INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




WebSockets e a sua aplicação no mundo Web

PEDRO MANUEL OLIVEIRA ALMEIDA
outubro de 2019

POLITÉCNICO
DO PORTO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




WebSockets e a sua aplicação no mundo Web

PEDRO MANUEL OLIVEIRA ALMEIDA
Outubro de 2019

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

WebSockets e a sua aplicação no mundo Web

Pedro Manuel Oliveira Almeida

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Paulo Manuel Baltarejo de Sousa

Porto, 13 Outubro de 2019

Dedicatória

Aos meus pais, família, amigos e namorada pelo apoio incondicional sempre que precisei.

Pedro Almeida

Resumo

Hoje em dia as aplicações são desenhadas e implementadas de forma a que sejam constantemente conectados ao mundo *web*. Existe um crescimento exponencial da utilização de aplicações *web* atualmente, e à medida que este crescimento vai acontecendo, novos e inovadores requisitos são enunciados, com o avanço tecnológico.

As aplicações *web* desenvolvidas hoje em dia, baseadas em *WebServices* (HTTP), genericamente apresentam falhas de comunicação e sincronização entre componentes numa arquitetura cliente-servidor. Problemas de sincronização esses que representam lacunas na comunicação existente entre componentes, atrasos na entrega de mensagens, ou ainda excesso de *bytes* transferidos, o que leva ao mau desempenho de uma aplicação *web*, e ao possível abandono por parte dos seus utilizadores por insatisfação.

Grande parte destas aplicações são desenvolvidas utilizando *WebService* APIs, que têm por base o protocolo HTTP. Este protocolo, apesar das suas características o tornarem versátil na sua utilização em aplicações *web*, existem mecanismos (em determinados contextos) que não são tão eficientes, nomeadamente os utilizados em aplicações *web* que necessitem de atualizações de dados em tempo real.

Nesse sentido foi desenvolvido um protótipo implementando um protocolo diferente – *WebSockets* – de forma a tentar obter resultados sobre o desempenho do mesmo e que provasse ser a abordagem a seguir na tentativa de solucionar os problemas de comunicação existentes atualmente no mercado das aplicações *web*. Esses resultados derivam de métricas definidas através de estudos e artigos que definem formas e tipos de testes que se executam nas aplicações *web* para obter relatórios de desempenho.

Após análise das características do protocolo e dos resultados obtidos através do protótipo desenvolvido, conclui-se que o protocolo *WebSockets* é efetivamente mais eficiente do que o utilizado em *WebServices* (HTTP), visto que possui mecanismos internos que permitem atingir menores tempos de resposta médios de pedidos efetuados entre cliente-servidor, e também menos dados transferidos.

Palavras-chave: *WebSockets*, cliente-servidor, comunicação *web*, aplicação *web*, servidor, REST, HTTP, tempo de resposta, *WebService,* versatilidade, sistemas em tempo real

Abstract

The products that are designed and developed today are done in a way so that they are constantly connected to the world wide web. There has been an exponential growth in web applications' usage nowadays, and as growth keeps happening, new and innovative requirements come up.

In the context of the main problem of this thesis, web applications have some issues in terms of communication and data synchronization amongst components within a client-server architecture. These kinds of problems represent not only failures in communication between components, but also delays in message delivery and overhead of transferred information, which leads to a bad performance of the app, and a possible abandonment of its users.

A great majority of the web apps are developed using WebService APIs, which implement HTTP as a basis protocol. This protocol, despite its characteristics turn it into a versatile one, there are certain mechanisms (in certain contexts) that are not that efficient, specifically the ones used in web apps that need real time data updates.

That said, a prototype has been developed using a different protocol – WebSockets – so that results from its performance could be obtained and it proved itself to be the solution of the problem stated above. These results come from multiple studies and articles mentioning different metrics and ways of testing web applications' performance.

After analysing the protocol's characteristics and the results that come from the prototype development, the conclusion is that WebSockets is indeed more efficient than the one WebServices use – HTTP. This comes from observing that the average time response and also the amount of data transferred between components is lower, and therefore the app provides a greater user experience.

Keywords: WebSockets, client-server, web communication, web application, server, REST, HTTP, bandwidth, average response time, WebService, real time system, versatiliy

Agradecimentos

Quero agradecer a todos os mencionados na dedicatória e a todas as pessoas que estiveram presentes no percurso académico que fiz, e que me acompanharam desde sempre, independentemente das circunstâncias.

Quero também dar um agradecimento especial ao meu orientador por todo o apoio na elaboração da presente dissertação, bem como toda a orientação que forneceu ao longo da unidade curricular.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Acrónimos e Símbolos

Lista de Acrónimos

ACK	*Acknowledge*
AHP	*Analytic Hierarchy Process*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
DNS	*Domain Name System*
FFE	*Fuzzy Front End*
FTP	*File Transfer Protocol*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
HTTPS	*HyperText Transfer Protocol Secure*
IP	*Internet Protocol*
MIME	*Multipurpose Internet Mail Extension*
MIT	*Massachusetts Institute of Technology*
NCD	*New Concept Development*
OOP	*Object Oriented Programming*
REST	*Representational State Transfer*
RFC	*Request For Comment*
RPC	*Remote Procedure Call*
SEO	*Search Engine Optimization*
SLA	*Service Level Agreements*
SMTP	*Simple Mail Transfer Protocol*
TCP	*Transmission Control Protocol*
TCP/IP	*Transmission Control Protocol/Internet Protocol*

UDP	*User Datagram Protocol*
UML	*Unified Modeling Language*
URI	*Uniform Resource Identifier*
W3C	*World Wide Web Consortium*

Página em branco

1 Introdução

O primeiro capítulo desta dissertação apresenta uma contextualização e descrição do problema com o objetivo de enquadrar o leitor no tema da mesma. Será composta por um enquadramento teórico e um resumo da análise de valor e dos objetivos pretendidos. Por fim, será apresentada a estrutura do documento.

1.1 Enquadramento

Qualquer sistema informático atualmente possui uma determinada arquitetura de forma a ser operacional. Esta arquitetura consiste numa estrutura de vários subsistemas informáticos organizados de tal forma a que o desenvolvimento do sistema primário seja organizada e suficientemente eficiente[1].

Um determinado tipo de arquitetura necessita do suporte de várias ferramentas que auxiliem na sua implementação, e estas constituem pedaços de *software* com funcionalidades mais genéricas já previamente implementadas e reutilizáveis, facilitando, portanto, o desenvolvimento do novo sistema. No entanto, o uso destas ferramentas implica um alto esforço de formação de conceitos e uma possível dependência das entidades que codificam, no caso destas serem propriedade privada, o qual pode ser considerado uma desvantagem.

Tal como cada tipo de *software* pode conter uma determinada arquitetura, uma aplicação *web* possui um tipo de arquitetura muito específica, denominada de arquitetura cliente-servidor.

1.1.1 Arquitetura cliente-servidor

A arquitetura cliente-servidor, ilustrada na Figura 1, consiste na divisão de uma aplicação em duas partes, ou subaplicações – cliente e servidor. Estas são implementadas tendo como dependência o recurso a uma rede, que por sua vez realiza a interligação entre as duas partes.

O servidor providencia a funcionalidade central e permite com que várias aplicações clientes executem pedidos para realizar uma determinada tarefa. Este aceita o pedido, executa a tarefa, e por fim retorna o resultado ao cliente num determinado formato.

Por outro lado, o cliente é manipulado por um utilizador que executa determinadas ações consoante as funcionalidades disponíveis, podendo estas serem expandidas através da utilização de *plugins*[1].

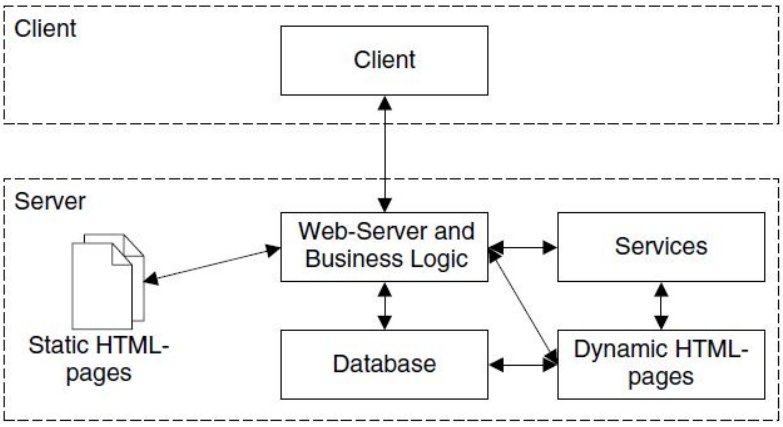

Figura 1 - Arquitetura cliente-servidor[1]

Em teoria a componente servidora pode ser distribuída em conjunto com a componente cliente, caso se trate de um monólito[2]. Porém, em todas as atualizações de código (quer seja no cliente ou servidor) ambos os componentes (cliente e servidor) têm de ser distribuídos novamente, o que pode não ser o mais eficiente. Idealmente o servidor é alojado numa máquina remota num determinado ambiente, enquanto que o cliente é alojado num outro ambiente independente, para que o processo de distribuição seja o mais controlado possível.

No entanto, existem outros componentes importantes na arquitetura de uma aplicação *web.*

[1] pedaços de código genérico que contêm uma funcionalidade específica

A base de uma aplicação *web* assenta numa ligação à rede, por isso é importante referir que há alguns componentes que têm impacto na sua eficiência, tais como:

- *Firewall* – *Software* ou *hardware* que contém regras e filtros específicos para a comunicação existente dispositivos/componentes, dentro redes seguras e/ou inseguras.
- *Web server* – *software* que suporta vários protocolos *web* (tais como HTTP, HTTPS, *etc*.) necessários para comunicação através de pedidos.
- *Database server* – servidor onde normalmente são armazenados dados sobre a lógica de negócio.

Hoje em dia existem muitas classificações no que diz respeito aos meios de comunicação entre aplicações *web*, mas apenas duas são significativamente importantes no mercado – *WebServices* e *WebSockets*[3].

Tipicamente, a implementação tanto dos *WebServices*, como dos *WebSockets* resultam no formato de *Application Programming Interface* (API). O conceito de API consiste numa interface de programação aplicacional que geralmente representa uma possível interação com múltiplos componentes de *software*[4].

A comunicação orientada a serviços (*WebService* API) tem como suporte o protocolo HTTP – *Hypertext Transfer Protocol*. Este protocolo surgiu em 1989 e é considerado o protocolo principal na *World Wide Web*, permitindo efetuar uma comunicação entre vários tipos de aplicações cliente e servidor. Com a ajuda deste protocolo é possível transmitir informação - tal como vídeos, imagens, ficheiros, *etc*. – e comunicar com *browsers*[2], como são exemplo o *Google Chrome*, *Mozilla Firefox*, *Internet Explorer*, entre outros. As classificações distinguem-se pelo tipo de arquitetura que ambas implementam.

O estilo arquitetural REST (*Representational State Transfer*) é a forma mais comum de estruturar um pedido a uma API. Uma API que adere aos princípios REST é designada de RESTful API, onde são utilizados modelos de pedido/resposta em que cada mensagem vinda do servidor representa uma resposta a um pedido com origem no cliente. Como estas estruturas se baseiam em entidades, existem pedidos característicos às mutações ou obtenção dessas mesmas entidades.

Por outro lado, o protocolo *WebSocket* foi normalizado em 2011 pela W3C[5] (*World Wide Web Consortium)* e marca o início de uma nova dinâmica no mundo *web*. Este permite a comunicação bidirecional entre cliente e servidor, sem dependência da resposta (ou da sua ausência) ao pedido feito anteriormente. O modelo de segurança utilizado por este protocolo é comum ao que é utilizado pelos *browsers*. Um dos objetivos tecnológicos deste protocolo é fornecer um mecanismo de comunicação para aplicações baseadas em *browsers* e que precisem de comunicações bidirecionais e que não dependa da existência de múltiplas

[2] navegadores de Internet

conexões HTTP, mas sim através do uso de uma única conexão TCP – *Transmission Control Protocol*[6].

Em suma, as *WebServices* APIs distinguem-se pela comunicação de dados através de interação direta entre entidades, enquanto que as *WebSocket* APIs sustentam uma arquitetura mais direcionada ao envio imediato de informação de volta ao sistema que invocou uma transferência de dados, também denominada *push.*

1.2 Problema

A arquitetura do tipo cliente-servidor é a mais utilizada no mercado das aplicações *web*. Esta arquitetura permite o isolamento de componentes e a sua possível reutilização, em diferentes contextos de negócio.
A necessidade de eficiência da comunicação que existe entre os dois componentes (cliente-servidor) de uma aplicação *web* constitui um fator crucial para o desempenho da mesma. Sem a devida comunicação, os componentes não funcionam de acordo com o que a sua arquitetura permite, tornando a aplicação obsoleta.
Dado que as ligações cliente-servidor são muito frequentes nas aplicações *web* do mercado atual, torna-se necessária uma abordagem que permitam com que estas sejam o mais estável possível, não comprometendo a funcionalidade da aplicação.

O problema reside no facto de que, apesar da existência de várias alternativas de comunicação entre componentes cliente-servidor, estas apresentam, em geral, problemas de sincronização entre os componentes (referidos na Subsecção 2.2.2), o que leva consequentemente à instabilidade de comunicação entre eles. Consideram-se que uma aplicação tem problemas de sincronização quando o utilizador não obtém o *feedback* que a aplicação necessita de fornecer a seu devido tempo, representando a existência de lacunas na comunicação existente entre os vários componentes (cliente-servidor), atrasos na entrega de mensagens, ou ainda excesso de *bytes* transferidos, o que leva a um mau desempenho da própria aplicação *web* em geral.

O mau desempenho de uma aplicação *web* é um fator negativo, pois provoca um possível abandono por parte dos seus utilizadores por insatisfação, visto que estes não conseguem obter o total benefício na sua utilização.

Os problemas de sincronização provêm principalmente do protocolo de comunicação utilizado hoje em dia na maioria das aplicações *web,* que apresenta condicionantes quando aplicados em sistemas de tempo real. Estes sistemas possuem uma necessidade de atualização constante de dados, representando inclusive a possibilidade de o componente servidor notificar o cliente quando existem mudanças de estado (por exemplo, receção de novos dados). As técnicas existentes no protocolo em questão (Subsecção 2.2.2) não são suficientes para responder às necessidades funcionais que este tipo de aplicações requer.

1.3 Objetivos

O principal objetivo desta dissertação é aferir se o protocolo de *WebSockets* realmente tem um impacto na implementação das aplicações *web,* e se é considerado mais adequado do que o protocolo HTTP, através de resultados de uma comparação teórica e prática de um protótipo com determinadas funcionalidades. O protótipo contém uma *WebSocket* API (suportada pelo protocolo de *WebSockets*) e uma REST API. A escolha da implementação de uma REST API prende-se com o facto de que esta existe através do suporte ao protocolo HTTP, e é considerada um dos mais utilizados atualmente a nível global [7].

O objetivo principal será repartido entre vários outros, pois cada uma das comparações realizadas a nível de desempenho ou custos, bem como medição final das vantagens e desvantagens de ambos os protocolos constituem outros objetivos.

Dos objetivos citados, existem várias questões que se pretendem responder ao longo da presente dissertação:

Q1. Que *benchmarks* devem ser implementados de forma a obter uma boa avaliação final entre cada protocolo?

Q2. O protocolo *WebSockets* tem melhor desempenho, escalabilidade e confiabilidade do que o HTTP, suportando também a conexão de vários clientes em simultâneo?

Todas estas questões levam a uma questão que será a final:

Q3. Qual o impacto que o protocolo *WebSocket* tem na implementação de uma aplicação *web* nos dias de hoje, comparativamente com HTTP?

1.4 Análise de Valor

A realização da análise de valor terá como referência o objetivo principal do presente trabalho que é a de trazer valor ao desenvolvimento de aplicações *web*, tendo em conta os custos associados e as métricas de desempenho da infraestrutura responsável pela comunicação interna e/ou externa com outras aplicações.

Para tal, a análise de valor irá ser realizada de acordo com os modelos *New Concept Development* (descrito na Secção 3.2) e *Analytic Hierarchy Process* (descrito na Secção 3.7), definir o valor, valor percecionado e proposta de valor.

Como complemento será também descrito na Secção 3.5 o documento relativo ao Modelo de Negócio CANVAS com o objetivo de clarificar o modelo de negócio.

1.5 Estrutura do documento

O presente documento está dividido em três partes fulcrais, introdução, corpo e conclusões.

Estas partes são constituídas pelos capítulos:

- **Capítulo 1 – Introdução**, onde será feita uma contextualização do projeto, dos seus objetivos, o problema que a solução a implementar tende a colmatar, a sua análise de valor e por fim a estrutura do documento.
- **Capítulo 2 – Contexto, Problema e Estado da Arte**, onde é abordado o contexto do problema, e o estado da arte e são analisadas e avaliadas as tecnologias relevantes, bem como a avaliação das abordagens existentes, de forma a que exista uma melhor leitura dos capítulos seguintes.
- **Capítulo 3 – Análise de Valor**, onde será descrita a proposta de valor do projeto através da identificação de oportunidade, análise da mesma e descrição das ideias geradas. É neste capítulo que é usado o método de avaliação hierárquico (AHP [3]) para determinação da ideia mais vantajosa a implementar.
- **Capítulo 4 – Padrões de *software* e *benchmarks***, onde é apresentada, e detalhada, a análise dos padrões de *software* a ter em conta e são também descritos *benchmarks*.
- **Capítulo 5 – Desenvolvimento do protótipo,** onde é descrito com algum detalhe todo o processo de implementação do protótipo desenvolvido de forma a obter resultados.
- **Capítulo 6 – Avaliação**, onde são testadas e comparadas as estruturas de comunicação entre *web apps*[4]*,* através das suas grandezas de avaliação e hipóteses a testar. São também apresentados resultados de *benchmark*.
- **Capítulo 7 – Conclusões**, onde são apresentadas as conclusões da dissertação, respondendo de forma sumária às questões propostas no capítulo 1.

[3] *Analytic Hierarchy Process* – modelo de auxílio a tomada de decisões de um negócio
[4] Aplicações web

Página em branco

2 Contexto, Problema e Estado da Arte

Este capítulo tem como principal objetivo a contextualização do âmbito do projeto, enquadrando-o num contexto a nível global. É feita uma análise do mercado e do que representa a área de transferência de dados entre aplicações *web*, de forma a que exista um maior conhecimento sobre a temática em que os *WebSockets* estão incluídos.

O capítulo está dividido em cinco partes: contexto e destaque do problema, enquadramento teórico e tecnológico com vista a perceber alguns conceitos que são considerados fundamentais, plataformas de desenvolvimento que sejam relevantes, e por fim apresentação de soluções e abordagens existentes.

