição teórica e técnica dos mesmos. Serão ainda apresentados algoritmos básicos essenciais para compreensão da arquitetura.

3.2.1. Arquitetura dos Componentes

O modelo deste trabalho baseia-se em um modelo construído através da ideologia de uma arquitetura distribuída (*ver Secção 2.5*). Este modelo é composto pelas componentes de Cliente e Servidor (ver Figura 20), o Cliente é também denominado de *Subscriber*, por sua vez o Servidor será nomeado *Server Interface*. Por fim o *Publisher* é a componente responsável por colocar mensagem na rede (*ver Secção 2.3*).

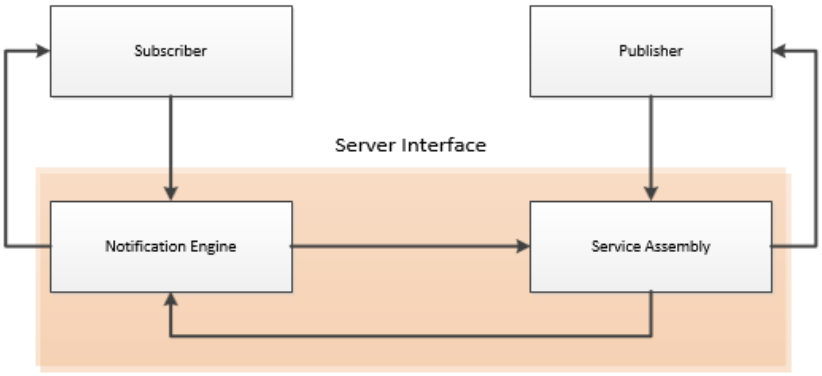


Figura 20: Proposta do Modelo de Comunicação

Na Figura 20, verifica-se a existência das três componentes-chave da implementação. A comunicação entre o cliente e o *Server Interface* é uma comunicação em 2 sentidos (*Two-way*), sendo que o *Subscriber* solicita pedidos ao servidor e este retorna com respostas que satisfaçam essa procura.

Internamente o Servidor processa a solicitação efetuada pelo *Subscriber* processando o pedido e guardando numa base de dados relacional (*Notification Engine*), toda a informação necessária por forma a manter persistente as configurações e atributos da ligação entre o *Publisher* e o *Subscribe.* O *Service Assembly* é o componente responsável por efetuar a validação de resultados entre os diversos componentes existentes bem como o responsável pelo *Map reduce* de mensagens.

O componente assume ainda a responsabilidade do tratamento de conteúdos, este tratamento é um conjunto de funções responsáveis pela conversão de mensagens para o formato de ficheiro suportado pelo sistema: o *JSON* (*JavaScript Serializable Object Notification*)

Por forma a manter a veracidade e a autenticidade de mensagens entre os diversos componentes da rede, o *Service Assembly*, apresenta uma camada de segurança (*Security Layer*) e uma camada de criptografia (*Cryptography Layer*), como é demonstrado na Figura 21. As mensagens são assinadas usando um certificado de autenticidade reconhecido e cifradas num algoritmo *AES (Advanced Encryption Standard)* com uma chave de 256 *bits*.

Por fim o *Publisher* é um componente que envia as mensagens para a rede, nesta infrastrutura o componente *Publisher* assume uma entidade de *Interface*, pois pode ser implementado sobre diversas tecnologias atuais, por forma a manter o sistema o mais abrangente possível. Neste trabalho serão demonstradas as implementações de *Interfaces Publishers* em sistemas de base de dados e em linhas de comando (*Command Line*)

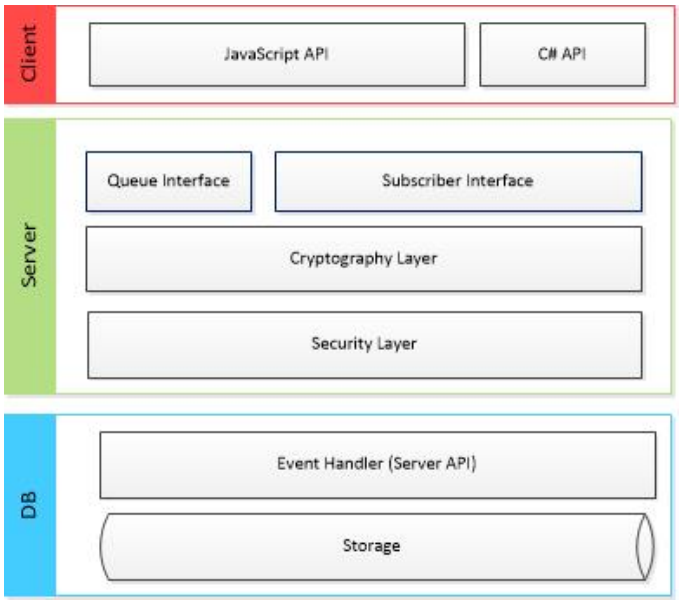


Figura 21: Proposta da Arquitetura do Componente

As estruturas dos componentes podem-se ainda dividir em três camadas diferentes: Cliente, Servidor e Base de dados (*DB*).

Na camada de Base de dados, estão incluídos os componentes de *Storage*, a cargo de um Servidor de base de dados relacional, nesta implementação optou-se por um *Microsoft SQL Server*, mas a existência de outro *SGBD* (Sistema de Gestão de Base de Dados) está contemplado, pois a estrutura está assente sobre classes *interfaces* da linguagem de desenvolvimento.

Como foi referida anteriormente esta camada de *DB* faz parte do *Notification Engine*, que é o componente responsável pela persistência dos dados, bem como do filtro de mensagens, denominado de *Matching (ver Secção 2.4)*. O subcomponente *Event Handler* efetua a gestão de eventos como é demonstrado na Figura 22.

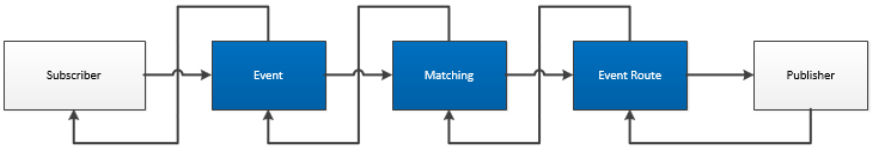


Figura 22: Visão geral do componente de Eventos (Event Handler)

Os eventos podem ser despoletados tanto pelo *Subscriber* como pelo *Publisher*, para simplificar a estrutura do sistema existe um único componente de eventos, que através de parâmetros identifica qual a sua origem e o seu destino. Este componente de eventos é o responsável pelo roteamento (*Event Routing*) de mensagens e sua distribuição pelos corretos canais disponíveis na rede em determinado momento.

A camada de *Server* na Figura 21, numa perspetiva genérica, é incluída no componente *Service Assembly* nesta camada estão incluídas as subcamadas de criptografia e de segurança.

Na camada de criptografia optou-se pelo *SHA1* pois é um algoritmo *standard* de criptografia validado e aprovado pela indústria informática e de serviços. Este algoritmo foi escolhido pois é o algoritmo padrão de criptografia na proposta do *WebSocket Protocol* (*RFC 6455*).

Na camada de Segurança estão incluídas várias funcionalidades como a gestão de utilizadores, grupos e níveis de acesso. A configuração deste mecanismo de segurança fica a cargo do *namespace System.Web.Security* mais especificamente as classes abstratas *UserProvider* e *RoleProvider* da *framework NET*, eliminando assim a

necessidade de desenvolvimento de um sistema para a gestão de utilizadores personalizada. Esta opção permite a implementação de diferentes tipos de suportes de permissões, como exemplo: *AzMan*, *SQL*[12], *Passport*[13], *Active Directory*[14], ou mesmo um serviço de permissões personalizados. **Nota**: A opção do *AzMan* é considerada uma boa prática em *software* desenvolvido sobre a plataforma *NET, segundo o* (Microsoft patterns & pratices, 2012).

Na implementação existem dois subcomponentes de armazenamento de mensagens o componente *Queue Interface* é o responsável pelo armazenamento de *Queues*, o método utilizado na implementação deste componente é o *FIFO* (*First in first out*), através deste subcomponente as mensagens são armazenadas temporariamente, a sua distribuição é o mais em tempo real possível.

Outro subcomponente responsável pelo envio de mensagens é o *Subscriber Interface*, neste quadro o componente existente é uma *Interface* de uma implementação, o que permite uma ou mais futuras implementações deste sistema de envio de conteúdos e mensagens, de forma a otimizar sinergias entre as mais diversas plataformas.

Na vertente de componentes *Client*, existem duas *API's* (*Application Programming Interface*) disponíveis. A *API JavaScript*, implementada sobre *WebSockets*, responsável pela implementação do sistema na web, é a componente que permite aos programadores escreverem programas que comuniquem diretamente com o *browser* cliente, estão disponíveis diversas plataformas atuais *IE 10*, *Firefox* e *Chrome*.

Foi incluída uma *API* em C# que permite comunicar nativamente com o servidor para suportar o transporte de mensagens em sistemas distribuídos entre diversos sistemas na mesma rede. As duas *API's* cliente são idênticas na sua interface.

3.3. API Servidor

As tecnologias usadas nesta implementação são o *C#* para o desenvolvimento servidor.

O servidor é composto pelas seguintes componentes: *Server Interface* (*Notification Engine* e *Service Assembly*) e pelo *Publisher*.

---

[12] *SQL – Structured Query Language* é a linguagem de pesquisa declarativa, padrão para bancos de dados relacionais.

[13] *Passport* – Serviço unificado de autenticação de utilizadores criado e mantido pela Microsoft que permite aos mesmos acederem a vários websites utilizando uma única conta.

[14] *Active Directory* – Serviço de diretório no protocolo LDAP que armazena informações sobre objetos em rede de computadores e disponibiliza essas informações a utilizadores e administradores dessa rede.

```
/// <summary>
/// Publishes the specified message.
/// </summary>
/// <param name="message">The message.</param>
/// <param name="callback">The callback.</param>
void Publish(string message, Action<string> callback);
```

Figura 23: Código do *IPublisher* em C#

O algoritmo demonstrado na Figura 23 é a *Interface* com o programador, o componente de *IPublisher*, este contêm o método *Publish ()* que permite ao sistema publicar informações no servidor, ou *middleware*. Além da mensagem enviada, a função contém uma ação de *Callback*, que permite ao executar código personalizado logo após o envio da mensagem. A utilização prática deste componente é demonstrada na Figura 24.

```
Console.WriteLine("The publisher started successfully, press key 'ctrl+c' to stop it!");

while (true)
{
    string str = Console.ReadLine();

    using (Publisher pub = new Publisher(new ServerConfig()
    {
        Host = "localhost",
        Port = 8081,
        Route = "/demo",
        Debug = true
    }))
    {
        pub.Publish(str
            , () =>
                {
                    Console.WriteLine("message sended sucessfuly.");
                });
    }
}
```

Figura 24: Exemplo de um envio de uma mensagem através do Publisher

Neste exemplo o programa começa por se ligar ao servidor *middleware* através de uma instancia do componente *Publisher* no endereço *ws://localhost:8081/demo* que representa o *Host*, a porta e o *Route*. O *Route* é o equivalente a um canal de comunicação no qual circulam informações sobre um determinado tema.