2.1 Contexto

O mundo aplicacional teve um exponencial crescimento nos últimos anos com a normalização da Internet no mundo[8]. Estima-se que cerca de 67% de todos os portadores de dispositivos móveis têm acesso à Internet através do seu dispositivo, como também se estima que existam aproximadamente 4.4 biliões de pessoas por todo o mundo com acesso à internet (dados de Janeiro 2019[9]).

Os números tiveram um aumento significativo desde 2014 e tendem a aumentar (Figura 2), sendo a causa do aumento a existência de cada vez mais serviços que realizam as suas tarefas através da Internet, com a população a ter também o acesso à informação cada vez mais facilitado. Desde a leitura de notícias, música, vídeos, acesso a serviços sociais e financeiros, hoje em dia é quase impossível encontrar um serviço que não tenha suporte online, e consequentemente uma aplicação *web* para o efeito.

Uma das áreas com maior crescimento dentro da indústria das aplicações *web* é o *e-commerce*. A aderência ao comércio online tem vindo a crescer, e a *Amazon* é já um dos sites mais visitados hoje em dia.

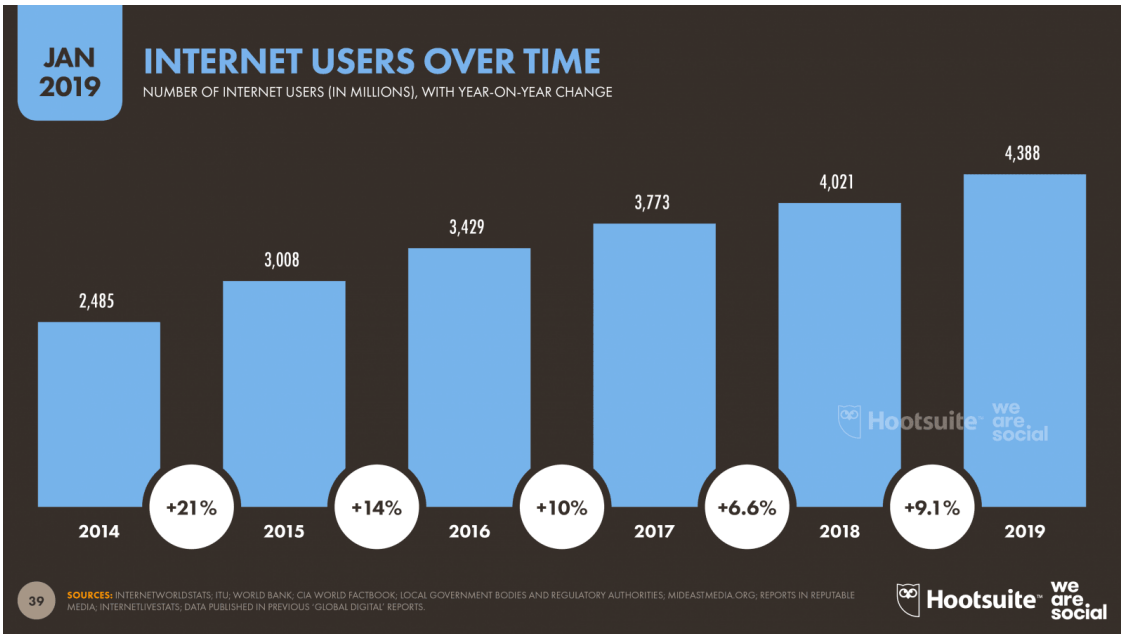

Figura 2 – Utilizadores de internet ao longo do tempo (Jan 2019)[9]

Figura 3 - Lista de sites mais visitados (Jan 2019)[9]

À medida que o crescimento da Internet e das aplicações *web* foi acontecendo, novas necessidades e requisitos foram incluídos nas aplicações, com o avanço tecnológico. Inicialmente os *websites* eram todos estáticos, mas com a introdução de novas tecnologias (e dos *websites* dinâmicos) foi possível identificar que os utilizadores poderiam obter, trocar e armazenar informação online, bem como realizar determinadas ações. A informação obtida seria então persistida em algum sistema externo, e que seria utilizada posteriormente, em algum momento da aplicação. A certa altura surgiu o conceito dos sistemas em tempo real, que permitiam ao utilizador obter informação atualizada instantaneamente.

2.1.1 Problema

Os sistemas em tempo real introduziram um novo paradigma às aplicações *web,* sobre o qual foi necessária a criação de novas técnicas no protocolo de comunicação utilizado na altura, e que permitissem a troca de informação de forma automática e rápida, bem como a notificação por parte do componente servidor quando existisse alguma atualização de estado relevante.

Com este novo paradigma o protocolo HTTP implementou duas novas técnicas: HTTP *Long polling* e HTTP *streaming* (ambos descritos na Subsecção 2.2.2). Ambas as técnicas permitem a ligação ao servidor por um determinado tempo, apesar de a técnica de *streaming* manter a ligação cliente-servidor durante mais tempo. Tecnicamente a solução ideal para os sistemas em tempo real seria a utilização de uma destas técnicas. No entanto, estas apresentam defeitos[10], que têm impacto na comunicação que existe entre componentes da arquitetura cliente-servidor.

Em suma, as técnicas existentes atualmente para garantir sincronização entre componentes (HTTP *long polling* e HTTP *streaming*) não são suficientemente eficientes, visto que a probabilidade de existirem términos de conexões inesperadas é alta, bem como a probabilidade de o cliente transferir mais dados do que aqueles que efetivamente necessita para o seu funcionamento esperado. Estes problemas representam problemas de sincronização entre cliente-servidor, resultando em atrasos na entrega de mensagens, ou ainda excesso de *bytes* transferidos, o que por sua vez leva a um mau desempenho da aplicação *web* em geral. Este fator é bastante negativo na consideração de utilização de uma aplicação *web*, pois pode provocar o abandono por parte dos seus utilizadores, visto que não beneficiam com a sua funcionalidade.

O problema desta dissertação assenta em perceber o impacto do protocolo *WebSocket* nas aplicações *web*, tendo em conta a escolha da implementação de um protocolo em detrimento do outro, com base nas suas características, vantagens e desvantagens, contextos em que se aplicam, *etc*.

2.1.2 Solução

O objetivo da presente dissertação é o desenvolvimento de um protótipo que contenha uma *WebSocket* API e uma REST API (que utiliza HTTP), com o objetivo de estudar e comparar as duas abordagens utilizando diversas métricas, e potencialmente considerar *WebSockets* como uma solução ao problema referido na Subsecção 2.1.1. O protocolo *WebSocket* é um protocolo criado especificamente para lidar com informação em tempo real, e permite a remoção do excesso de informação existente em outras técnicas, bem como permite reduzir a complexidade de desenvolvimento devido à sua fácil integração com os componentes *web*, sendo que providencia uma interface nativa para o fazer.

O protótipo terá de conter métricas para avaliação de desempenho de forma a verificar que os diversos problemas existentes noutras técnicas não estão presentes no protocolo a ser implementado.

2.2 Enquadramento teórico

Nesta secção é feita uma abordagem teórica aos conceitos que são fundamentais para a compreensão da dissertação e dos seus envolventes. Esses conceitos incluem as comunicações que existem entre cliente e servidor, bem como os protocolos que as suportam e alguns métodos que persistem entre os quais.

No mundo das aplicações *web* a comunicação que existe entre a aplicação cliente e servidora é fundamental para um bom desempenho da solução. Dentro da temática de comunicação cliente-servidor existem vários modelos arquiteturais adotados para aumentar a eficiência da mesma.

As aplicações *web* têm sobretudo suporte baseado no conjunto dos protocolos *Transmission Control Protocol/Internet Protocol* (TCP/IP), incluídos no modelo TCP/IP, que é implicitamente o modelo que mais impacto tem para o estudo incluído na presente dissertação. Este é altamente escalável e também compatível com qualquer sistema operativo de forma a comunicar com outro sistema.

2.2.1 Modelo TCP/IP

O modelo TCP/IP foi concebido de forma a ser independente do *hardware* e com possibilidade de execução em qualquer conexão, através de qualquer dispositivo. Este contém quatro camadas de comunicação:

- *Application* – camada que fornece às aplicações *standards* de troca de dados/ informação. Inclui os protocolos HTTP, FTP (*File Transfer Protocol*), SMTP (*Simple Mail Transfer Protocol*, utilizado na troca de emails, etc.)
- *Transport* – camada que é responsável pela manutenção de comunicações pela rede, entre vários dispositivos. Inclui o protocolo TCP (*Transmission Control Protocol*) e o protocolo UDP (*User Diagram Protocol*). Ambos utilizam a camada de rede implementada pelo protocolo de Internet (IP[11], abrangido na camada de *networking*).
- *Networking* – camada responsável por gerir e transportar pacotes de informação através de rede independentes, ultrapassando a própria fronteira da rede.
- *Datalink* – camada que consiste em protocolos que só atuam numa rede. Consiste num componente que interliga nós numa rede. Os protocolos incluídos nesta camada são o *Ethernet* e o ARP (*Address Resolution Protocol*).

O conjunto de protocolos TCP/IP referido anteriormente consiste nos vários protocolos que atuam numa das quatro layers do modelo de TCP/IP[12] (Figura 4).

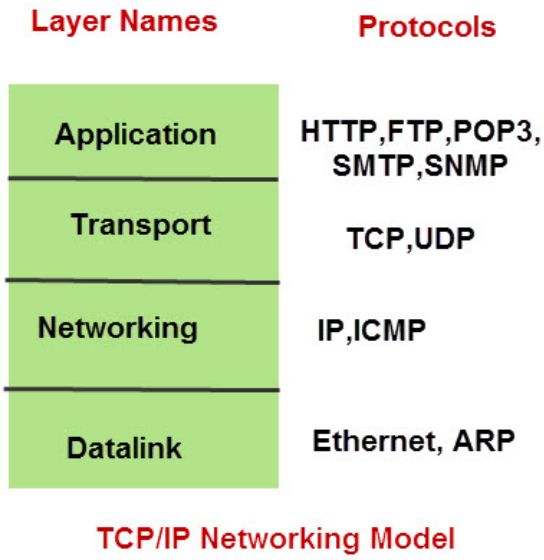

Figura 4 - Modelo de rede TCP/IP[12]

O foco da presente dissertação está centrado na camada de transporte de dados (*Transport*) e na camada aplicacional (*Application*).

2.2.2 Protocolo HTTP

O protocolo HTTP é a base de todas as comunicações em aplicações *web* que existem entre cliente e servidor e é definido como o tipo de comunicação caracterizado por uma troca de

mensagens entre ambas as partes, através de um processo pedido-reposta[13], tal como o apresentado na Figura 5.

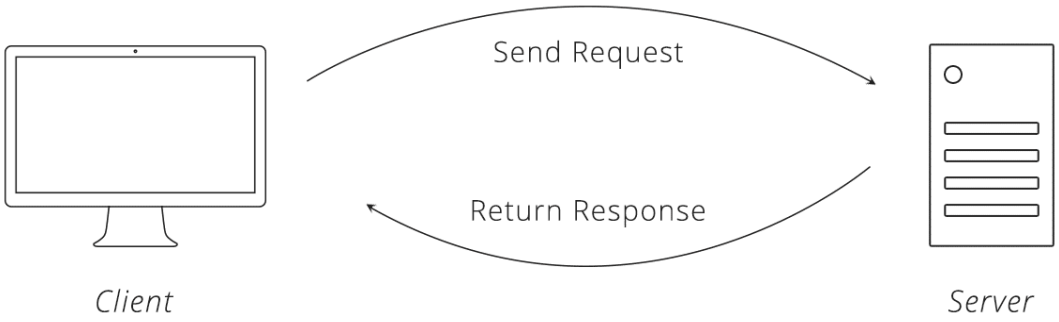

Figura 5 - Ciclo pedido-resposta do HTTP[14]

Cada pedido é constituído por um método, um *Uniform Resource Identifier*[15] (URI), seguido de uma mensagem do tipo *Multipurpose Internet Mail Extensions*[16] (MIME), contendo modificadores do pedido, informação sobre o cliente e possível conteúdo necessário para o servidor, tal como se pode ver na Figura 6.

O método relativo a um pedido pode variar consoante o tipo de operação que se queira efetuar. Normalmente para obtenção de dados utiliza-se o método GET, mas para submissão de dados ou atualização dos mesmos deve-se utilizar o POST e o PUT, respetivamente[17]. No entanto, estes tipos de operações são definidos no servidor e o cliente terá de respeitar as suas regras, visto que os dados do pedido terão de coincidir com o que o servidor está à espera de receber.

Uma resposta do servidor (identificado na Figura 7) é caracterizada por um objeto característico do seu estado, incluindo no corpo da mensagem, um código de sucesso ou erro, seguido de uma mensagem do tipo MIME[5], que contém informação sobre o servidor, bem como outra relacionada com as entidades a serem manipuladas na troca de informação.

O código de sucesso ou erro que é enviado pelo servidor constitui o que é denominado de *status code* [10] e representa o estado da resposta que o servidor tem a dar ao cliente.

[5] *Multipurpose Internet Mail Extension*[96] – consiste numa extensão do protocolo de email que permite a troca de vários tipos de dados (imagens, áudio, vídeo, etc.)

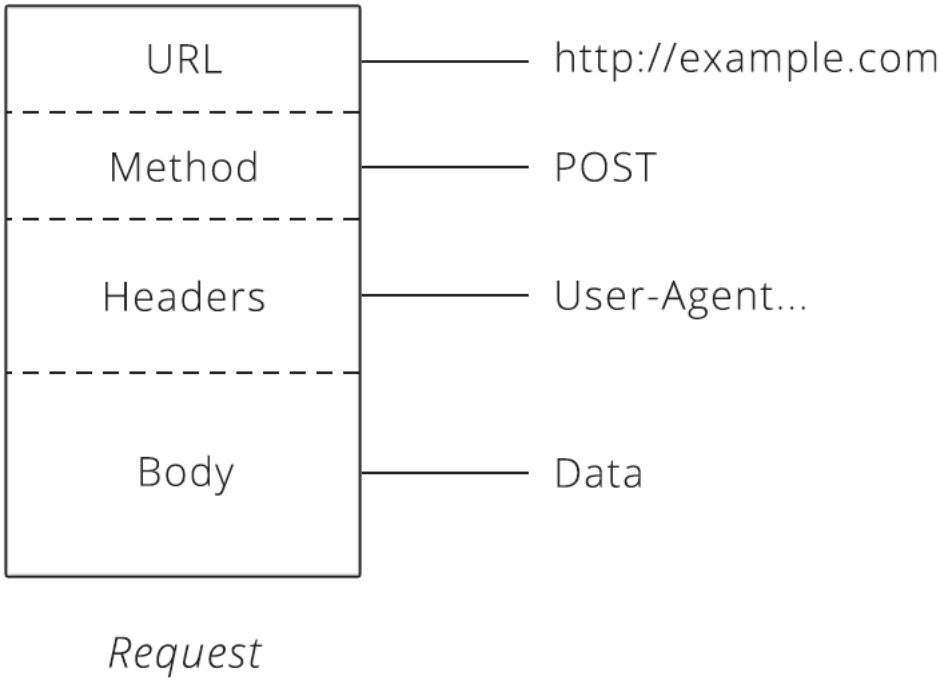

Figura 6 - Constituição de um pedido HTTP[14]

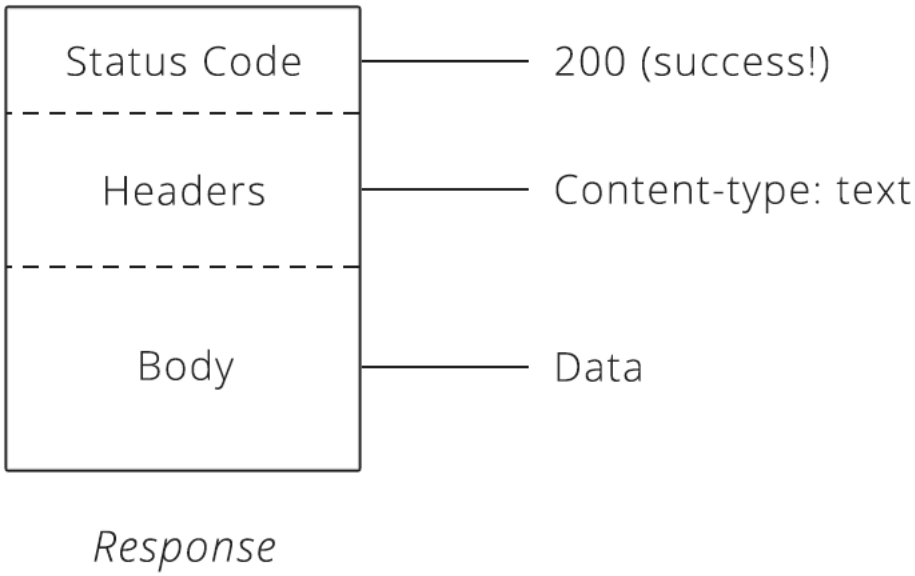

Figura 7 - Constituição de uma resposta HTTP[14]

Nos primórdios da Internet a maioria das páginas eram estáticas, e como consequência a comunicação entre cliente e servidor era bastante limitada visto que a aplicação cliente enviaria

pedidos ocasionais para o servidor, que de seguida responderia, acabando por terminar a comunicação até existir um novo evento que fosse despoletado pelo utilizador.

Em qualquer aplicação *web* existem dois modos de operação: *pull* e *push*[18]. O primeiro (*pull)* é relativo ao paradigma pedido/resposta, em que o cliente pede informação ao servidor em cada pedido e o pedido é síncrono, ou seja, existe um bloqueio na execução da aplicação até ser obtida uma resposta do servidor. Por outro lado, o *push* consiste num processo de envio de informação com base no paradigma de *publish/subscribe*[19], em que um cliente subscreve um determinado evento ou canal de comunicação e o conteúdo é enviado do servidor para o cliente assim que existe nova informação a ser enviada. Este tipo de comunicação é assíncrono, significando que pode acontecer a qualquer altura e a execução do cliente não é bloqueada.

O protocolo HTTP baseia-se no paradigma de pull, não sendo originalmente criado com o objetivo de suportar comunicações em tempo real, foram desenvolvidos vários modelos alternativos para atingir esse objetivo, sendo eles: *polling*, *long polling* e *streaming*.

2.2.2.1 Polling

O modelo *polling* representa a primeira tentativa em direção à comunicação em tempo real das aplicações *web*.

Através deste, o *browser* envia vários pedidos ao servidor de forma periódica e recebe uma resposta imediata por parte do mesmo.

Esta técnica poderia ser uma boa solução para o problema das comunicações em tempo real se fossem conhecidos os intervalos de tempo exatos para que a mensagem entregue fosse consistente, pois existiria a possibilidade de sincronizar o pedido para que este ocorresse apenas quando houvessem garantias de que o servidor conseguiria responder com algo considerado útil.

Contudo, os dados em tempo real são geralmente imprevisíveis, e como tal resultam pedidos desnecessários com a consequência de existirem várias conexões a serem abertas e fechadas num curto espaço de tempo.

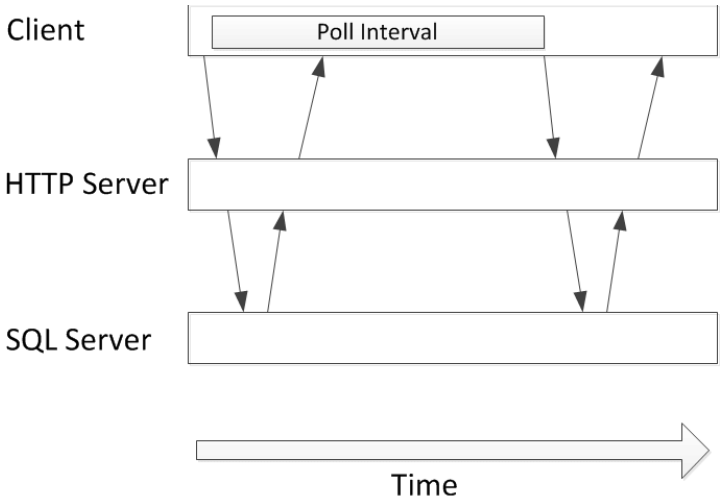

Figura 8 - HTTP Polling/Short Polling[20]

2.2.2.2 Long Polling

O *long polling* (Figura 9) é semelhante ao *polling* (apresentado na Subsecção 2.2.2.1), com a exceção de que o servidor mantém o pedido HTTP aberto se existir algum recurso ainda ocupado e/ou se este não tiver nenhuma resposta imediata disponível para o cliente. O tempo em que o pedido (também conhecido por *hanging* GET[21]) fica aberto é determinado pelo servidor.

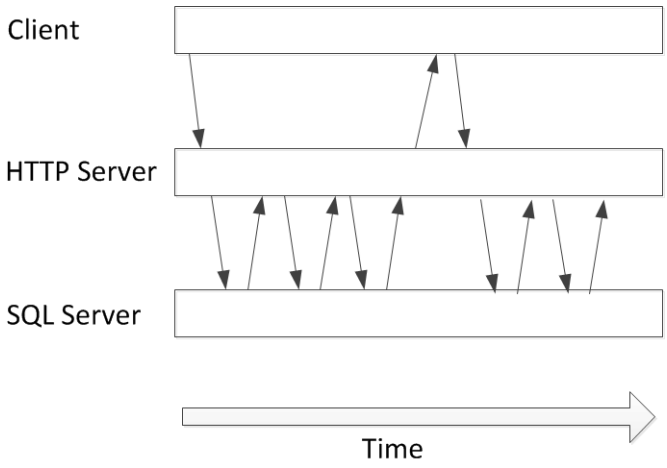

Figura 9 - HTTP Long Polling[22]

Se existirem novos dados disponíveis dentro do intervalo especificado pelo servidor, a informação é enviada para o cliente e é fechada a conexão. Caso contrário, o servidor irá notificar o cliente acerca do término da conexão em aberto.

É importante que se perceba que se existir um elevado tráfego de informação entre cliente e servidor, este método não garante melhorias de performance sobre o tradicional *polling* podendo até ser pior, visto que a probabilidade de existir um ciclo de *polls*[6] imediatos aumenta substancialmente[23].

A técnica de HTTP *Long Polling* introduz um defeito de *header overhead*[24], que consiste no excesso de informação que é trocada em cada pedido-resposta que o cliente efetua e recebe do servidor. Cada pedido corresponde a uma mensagem HTTP, contendo todos os *headers*. No caso do tamanho dos dados da resposta serem pequenos, e os pedidos não serem feitos com muita frequência, a percentagem de dados que representam os *headers* pode ser elevada e tem um impacto negativo na quantidade de dados transferidos através da rede. Além deste problema, esta técnica também tem um outro problema denominado de *maximal latency*[24], que consiste na alta latência que existe entre pedidos. Depois de enviada uma resposta ao cliente, o servidor espera até pelo próximo pedido por parte do cliente, até que possa enviar uma nova resposta. Isto significa que o servidor irá ter um nível de latência elevado na rede, podendo levar à perda de pacotes de rede e/ou à retransmissão dos mesmos.

2.2.2.3 Streaming

O *streaming* (Figura 10) é semelhante aos métodos anteriores onde existe um pedido do cliente ao servidor, mas com uma diferença por parte da resposta do servidor. A comunicação é baseada numa conexão HTTP persistente e o servidor nunca notifica o cliente que a mensagem de resposta é completa. Desta forma a conexão é mantida em aberto e sempre disponível para transmissão de novos dados, até que exista um período de tempo expirado onde a conexão é terminada.

Contudo, devido ao facto deste método ser encapsulado numa conexão HTTP, existe a possibilidade de que os servidores proxies[7] consigam aumentar o tamanho da resposta e consequentemente aumentar a latência da transmissão de informação, resultando na obtenção de altos níveis de latência de rede e a possibilidade de haver problemas na transferência de dados numa resposta a um pedido[24].

Por esse motivo, o servidor tem uma verificação de deteção de proxy e caso se confirme a sua utilização, este passa a usar o método de *long polling* (apresentado na Subsecção 2.2.2.2) automaticamente.

[6] Pedidos ao servidor
[7] Servidor que atua como um intermediário entre a ligação de um cliente e um servidor

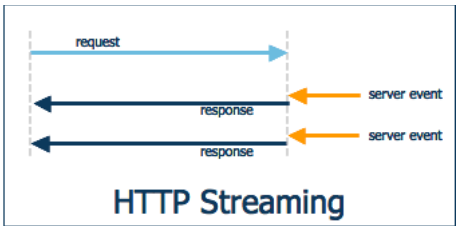

Figura 10 – HTTP Streaming[25]

2.2.2.4 Comparação entre métodos de comunicação em tempo real (HTTP)

De forma a sumarizar e comparar as características de cada método de comunicação descrito nas subsecções anteriores, estas estão descritas na Tabela 1.

Tabela 1 - Caracterização dos métodos de comunicação utilizados em HTTP

	Polling (short polling)	***Long Polling***	***Streaming***
Forma de execução de pedidos ao servidor	São feitos vários pedidos intervalados entre si.	Existe um pedido ao servidor.	É feito um único pedido ao servidor.
Processo de resposta pelo servidor	Recebe uma resposta imediata a cada pedido.	O pedido fica em aberto por um determinado período de tempo.	O pedido fica em aberto por tempo indeterminado (ou por um tempo configurável)
Frequência com que existe resposta por parte do servidor / Tipo de resposta enviada ao cliente	O servidor não envia mais respostas, a não ser que exista um outro pedido.	Se existir uma notificação dentro do tempo estimado, uma resposta é enviada de volta. Caso contrário existe um término da conexão por parte do servidor.	O servidor envia uma resposta que é continuamente atualizada.

Em suma, as técnicas de *streaming* e *long polling* são bastante semelhantes, visto que ambos executam apenas um pedido que permanece em aberto por um determinado período de tempo (ou não, dependendo da técnica). A única diferença entre estas duas técnicas é o facto de o *streaming* ser uma técnica mais completa, visto que a resposta enviada no pedido inicial pode ser atualizada, enquanto que no *long polling* é enviada uma nova resposta em caso de atualização por parte do servidor. O short polling (ou polling), por outro lado, é a técnica mais básica visto que existem vários pedidos ao servidor, e não existe resposta por parte do servidor enquanto não houver um novo pedido por parte do cliente.

2.2.3 TCP

O TCP (*Transmission Control Protocol*) pertence também à camada de transporte no modelo TCP/IP e é um dos protocolos de transmissão de dados mais importantes, visto ser o mais fiável, pois possui um controlo de transmissão que permite assegurar a consistência dos dados que são transmitidos [26].

Os dados são transmitidos *byte*[8] a *byte* em segmentos independentes, de acordo com tempos específicos.

O processo de controlo do protocolo (demonstrado na Figura 11) consiste numa iniciação de pedido por parte do cliente para o servidor, onde este responderá com um comando de ACK[27] (*Acknowledge*), para que exista uma segurança de que os dados serão efetivamente transmitidos para o destino correto.

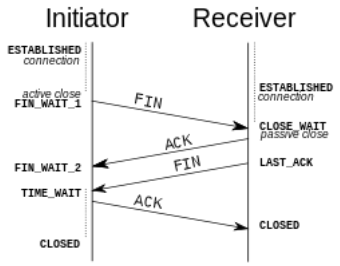

Figura 11 – Mecanismo de acknowledge do TCP

Em suma, os aspetos mais importantes do protocolo TCP são:

- É *connection-oriented*, ou seja, para que exista troca de dados, uma conexão tem de ser estabelecida em primeiro lugar. Só quando existir uma conexão é que se torna possível enviar dados a qualquer momento.
- É fiável, visto que garante a entrega de todos os pacotes enviados através do mecanismo de *acknowledge*.
- É bidirecional e *full-duplex*, o que significa que ambas as partes da comunicação podem fazê-lo quando pretenderem.

Uma conexão TCP denomina-se *socket*, que consiste numa abstração de uma estrutura de dados que pode tanto ser escrita como lida como se fosse um ficheiro. A programação feita através de *sockets* denomina-se *socket programming*.

[8] Unidade de informação digital

2.2.3.1 TCP *Keep-Alive*

Durante uma conexão TCP inativa, não existe troca de dados entre duas máquinas. Isto implica que a partir do momento que uma conexão TCP é estabelecida, esta pode tornar-se inativa por períodos prolongados, excedendo horas ou dias. A conexão só é terminada quando ambos os participantes na conexão se desconectam ou reiniciam. No entanto, por vezes existe a necessidade de o servidor detetar se um cliente ainda está conectado ou não, e para isso existe uma funcionalidade no TCP denominado *keep-alive*. Esta funcionalidade implica que a aplicação servidora tente utilizar recursos *web* como se se tratasse de uma aplicação cliente, para determinar se existe a necessidade de acesso a estes recursos.

Esta funcionalidade tem um *timeout* de 2 horas (configurável), sendo que ao fim desse tempo o servidor envia informação para o cliente com indicação de ACK (*Acknowledge).* Caso a aplicação cliente não responda de volta com um outro ACK, assume-se que a conexão pode ser terminada visto que não está a ser mais utilizada.

Apesar da principal vantagem desta funcionalidade ser a de poder terminar uma conexão TCP, existem vários motivos para que o seu uso seja desencorajado, listados pelo *Host Requirements RFC*[28]:

- Pode causar términos de conexão indesejados durante determinadas falhas;
- Consomem *bandwidth*[9] desnecessário;
- Podem ter alto custo monetário para utilizadores de internet que paguem por cada pacote transferido.

2.2.4 Protocolo WebSocket

Nesta subsecção é feita uma abordagem ao protocolo *WebSocket* quanto à sua definição de conceito, bem como as suas características base.

2.2.4.1 Definição do protocolo

O protocolo *WebSocket* caracteriza-se por ser o início de uma revolução digital no mundo das comunicações *web*. A sua origem permitiu a criação de múltiplas aplicações direcionadas à utilização desta mesma tecnologia, com o objetivo de permitir a conexão permanente à *web*[29].

Segundo o *Request For Comment* (RFC)[6], o protocolo *WebSocket* é definido como o protocolo que permite a comunicação bidirecional entre uma aplicação cliente a correr código num

[9] Banda larga de internet

ambiente controlado e um servidor remoto que tem a possibilidade de optar por comunicações provenientes do cliente. A nível de segurança, utiliza o mesmo padrão que os *browsers* utilizam, que se baseiam na correspondência equivalente entre a origem do pedido e o seu destino. O protocolo consiste numa conexão encapsulada em TCP (detalhado na Subsecção 2.2.3) e tem o objetivo de colmatar a necessidade de existência de várias ligações HTTP para comunicação cliente-servidor.

Para iniciar uma comunicação por *WebSocket*, é necessário que exista um HTTP *handshake*[30], que será explicado posteriormente (Subsecção 2.2.4.2).

2.2.4.2 WebSocket handshake

Quando surgiu a primeira versão do protocolo *WebSocket*, este foi pensado para que a sua integração numa infraestrutura *web* já existente fosse um processo garantido. Desta forma, decidiu-se que o protocolo teria de ser desenvolvido de forma a que tivesse retro compatibilidade, tendo sido optado por uma conexão HTTP para o efeito. A aplicação cliente teria então de iniciar uma conexão HTTP, onde enviaria um *header* denominado de *Upgrade* para informar o servidor que pretenderia iniciar uma conexão *WebSocket*. O processo de transformar uma conexão que utilize o protocolo HTTP para uma que utilize *WebSocket* é denominado de *handshake*.

Caso o servidor suporte esse tipo de conexão, este enviar de volta um *header* na resposta a identificar que iniciou uma conexão WebSocket.

Quando o *handshake* for completo, a conexão *WebSocket* fica ativa, e ambos podem enviar dados, de forma bidirecional, através de um pedido de comunicação TCP/ IP. A informação é contida em *frames*[10], em que cada um consiste em dados entre 4 a 12 *bytes* para assegurar que a mensagem pode ser reconstruída.

2.2.4.3 *WebSocket* API

A integração de *WebSockets* num *browser* é demasiado facilitada, quando utilizada a sua API JavaScript – *WebSockets* API[29].

Nela existem várias subscrições a eventos importantes que acontecem antes, durante, e depois das conexões estabelecidas entre cliente e servidor.

Uma conexão é estabelecida utilizando um URI num dos seguintes formatos:

[10] Pedaços de informação

- ws:// para comunicação normal (insegura)
- wss:// para comunicação em canais seguros (semelhante ao HTTPS)

A criação de uma conexão é tão simples como invocar uma linha de código - ver Código 1.

```
var myWebSocket = new WebSocket("ws://examplehost.org");
```

Código 1 - Inicialização de uma conexão *WebSocket*

Posteriormente existirão três possíveis estados para a conexão: *connecting*, *open* e *closed.* Como o JavaScript é baseado em eventos, a API define vários eventos que são invocados relativos aos estados em que a conexão se encontra:

- *onopen* – ocorre quando uma conexão é estabelecida
- *onmessage* – ocorre quando o cliente recebe dados do servidor
- *onerror* – ocorre quando existe um erro na comunicação cliente-servidor
- *onclose* – ocorre quando a conexão é terminada.

A personalização e o controlo sobre esses eventos permitem aos desenvolvedores muita flexibilidade no uso de conexões *WebSockets* consoante as suas necessidades.

2.2.4.4 Suporte de *WebSockets* API em *browser*

Após *release* do *standard* RFC 6455[29], os navegadores de internet foram relativamente rápidos na integração e no suporte ao protocolo *WebSocket*. Entre Dezembro de 2011 e Março de 2012, os principais *browsers* – Google Chrome e *Mozilla Firefox* – anunciaram o seu suporte ao protocolo – (Tabela 2).

Versão protocolo WebSocket	Internet Explorer	Firefox	Chrome	Safari	Opera	Edge
13 RFC 6455	10	11	16	6	15	12

Tabela 2 - Suporte mínimo do protocolo *WebSocket*[29]

2.2.4.5 Problemas de segurança do protocolo *WebSocket*

Os problemas de segurança na utilização de *WebSockets* são bastante conhecidos[31], tendo que ser investigados e mitigados de forma a assegurar um canal seguro de comunicação. Alguns destes problemas incluem:

- *Cross-site Websocket Hijacking*[43] – como o servidor *WebSocket* só verifica se o utilizador está autenticado através de *cookies* (um tipo de dado enviado no pedido) e não verifica a sua origem, é possível simular que está autenticado e manipular dados que são enviados ao servidor, ou até mesmo interpretar respostas.
- *Denial of Service* (*DoS*) – como o protocolo permite conexões ilimitadas, é possível criar *scripts* que conectem ao servidor infinitas vezes, criando falha no sistema.

2.2.5 Web / WebService API

Uma *Application Programming Interface* (API) consiste num conjunto de rotinas, protocolos e ferramentas para construir uma aplicação. Em termos gerais, é um conjunto definido de métodos de comunicação entre vários componentes de *software*[4].

Genericamente uma API define uma série de regras que os programadores devem seguir de forma a que interajam com uma linguagem de programação, biblioteca, ou outra ferramenta do tipo.

Atualmente são considerados vários tipos de API[7]:

- *Web*/*Internet* – uma API que é implementada através da rede, para uso único através da Internet.
- *Product* – API que apesar de passível de ser utilizada através da rede, é implementada dentro de um produto, de forma a que aqueles que o instalem herdem as suas rotinas. Um exemplo deste tipo de API são as que estão implementadas em produtos como o SugarCRM[32].
- *Standard* – API standard que não contêm qualquer tipo de especificações de *browser*, como é o exemplo da *Mobile Data Plan* API[33] da *Google*, ou ainda da *Web Audio* API[34], da W3C[5].
- *Browser* – APIs que podem ser encontradas num *browser*, como é o exemplo da implementação da *Web Audio* API[35], da Mozilla.
- *System/Embedded* – APIs que são passíveis de serem integradas em aplicações *web*, mas que podem ser específicas a um sistema operativo ou dispositivo, como é o exemplo de uma API utilizada numa aplicação *web* para acesso ao sensor de impressão digital.

Estudos feitos pela *ProgrammableWeb*[7] para determinar o tipo de APIs mais utilizado comprovam que em 2014 82% das APIs que contêm valor são aquelas do tipo *Web/Internet*, o que demonstra um enorme crescimento e investimento do mundo das aplicações *web* neste tipo de serviços. O resultado deste estudo pode ser revisto graficamente na Figura 12.