Para efetuar o envio de informação, o programador deve aceder à função *Publish ()* que como explicado na Figura 23 é o responsável pelo envio da informação para o *middleware*.

Após o envio da mensagem por parte do componente *Publisher* para o *middleware* este trata de processar a informação e enviá-la aos seus destinatários. O processamento dessa informação passa pelo componente de *Notification Engine*, demonstrado na Figura 25.

```
public override void ExecuteCommand(RealtimeSession session, SubRequestInfo requestInfo)
{
    string route = session.Path;

    Parallel.ForEach<RealtimeSession>(session.AppServer.GetSessions((s) =>
    {
        return s.Path == route;
    }), (ss) =>
    {

        Regex regex = new Regex(session.Matching ?? "*", RegexOptions.IgnoreCase |
            RegexOptions.Multiline);

        if (regex.IsMatch(requestInfo.Body))
        {
            ss.Send(requestInfo.Body);
        }

    });
}
```

Figura 25: Componente de *Notification Engine*

No código acima demonstrado referente ao componente *Notification Engine* existem elementos de relativa importância. O elemento mais importante no trecho de código demonstrado é o *RealtimeSession*, este é uma representação de todas as sessões que estão ligadas ao *middleware*, uma sessão representa uma conexão a um componente *ISubscriber*. O filtro de identificação do canal (ex: */demo*) é efetuado através da expressão de uma comparação de igualdade e usando uma função *Paralell.ForEach*, esta função permite o executar de um código cíclico em paralelismo, diversificando o acesso ao ciclo pelos vários processadores existentes no servidor, tirando assim maior partido dos recursos da máquina.

Por fim e após o filtro de canais, cada expressão de igualdade verdadeira é comparada através de uma *regular expression*[15] que verifica se o conteúdo foi requisitado pelo *ISubscriber* através do método *Subscribe (match).* Neste caso e apenas se a expressão for verdadeira o conteúdo é enviado para o cliente em formato de *JSON.*

[15] Uma regular expression providência uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres.

3.4. API Cliente

As tecnologias que serão usadas nesta implementação serão o *JavaScript* e o C# para o desenvolvimento cliente.

Para simplificar possíveis migrações e melhorias a versão cliente foi desenvolvida utilizando métodos que são implementados numa *interface* comum. Para entendermos o conceito é necessário intender o significado de interface em programação, sendo que a uma *interface*, é a fronteira que define a forma de comunicação entre duas entidades. Pode ser entendida como uma abstração que estabelece a forma de interação da entidade com o mundo exterior, através da separação dos métodos de comunicação externa dos detalhes internos da operação, permitindo que esta entidade seja modificada sem afetar as entidades externas que interagem com ela.

A *Interface* da versão C# é demonstrada na Figura 26.

```
/// <summary>
/// Subscribes with the specified match.
/// </summary>
/// <param name="match">The match.</param>
void Subscribe(string match = null);

/// <summary>
/// Open a real-time connection with the server
/// </summary>
void Open();

/// <summary>
/// Close the existing connection
/// </summary>
void Close();

/// <summary>
/// Send a message to the server
/// </summary>
/// <param name="message">The message.</param>
void Send(string message);

/// <summary>
/// Raises an event related with the connection
/// </summary>
/// <param name="event">The event.</param>
/// <param name="callback">The callback.</param>
void On(string @event, Action<RealtimeMessage> callback);
```

Figura 26: Código do *ISubscriber* em C#

Nesta demonstração, existem métodos importantes que devem ser referidos como o *Open ()* responsável pela abertura da comunicação entre o cliente e o servidor. O *Subscribe ()* é o método responsável pela subscrição de um canal de comunicação, é

neste método que o servidor é informado que o cliente está disponível para começar a conversação. Por fim o método *On ()* é onde são executados os diversos eventos suportados pelo interface.

Outros métodos contidos na *API* permitem o envio de mensagens para a modificação de estados de conexão como o método *Send ()* e o *Close ()* responsável pelo encerramento da comunicação.

A *API* cliente é um dos componentes-chave da solução pois é o responsável pela comunicação de informação entre o navegador e o servidor. A implementação deste mecanismo de subscrição é demonstrada na Figura 27:

```
ISubscriber subscriber = new Subscriber(Guid.NewGuid(), new ServerConfig()
{
    Host = "localhost",
    Port = 8081,
    Route = "/demo",
    Debug = true
});

subscriber.Open();
subscriber.On("open"
    , (msg) =>
        {
            subscriber.Subscribe("*");
        });

subscriber.On("receive"
    , (msg) =>
        {
            if (textBox1.InvokeRequired)
            {
                textBox1.Invoke((MethodInvoker)delegate
                    {
                        textBox1.Text += msg + Environment.NewLine;
                    });
            }
        });
```

Figura 27: Exemplo de um pedido *Subscriber*

Nesta demonstração, do componente *Subscriber*, o algoritmo inicializa uma conexão com o servidor através do método *Open ()*, sendo que em seguida executa o método *On ()* que contem os eventos de *open*, *close* e *receive*.

Como nota importante é na função *On ('open', ...)* a altura ideal para executar o método *subscribe ()*, que permite ao programa assinar e filtrar (parâmetro *match*) as mensagens que deseja receber.

O mapa de estados do modelo, na Figura 28, demonstra todo o caminho percorrido pela implementação no seu estado mais comum.

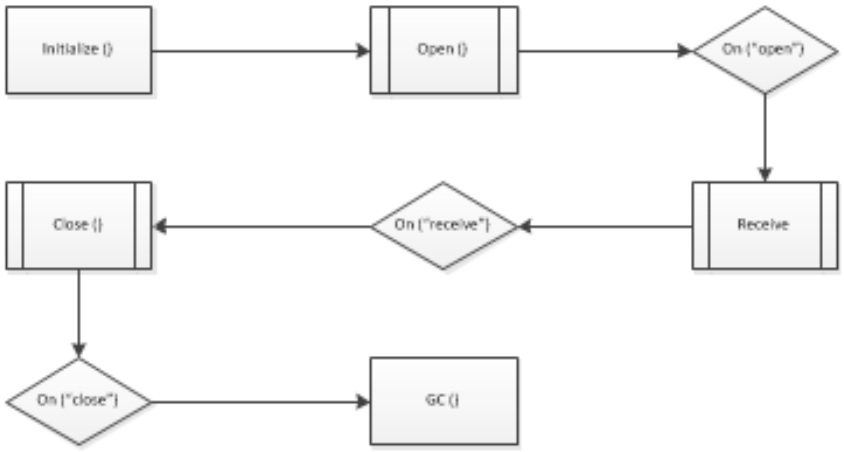


Figura 28: Mapa de estados do componente *Subscriber*

O restante código referente à *API* de *JavaScript e C#* é demonstrado no anexo I e II deste documento.

4. Validação e Avaliação

Este capítulo apresenta resultados obtidos com a implementação de referência realizada a fim de verificar a validade das contribuições da solução proposta. Desta forma, experimentos realizados são apresentados e analisados.

Para a validação e avaliação do trabalho, proponho três tipos de testes: i) Testes unitários; ii) Testes de desempenho e iii) Inquéritos.

4.1. Prova de Conceito – Um *site* de notícias

É apresentada uma prova de conceito para a implementação proposta, um *Site* de notícias que obtém a informação colocada pelo jornalista em tempo real.

O funcionamento do *site* é simples pois é uma prova de conceito, conforme o *mockup* apresentado na Figura 29. O *site* inclui uma zona de destaque (zona da esquerda), que é atualizado em tempo real mediante a publicação de um artigo por parte do jornalista, sem a necessidade de qualquer comunicação por parte do cliente ao servidor.

Sempre que for atualizado o cliente (navegador de *internet*) este manterá o estado da última notícia em destaque, atualizando novamente quando assim houver a necessidade.

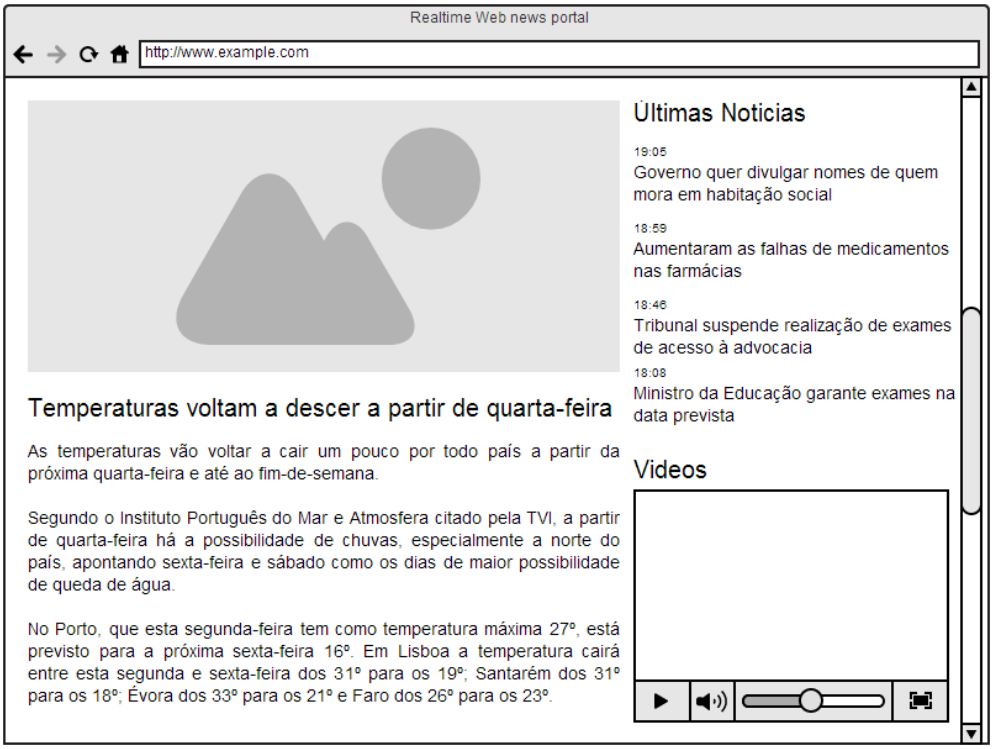


Figura 29: Mockup do funcionamento da prova de conceito

Os dados das últimas notícias (zona da direita) são armazenados em uma base de dados como demonstrado Figura 30:

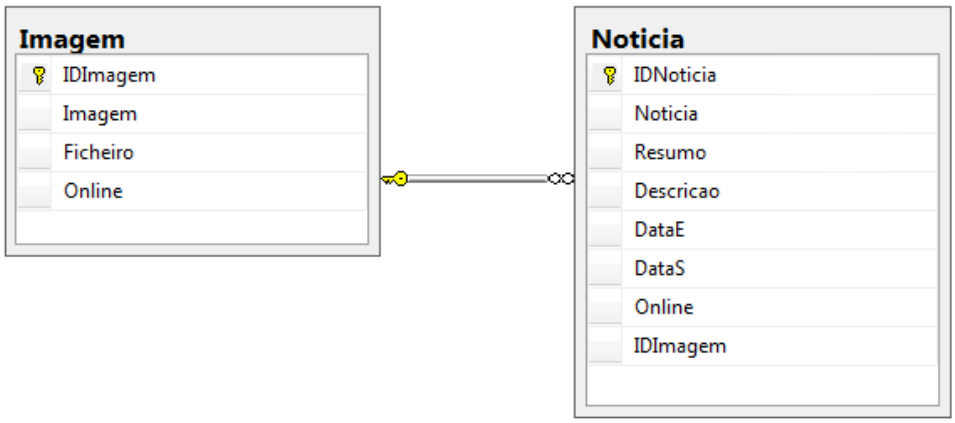


Figura 30: Estrutura da Base de dados

Este funcionamento permite que a página de internet seja carregada inicialmente através de uma conexão à base de dados. Deste modo a colocação da última notícia acontece na zona da esquerda e as restantes na zona da direita, e sempre que ocorrer uma atualização, acontece os seguintes procedimentos:

1. Elimina-se o último item da zona da direita;
2. O item da zona da esquerda passa a ser o primeiro item da zona da direita;
3. Colocado o item na zona da esquerda através de tecnologia Real time web

4.2. Testes Unitários

Os testes unitários permitem o teste do código implementado, verificando e retornando valores booleanos de cada função. Através destes testes é obtida uma precisão completa, do correto funcionamento da função programada.

Serão efetuados os testes unitários a todas as funções que contêm um corpo. Os elementos que apenas contêm Interfaces não serão testados. Os Interfaces foram criados para facilitar o programador na implementação de código personalizado, tornando assim a *API* mais extensível. Por esse mesmo motivo as Interfaces não têm um corpo logo não podem ser devidamente testadas, apenas podem ser testados as classes que implementam tais interfaces.

Os resultados obtidos dos testes foram bastante satisfatórios. Ficou comprovado que a utilização de técnicas como o paralelismo ou sistemas distribuídos aumentam a

capacidade do servidor, pois conseguem de uma simples distribuir carga entre os diversos componentes do sistema, tal como é demonstrado na Figura 31.

```
Reading protocols ... Succeed

The server started successfully, press key 'q' to stop it!

Sending welcome message 8f64c076-bc87-46d3-955e-6efb0c1d1e63 ... Succeed
Registering session 8f64c076-bc87-46d3-955e-6efb0c1d1e63 ... Succeed
Sending welcome message 8bba5d97-7a9c-43bd-8779-bcd0ef4f1ff3 ... Succeed
Registering session 8bba5d97-7a9c-43bd-8779-bcd0ef4f1ff3 ... Succeed
Founded a route match with /demo ... Succeed
Founded a route match with /demo ... Succeed
Sending message with matching * ... Succeed
Sending message with matching * ... Succeed
Sending close message 8bba5d97-7a9c-43bd-8779-bcd0ef4f1ff3 ... Succeed
Sending close message 8f64c076-bc87-46d3-955e-6efb0c1d1e63 ... Succeed
Sending welcome message 34a5dc8f-16f6-4fcf-8da7-a0da9468e66a ... Succeed
Registering session 34a5dc8f-16f6-4fcf-8da7-a0da9468e66a ... Succeed
Sending welcome message 9a2abf71-172f-44db-a4f7-b315ba92cdf9 ... Succeed
Registering session 9a2abf71-172f-44db-a4f7-b315ba92cdf9 ... Succeed
Founded a route match with /demo ... Succeed
Sending message with matching * ... Succeed
Sending close message 9a2abf71-172f-44db-a4f7-b315ba92cdf9 ... Succeed
Sending close message 34a5dc8f-16f6-4fcf-8da7-a0da9468e66a ... Succeed
Sending welcome message 2cab326e-7865-4bfb-b084-583fe365fb47 ... Succeed
Registering session 2cab326e-7865-4bfb-b084-583fe365fb47 ... Succeed
Sending welcome message fa2de01b-f0b3-447a-ad3f-fa1b08c55923 ... Succeed
Registering session fa2de01b-f0b3-447a-ad3f-fa1b08c55923 ... Succeed
Founded a route match with /demo ... Succeed
Sending message with matching * ... Succeed
Sending close message fa2de01b-f0b3-447a-ad3f-fa1b08c55923 ... Succeed
```

Figura 31: Resultado da bateria de testes

Nos testes da Figura 31 foram realizadas verificações às diversas componentes do sistema. As verificações principais ocorreram em partes do sistema como o envio de mensagens, a mensagem de boas vindas, a mensagem de fecho, a receção de mensagens e comandos, a verificação de *Routing*, bem como o início e término da sessão entre o servidor e o cliente.

O teste foi efetuado no dia 20 de Junho de 2012 sendo que foi testado num equipamento idêntico ao utilizado na secção Testes de Desempenho.

4.3. Testes de Desempenho

Os testes de desempenho apresentam a medição de performance de rede e de análise de código. Além disso os testes de desempenho servem como com outras soluções apresentadas (ver a secção 2.8). Os testes foram efetuados no dia 16 de Março de 2013 utilizando o navegador *Google Chrome* v.25. A arquitetura utilizada foi um *Intel i7* com

8GB de *RAM* e um disco a *7200 RPM*, a executar o *Windows 7*. A versão do protocolo em vigor nesta versão era o *HYBRD 13* (*RFC 6455*).

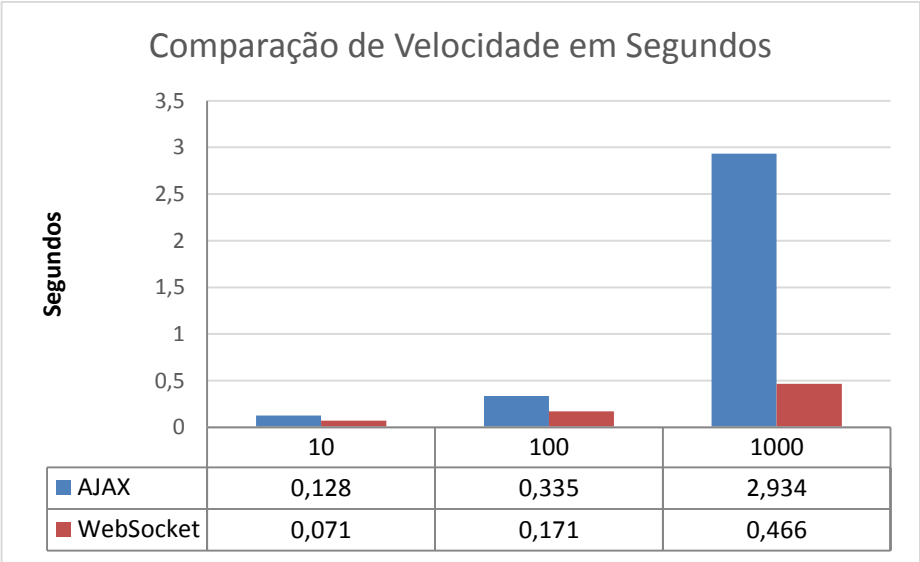


Figura 32: Comparativo de velocidade de obtenção de dados

A Figura 32 mostra a diferença de resultados obtidos entre a comparação do método de *reverse-ajax* (tradicional) e da implementação da tecnologia *WebSocket* disponível no *HTML 5.0*, mostrando claramente que o sistema obtém melhor desempenho quando existe um maior número de informação. Melhorando assim a latência da comunicação.

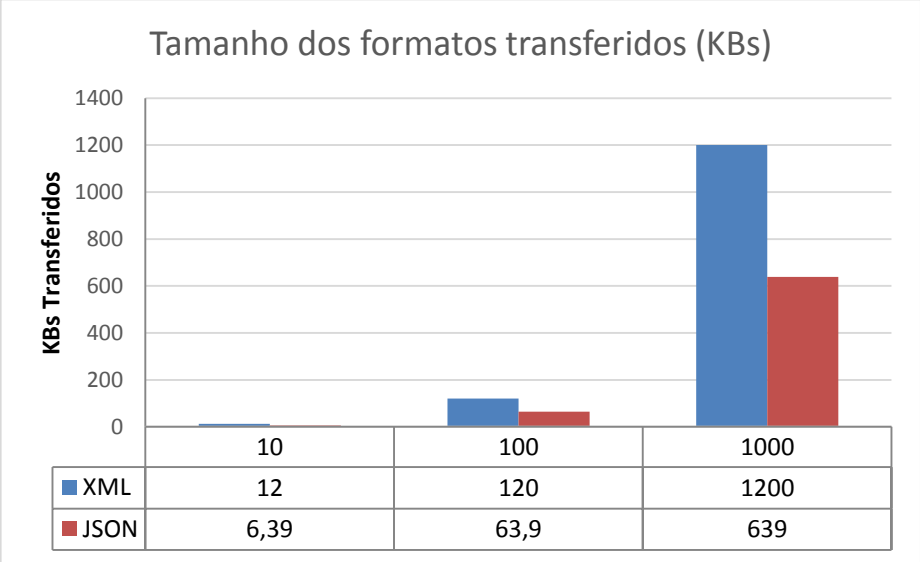


Figura 33: Tamanho dos formatos transferidos (*KBs*)

Para validar este desempenho, utilizou-se o mesmo ficheiro usado para comparar o formato de ficheiros (ver a secção 2.8), no teste de desempenho verificou-se que o formato de *JSON* é 47% mais pequeno em média que o formato de *XML*. Este teste demonstra que os dados são tratados mais depressa pelo navegador resultando assim num menor tempo de resposta na apresentação de conteúdos.