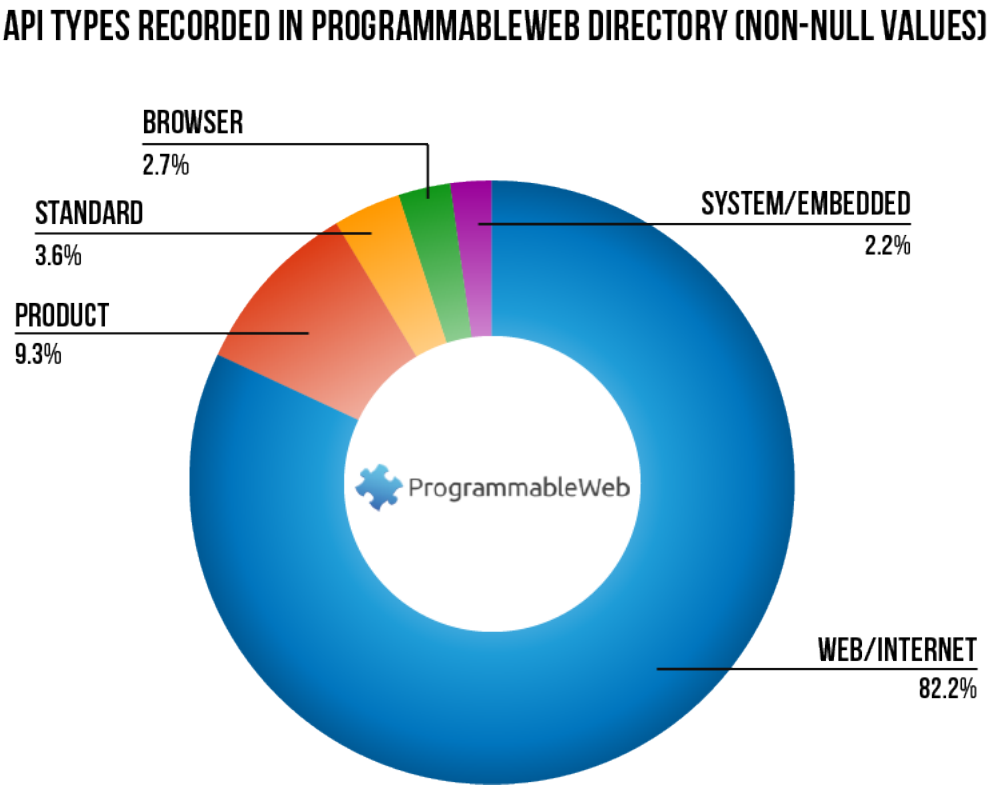

Figura 12 - Resultado de estudo relativo a tipos de API[7]

Dentro dos tipos de APIs, cada um segue de certa forma o seu estilo arquitetural, existindo vários que se destacam, entre eles:

1. REST – dedicado a comunicação sobre métodos HTTP para fácil manipulação de entidades.
2. RPC[36] (*Remote Procedure Call*) – API que invoca vários procedimentos num serviço *web*, como é o caso do SOAP[37].
3. *Push/Streaming* – APIs em tempo real onde os dados são transmitidos de forma bidirecional, como é o exemplo de *WebSockets.*
4. *GraphQL*[38] – APIs utilizadas para efetuar *queries*[11] (pedidos de informação) de forma mais eficiente e rápida.

[11] Designação para pedido de informação contido numa base de dados

Um outro estudo pela ProgrammableWeb[7] cujo resultado está demonstrado na Figura 13 aufere que o estilo arquitetural mais utilizado numa API atualmente é o REST, com 81.53% de usabilidade.

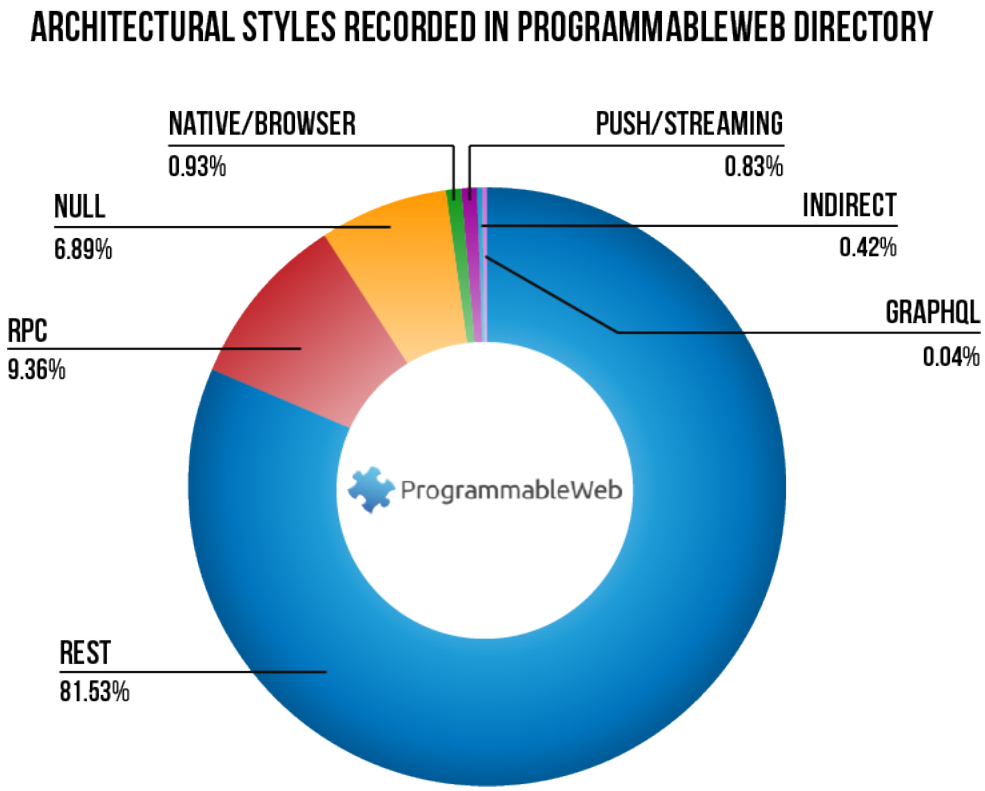

Figura 13 - Resultado de estudo para estilos arquiteturais de API[7]

As API mais utilizadas atualmente são as *Web* API, ou também denominadas de *WebServices* API. Conhecidas atualmente a partir da época em que os serviços de Internet permitiam aos seus utilizadores armazenar informação *online*, estas consistem num único tipo de interface onde a comunicação é realizada com recurso à internet e a protocolos *Web*[39]. Num modo geral, as *Web* APIs funcionam através de uma interface HTTP e para isso são definidos *endpoints*[40] (caminhos e/ou rotas), bem como estruturas e formatos de pedidos e resposta para cada *endpoint*.

2.3 Plataformas de desenvolvimento

Nesta subsecção é apresentado o conjunto de plataformas que são consideradas para a implementação do protótipo que visa estudar a possibilidade de o protocolo *WebSocket*

resolver o problema descrito, bem como todas as outras tecnologias que fazem parte da sua contextualização, em função da presente dissertação.

Estas incluem o *JavaScript* (visto ser a linguagem base de programação para aplicações *web*), *Node.js* (plataforma bastante utilizada no desenvolvimento de APIs que servem de suporte às aplicações *web* atuais) e *Socket.IO*, pois é uma das possíveis abordagens na implementação de uma *WebSocket* API, que tem como suporte o protocolo *WebSocket*, descrito na Subsecção 2.2.4.

2.3.1 JavaScript

Linguagem de programação geralmente utilizada no desenvolvimento de aplicações *web* e originalmente criada pela Netscape[41] como forma de adicionar elementos interativos e dinâmicos às páginas *Web*[42].

Esta linguagem de programação foi originalmente considerada *client-side*, ou seja, o código fonte é diretamente processado pelo *browser*, onde as respetivas funções são executadas logo após o carregamento da página *web*, sem prévia comunicação com o servidor. Como esta não é compilada, todo o *JavaScript* fica visível diretamente no código fonte da página *web*, exceto nas situações em que se usam mecanismos que previnam a sua leitura na íntegra (por exemplo, ofuscação e/ou minificação de código).

Apesar de a área de implementação de *Java*[43] e *JavaScript* serem diferentes, estas têm algo em comum e que faz com que sejam equiparadas na sua forma de trabalhar, sendo que ambas constituem uma base programação fundamentada em OOP (*Object Oriented Programming)*.

Por norma, o código *JavaScript* está localizado dentro de uma *tag* HTML (*script*), que indica ao *browser* que o bloco de código contido nessa mesma *tag* terá de ser interpretado e executado posteriormente.

De notar que esta inserção de *JavaScript* em código HTML é uma das formas para que este seja executado. No entanto, existem também outras variantes que não requerem esta mesma inserção em HTML.

Uma das variantes corresponde à tecnologia *Node.js* (explicado na subsecção seguinte), que utiliza *JavaScript* na sua base.

Existem, no entanto, várias desvantagens[44] no uso de *JavaScript*, entre as quais:

- Necessidade de reforço de segurança a nível *client-side*, visto que se o código é visível para os utilizadores, pode ser utilizado de forma maliciosa

- Suporte dos *browsers* – cada *browser* interpreta código *JavaScript* de forma diferente, existindo a possibilidade de algumas funções nativas não estarem disponíveis em certos *browsers,* o que leva o programador a ter isso em conta.
- Paragem de funcionamento em caso de erro – caso exista algum erro inesperado e não tratado, toda a aplicação deixará de funcionar, visto que se os *browsers* não são tolerantes a erros.

2.3.2 Node.js

Node.js é uma plataforma que funciona do lado de servidor e é construída através do *engine*[12] de *JavaScript* to *Google Chrome*. Desenvolvida em 2009 por Ryan Dahl, sendo a última versão a v0.10.36 [45].

O seu objetivo é a construção de aplicações web que sejam rápidas e facilmente escaláveis, utilizando um mecanismo direcionado a eventos e não bloqueando operações I/O[13] de forma a que a plataforma seja eficiente e leve, ideal para aplicações que correm através de vários dispositivos e que contenham a necessidade de comunicação em tempo real com uma quantidade significativa de dados.

É uma plataforma *open source*, o que significa que o código fonte é visível à comunidade, e onde todos podem contribuir para melhorar as suas funcionalidades.

A linguagem de programação que está associado ao *Node.js* é o *JavaScript* e a plataforma tem a possibilidade de ser executada em diversos ambientes, tais como *OS X*, *Microsoft Windows* e *Linux*.

Algumas das funcionalidades mais importantes que fazem com que o *Node.js* seja uma das primeiras escolhas no desenvolvimento *web* são:

- É uma plataforma direcionada a eventos assíncronos – todas as APIs que utilizam *Node.js* são assíncronas, ou seja, o servidor não espera pelo retorno de dados, prosseguindo sempre para o próximo pedido contando com o funcionamento de um mecanismo de eventos e notificações para que todos os pedidos feitos à API tenham resposta.
- Biblioteca rápida e eficiente – tendo em conta que esta é construída a partir do *engine* V8[46] de *JavaScript* do *Google Chrome*, toda a biblioteca *Node.js* é considerada rápida em termos de execução de código.

[12] Termo para designar um *engine* de execução de uma determinada plataforma
[13] Input / output

- Contém uma única *thread*[14], mas é altamente escalável – *Node.js* utiliza um modelo de *single thread* com *event looping*[15] e que consegue, através do seu mecanismo de eventos, ajudar o servidor a responder de uma forma assíncrona e que faz com que este seja altamente escalável, ao contrário dos servidores tradicionais que criam *threads* limitadas para responder aos pedidos recebidos.
- A plataforma foi desenvolvida de acordo com uma licença MIT[47], o que permite que seja utilizada à forma do programador sem quaisquer repercussões.

Devido a todos estes fatores, existem muitos projetos e empresas que utilizam *Node.js* nas suas aplicações. Estas incluem o *eBay*, *Microsoft*, *PayPal*, *Uber*, etc [45].

2.3.3 Socket.IO

Socket.IO é uma biblioteca que permite a comunicação em tempo real, de forma bidirecional e que seja à base de eventos entre cliente e servidor [48].

Esta é constituída por um servidor em *Node.js* e uma biblioteca *JavaScript* pronta a ser utilizada por uma aplicação *web.*

As conexões são estabelecidas por via de *proxies* e *load balancers* (balanceadores de carga) e respeitam os princípios de uma conexão que obedece ao método *long-polling*, que posteriormente evolui para uma outra de *streaming*, como é o caso do *WebSocket*.

2.4 Estudo de abordagens

O objetivo é identificar se o protocolo *WebSocket* é considerado aquele a implementar para que as comunicações entre aplicações *web* sejam o mais eficiente possível, para que resolvam o problema proposto.

Enquanto o HTTP é um protocolo unidirecional onde um pedido é sempre iniciado por um cliente, o *WebSocket* é bidirecional onde tal padrão de pedido/resposta não existe, o que pode ser vantajoso na eficiência da comunicação, impedindo que haja abrandamento do fluxo de informação. Além disso, o *WebSocket* tem uma comunicação *full-duplex*, ou seja, as aplicações cliente e servidora comunicam entre si independentemente um do outro, ao invés do protocolo HTTP onde só é permitido existir uma resposta do servidor se existir um pedido do cliente.

[14]Processo de execução de um determinado programa
[15] Sequência de eventos consecutivos

Um outro fator importante na eficiência de comunicação é a existência de múltiplas conexões que persistem no protocolo HTTP (e numa RESTful API) e que podem provocar um esgotamento de recursos. Isto irá ser tido em conta no desenvolvimento do protótipo.

Numa *WebSockets* API a comunicação entre cliente e servidor é feita a partir de uma única conexão TCP que é iniciada a partir de uma conexão HTTP através de um mecanismo de Upgrade do próprio HTTP, e que tem o seu próprio *lifecycle* (ciclo de vida).

2.5 Sumário

Neste capítulo foram feitas uma contextualização e uma descrição do problema de forma mais detalhada, bem como foi apresentada uma possível solução para o problema, destacando aqueles que são os objetivos da dissertação. Seguiu-se um enquadramento teórico com conceitos que são considerados importantes na compreensão do protótipo e que ajudam também a perceber o problema de forma mais profunda. Foram, sucintamente, também descritas algumas plataformas de desenvolvimento que podem ser incluídas no protótipo, de uma forma ou outra. Por fim fez-se uma avaliação detalhada de soluções e abordagens a ter em consideração na implementação do protótipo.

3 Análise de Valor

Neste capítulo é apresentada uma descrição em detalhe da análise de valor à solução ao problema referido anteriormente (Secção 1.2), bem como a referência aos vários modelos que lhe dá suporte.

3.1 Descrição

A análise de valor é uma aplicação sistemática de técnicas que identificam as funções de um produto ou serviço, estabelecem um Valor para essas funções e proporcionam as funções necessárias ao mais baixo custo. Para tal, é necessário compreender o propósito do produto e os requisitos do seu cliente.

O seu principal objetivo é dignificar o aumento de valor de um produto ou serviço, tendo sempre em conta os custos associados e mantendo sempre a qualidade do produto ou serviço.

Contudo, o processo de inovação, identificado na Figura 14, contém sempre um risco elevado, principalmente a nível económico. Para existir um lançamento de novos produtos ou serviços, deve-se sempre procurar atingir a diminuição ao máximo desses riscos. Para tal, existem vários métodos que ajudam a alcançar essa diminuição de risco.

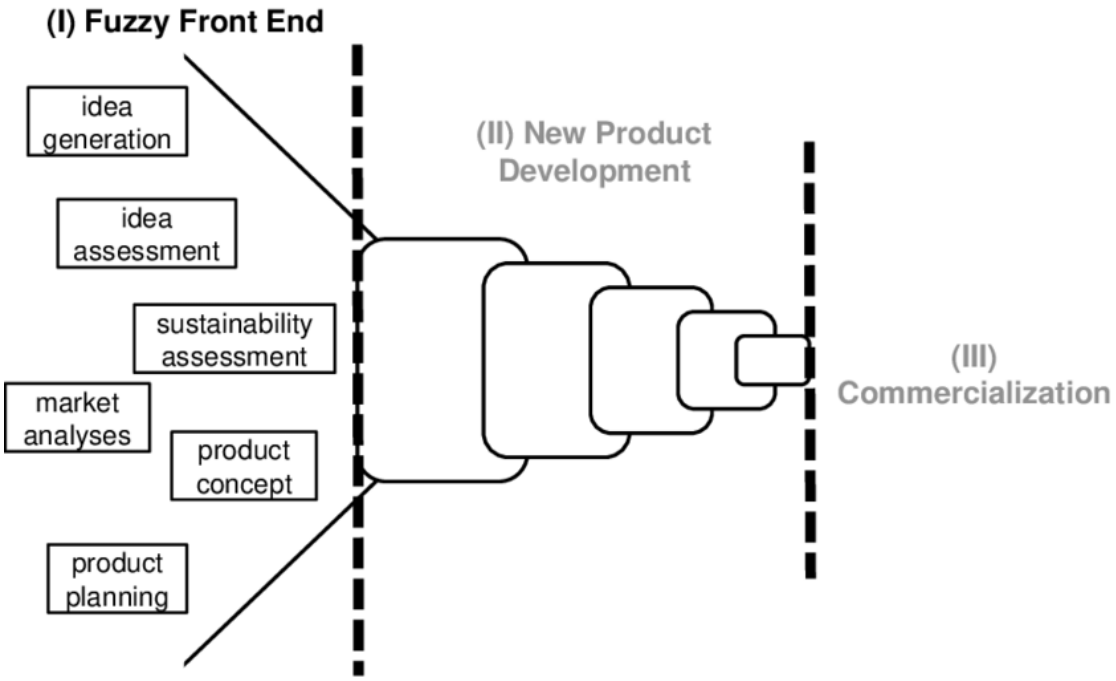

Figura 14 - Processo de inovação[49]

O processo acima identificado refere-se então em três partes fundamentais:

1. ***Fuzzy Front End*** **(FFE)** – fase de incerteza e imprevisibilidade, tanto a nível de calendarização, como de análise de ideias e identificação de possíveis oportunidades.
2. ***New Product Development*** – fase com ambiente disciplinado e planeado, com objetivos bem definidos e preparado para o desenvolvimento do novo produto ou serviço.
3. **Comercialização** – Divulgação e venda do produto ou serviço desenvolvido.

A presente dissertação irá abordar a notação do modelo NCD de Koen[50], onde irá apresentar as diversas fases do processo de invocação, desde o problema inicial até à geração de ideias e a sua respetiva seleção.

3.2 Modelo New Concept Development – NCD

O modelo *NCD*[50] (Figura 15) providencia um bom sumário de todas as atividades que acontecem antes da fase do desenvolvimento do novo produto (NPD) e consiste em três partes – os fatores sobre os quais não existe controlo, o *engine* que conduz as atividades do FFE e ainda os cinco elementos importantes que serão descritos posteriormente.

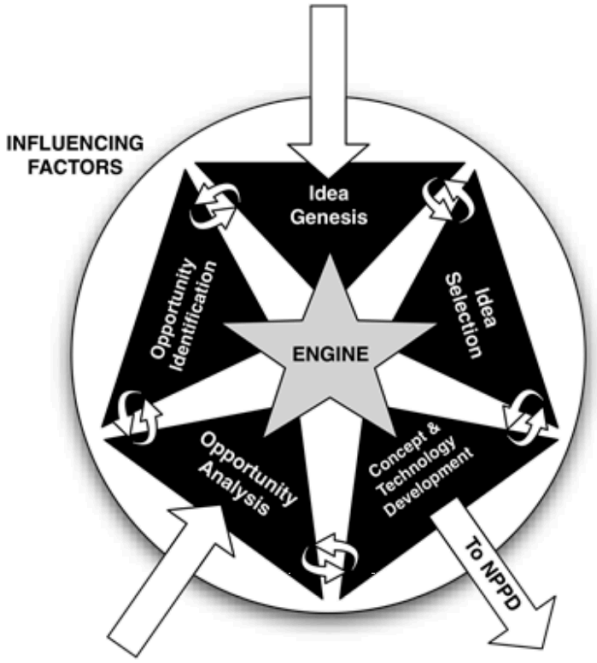

Figura 15 - Modelo New Concept Development (NCD)[51]

3.2.1 Engine

O *engine* representa a liderança, cultura e estratégia de negócio da organização a nível executivo e que conduz os cinco elementos principais.

3.2.2 Elementos da Atividade Inovadora

O modelo NCD contém cinco elementos fundamentais:

3.2.2.1 Identificação de Oportunidade

O mercado das aplicações *web* está em constante alteração, e tem-se verificado nos últimos tempos um aumento na sua utilização e produção[52], tal como representado na Figura 16. Este crescimento deve-se sobretudo à evolução da Internet como um todo, e à necessidade que existe em integrar vários serviços de rede em vários ambientes aplicacionais. O avanço da tecnologia é também um fator importante, visto que este resulta num aumento de interesse por parte dos programadores de forma a desenvolverem *software* atualizado.

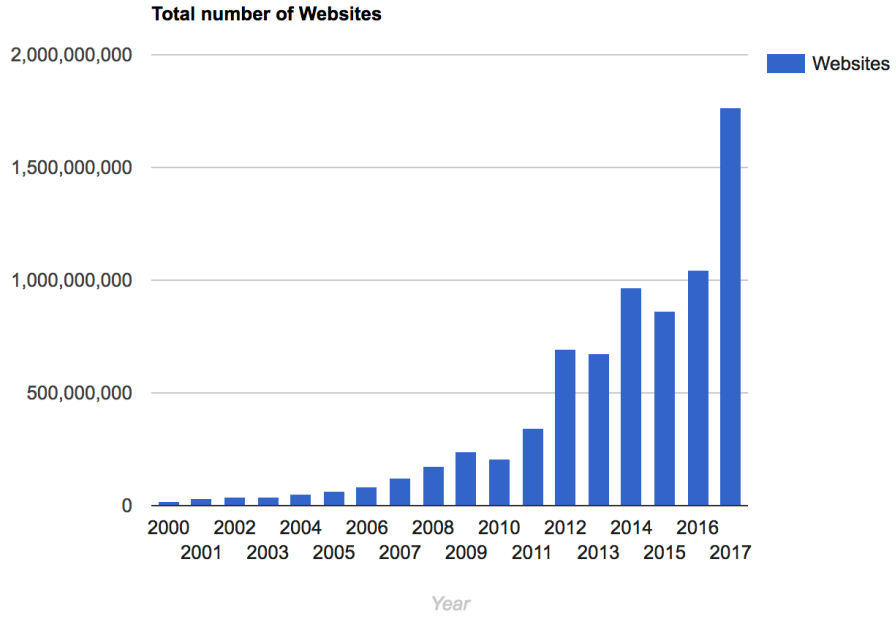

Figura 16 - Número de websites (*InternetLiveStats*)[52]

Com este crescimento é normal que exista tendência a que este tipo de produto seja cada vez mais utilizado, visto que hoje em dia existem muitas mais capacidades de integração do que alguma vez existiu. Esta tendência é auferida pela Figura 17.

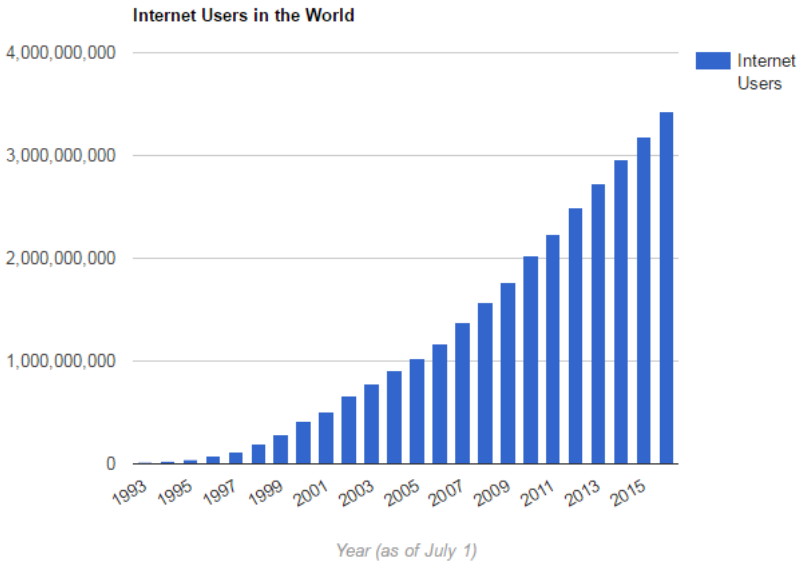

Figura 17 - Número de utilizadores de Internet (*InternetLiveStats*)[52]

A par deste crescimento, o número de aplicações que necessitem de informação em tempo real é cada vez maior, visto que é possível, nos dias de hoje, aceder a resultados de desporto, realizar apostas online, analisar descontos em plataformas de compras, etc. Com isto, a necessidade de implementação de um mecanismo de comunicação de dados que seja eficiente e que não comprometa o sistema em qualquer altura (caso a plataforma onde seja integrada tenha algum tipo de sensibilidade de informação, como é o caso das apostas) é essencial, ou prejudicará a aplicação como um todo, podendo levar à sua inutilização.

O conjunto destes fatores constitui a oportunidade que é discutida na presente dissertação e que se acredita trazer valor para os utilizadores do mundo *web* em geral.

3.2.2.2 Análise de Oportunidade

De forma a que a oportunidade identificada seja suportada, recorreu-se a dados estatísticos do mercado, nomeadamente à utilização de sistemas *web* por parte dos utilizadores, bem como o crescimento de desenvolvimento de aplicações *web*.

O crescimento do mercado é uma implicação direta nesta dissertação, que incide sobre o desenvolvimento de aplicações *web* em tempo real.

Devido a este conjunto de dados adquiridos, é possível afirmar a viabilidade da oportunidade e avançar com a ideia de forma a obter mais estatísticas sobre a mesma, para compreender se a decisão de alteração da infraestrutura para utilização de *WebSockets* é a mais correta. Esse estudo será suportado pela presente dissertação.

3.2.2.3 Geração de Ideias

Após a análise da oportunidade definida anteriormente e o seu respetivo problema, foi realizado um conjunto de processos de brainstorming, onde surgiram algumas ideias de como se poderá acrescentar valor.

Como resultado dessas sessões surgiu a ideia principal – o melhoramento da eficiência da infraestrutura de comunicação que está integrada nas aplicações *web*.

Para tal, surgiu a ideia da implementação de *WebSockets* API que atua num protocolo diferente do habitual, e para isso será necessário o desenvolvimento de um estudo que irá avaliar várias métricas que serão comparadas com outras infraestruturas (que utilizam outros protocolos), como é o caso das RESTful APIs (tendo como suporte o protocolo HTTP). Todas as grandezas de avaliação e testes de hipóteses serão definidas posteriormente na presente dissertação.

3.2.2.4 Seleção de Ideias

Após reflexão sobre as ideias geradas decidiu-se realizar o estudo que foi referido na subsecção acima, incindindo este sobre a implementação de infraestruturas de dados que tenham como suporte o protocolo *WebSocket*. Dependendo do resultado desse estudo, será analisada a possibilidade de execução da ideia considerada como principal.

3.2.2.5 Definição de Conceito

A definição de conceito surge após conclusão da definição da ideia principal, que constitui o melhoramento da eficiência da infraestrutura de comunicação utilizada numa aplicação *web*. Para isso, será necessário efetuar o estudo definido na geração de ideias.

3.3 Valor, Valor Percecionado e Valor para o Cliente

Nesta secção será apresentada a definição de valor e a quem se destina o mesmo. Será também analisado o valor percecionado, tendo como suporte a tabela de benefícios e custos.

Por fim, será analisado e apresentado o valor para o cliente.

3.3.1 Valor

O conceito de valor é subjetivo de indivíduo para indivíduo devido à sua análise particular a um produto ou serviço. Um negócio resulta da troca de produtos ou serviços, e como tal a criação de valor para quem usufrui destes é particularmente essencial.

Relativamente a esta dissertação o valor criado irá tentar ser alcançado através da comparação entre o desenvolvimento de aplicações *web* utilizando a infraestrutura de comunicações mais comum (RESTful APIs, com HTTP) e utilizando *WebSockets* API (com protocolo *WebSocket*).

Na Tabela 3 estão apresentados os benefícios e sacrifícios do valor constituinte da possível solução ao problema descrito no início deste documento.

	Serviço	Relação
Benefícios	Substituição da infraestrutura utilizada atualmente no desenvolvimento de aplicações *web;* Sem custos de licença associados;	Aumento de satisfação relativo aos utilizadores das aplicações *web*, visto obterem melhor *performance*; Mais utilizadores abrangidos por essas aplicações *web*; Fortalecimento de conhecimentos por parte dos programadores, relativos a novas tecnologias.
Sacrifícios	Perda de algumas características técnicas e consequentemente de funcionalidades.	Formação na tecnologia em questão.

Tabela 3 - Benefícios e sacrifícios

3.3.2 Valor Percecionado

O valor percecionado resulta da expetativa que existe no cliente relativamente a um produto ou serviço.

No caso da presente dissertação e tendo em conta que o cliente é definido como sendo os desenvolvedores de aplicações *web* de forma geral, o valor percecionado para estes é alto, uma vez que será possível com que as interações entre os seus utilizadores com as suas aplicações sejam mais eficientes e proveitosas, podendo resultar obviamente num crescimento da sua base de dados de utilizadores.

3.3.3 Valor para o Cliente

Esta métrica é determinada através da relação entre os benefícios e sacrifícios associados a um produto ou serviço, sendo preciso avaliar o peso de cada variável.

Ambas as vantagens e desvantagens têm de ser analisadas pelo cliente de forma a que este decida se existe a possibilidade de substituição de uma tecnologia pela outra, a seu favor. A presente dissertação tem como objetivo ajudar o cliente nesse aspeto.

3.4 Proposta de Valor do Projeto

A solução apresentada na presente dissertação tem como objetivo solucionar o problema descrito na Secção 2.2, procurando saber se os desenvolvedores de aplicações *web* deverão substituir a atual infraestrutura de comunicação entre aplicação cliente e servidora por uma outra que poderá ter mais benefícios.

Esta solução irá criar valor para os programadores, assumindo que além do conhecimento e investigação que teriam ao envergar por esta opção, poderiam ainda alargar o número de utilizadores da sua aplicação *web* devido à sua eficiência e rapidez de comunicação após interações com o utilizador.

Tendo em conta o crescimento deste novo tipo de infraestrutura, os desenvolvedores de aplicações *web* teriam também a eventual possibilidade de participação em outros projetos do mesmo nível, visto esta infraestrutura estar em grande crescimento no mercado atual.

Todas estas vantagens e desvantagens serão estudadas no decorrer da presente dissertação.

3.5 Modelo de Negócio Canvas

Com o objetivo de auxiliar o negócio e a sua compreensão a uma escala global foi criado um modelo de negócio Canvas, representado na Figura 18.

São apresentados os componentes que estão descritos no modelo.

- **Parcerias Chave**
  A principal parceria chave seria a comunidade *open-source* de desenvolvimento de aplicações *web*, no seu geral.

- **Atividades Chave**
  A atividade chave seria a execução da comparação entre as diversas tecnologias que são utilizadas no desenvolvimento de infraestruturas de comunicação envolvidas nas aplicações *web*, bem como as respetivas aplicações que serviriam de protótipo a essa comparação.

- **Recursos Chave**
  O único recurso chave é toda a infraestrutura da aplicação em si, bem como os próprios desenvolvedores.

- **Estrutura de Custos**
  Não existe nenhum custo associado ao negócio, visto que a ferramenta para o desenvolvimento deste tipo de infraestrutura está disponível de forma gratuita.

- **Proposta de Valor**
  A proposta de valor é o aumento de conhecimento por parte dos desenvolvedores de aplicações *web*, bem como o aumento substancial da eficiência das aplicações que efetivamente construírem e que usufruem deste tipo de infraestruturas.

- **Relação com os Clientes**
  A relação com os clientes será considerada uma relação direta, ou indireta, visto que a aplicação em que estão a trabalhar poderá ser vendida por outra entidade.

- **Canais**
  Existe uma mudança de infraestrutura diretamente na fonte, ou seja, a partir dos desenvolvedores de aplicações *web*.

- **Segmentos de Mercado**
  Existem vários segmentos de mercado à qual esta infraestrutura é uma forte candidata a ser utilizada dentro do mercado que é as aplicações *web*, principalmente as que necessitem de dados em tempo real, como por exemplo o mercado das apostas, da banca ou até do desporto.

- **Fontes de Rendimento**
  A fonte de rendimento não é apresentada visto que não é diretamente executada, mas sim a partir do eventual número de vendas de aplicações que são consideradas eficientes e bem construídas.

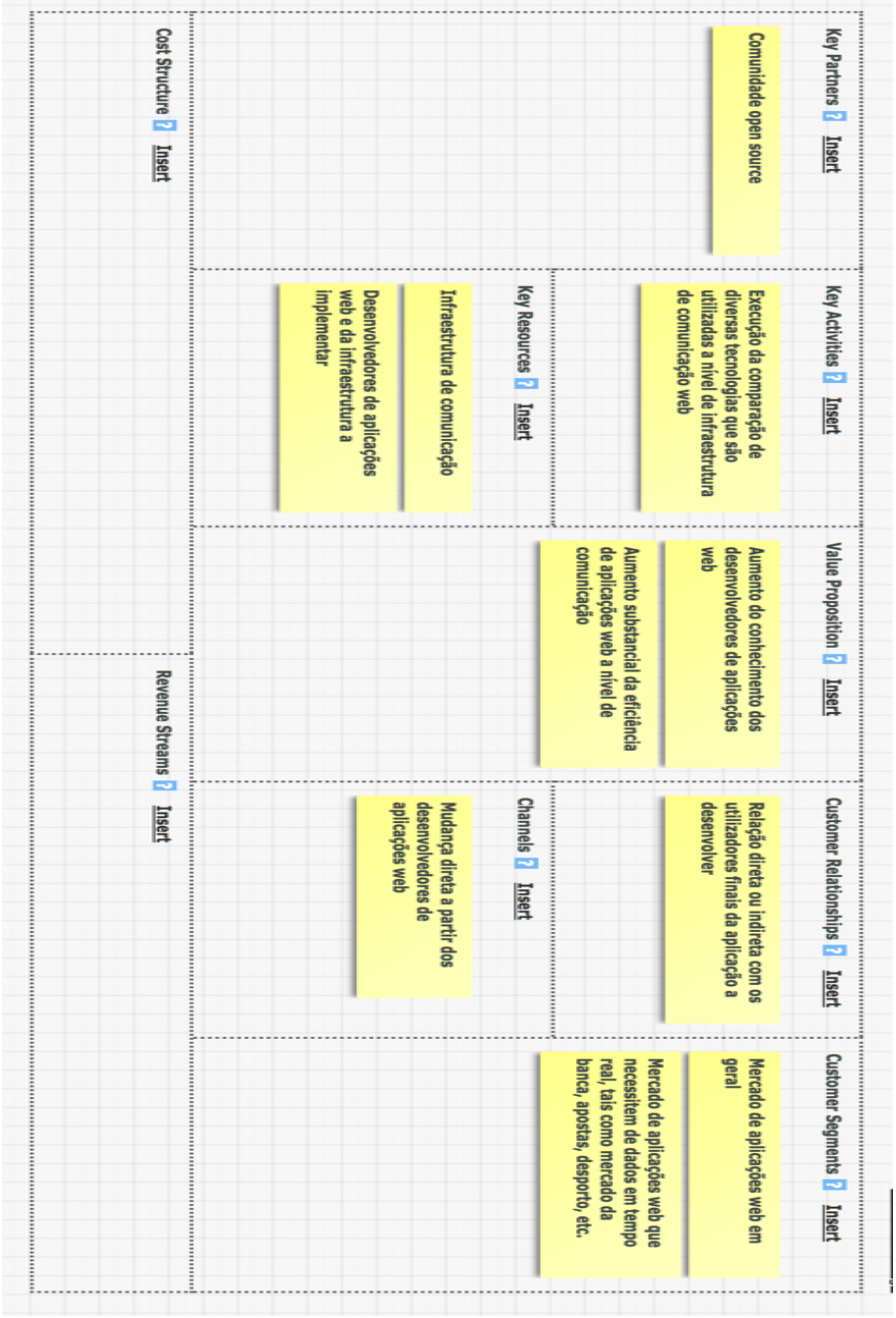

Figura 18 - Modelo de negócio Canvas

3.6 Rede de Valor

Uma rede de valor consiste num conjunto de relações complexas que poderão gerar valor tangível ou intangível. Esta metodologia que permite estudar este tipo de relações foi criada pela autora *Verna Allee* e pretende demonstrar que estas normalmente acontecem entre dois

ou mais indivíduos, grupo ou organização. É considerada uma rede de valor toda a organização que esteja envolvida na troca de valores tangíveis ou intangíveis [53].

No caso da presente dissertação será necessária uma relação de aprendizagem entre os vários programadores que de alguma forma já utilizam (ou utilizaram) este tipo de infraestrutura. Devido ao facto desta infraestrutura ter sido implementada através do *open source*, esta contém obrigatoriamente uma comunidade, que poderá ser considerada a rede de valor. Dentro da rede de valor é sempre instigada a partilha de conhecimento e a troca de valores entre os vários membros.

3.7 Análise Hierárquica – Modelo AHP

O modelo *Analysis Hierarchy Process* (AHP) foi desenvolvido por *Thomas L. Saaty* em meados de 1970 e tem sido estudado de forma exaustiva, sendo atualmente utilizado para auxílio na toma de decisões em cenários complexos[54].

O modelo AHP baseia-se na definição de vários critérios qualitativos e quantitativos (representado na Figura 19) que são trabalhados de forma a se obter um resultado final através de cálculos e que se traduzem numa decisão relativamente a um conjunto de opções iniciais.