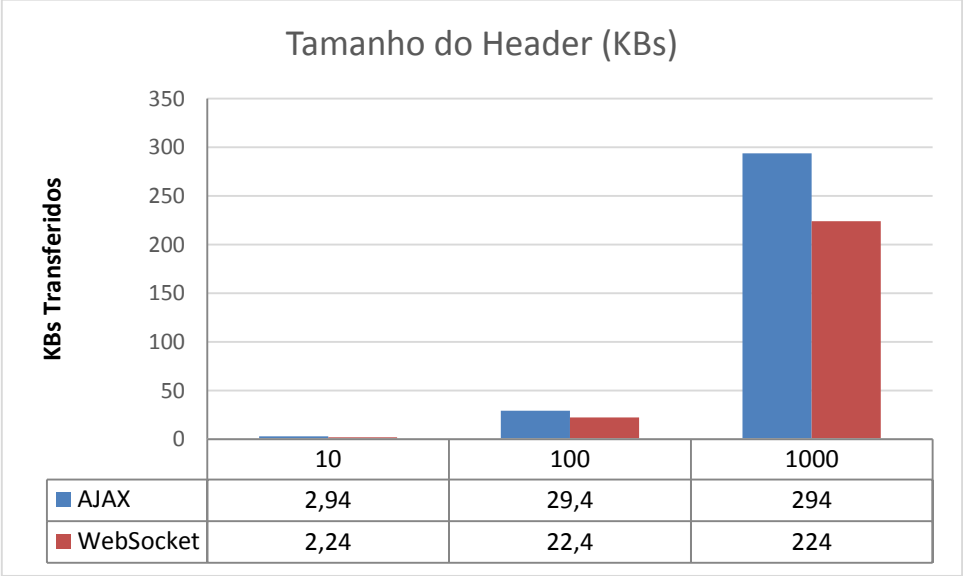


Figura 34: Tamanho do *Header* (Handshake)

Para validar este resultado utilizou-se o tamanho dos cabeçalhos da prova de conceito. Para o *AJAX* usou-se o *Header* do *HTTP*, pois o cabeçalho é composto das mesmas informações, para o *WebSocket* usou-se o cabeçalho gerado pela aplicação que é idêntico ao descrito do *RFC 6455*. Verificou-se que o cabeçalho de *WebSocket* é ligeiramente mais pequeno (224 *bytes*) comparado com o *AJAX* (229 bytes) o que traduz em uma melhoria de desempenho de 33%.

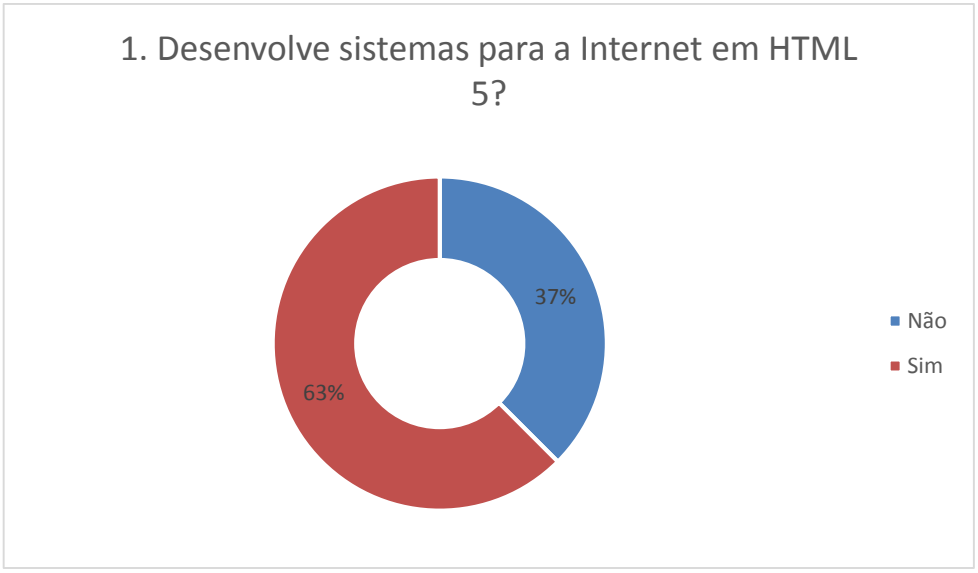
Os testes de desempenho efetuados demostram que a tecnologia *WebSockets* pode ser a escolha na implementação de uma solução de transmissão de dados em tempo real para páginas de internet, desenvolvidos sobre as últimas especificações da *W3C*. Esta conclusão permite-nos responder às perguntas de investigação 1) e 3).

4.4. Inquérito

Foram apresentados inquéritos a programadores que fornecerão respostas que permitam validar a correta implementação da proposta[16].

O Inquérito é composto por dez (10) perguntas simples. As perguntas foram divididas em dois grupos de cinco. Sendo que as perguntas de 1 a 5 estão relacionadas com o desenvolvimento e as perguntas 6 a 10 com a utilização. O inquérito foi enviado a 50 pessoas e foi obtida uma taxa de resposta 24%.

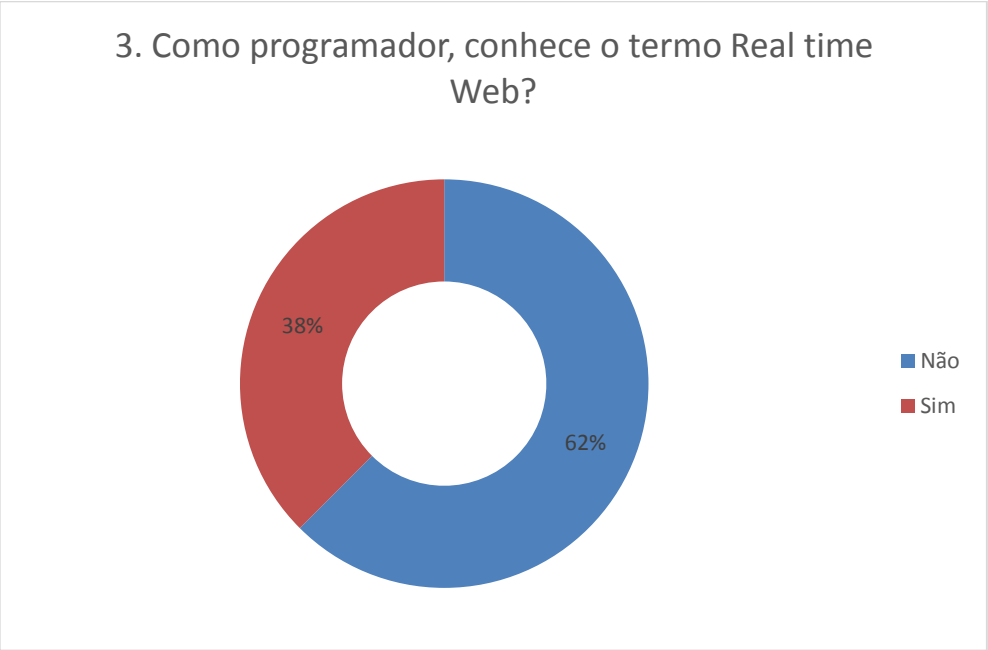
Os resultados são apresentados segundo um gráfico de anel pois é o tipo de gráfico que se enquadra melhor no tipo de dados que se deseja obter deste inquérito, ou seja, as percentagens a cada resposta. Os gráficos estão identificados com o número da pergunta e o título à pergunta sendo que a legenda identifica o nome das respostas possíveis. O valor numérico é expresso em percentagem dentro do anel.



O objetivo da resposta à pergunta 1 é perceber na amostragem a quantidade de programadores que utiliza tecnologia *HTML 5*. Esta é uma pergunta pertinente pois a solução *RTML* funciona sobre a tecnologia *HTML e JavaScript* logo é importante definir a quantidade de utilizadores potenciais da solução. No total, as respostas (ver Gráfico 1 do Anexo III) foram as esperadas, com 63% dos inqueridos a confirmarem a utilização do *HTML5* nos seus produtos, contra os restantes 37% que preferem a continuação da versão anterior do *HTML* ou outros tipos de tecnologias idênticas como a combinação de *XML/XSL*. Estes resultados demonstram que têm existido diversos

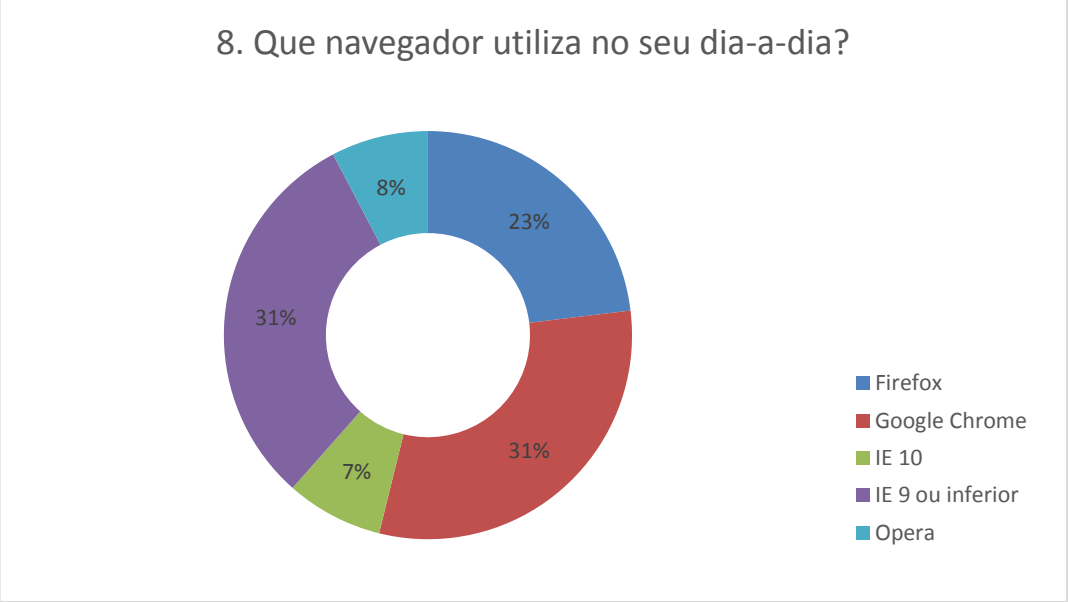
[16] Devido ao sistema proposto ser um sistema especializado, o número de entrevistas está limitada ao número de pessoas disponíveis e com capacidade de resposta.

esforços das produtoras de *Software* nacionais para acompanharem a tendência do mercado, principalmente no desenvolvimento de *web sites*.



A pergunta 3 visa responder à questão mais pertinente do inquérito, saber quantos programadores conhecem o termo *Real Time Web*. Esta questão é essencial para conhecer o estado de utilização da tecnologia no mercado. Nesta questão a resposta apesar de estar claramente abaixo dos cinquenta por cento ofereceu um grau de satisfação muito elevado pois representa quase quarenta por cento das respostas como um Sim.

Apesar de esta tecnologia ser muito pouco utilizada no mercado nacional para o desenvolvimento de produtos de *software*, é importante e satisfatório saber que uma grande percentagem dos programadores reconhece este tipo de tecnologias.



À pergunta 8 era tido como objetivo principal reconhecer quais os navegadores mais utilizados pelos programadores/utilizadores, por forma, a reconhecer qual a percentagem de utilizadores que usam navegadores com suporte a *HTML5* e *WebSockets*. De notar que nesta questão utilizou-se como possíveis respostas os navegadores de *internet* mais utilizados em Portugal.

As respostas surpreenderam pela positiva, pois demonstram que os dados fornecidos pela *W3Schools* não são os mais adequados à realidade nacional. Mesmo assim verificou-se que existe uma taxa superior a cinquenta por cento na utilização de navegadores com o suporte às tecnologias inerentes ao desenvolvimento de soluções em tempo real.

Devido à extensa quantidade de informação extraída nesta subsecção estão incluídos os gráficos mais importantes dos elementos de informação, sendo que os restantes foram incluídos no **Anexo III**.

4.5. Avaliação de resultados

Os resultados obtidos através de testes e do protótipo mostram que o protocolo *WebSocket* permite obter uma oferta melhorada face à atualmente existente para a distribuição de conteúdo em tempo real. A *API* de *WebSockets* fornecida pela *W3C* adequa-se claramente á maioria dos casos em que é necessária a utilização deste tipo de tecnologias.

Na prova de conceito conseguiu-se atingir o objetivo proposto, recordando que o objetivo era o de efetuar uma comunicação funcional entre um servidor e uma *web site*

através da *framework* implementada neste trabalho. Este protótipo serviu como base para o desenvolvimento dos testes de desempenho.

Os testes de desempenho provaram como já foi descrito que a tecnologia *WebSockets* e o paradigma *publisher/subscriber* são as tecnologias mais adequadas na implementação de soluções que necessitem de integrar dados em tempo real. Soluções que podem ser bastante diversificadas como já foram referidas: finanças, apostas desportivas, dados, ou mesmo o *Big Data*.

Os testes unitários são testes que devolvem um valor esperado, geralmente um valor booleano ou numérico que indica ao programador que a função devolve um valor esperado. Estes testes são importantes pois permitem saber se a função executa corretamente determinados parâmetros obtendo resultados esperáveis. Neste tipo de testes à *framework*, o sistema *Unit Test* da *Microsoft Visual Studio 2012* executou corretamente os testes que foram processados, obtendo na maioria dos casos resultados positivos como era espectável.

As respostas ao inquérito geraram resultados satisfatórios. Apesar dos resultados da Pergunta 4 serem bastante insatisfatórios por falta de qualquer resposta à pergunta, os resultados obtidos no geral permitiram perceber que os programadores ainda desconhecem este tipo de tecnologias, optando por outras tecnologias com maior visibilidade no mercado como o *AJAX* ou o *Postback*, apesar de a maioria dos inqueridos já optar por linguagens semânticas mais acuais como o *HTML 5*.

5. Considerações Finais

Este capítulo apresenta as conclusões obtidas do trabalho, avaliando os resultados alcançados, bem como discutindo trabalhos futuros.

A *Internet* hoje está a passar por um período de elevada transformação e os conteúdos são cada vez mais. Estima-se que a internet hoje duplique os seus conteúdos a cada dois anos, sendo que as estimativas para os próximos dez anos sejam que os conteúdos da internet sejam duplicados a cada sete dias. Com esta taxa de atualização o atual modelo é impossível de servir tantos pedidos de informação, o que requer uma atualização ao nível da infrastrutura computacional.

A pensar neste problema a *real time web* parece ser o modelo ideal, pois como referido, funciona como um sistema distribuído. Este funcionamento trás inúmeras vantagens, uma delas é a libertação do servidor *WWW* nos vários pedidos constantes ao servidor, libertando assim largura de banda necessária para a receção de mais pedidos e de envio de informação.

Neste contexto, o desenvolvimento de uma *framework* que possa agilizar a transação da plataforma atual para um *WWW* mais responsável, tem vindo a ganhar foco e interesse por parte de grandes grupos económicos ligados às *TI.* Alguns produtos interessantes surgiram nos últimos dois anos, entre eles o *SignalR* e o *pubnub.com*, que oferecem uma quantidade de funcionalidades interessantes mas pecam pela falta de suporte a funcionalidades como o *Matching* ou o filtro de informação.

A partir deste "*gap*", que não é uma falha mas sim a ausência de uma funcionalidade não introduzida nas *API's* por opção, surgem então outras alternativas como a solução aqui apresentada.

Como referido anteriormente ao longo do documento, um problema encontrado foi a complexidade da utilização das soluções existentes. Muitas delas possuem o código fechado e outras, um código complexo de entender. A solução aqui apresentada visa simplificar este processo, oferecendo funções simples e num modelo *open source*.

Finalmente, durante a elaboração deste documento verificou-se uma crescente oferta de soluções que oferecem um modelo idêntico ao proposto, através da disponibilização de *API's*.

Este crescente interesse deve-se ao facto de as principais empresas do mercado tecnológico estarem a apostar em serviços na *Cloud* oferecendo aos seus utilizadores soluções com um valor acrescentado, por vezes difíceis de suportar. Tal facto incentivou a descrever o melhor possível a tecnologia proposta neste documento, bem como o desenvolvimento de um protótipo funcional.

Quanto aos objetivos, atingiu-se os esperados, apesar de algumas limitações. As perguntas de investigação foram devidamente respondidas através da verificação do estado da arte, e com o desenvolvimento do protótipo. A tecnologia descrita neste documento está em constante desenvolvimento pois é uma tecnologia muito atual e difícil de compreender por vezes, pois envolve diversos temas e tecnologias relacionadas entre si, muitas delas atualizadas frequentemente.

5.1. Limitações

A implementação de uma *framework* passa pelo desenvolvimento de diversas componentes que têm de funcionar em conjunto. Essas componentes podem ser tanto bibliotecas de desenvolvimento como funções e suas dependências, o que em situações de alta pressão podem funcionar com poucas otimizações e deficientemente parametrizadas.

Pode-se ter ocorrido esse erro. Futuramente tenciona-se continuar com o projeto, pois este foi disponibilizado à comunidade de programadores através da rede social de programação *Github.com – https://github.com/pmcfernandes/rtml*, e por acreditar que este tipo de tecnologia vai melhorar muito as soluções de internet futuras e principalmente os dispositivos móveis sensíveis ao toque que utilizem *Apps* com acesso à internet para a obtenção de informação.

O tempo foi um fator limitativo no desenvolvimento deste projeto. A dificuldade na implementação de tecnologias que utilizam camadas de comunicação de *Sockets* tornou a implementação deste projeto mais limitado do que inicialmente previsto. Por isso optou-se pela utilização de uma biblioteca previamente desenvolvida denominada *SuperWebSocket* implementada por *Kelly Jiang*.

Outra limitação que pode ser encontrada neste documento é a quantidade de respostas obtidas no inquérito proposto. Dos 50 inqueridos apenas 24% responderam, ou seja 12 inquiridos deram a sua contribuição. Este número pode-se dever a várias situações, desde a falta de interesse por parte dos inqueridos em preencher inquéritos, ou pelo curto prazo de dois meses para o preenchimento do inquérito.

Em contraste a estas limitações, o documento foi compensado com uma descrição abrangente de todos os elementos que compõem a tecnologias bem como o estado da arte da mesma.

5.2. Trabalhos Futuros

Os trabalhos futuros passam por remover o suporte à biblioteca de desenvolvimento que foi usada na implementação deste projeto. A ideia será desenvolver uma *API* que substitua essa biblioteca, desenvolvendo uma biblioteca que seja compatível com o formato proposto no *RFC 6455*. Ao mesmo tempo desenvolver uma interface que permita a rápida adoção dos diversos formatos de *WebSockets* existentes *DRAFT-75, DRAFT-76, HYBRID 10, HYBRID 13 (RFC 6455).*

Na camada de Criptografia, implementar os diversos tipos de encriptação como por exemplo o *AES* por forma a não estar dependente do *SHA1* oferecido nativamente, bem como implementar o suporte a *Secure WebSockets* através de certificados digitais.

No *Event Handler*, mais especificamente no *Matching*, suportar outros formatos além de *Regular Expressions*, visto que estas apresentam algumas dificuldades de interpretação à maioria dos programadores.

Bibliografia

AYR Consulting. (2011). *Real time and the birth of the Web 3.0.* Lisboa: AYR Consulting. Retrieved from http://www.realtime.co/cache_bin/XPQlr0QXX818A7SyScZMb1ZKU.pdf

Baldoni, R., Querzoni, L., & Virgillito, A. (2009). *Distributed Event Routing in Publish/Subscribe.* Roma, Italia: Universit`a di Roma "La Sapienza".

Baptista, G., Endler, M., Rubinsztejn, H., & Sacramento, V. (2005). *Uma API Pub/Sub para Aplicações Móveis Sensíveis ao Contexto.* Rio de Janeiro, Brasil: Pontifícia Universidade Católica.

Berners-Lee, T. (1996, Maio 1). *IETF RFC 1945*. Retrieved from Hypertext Transfer Protocol - HTTP/1.0: http://www.ietf.org/rfc/rfc1945.txt

Braga, M. (2012, Maio 16). *Say Hello to the real real-time web*. Retrieved Outubro 23, 2012, from Ars Techinica: http://arstechnica.com/business/2012/05/say-hello-to-the-real-real-time-web

Cao, G. (2011). CSE 513: Distributed Systems. *Mobile Computing and Networking Conference* (pp. 4,5). Pennsylvania: The Pennsylvania State University.