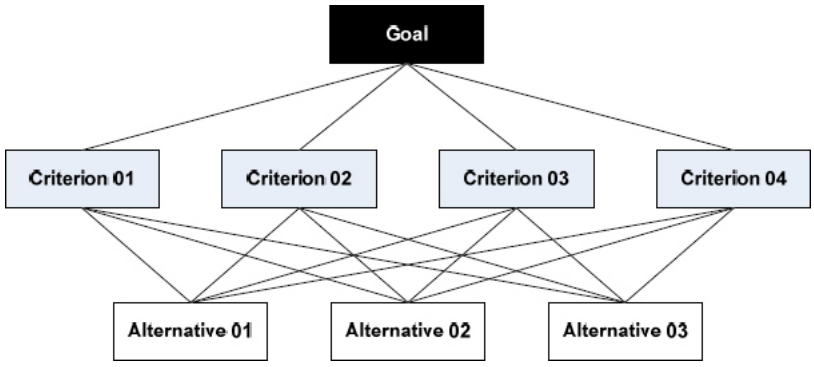

Figura 19 - Exemplo de hierarquia de critérios e objetivos

Na presente dissertação é utilizado o modelo AHP de forma a que seja distinta a melhor tecnologia / protocolo a ser desenvolvida, sendo os critérios os seguintes:

- **Desempenho**
- **Escalabilidade**
- **Complexidade**

De forma a quantificar de forma relativa os critérios, *Saaty* criou uma escala de valores (representada na Tabela 4) que se traduzem a importância de relacionamento entre critérios.

Tabela 4 - Escala de Saaty[55]

AHP Scale of Importance for comparison pair (aij)	Numeric Rating	Reciprocal (decimal)
Extreme Importance	9	1/9 (0.111)
Very strong to extremely	8	1/8 (0.125)
Very strong Importance	7	1/7 (0.143)
Strongly to to very strong	6	1/6(0.167)
Strong Importance	5	1/5(0.200)
Moderately to Strong	4	1/4(0.250)
Moderate Importance	3	1/3(0.333)
Equally to Moderately	2	1/2(0.500)
Equal Importance	1	1 (1.000)

A utilização de números pares só deverá acontecer se existir uma necessidade de negociação entre avaliadores (participantes na decisão). Caso contrário, é preferencial a utilização de números ímpares para comprovar a existência de uma distinção razoável perante os critérios.

Tendo em conta a escala referida, pode-se afirmar que o critério de **Desempenho** é o mais importante, sendo 3 vezes mais do que a **Escalabilidade** e 9 vezes mais do que a **Complexidade**. Por sua vez, a **Escalabilidade** é 3 vezes mais importante do que a **Complexidade**.

A Tabela 5 tem o objetivo de demonstrar a matriz de importância dos critérios referidos anteriormente.

Tabela 5 - Matriz de importância de critérios

	D	**E**	**C**
D	**1**	**3**	**9**
E	**1/3**	**1**	**3**
C	**1/9**	**1/3**	**1**

Através da matriz de importância é possível aferir o vetor da Tabela 6, que consiste no vetor de prioridades e irá indicar o peso de cada critério. Este cálculo é feito através da divisão de cada elemento de uma coluna da matriz com a soma dos valores dessa mesma coluna, em conjunção com o cálculo da média de cada nova linha da matriz.

Tabela 6 - Vetor de prioridades

D	**0.69**
E	**0.23**
C	**0.08**

Obtendo o vetor de prioridades (Tabela 6) é agora possível aplicar o conceito de cada peso de critério aos diferentes protocolos.

Consegue-se auferir que o critério de desempenho tem aproximadamente 69.2% de prioridade, seguido do critério de escalabilidade com 23.1%, e por fim o critério de complexidade com 7.7% [56].

3.7.1 Desempenho

Tal como indica a Matriz da Tabela 7, pode-se auferir que o desenvolvimento de API com suporte a HTTP demoram em média mais tempo a resolver os seus pedidos segundo um estudo[57], e daí concluir que terá, em média, ligeiramente menos desempenho do que *WebSockets*.

Tabela 7 - Matriz de comparação de desempenho

	HTTP	**WebSockets**
HTTP	**1**	**1/3**
WebSockets	**3**	**1**

3.7.2 Escalabilidade

Segundo a Matriz da Tabela 8, pode-se concluir que as API que tenham como suporte o HTTP (nomeadamente as RESTful API) têm uma ligeira vantagem em relação aos *WebSockets* e à sua escalabilidade, por serem *stateless*, o que significa que qualquer servidor consegue responder a qualquer pedido, não existindo necessidade de guardar estado entre sessões e/ou pedidos[57].

Tabela 8 - Matriz de comparação de escalabilidade

	HTTP	WebSockets
HTTP	**1**	**2**
WebSockets	**1/2**	**1**

3.7.3 Complexidade

Não existindo qualquer estudo referente à comparação de complexidade entre o desenvolvimento de API que suportem os diferentes tipos de protocolos, não é possível analisar com detalhe este critério.

3.7.4 Classificação de protocolos / APIs

Após a análise detalhada de cada critério e da definição das matrizes respetivas, é necessário calcular o vetor prioridade para cada um dos critérios, tal como representado naTabela 9.

Tabela 9 - Matriz das prioridades

	D	C	E
HTTP	**0.25**	**0**	**0.67**
WebSockets	**0.75**	**0**	**0.33**

Tendo estes dados todos salvaguardados, pode-se auferir a classificação dos protocolos (Tabela 10) através da multiplicação de cada elemento da matriz de prioridades (Tabela 9) com o peso dos critérios calculadores anteriormente (Tabela 6), somando posteriormente a linha de forma a obter o resultado final.

Tabela 10 - Classificação de protocolos / APIs

	D	C	E	Resultado
HTTP	**0.1725**	**0**	**0.1541**	**0.3266**
WebSockets	**0.5175**	**0**	**0.0759**	**0.5934**

Tal como se pode confirmar pela classificação de protocolos, o de *WebSockets* possui um resultado favorável de cerca de 59% em relação aos aproximados 33% do HTTP (REST).

3.8 Sumário

Este capítulo descreve a análise do problema através da sua análise de valor, onde foram seguidos os modelos NCD e AHP. Foram também definidos o valor, valor percecionado, valor para o cliente e por fim foi apresentada a proposta de valor.

4 Padrões de *Software* e *Benchmarks*

Neste capítulo são apresentados os padrões de *software* que servirão de suporte ao desenvolvimento do protótipo, bem como será descrito o conceito de testes de *performance* em aplicações *web*, e ainda ferramentas que são importantes na aferição de métricas e resultados desses testes.

4.1 Padrões de desenvolvimento de *software*

Para a implementação do protótipo é necessário ter em mente princípios básicos de desenvolvimento de *software* que são fulcrais para a sua operabilidade.

Segundo o IEEE[58] o *design* arquitetural é denominado como um processo de definição de uma coleção de componentes de *hardware* e *software* e as suas respetivas interfaces de forma a estabelecer uma ferramenta para o desenvolvimento de um sistema de computação.

Esta deverá ser uma das primeiras etapas no desenvolvimento de *software* visto que é nesta fase que são traçados os planos e princípios e/ou padrões arquiteturais que a solução deve respeitar, de forma a que esta seja coerente e coesa.

A presente secção terá como suporte o uso das vistas lógica e de processo do modelo 4+1 [59].

4.1.1 Princípios arquiteturais

Nesta subsecção são apresentados alguns dos pontos-chave que constituem parte do que são consideradas boas práticas no desenvolvimento de software[60] e sobre os quais a solução terá de respeitar.

4.1.1.1 Separation of concerns (SoC)

Separation of concerns representa um princípio que defende que a aplicação deverá ser separada em diversas secções e/ou componentes, de forma a que cada uma/um seja resposta a uma necessidade apenas. Isto resultará numa maior eficiência na manutenção do *software*, visto que todo o comportamento relativo a uma funcionalidade e/ou conjunto de

funcionalidades relacionadas estará isolado e não deverá ter qualquer impacto noutros componentes.

4.1.1.2 Single Responsibility Principle

O princípio de *Single Responsibility* está de certa forma relacionado com o princípio referido anteriormente, *SoC*, visto que este também afeta a estrutura do *software* de forma a que cada componente responda a uma necessidade e que tenha apenas uma responsabilidade, restringindo e controlando dessa forma o comportamento que é esperado o *software* ter perante uma determinada situação ou contexto.

4.1.1.3 Principle of Least Knowledge (or Law of Demeter)

O princípio do menor conhecimento (ou também conhecido com o a Lei de *Demeter*[61]) consiste na limitação de conhecimento que está associado a um objeto / componente, de forma a evitar com que vários objetos diferentes se cruzem entre si de forma a existir uma incoerência de comportamentos.

4.1.1.4 Camada de serviço

Na maioria das aplicações que necessitem de interação com dados dinâmicos é importante que exista uma camada de serviço (denominada de *Service Layer*[62]), representada na Figura 20.

Esta consiste numa *interface* que é comum à lógica da aplicação e que permita o acesso externo a dados e funções de serviços que manipulem determinados dados. O objetivo desta camada é ajudar na centralização do acesso aos dados e às funções dos serviços, bem como ajudar, dessa forma, à abstração da implementação interna e a posteriores alterações. Além destas vantagens, é também possível atribuir versões aos serviços para que sejam mais facilmente identificados em caso de erro e/ou investigação de problemas.

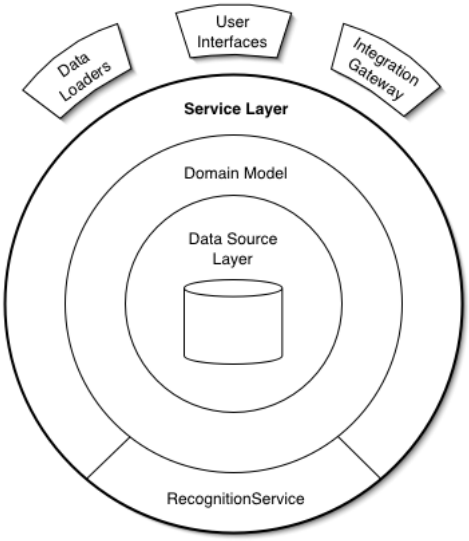

Figura 20 - *Service Layer*

4.2 Testes de performance web

O objetivo da dissertação é considerar o impacto que o protocolo *WebSockets* tem quando implementado numa aplicação *web*, podendo ser a solução ao problema descrito na Subsecção 2.1.1, resolvendo desta forma os problemas de sincronização entre componentes numa arquitetura cliente-servidor. Este conceito de sincronização ente componentes pode ser apoiado por testes de *performance*. A realização de testes de *performance* é um aspeto importantíssimo na manutenção de uma API (independentemente do seu tipo) ou aplicação *web*.

Quando aplicado ao mundo *web,* não é suficiente afirmar que se fazem testes de *performance* numa aplicação *web* ou API. *Performance* não significa apenas a medição de tempos de resposta, nem afirmar que a única implicação que um mau desempenho tem numa aplicação é o facto de ela poder operar de forma lenta.

O objetivo de testar e monitorizar o desempenho de uma aplicação *web* ou API é o de poder encontrar *bottlenecks*[63] de forma eficiente. *Bottlenecks* representam causas de problemas que uma determinada aplicação pode ter devido a limitações num certo contexto, e que podem impactar negativamente a *performance* desta[64]. A existência destes problemas pode causar utilizadores insatisfeitos, ou no pior caso à não utilização da aplicação.

O investimento em ferramentas que permitam ajudar a encontrar *bottlenecks* é considerado um passo importante na entrega de *software* consistente e de alta qualidade.

De referir que com a informação presente neste capítulo será possível responder à questão **Q1** (1.3).

4.2.1 *Tipos de testes de desempenho*

Apesar de existirem vários tipos de teste de desempenho, o importante será avaliar (consoante o contexto de negócio onde a aplicação ou API se inclui) quais os tipos de teste que melhor se adaptam ao problema. Vários domínios aplicacionais podem requerer diferentes tipos de testes e/ou métricas, daí ser importante ter um conhecimento vasto sobre o tipo de testes que existe, de forma a obter os melhores dados sobre a aplicação/API.

A Microsoft juntou uma lista de conceitos relativos aos tipos de teste de *performance* numa aplicação *web*[65]. De seguida será apresentada uma lista com os pontos essenciais por tipo de teste, de forma a sumarizar o que foi referenciado pela documentação da Microsoft.

4.2.1.1 Testes de carga (*load tests*)

Os testes de carga são usados de forma a determinar qual o desempenho de uma aplicação, com base num determinado volume de utilizadores, num certo período de tempo. Genericamente, estes testes aumentam a quantidade de pedidos ao longo da sua duração, apesar de serem polivalentes na medida em que conseguem ser configuráveis para aferir um espetro alargado de valores (baixa, média e alta carga).

Este tipo de testes têm como objetivo a resposta aos *Service Level Agreements*[66] (SLAs), que representam os objetivos de desempenho que foram especificados para uma aplicação em concreto.

4.2.1.2 Testes de stress (stress *testing*)

Os testes de stress são semelhantes aos testes de carga à exceção do facto de que estes foram especificamente criados de forma a testar a máxima capacidade de desempenho de uma aplicação, com valores de carga máxima, ou além de máxima. O objetivo deste tipo de testes é revelar problemas aplicacionais que só apareceriam quando o sistema opera com valores máximos de carga. Estes problemas podem ser caracterizados como problemas de sincronização, concorrência ou memória (*memory leaks*)[65]. Apesar dos problemas identificados, é expectável que a aplicação continue a funcionar.

4.2.1.3 Testes de resistência (*endurance testing*)

Os testes de resistência são um tipo de testes que focam na reprodução do nível de carga considerado normal para uma aplicação em específico, tendo como objetivo avaliar a capacidade de esta operar a um nível de utilização expectável, por um longo período de tempo[67].

4.2.1.4 Testes de capacidade (*capacity tests*)

Os testes de capacidade têm como objetivo ajudar a determinar a capacidade que uma aplicação tem de operar quando o número de utilizadores aumentar, num futuro. Além da capacidade é também possível determinar que recursos serão necessários no futuro para conseguir suportar uma determinada carga, superior à existente no momento do teste.

4.2.1.5 Outro tipo de testes

Apesar de existirem vários tipos de testes, tal como os mencionados anteriormente, é necessário que existam ações a retirar após cada execução de um conjunto de testes. Caso contrário, o teste não ajudará a melhorar o desempenho da aplicação *web*.

De seguida serão apresentadas algumas métricas e *benchmarks* que podem ser tidos em conta numa avaliação de desempenho de uma aplicação *web* ou API.

4.2.2 Benchmarks

Um dos aspetos críticos de realização de testes de *performance* implica encontrar o conjunto de métricas que são consideradas importantes no contexto da aplicação *web* e do domínio onde esta se encontra. Por exemplo, caso a aplicação a ser testada se encontre integrada na indústria do *e-commerce*, uma das métricas de desempenho consideradas pode ser o número de vendas. No entanto, as métricas que se pretendem considerar na dissertação são mais técnicas e não são relativas a conceitos de domínio, visto que o que se pretende demonstrar será mesmo o desempenho de um determinado protocolo.

A avaliação do conjunto de métricas errado pode levar a conclusões erradas pois os dados obtidos não são válidos no contexto. Desta forma, os tipos de dados obtidos são importantíssimos na avaliação de uma solução.

De seguida vão ser descritas as métricas mais utilizadas no que diz respeito à avaliação de *performance* nas aplicações *web* e nas API atualmente[67].

4.2.2.1 Tempo de resposta médio

O tempo de resposta médio é provavelmente a métrica mais utilizada nos testes de *performance* (Figura 21). É sem dúvida uma métrica que deve estar presente no conjunto de testes a fazer, visto que através dela é possível verificar de que forma a aplicação se comporta consoante os diferentes níveis de carga. É benéfico na visibilidade de possíveis regressões em caso de alguma alteração de código recente.

Para uma avaliação eficaz é necessário definir previamente em que ponto é considerado que a aplicação seja demasiado lenta, ou seja, qual o máximo tempo de resposta que é esperado ter.

Em suma, quanto menor for o valor, mais fluída será a aplicação pois responde às ações do utilizador de forma rápida e interativa.

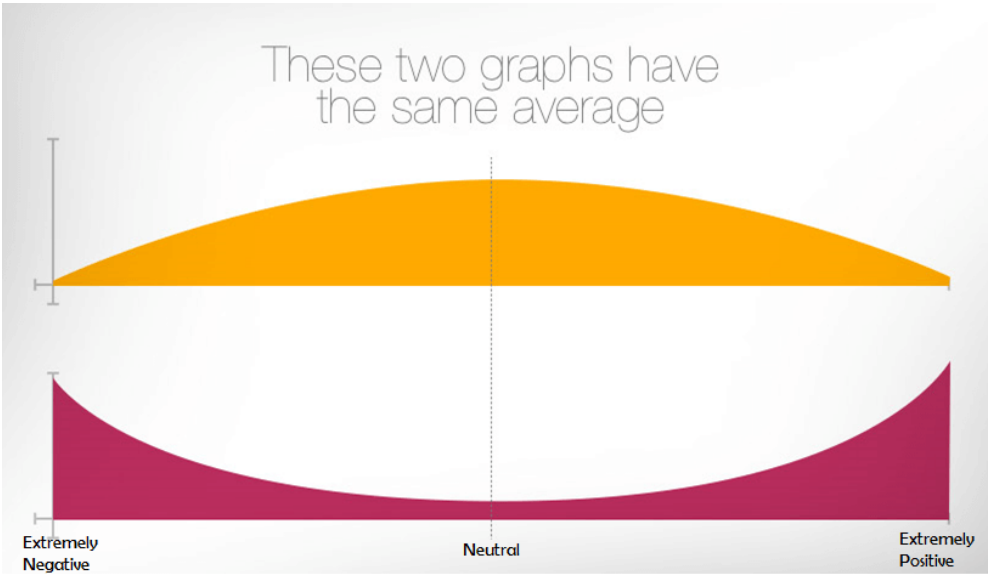

Figura 21 - Exemplo de testes de tempo de resposta médio[68]

4.2.2.2 Tamanho de *payload*

Em qualquer pedido feito numa aplicação que assente na arquitetura cliente-servidor é feita uma transferência de dados do servidor para o cliente.

Uma das métricas mais importantes na averiguação do desempenho de uma aplicação *web* é a análise feita aos dados enviados tanto ao cliente como ao servidor[69].

À medida que as aplicações *web* incorporem funcionalidades mais robustas e complexas, o tamanho do *payload* enviado para o cliente é essencial, especialmente para os utilizadores que possuam uma ligação à internet mais lenta. Respostas do servidor que contenham mais informação do que é suposto levam a um grande impacto negativo no desempenho da aplicação, visto que esta irá tornar-se mais lenta, o que consequentemente levará ao abandono dos utilizadores. A Google refere que, em média, uma página de uma aplicação *web* que demore no mínimo três segundos a carregar leva a uma hipótese de 53% dos utilizadores abandonarem a página antes que seja feito o seu carregamento total[70].