Carbou, M. (2011, Junho 19). *Reverse AJAX, Introduction Comet*. Retrieved Outubro 20, 2012, from DeveloperWorks, IBM: http://www.ibm.com/developerworks/web/library/wa-reverseajax1/index.html

Carvalho, A. (2008). *Manual de Ferramentas da web 2.0 para Professores.* Lisboa: Ministério da Educação.

Chakinala, R. C., Kumarasubramanian, A., & Laing, K. A. (2007). *Playing Push vs Pull: Models and Algorithms for Disseminating Dynamic Data in Networks.* Madras, India: Dept. of Comp. Sci. & Engr IIT Madras.

Chockler, G., Melamed, R., Tock, Y., & Vitenberg, R. (2006). *SpiderCast: A Scalable Interest-Aware Overlay for Topic-Based Pub/Sub Communication.* Oslo: Department of Informatics, University of Oslo, Norway.

Dignan, L. (2011, Abril 22). *Cloud computing market: $241 billion in 2020*. Retrieved Dezembro 28, 2012, from Zdnet.com: http://www.zdnet.com/blog/btl/cloud-computing-market-241-billion-in-2020/47702

Eugster, P., Felber, P., Guerraoui, R., & Kermarrec, A.-m. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 114,131.

Eugster, P., Holzer, A., & Garbinato, B. (2005). *Location-based Publish/Subscribe.* Lausanne, Switzerland: Distributed Object Programming Lab.

Fette, I., & Melnikov, A. (2011, Dezembro 1). *IETF RFC 6455*. Retrieved from The WebSocket Protocol: http://tools.ietf.org/html/rfc6455

Fromm, K. (2008). The Real-Time Web and its Future. (M. Kirkpatrick, Interviewer)

Furukawa, Y. (2011). Web-based control application using WebSocket. *Proceedings of ICALEPCS2011*, 673.

Gilmour, S., & MG, S. (2009, Dezembro 7). *Google Aims To Push the Speed of Light with Realtime results*. Retrieved Outubro 20, 2012, from TechCrunch: http://techcrunch.com/2009/12/07/google-realtime

Google Code University, Google Inc. (2011, Janeiro 1). *Introduction to Distributed System Design*. Retrieved Dezembro 21, 2012, from Google Code University: http://code.google.com/intl/pt-PT/edu/parallel/dsd-tutorial.html

Hämäläinen, H. (2012). *HTML5: WebSockets.* Aalto University, Department of Media Technology.

Kirkpatrick, M. (2012, Setembro 22). *Explaining the Real-Time Web in 100 Words or Less*. Retrieved Outubro 2012, 27, from Read Write Magazine: http://readwrite.com/2009/09/22/explaining_the_real-time_web_in_100_words_or_less

Lagutin, D. (2011). Security: Packet Level Authentication and Pub/Sub Security Solution. *Pursuing a Pub/Sub Internet* (p. 20). Helsinki: Helsinki Institute for Information Technology.

Lopes, J. (2012). *Suporte para Interface Web de Aplicações Distribuídas: Estudo da Tecnologia Websocket.* Rio Grande do Sul: Universidade Federal do Rio Grande do Sul.

Lubbers, P., & Greco, F. (2012). *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web* . Sofia: Kaazing Corporation.

Mason, R. (2011, Julho 14). *Real-time Web and Streaming APIs*. Retrieved Dezembro 12, 2012, from MuleSoft Blog: http://blogs.mulesoft.org/real-time-web-and-streaming-apis/

Microsoft patterns & pratices. (2012, 1 1). *Publish/Subscribe*. Retrieved Dezembro 15, 2012, from Microsoft patterns & pratices: http://msdn.microsoft.com/en-us/library/ff649664.aspx

Morimoto, C. (2008). *Redes: Guia Prático.* São Paulo: GDH Press/Sul Editores.

Parr, B. (2011, Agosto 4). *Google To Revive Realtime Search, Thanks to Google+*. Retrieved 1 10, 2013, from Mashable: http://mashable.com/2011/08/03/google-realtime-search-revive/

Pierro, G., Cavallari, F., Di Guida, S., & Innocente, V. (2010). Fast access to the cms detector condition data employing. *Technical Report CMS-CR-2010*, 222.

Porto Editora. (2012, Janeiro 1). *Definição de ambiente.* Retrieved Dezembro 28, 2012, from Dicionário da Língua Portuguesa: http://www.infopedia.pt/lingua-portuguesa/ambiente

Rumpe, B., & Wimmel, G. (1999). A Framework for Realtime Online Auctions. *Managing Information Technology in a Global Economy* (pp. 908--912). Munique: Munich University of Technology. Retrieved from http://www.se-rwth.de/~rumpe/publications/papers/RW01/RW01.pdf

Silvestre, B. O. (2005). *Serviços de Notificação de Eventos Baseados em Publish/Subscribe.* Rio de Janeiro: Universidade Católica do Rio de Janeiro.

Tanenbaum, A. (1995). *Distributed Operating Systems.* New Jersey: Prentice-Hall.

Varela, T. (2012). *Implementação e Análise da Utilização de Websockets em Sistemas Computacionais.* Rio Grande do Sul: Universidade Luterana do Brasil.

Varela, T., & Stanley, L. (2012). *Implementação e análise da utilização de websockets em sistemas computacionais.* Rio Grande do Sul: Universidade Luterana do Brasil.

w3schools. (2012, Janeiro 1). *w3Schools*. Retrieved from w3Schools AJAX: http://www.w3schools.com/ajax/default.asp

Wegner, W. (2012, Agosto 22). *Generating C# Classes from JSON*. Retrieved Janeiro 11, 2012, from From the whiteboard to the keyboard: http://www.wadewegner.com/2012/08/generating-c-classes-from-json/

Wikimedia Foundation, Inc. (2013, Março 21). *Wikipedia*. Retrieved from Tecnología Push: http://en.wikipedia.org/wiki/Push_technology

Zhangling, Y., & Mao, D. (2012). A Real-Time Group Communication Architecture Based on WebSocket. *International Journal of Computer and Communication Engineering*, 408.

Anexos

Anexo I

Versão integral do componente de *JavaScript* da *framework* proposta.

```
// Rtml.js
// Copyright (C) 2013 Pedro Fernandes

/* This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA */

var rtml = Class.create();

rtml.prototype = {

        /**
         * Creates a new instance of the rtml object
         *
         * @param {string} id       Unique identifier for client
         * @param {json}   options  Server connections options
         * @return
         *
         * <code>
         *    var subscriber = new rtml('B3CD21CE-3A52-4F8E-904D-2BB80E44FFD6', { host:
'www.pfernandes.pt', port: 8080, debug: true });
         * </code>
         */
        initialize: function (uniqueID, options) {
                this._opt = {
                        host: window.location.hostname,
                        port: 80,
                        route: '/',
                        debug: false
                }

                Object.extend(this._opt, options);

                this._uniqueID = uniqueID;
                this._host = "ws://" + this._opt.host + ":" + this._opt.port;
        },

        /**
         * Gets the unique id of the current customer
         *
         * @return {string} id
         *
         * <code>
         *    var subscriber = new rtml(...); (...)
         *    alert(subscriber.id());
         * </code>
         */
        uniqueID: function() {
                return this._uniqueID;
        },

        /**
         * Check if the real-time connection was established
         *
         * @return {boolean}
```

```
 *
 * <code>
 *    var subscriber = new rtml(...); (...)
 *    alert(subscriber.connected());
 * </code>
 */
connected: function() {
        if (!this._socket) {
                return false;
        } else {
                return !(this._socket.readyState == 2);
        }
},

/**
 * Set contents to subscribe in message
 *
 * @param {regex} match    Regular expression to filter messages
 * return {string}
 *
 * <code>
 *    var subscriber = new rtml(...);
 *    subscriber.open('/realtime/server');
 *        subscriber.on('open', function() {
 *                        subscriber.subscribe(/\[A-Z]\);
 *        });
 * </code>
 */
subscribe: function(match) {
        if (typeof match == 'undefined') {
                return this._match;
        }

        this._match = match || "*";
        this.send({
                subscribe: match
        });
},

/**
 * Open a real-time connection with the server
 *
 * @return
 *
 * <code>
 *    var subscriber = new rtml(...);
 *    subscriber.open('/realtime/server');
 * </code>
 */
open: function() {
        var _debug = this._opt.debug;
        var _id = this.uniqueID();

        if(!("WebSocket" in window)) {
                console.error("WebSocket is not supported.");
                return;
        }

        this._socket = new WebSocket(this._host + this._opt.route);

        this.on("close", function() {
                if (_debug == true) {
                        console.log ("Client " + _id + " is closed.");
                }
        });

        this.on('receive', function(msg) {
                if (_debug == true) {
                        console.log(msg);
```

```
                }
        });

        this.on("open", function() {
                if (_debug == true) {
                        console.log ("Client " + _id + " is connected.");
                }

                this.send({
                        uniqueID: _id
                });

        });
},

/**
 * Close the existing connection
 *
 * @return
 *
 * <code>
 *    var subscriber = new rtml(...); (...)
 *    subscriber.destroy();
 * </code>
 */
destroy: function() {
        if (this.connected()) this._socket.close();
},

/**
 * Send a message to the server
 *
 * @param {json} msg   Message to send to server
 * @return
 *
 * <code>
 *    var subscriber = new rtml(...);
 *    subscriber.open('/realtime/server');
 *    subscriber.send({
 *      uniqueID: (...),
 *              subscribe: (...),
 *              msg: {
 *              (...)
 *      }
 *    })
 * </code>
 */
send: function(msg) {
        if (!this.connected()) this.open();

        this._socket.send(Object.toJSON(msg));
},

/**
 * Raises an event related with the connection
 *
 * @param {string}   event     Event want to fire
 * @param {function} callback  Function that returns the server response
 * @return
 *
 * <code>
 *    var subscriber = new rtml(...); (...)
 *    subscriber.open('/realtime/server');
 *    subscriber.on('receive', function(msg) {
 *      (...)
 *    });
 *    subscriber.on('open', function()  {
 *      (...)
 *    });
```

```
         *    subscriber.on('close', function() {
         *      (...)
         *    });
         * </code>
         */
        on: function(event, callback) {
                if (event == "open") {
                        this._socket.onopen = function() {
                                callback();
                        };
                }

                if (event == "close") {
                        this._socket.onclose = function() {
                                callback();
                        };
                }

                if (event == "receive") {
                        this._socket.onmessage = function(msg) {
                                callback(msg.data.toString().evalJSON());
                        };
                }
        }

};
```

Anexo II

Versão integral do componente de *C#* da *framework* proposta.

```
// Publisher.cs
// Copyright (C) 2013 Pedro Fernandes
/* This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Place, Suite 330, Boston, MA 02111-1307 USA
*/

using Realtime.Common;
using System;
using System.Threading;

namespace Realtime.Publisher
{
    public class Publisher : IPublisher, IDisposable
    {
        private WebSocket4Net.WebSocket _socket;
        private ServerConfig _config;

        /// <summary>
        /// Initializes a new instance of the <see cref="Subscriber" /> class.
        /// </summary>
        public Publisher()
        {
            _config = new ServerConfig() {
                Host = "localhost",
                Port = 8080,
                Route = "/",
                Debug = true
            };
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Subscriber" /> class.
```

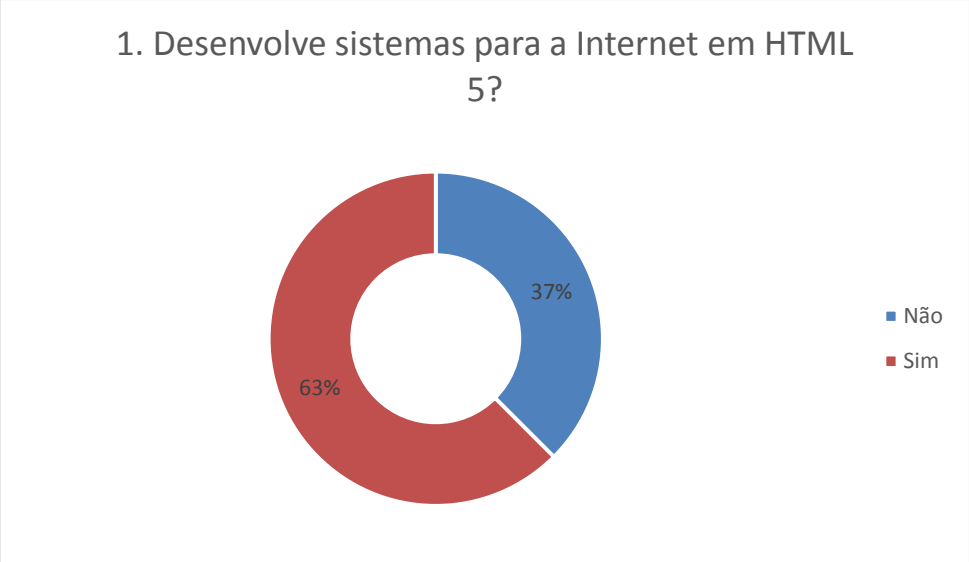
```
/// </summary>
/// <param name="id">The id.</param>
/// <param name="config">The config.</param>
public Publisher(ServerConfig config)
    : this()
{
    this._config = config;
}

/// <summary>
/// Check if the real-time connection was established
/// </summary>
/// <returns>
///   <c>true</c> if this instance is connected; otherwise, <c>false</c>.
/// </returns>
private bool IsConnected()
{
    if (_socket == null)
    {
        return false;
    }

    return (_socket.State == WebSocket4Net.WebSocketState.Open);
}

/// <summary>
/// Open a real-time connection with the server
/// </summary>
private void Open()
{
    _socket = new WebSocket4Net.WebSocket(_config.ToString());
    _socket.EnableAutoSendPing = true;
    _socket.AutoSendPingInterval = (1000 * 60); // 1 Minute
    _socket.Open();
}

/// <summary>
/// Publishes the specified message.
/// </summary>
/// <param name="message">The message.</param>
/// <param name="callback">The callback.</param>
public void Publish(string message, Action<string> callback)
{
    this.Publish(message, callback);
}

/// <summary>
/// Publishes the specified message.
/// </summary>
/// <param name="message">The message.</param>
/// <param name="callback">The callback.</param>
public void Publish(object message, Action callback)
{
    if (_socket == null)
    {
        this.Open();
    }

    while (!this.IsConnected())
    {
        Thread.Sleep(100);
    }

    dynamic d = new { msg = message };
    _socket.Send("PUB " + ((object)d).Stringfy());

    callback.Invoke();
}
```

```
        /// <summary>
        /// Performs application-defined tasks associated with freeing, releasing, or resetting
unmanaged resources.
        /// </summary>
        public void Dispose()
        {
            if (_socket != null) _socket.Close();
            if (_socket != null) _socket = null;
            if (_config != null) _config = null;
        }
    }
}
```

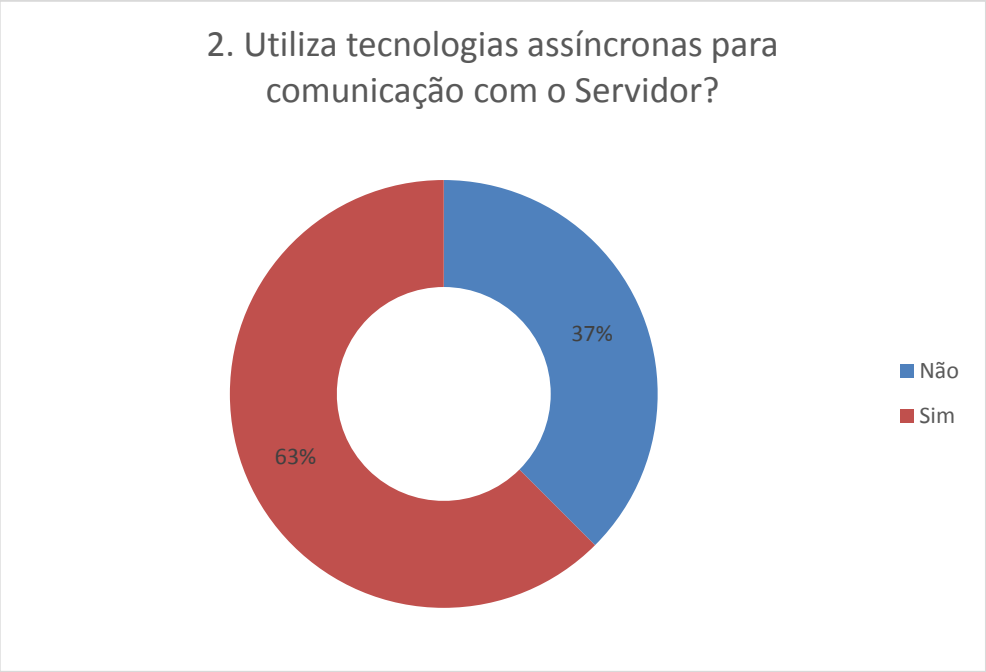
Anexo III

Este anexo mostra a informação extraída do inquérito proposto a 50 participantes. Para mais informações consultar a secção Inquérito do documento.

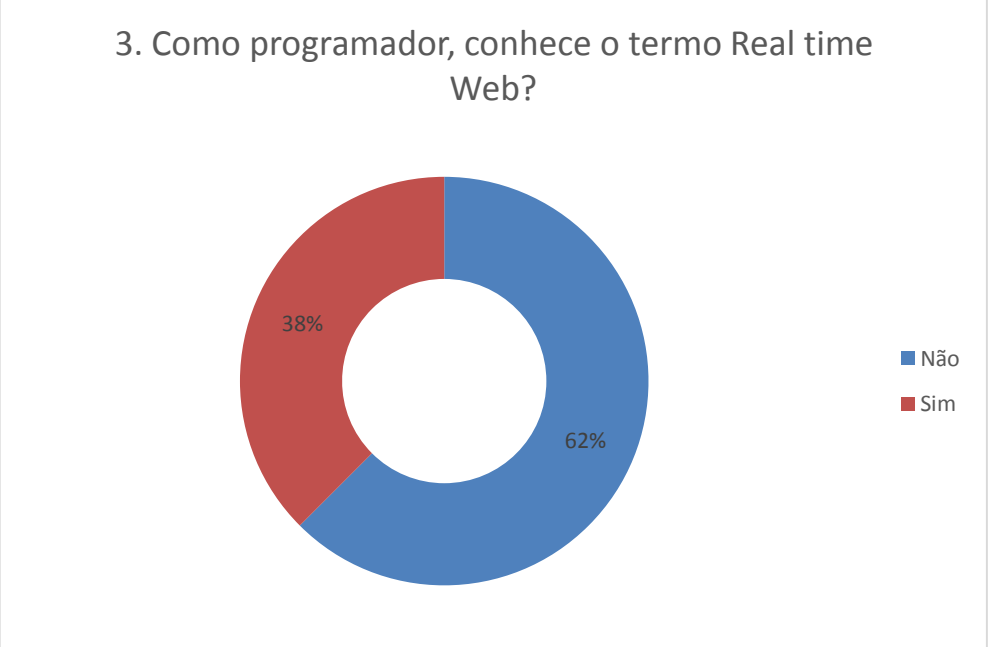
A **Pergunta 1** permite saber que o programador está a par do *HTML 5*.



Através da **Pergunta 2** permite-se saber se o programador conhece as técnicas *AJAX* através do desenvolvimento assíncrono entre o servidor e o cliente. A resposta à pergunta com um Não indica que o programador não tem esta técnica como uma utilização contante usando claramente o método *Postback (POST/GET).*

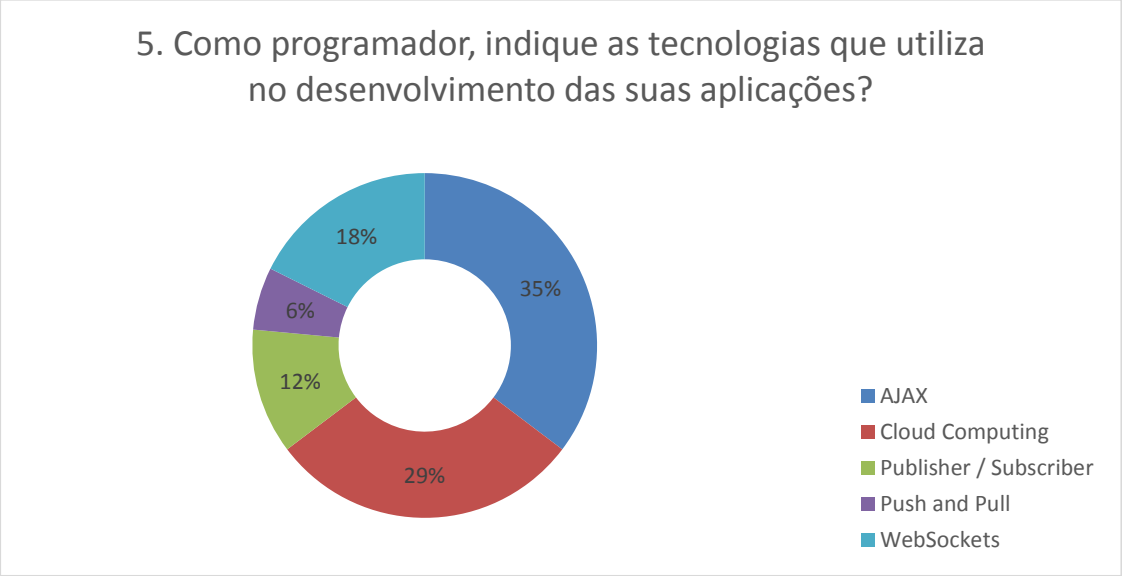


A **Pergunta 3** permite saber se conhece o termo real time.