4.2.2.3 Percentagem de erros

A análise de percentagem de erros que ocorrem na aplicação são parte do conjunto de métricas a ter em conta na avaliação de desempenho uma aplicação *web.* O objetivo é fornecer uma perspetiva de que componentes falham num determinado nível de utilização da aplicação. Em geral quando acontece um problema na aplicação, pode não haver registos de falha de pedidos (independentemente da carga do servidor).

Existem sobretudo três tipos de potenciais erros que podem ser tidos em conta na análise desta métrica:

1. Percentagem de erros HTTP – número de pedidos HTTP que resultam em erro;
2. Exceções aplicacionais registadas – número de erros registados pela aplicação;
3. Exceções em geral – número de todas as exceções.

O número de exceções que são ignoradas numa aplicação tem a probabilidade de ter grande impacto de desempenho.

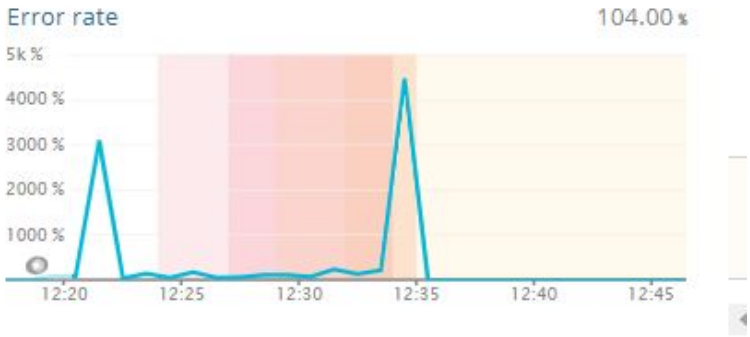

Figura 22 – Exemplo de registo de percentagens de erros[68]

4.3 Tecnologias e ferramentas relevantes

Para o desenvolvimento do protótipo que serve de exemplo para o estudo foi utilizado um dos IDE[16] mais utilizados no mercado do desenvolvimento Web - *Visual Studio Code*[74]. A decisão de opção por este IDE baseia-se sobretudo na familiaridade que já existia para com o trabalho no mundo profissional, mas sobretudo na qualidade de integração deste com as várias tecnologias que constituem parte do desenvolvimento. Um outro fator importante na escolha do IDE é o custo do mesmo, sendo que o *Visual Studio Code* é um projeto *open source*, ou seja, não tem qualquer custo de uso, contando também com uma comunidade bastante ativa.

A linguagem a ser utilizada foi o *Node.js* (que se baseia em *JavaScript*), visto que este foi inventado como objetivo de criar *websites* em tempo real, contendo capacidades de *push* [75] - estilo de comunicação que representa o *envio* de informação para um cliente através de um paradigma I/O[17] baseado em eventos[76].

Esta secção não tem como objetivo a comparação de soluções e alternativas de implementação, mas sim a justificação da razão de opção por uma delas. Para a realização do estudo no âmbito do presente documento foi considerado que Node.js seria o ambiente de desenvolvimento perfeito, visto que devido à sua arquitetura, e comparado com outras linguagens, tem uma melhor performance em sistemas de alta concorrência, pois esta não utiliza *threads* na gestão de tarefas paralelas, mas sim um ciclo baseado em eventos e *callbacks* assíncronas.

Apesar das várias alternativas à implementação relativa à tecnologia de *WebSockets* foi escolhida a utilização do *Socket.IO*, visto que o protótipo contém uma API desenvolvida em *Node.js*, construída com a ajuda da ferramenta *Feathers* (ver Subsecção 4.3.2) que por sua vez

[16] *Integrated development environment,* ou seja, *software* que auxilia na programação de várias linguagens
[17] Input/Output

inclui uma grande integração com a tecnologia *Socket.IO*. A ferramenta em questão (*Feathers*) apoia na resolução de um requisito mencionado no início do Capítulo 5, e permite que existam várias implementações de vários tipos de API numa só aplicação (tal como explicado na Subsecção 4.3.2).

4.3.1 Ferramentas de monitorização

As ferramentas que auxiliaram no estudo e medição de métricas e testes necessários para discussão na presente tese foram obtidos sobretudo através da ferramenta de *debug* incorporada no *Google Chrome*, *Google Chrome Developer Tools*. Todas as informações estatísticas de pedidos (contendo *status code* de cada pedido, resposta, etc.) estão presentes na respetiva secção de *Network* do *Developer Tools*.

Além do *Developer Tools* foi também utilizada a aplicação *desktop*[18] do *Wireshark*, que permite analisar o tráfego numa determinada rede, contendo informação mais detalhada de cada pedido (tal como endereços IPs, *bytes* transferidos entre servidores, *etc*.).

Ambas as ferramentas foram importantes na aferição de valores e variáveis que serão explicadas posteriormente e demonstradas em gráficos na Secção 5.4.

4.3.2 Feathers

Feathers [77] é uma ferramenta *web* que permite a criação de aplicações que necessitem de dados em tempo real. É polivalente pois funciona com múltiplos sistemas de gestão de base de dados e com praticamente qualquer tecnologia de *front-end.*

Esta ferramenta tem inúmeras vantagens na sua utilização, entre as quais: flexibilidade no desenvolvimento de aplicações visto que suporta várias linguagens (JavaScript, TypeScript, etc.) e não obriga a que a implementação seja de uma certa forma específica, apesar de ser providenciada a estrutura necessária para um bom funcionamento de uma aplicação; existem vários *adapters*[19] que permitem a conexão a vários sistemas de gestão de base de dados (*mySQL*, *postGres*, etc.); a estrutura que a ferramenta providencia para o desenvolvimento é orientada a serviços, o que permite, caso necessário, que a transição para uma arquitetura de *microservices*[78] seja facilitada; a ferramenta é bastante fácil de utilizar e é escalável à necessidade do programador, visto que existe um vasto ecossistema de *plugins* que podem ser instalados e prontamente utilizados; por fim, esta ferramenta garante uma funcionalidade

[18] versão PC/Mac de um software
[19] camadas abstratas de conexão a uma determinada infraestrutura

CRUD (*Create, Read, Update, Delete) out-of-the-box*[20], através de serviços que estão disponíveis tanto em REST, como em *WebSockets*.

Um serviço do *Feathers* (Figura 23) caracteriza-se por um objeto que contém métodos pré-definidos para manipulação (leitura, criação, remoção e atualização) de entidades que sejam incluídas na aplicação.

Figura 23 - Exemplo de um serviço no *Feathers*

A aplicação final consiste num projeto desenvolvido em *Node.js* e que inclui a configuração *Feathers* necessária para que sejam permitidos os devidos testes. O objetivo desta configuração é criação da possibilidade de a aplicação usar dinamicamente um protocolo ou o outro, consoante necessidade e/ou indicação do programador.

Para que isto aconteça, é necessário importar os dois pacotes necessários *(@feathers/express* e *@feathers/socketio*). Estes pacotes são bibliotecas utilizadas para suporte de protocolo HTTP (*express*) e WebSockets (*socketio*). Optou-se pela utilização do pacote relativo ao *Socket.IO*, por ser um dos mais utilizados e também por estar referido na Subsecção 2.3.3 como sendo uma das soluções e/ou abordagens a ter em conta.

Após instalação das respetivas bibliotecas, é necessário configurar a aplicação servidora (ver Código 2) para que estes estejam disponíveis para utilização posterior. Para isso basta adicionar duas linhas de código para o efeito.

```
// Add REST API support
app.configure(express.rest());
// Configure Socket.io real-time APIs
app.configure(socketio());
```

Código 2 - Feathers - Configuração de Socket.IO e REST

[20] sem necessidade de configuração, pronta a utilizar

Após tudo configurado, é necessário proceder à implementação dos requisitos funcionais que foram especificados na Subsecção 5.1.1.

4.4 Sumário

Este capítulo é iniciado com uma apresentação dos padrões de desenvolvimento de software a ter em consideração no desenvolvimento do protótipo. De seguida é feita uma descrição em detalhe dos testes de *performance web*, e dos seus diversos tipos de teste que podem e devem ser feitos quando implementada uma aplicação *web*. Por fim, é feito o levantamento dos *benchmarks* e métricas essenciais para avaliação assertiva do desempenho de uma aplicação *web*, bem como são mencionadas algumas tecnologias e ferramentas relevantes ao desenvolvimento.

De referir que o capítulo responde em parte à questão **Q1** levantada na Secção 1.3, tendo sido identificadas métricas e *benchmarks* importantes para a avaliação do desempenho de uma aplicação *web*, os quais incluem a monitorização do tempo de resposta médio entre pedidos, tamanho de dados transferidos durante a comunicação cliente-servidor e percentagem de erros.

5 Desenvolvimento do protótipo

Este capítulo consiste na apresentação da síntese de toda a implementação do projeto que serve para demonstração do impacto que o protocolo *WebSockets* tem nas aplicações *web*, e apresenta alguns fatores importantes que foram considerados ao longo do desenvolvimento.

De forma a garantir que a implementação do projeto seja coerente com as grandezas, metodologias de avaliação (ver Subsecção 6.1) e princípios arquiteturais (ver Subsecção 4.1.1), é necessário garantir que este desenvolvimento seja feito num ambiente controlado, e que este seja idêntico aquando a comparação entre as duas camadas de comunicação – *WebServices* e *WebSockets*.

A abordagem para o desenvolvimento do projeto seria a implementação de duas aplicações que funcionariam de API – uma API *RESTful* que utilizaria o protocolo HTTP, e outra que seria construída tendo por base o protocolo *WebSockets*. Seria também desenvolvida uma aplicação *web* (com componentes de *front-end*), que interagia com estas APIs e executava alguns testes consoante algumas métricas, expondo por fim os resultados dos mesmos.

No entanto, após longa investigação acerca das possíveis alternativas que ajudassem na implementação das respetivas API foi descoberta uma opção que ajudaria a satisfazer este requisito. Após um ponderado estudo sobre a ferramenta em questão, *Feathers*, esta foi considerada válida para o âmbito do projeto, e seria por isso a ferramenta a ser utilizada para o desenvolvimento do protótipo da presente dissertação.

O presente capítulo começa por apresentar uma visão sobre a análise e *design* do protótipo a desenvolver, passando de seguida à demonstração da implementação dos requisitos funcionais e da interface do utilizador.

5.1 Análise e design

Nesta secção serão analisadas as funcionalidades a implementar, bem como os requisitos funcionais e não funcionais.

5.1.1 Requisitos funcionais

As funcionalidades definidas para implementação da solução em questão surgiram através de necessidades que existem no mercado atual de aplicações em tempo real, como é o caso do mercado de desporto, ou até da banca.

Os requisitos funcionais derivam das necessidades de avaliação de métricas e *benchmarks* (referidas na Subsecção 4.2.2), bem como de funcionalidades mais implementadas atualmente em sistemas de tempo real (nomeadamente na indústria de desporto e apostas desportivas[79]). Estes estão representados num diagrama de casos de uso (Figura 24) e constituem:

- **UC01** – Obter resultados desportivos em tempo real.
- **UC02** – Visualizar em detalhe informações estatísticas sobre os resultados.
- **UC03** – Criar perfil de utilizador.
- **UC04** – Comentar em direto (através de um chat) os resultados desportivos.
- **UC05** – Obter quotas de mercado relativas às equipas envolvidas num determinado jogo.
- **UC06 –** Obter *odds*[21] de apostas desportivas.

O objetivo posterior de cada requisito funcional é também obter resultados de *benchmarking* sobre cada um, de forma a ter métricas para chegar a uma conclusão final.

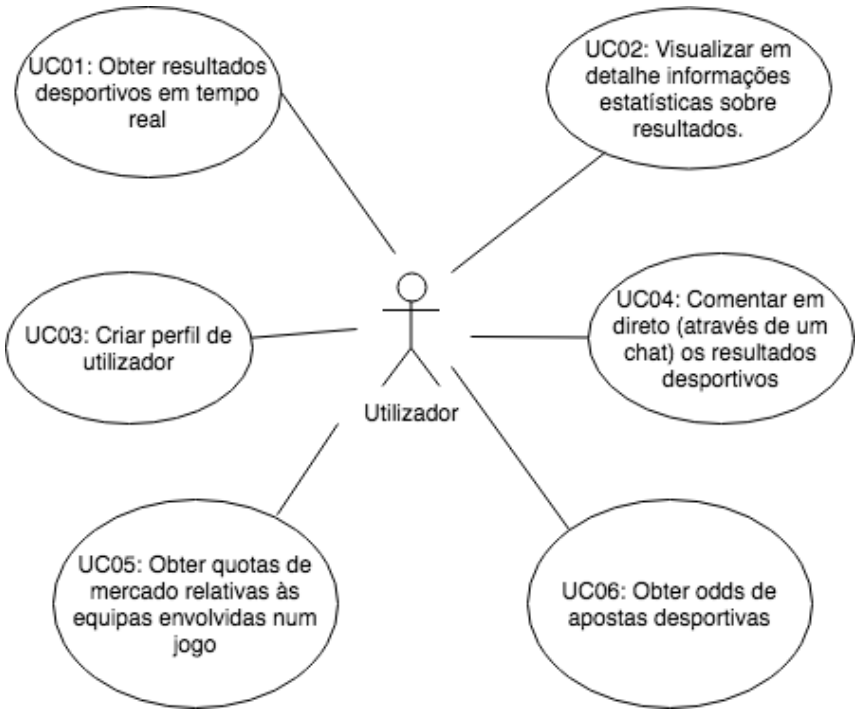

Figura 24 - Diagrama de casos de uso

[21] Probabilidades

Todos os casos de uso terão impacto na comparação dos protocolos visto que ambos derivam de funcionalidades que refletem e/ou interferem com dados em tempo real, ou seja, sem necessidade de alguma interação com o utilizador[80].

O *upload*[22] de dados do tipo de media (por exemplo imagem de perfil de utilizador) é também um fator importante visto que é uma transferência de um tipo de dados diferente dos pedidos comuns (respostas em formato objeto), e contribuirá para mais uma métrica de avaliação e comparação do protocolo.

5.1.2 Requisitos não funcionais

Relativamente aos requisitos não funcionais é importante que a aplicação a ser desenvolvida cumpra as seguintes temáticas, garantidamente:

- **Tecnologicamente atualizado** – as tecnologias envolvidas no processo de desenvolvimento de ambas as aplicações protótipo serão sobretudo o *JavaScript* e o *Node.js*.
- **Desempenho** – ambas as aplicações têm de garantir tempos de resposta baixos (até 10 segundos[81]) e serem suficientemente escaláveis.
- **Multiplataforma** – todos os *browsers* terão de ser capazes de executar ambas as soluções desenvolvidas.
- **Ambiente de testes suficientemente robusto e semelhante** – é necessário garantir que ambas as aplicações se encontrem no mesmo contexto e ambiente de teste, de forma a assegurar que os resultados obtidos posteriormente sejam o mais coerente possível. É também necessário garantir a robustez desses sistemas, visto que os testes a serem executados serão de carga considerada alta.

5.1.3 Vista lógica

A solução a desenvolver fará uso dos princípios que foram mencionados na Subsecção 4.1.1 e estará dividida em três grandes camadas:

- Camada de *User Interface* – interface gráfica sobre a qual o utilizador irá interagir e parte visual da solução e todos os componentes *front-end* que darão auxílio à interação do utilizador com a aplicação;
- Camada de lógica de negócio – camada que contém os serviços e é responsável pela obtenção e manipulação de dados.

[22] Carregamento

A camada de *User Interface* constitui a interação entre o utilizador final e a componente gráfica da solução, sendo o foco de entrada para a sua utilização. Esta é representada pelos componentes *front-end*, cada um com a sua devida função e comportamento específicos.

Por fim, a camada de lógica de negócio é responsável pela definição de múltiplos serviços que têm comportamentos específicas às suas funções e objetivos de existência, bem como a manutenção de dados dinâmicos existindo uma interação com um sistema de gestão de base de dados. Nela está também incluído o componente Feathers, que é um módulo que intervém na definição do componente de API.

O diagrama da Figura 25 pretende auxiliar na perceção e interação que existe entre componentes participativos da solução.

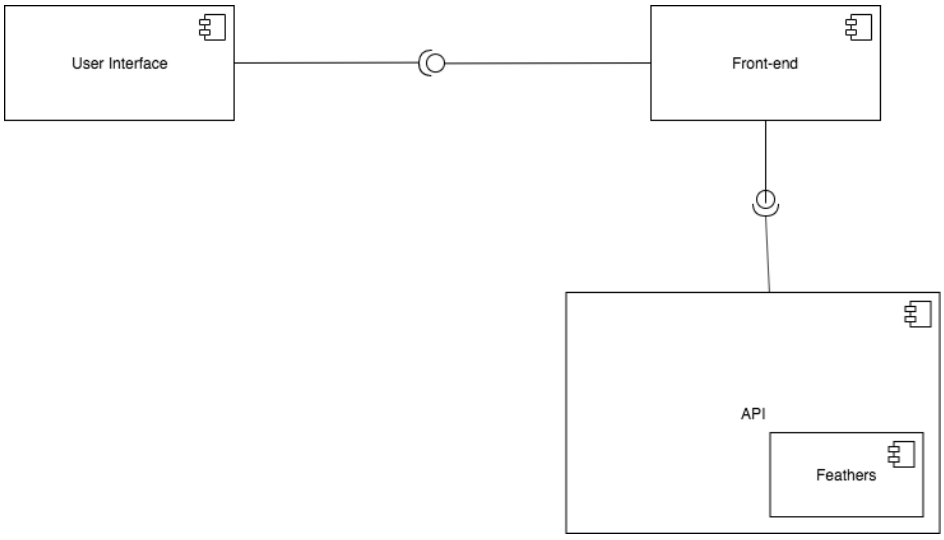

Figura 25 - Diagrama de componentes

5.1.4 Vista de processo

A vista de processo pretende dar a entender todo o *flow* que uma interação do utilizador provoca no sistema.

De forma a que exista um exemplo para compreender de alguma forma a estrutura da solução, foi criado um diagrama de sequência (Figura 26) para o UC02 -Visualizar em detalhe informações estatísticas sobre os resultados.