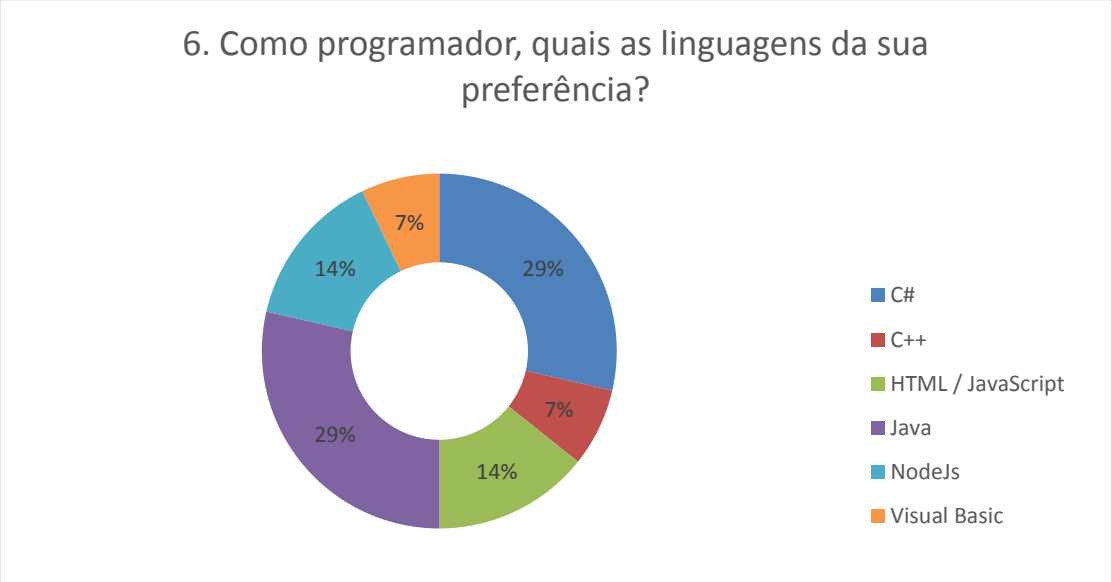


Na **Pergunta 4**, tentou-se obter mais informações ao utilizador perguntando: "Indique que tipo de funcionalidades deseja ver presente em uma framework real time web." No inquérito submetido não se obteve qualquer resposta a esta pergunta. Apenas é possível especular o motivo pelo qual não houve respostas sendo que a principal é por ser uma pergunta de texto livre o que inibe o utilizador a preencher a resposta.

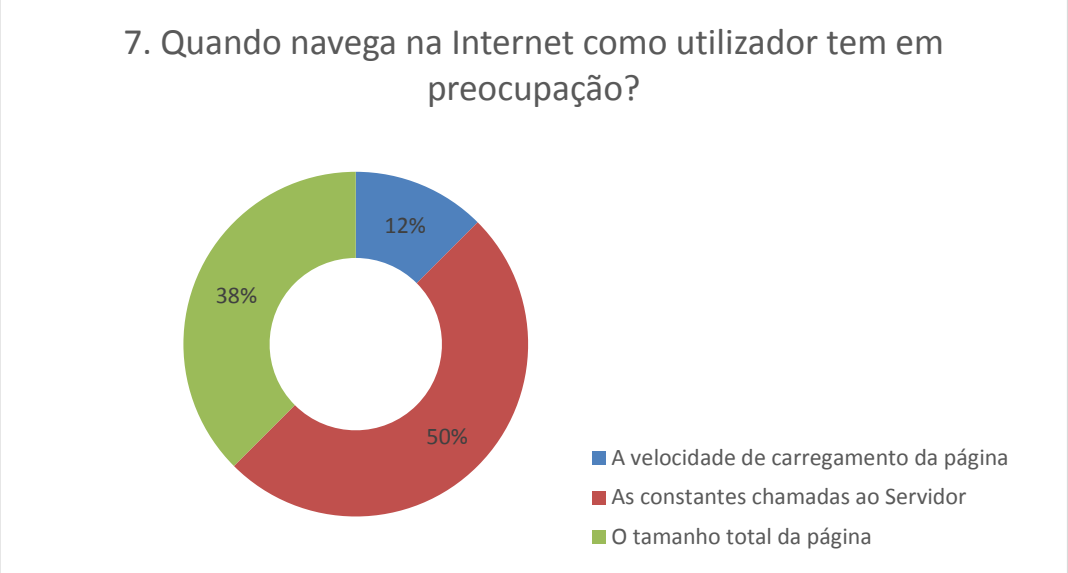
A **Pergunta 5** permite saber se o utilizador conhece ou utiliza as tecnologias descritas como tecnologias na implementação de soluções em tempo real, sendo elas o *AJAX* ou *WebSockets* e o paradigma *pub/sub* bem como a utilização de *Cloud Computing*. As várias respostas possíveis oferecem-me a indicação do caminho a tomar por exemplo optar por uma alternativa *AJAX* e *Push and Pull* ou uma plataforma *pub/sub* em *Cloud Computing*.



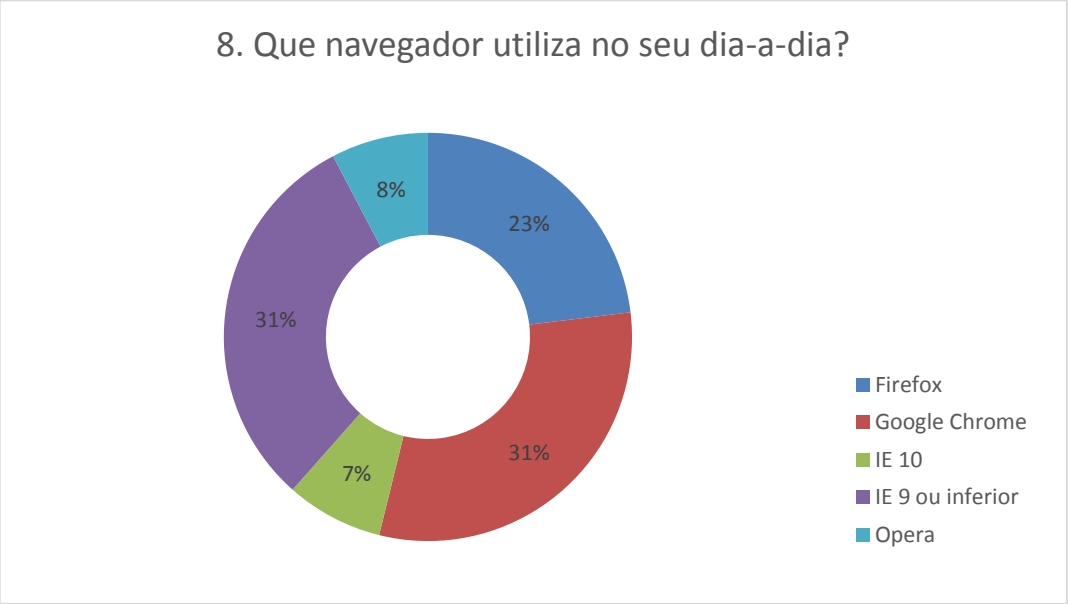
O desenvolvimento atual será feito em *C#*, sendo a **Pergunta 6** importante para perceber futuramente se é viável migrar a mesma implementação para outras linguagens.



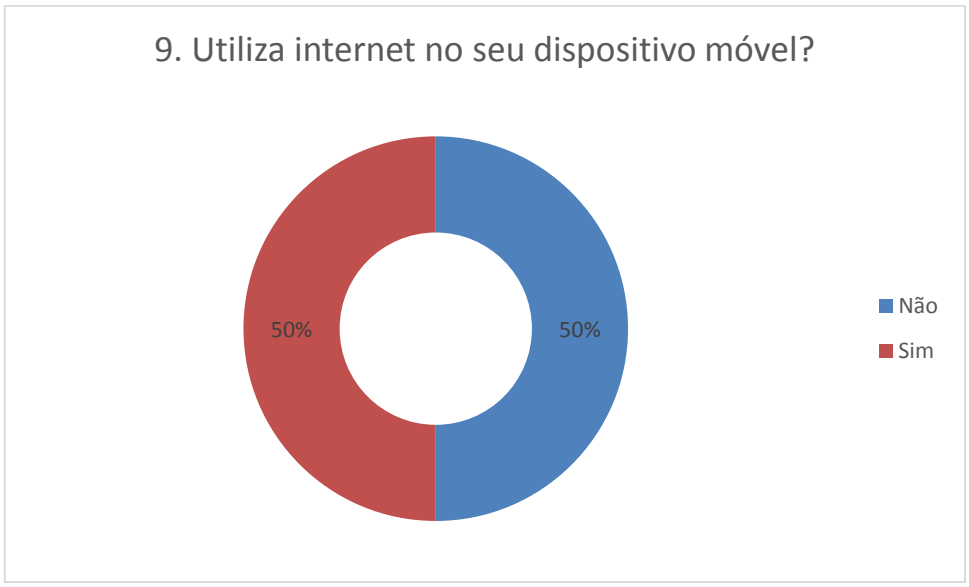
A **Pergunta 7** permite confirmar quais as principais preocupações de um utilizador ao navegar em uma web *site*.



A **Pergunta 8** tenta responder a quais os navegadores mais usados pelos utilizadores neste caso também programadores. Esta pergunta é um complemento a **Pergunta 5** pois os navegadores mais antigos (*IE 9* ou inferior) não suportam estas tecnologias de *WebSockets*.



A **Pergunta 9** é uma pergunta relacionada com a utilização em dispositivos móveis que é uma das razões da adoção desta metodologia devido à sua baixa capacidade de transferência de informação em rede.



Sendo que a **Pergunta 10** complementa a resposta à **Pergunta 9.**