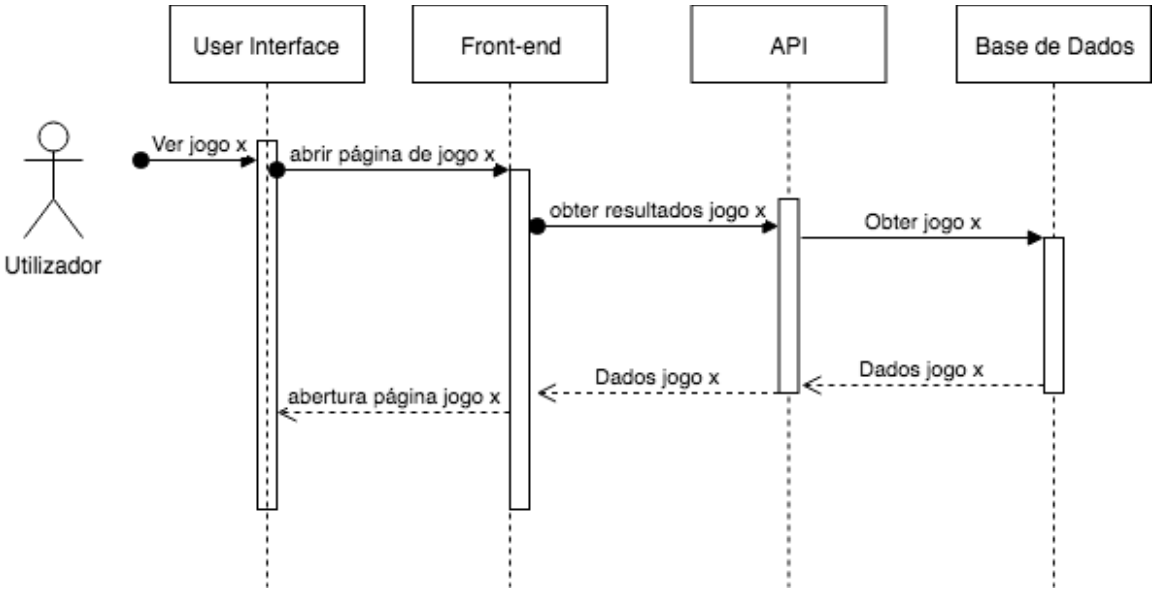

Figura 26 - Diagrama de sequência UC02

De acordo com o diagrama, a ação de abertura de uma página de resultados de um jogo desportivo faz com que exista uma interação da *User Interface* com o componente de *Front-End* de forma a que este desenhe essa página. No entanto, este componente tem de comunicar com a API em questão (REST ou *WebSocket*) para obter os dados em tempo real do jogo, bem como outros detalhes (equipas, equipamentos, estádio, etc.). Este componente API, localizado no *package* da camada de lógica de negócio faz uma pesquisa no componente de Base de Dados e devolve a informação sobre o jogo para o componente de *Front-End*, que por sua vez retorna ao utilizador uma página de resultados completamente atualizada.

A vista de implantação está no anexo A, visto não ser relevante no que diz respeito ao protótipo, por ser muito simples.

5.2 Implementação de requisitos

Esta secção contém a descrição completa do protótipo, com a devida implementação dos requisitos (funcionais e não funcionais) e serão apresentados, se relevantes, excertos de código de forma a explicar certos comportamentos.

Ao longo da implementação foi verificado que não existia nenhum benefício na implementação de casos de uso que foram identificados como funcionalmente semelhantes.

São considerados funcionalmente semelhantes os requisitos que tecnologicamente representam a mesma base de implementação, como é o caso da obtenção de resultados

desportivos em tempo real e a obtenção de *odds* para cada jogo que estivesse a decorrer. Ambos constituem um pedido de informação e obtenção de dados (independentemente da sua estrutura), e baseiam-se na implementação de um pedido HTTP *GET*.

Visto que a implementação do requisito extra não acrescentaria maior valor à experiência nem ajudaria nas conclusões finais para o âmbito deste documento, optou-se por não se proceder ao seu desenvolvimento.

Desta forma, só dois requisitos funcionais foram implementados na sua totalidade, visto que são cruciais para a análise das grandezas que foram referidas anteriormente e serviriam de caso de estudo para obtenção de possíveis conclusões.

De notar que todos os requisitos funcionais foram desenvolvidos tendo sido em conta os requisitos não funcionais (ver Subsecção 5.1.2).

Os padrões de desenvolvimento de *software* (Secção 4.1) foram também respeitados na medida em que as funcionalidades desenvolvidas estão contidas nos respetivos componentes (neste caso, classes) isolados, e que incluem uma única responsabilidade (por exemplo, a criação do perfil de utilizador está contida num componente específico à manipulação de entidades relativas a utilizadores). Desta forma a manutenção do *software* é facilitada, visto que todo o comportamento relativo a uma funcionalidade está isolado num componente, e não tem qualquer impacto noutros.

5.2.1 Obter resultados desportivos em tempo real

Na funcionalidade referente à obtenção de resultados desportivos em tempo real foi necessário definir um serviço *Feathers* (ver Subsecção 4.3.2), de forma a que a aplicação cliente possa requisitar os resultados desportivos em tempo real.

```
app.use('/livegames', {
  async get(id) {
    return { id, results: generateResults() };
  }
});
```

Código 3 - Serviço *Feathers* para obtenção de resultados desportivos

Este serviço implementa um método *GET*, no qual irá obter os resultados desportivos em tempo real. Este método é invocado através de um pedido AJAX[82] feito pelo *front-end*.

A lista de resultados desportivos resulta da criação de um algoritmo (Código 4) que foi elaborado e desenvolvido com a responsabilidade de geração de resultados de forma aleatória e dinâmica, para que estes estejam sempre disponíveis. Desta forma foi garantido que os resultados obtidos pelas experiências não sofrem impacto nem têm qualquer dependência de qualquer *third-party* (APIs externas, bases de dados, etc.).

Esta abordagem foi considerada fundamental no desenvolvimento, visto que um dos fatores que pode prejudicar a aferição de valores tais como o tempo de resposta de um determinado *endpoint* é precisamente a conexão a infraestruturas externas, que pode variar consoante o sistema em consideração. Desta forma foi possível descartar esta variável e ter a certeza que não existem falsas conclusões, pois pode-se afirmar que os testes são executados em ambientes controlados.

```
const generateResults = () => {
  let results = [];
  const teams = ['FC Porto', 'SL Benfica', 'Sporting CP', 'Liverpool FC',
 'Chelsea', 'Manchester United', 'Juventus', 'Ajax FC'];
  var arr1 = teams.slice(), // copy array
    arr2 = teams.slice(); // copy array again

    arr1.sort(function () { return 0.5 - Math.random(); }); // shuffle ar
rays
    arr2.sort(function () { return 0.5 - Math.random(); });

    while (arr1.length) {
      var name1 = arr1.pop(), // get the last value of arr1
        name2 = arr2[0] == name1 ? arr2.pop() : arr2.shift();

      const homeTeam = name1;
      const awayTeam = name2;
      const homeGoals = Math.floor(Math.random() * 3) + 1;
      const awayGoals = Math.floor(Math.random() * 3) + 1;

      const result = {
        homeTeam,
        awayTeam,
        homeGoals,
        awayGoals
      };

      results.push(result);

      arr1.splice(arr1.indexOf(name2), 1);
      arr2.splice(arr2.indexOf(name2), 1);

    }
  return results;
}
```

Código 4 - Algoritmo de geração de resultados desportivos

O algoritmo consiste numa função que dado um conjunto de equipas (*teams)*, cria duas estruturas de *arrays* e reordena-os de forma aleatória. De seguida obtém o último valor do primeiro *array* (sendo constituída a equipa da casa), e o primeiro do segundo *array*, caso não seja o mesmo valor obtido em primeiro lugar (equipa da casa). Caso seja efetivamente um valor repetido, este avança para a próxima posição do *array*. De seguida sorteia o número de golos para cada equipa, tendo um máximo de 3 golos por equipa (valor máximo arbitrário).

Por fim, ambos os elementos são removidos dos respetivos *arrays* para evitar jogos com equipas repetidas.

5.2.2 Criar perfil de utilizador

Para satisfação deste requisito funcional foi necessário definir um serviço *users* (Código 3), que é responsável pela manutenção e gestão de entidades relativas a utilizadores do sistema em questão. Este é criado segundo as regras definidas pelo *Feathers*, tal como foi explicado no requisito anterior.

```
app.use('/users, {
  async get(id) {
    return { id };
  },
  async create(data) {
    return data;
  }
});
```

Código 3 - Serviço Feathers para gestão de utilizadores

Como o requisito funcional especifica a criação de um utilizador é necessário definir, além do *get*, o método *create* que possibilita a inserção de um utilizador consoante os dados fornecidos (nome, morada, email, etc.).

Este método *create* consiste num pedido HTTP *POST*, que é utilizado pelas APIs para identificar uma criação/geração de um recurso/entidade, segundo a arquitetura REST[83].

Os vários perfis de utilizadores criados para efeitos de teste são gerados de forma aleatória utilizando uma biblioteca *JavaScript* externa – *Faker*[84]. Esta é capaz de produzir dados fictícios que caracterizam a identidade de uma pessoa.

Após configuração e definição dos vários *endpoints* e serviços, é necessária a construção de uma interface para que o utilizador possa invocar os mesmos e obter resultados.

5.3 User interface

A interface do protótipo é bastante simples e implementada de forma a que seja notório o objetivo do seu desenvolvimento – obtenção de *benchmarks* (explicado na Secção 5.4) e métricas relativas a várias funcionalidades, tendo sempre em consideração ambas as tecnologias que estão incluídas na experiência.

Foi desenvolvida utilizando HTML5 e *JavaScript*, com componentes base já previamente carregados (botões, *labels*, *etc*.)

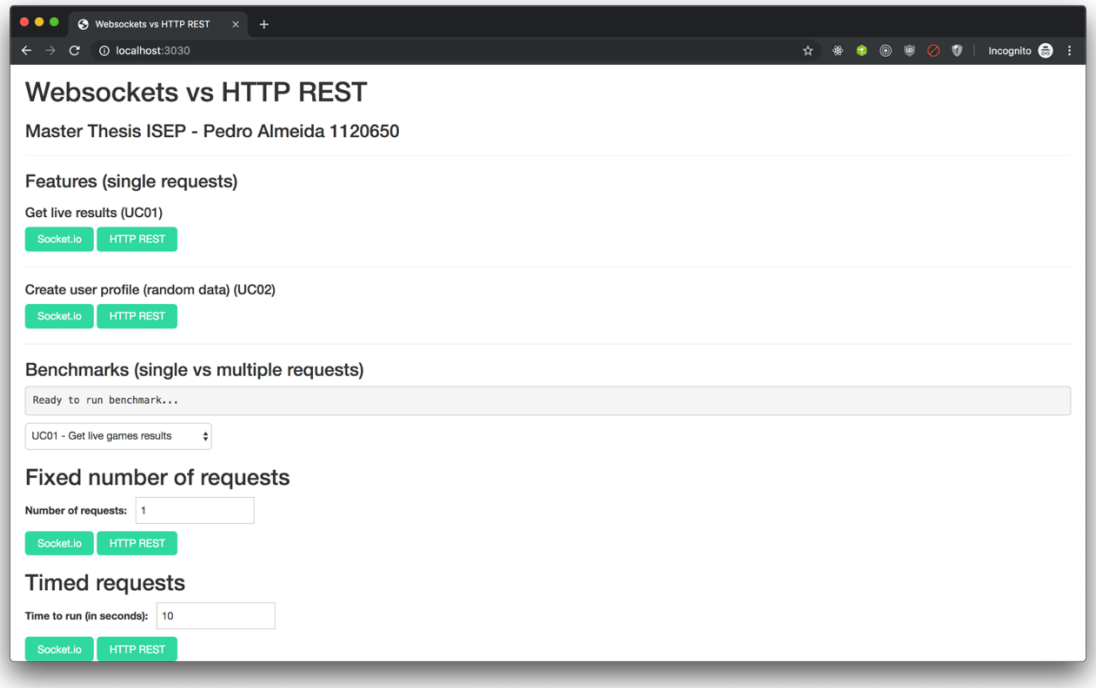

Figura 27 – Interface do protótipo desenvolvido

Como ilustrado na Figura 27, a interface consiste em duas secções principais – uma relativa à execução única de pedidos de forma a obter uma só resposta (consoante o tipo de API que queira utilizar – HTTP ou *WebSockets*), e outra secção destinada à execução de *benchmarks*, que por sua vez executam *x* pedidos em paralelo, ou *y* pedidos em *z* segundos, para um determinado requisito funcional.

Como resultado da invocação de qualquer serviço (ou conjunto deles) é apresentado numa *label*[23] em formato texto informação sobre o tempo de execução do mesmo.

[23] Caixa de texto

Tendo a interface pronta a ser utilizada, procedeu-se aos *benchmarks* sobre o protótipo desenvolvido, tendo sido recolhido dados sobre cada tentativa.

5.4 Benchmarks

Tal como foi apresentado na subsecção relativa à *user interface* (Subsecção 5.3), a aplicação inclui duas formas de operação – uma permite a execução de *x* pedidos em paralelo, e a outra permite a execução de uma série de pedidos sequenciais em *y* segundos, tendo em conta o tipo de conexão (HTTP ou *WebSocket*). Esta funcionalidade vai suportar a realização de teses de carga (ver Subsecção 4.2.1), sendo estes um fator importante na medição de desempenho de uma aplicação *web*.

Estes pedidos são caracterizados pelos respetivos requisitos funcionais, ou seja, é possível facultar informação sobre a obtenção de uma lista de resultados desportivos, bem como a criação de novos perfis de utilizadores. Esta possibilidade de incluir *benchmarks* resulta da necessidade de extração de valores e algumas métricas (tais como tempo de resposta, transferência de dados entre pedidos, etc.) para futura construção de conclusões (Capítulo 7).

O conceito de *benchmark* é muito utilizado no mundo tecnológico e consiste num teste (ou conjunto de testes) que é/são usados de forma a comparar desempenho entre uma aplicação e outra/s ou entre determinados critérios[85].

De forma a assegurar a execução de *x* pedidos em paralelo, bem como a execução de *y* pedidos em *z* segundos, foi necessária a criação de dois algoritmos que concretizassem a ideia.

A sequência de eventos que é comum às duas abordagens é a seguinte:

- Utilizador introduz o número de pedidos que quer executar em paralelo na caixa de *input* para o efeito e a funcionalidade sobre a qual pretende fazer o teste.
- O utilizador escolhe o protocolo para o efeito (HTTP ou *WebSocket*), consoante o botão de execução em que carregar.
- A aplicação deteta o botão que foi premido e, consoante o resultado, opta por executar um ou outro algoritmo desenvolvido.
- É apresentada a informação relativa à execução dos pedidos em questão na *label* destinada para o efeito

O seguinte diagrama de atividades (Figura 28) permite ter uma visão geral mais consistente relativamente ao comportamento esperado, resultante da ação do utilizador.

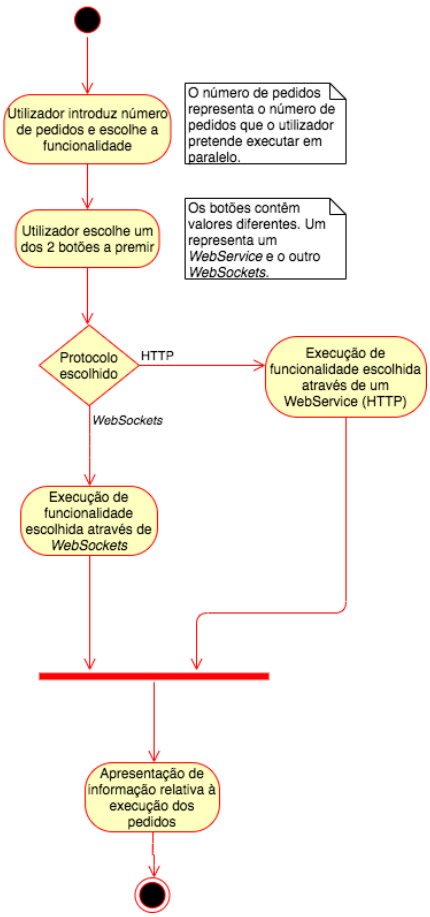

Figura 28 - Diagrama de atividades relativo à funcionalidade da aplicação

Tal como se pode observar através do diagrama de atividades da Figura 28, o protótipo inclui lógica de decisão para efetuar um tipo de pedido ou outro, consoante *input* do utilizador(ver Código 5). A escolha da funcionalidade é feita através da seleção de um valor numa caixa de seleção, sendo depois internamente (na aplicação) aferida qual o valor que o utilizador escolheu, procedendo de seguida à próxima ação - escolha entre executar os pedidos através de um *WebService* ou *WebSockets*.

5.4.1 *Desenvolvimento de algoritmos de* benchmark

Com o objetivo de conseguir obter resultados a partir das funcionalidades desenvolvidas, foram também implementados algoritmos que pudessem fornecer algumas métricas e valores estatísticos (tal como tempos de resposta, etc.). Na sequência do diagrama ilustrado na Figura 28, a aplicação possui lógica para execução dos algoritmos de *benchmarking* (Código 5). Após toda a interação necessário por parte do utilizador, é a vez da aplicação determinar qual a próxima ação.

```
if (request === 'count') {
    setStatus('Running request count benchmark...');

    const counter = parseInt(document.getElementById('request-count').value, 10);
    const { average } = liveGamesFeature ? await makeRequests(type, counter) : await
createUserProfile(type, counter);


    setStatus(`Making ${counter} ${types[type]} request(s) took ${average}ms`);
  }

  if(request === 'timed') {
    setStatus('Running timed request benchmark...');

    const time = parseInt(document.getElementById('request-timer').value, 10) * 1000;
    const counter = await runTimed(type, time, liveGamesFeature);

    setStatus(`Made ${counter} ${types[type]} requests within ${time / 1000} seconds`
);
  }
```

Código 5 - Lógica de decisão para determinada operação

A variável *request* representa uma *string* que provém do *input* que o utilizador inserir, relativamente ao tipo de operação. Existem dois tipos *input*, um para cada tipo de operação – execução de pedidos em paralelo, ou execução de pedidos durante um certo período de tempo.

Caso o utilizador escolha a operação de execução de pedidos em paralelo (*count*), é recolhido o número de pedidos (*counter*) que terão de ser feitos (valor que se encontra no *input* de texto para o efeito), e consoante a funcionalidade escolhida é executada uma ou outra. Se a funcionalidade for a obtenção da lista de resultados, a aplicação executará a função *makeRequests* (ver Código 6), recolhendo posteriormente o tempo de resposta a completar os *n* pedidos que o utilizador requisitou.

```
async function makeRequests (type = 'rest', times = 1) {
  const app = apps[type]();
  const promises = [];
  const start = new Date().getTime();

  for (let i = 0; i < times; i++) {
    promises.push(app.service('livegames').get('getlivegames'));
  }

  const result = await Promise.all(promises);

  return { average: new Date().getTime() - start, result };
}
```

Código 6 - Algoritmo de execução de *n* pedidos em paralelo

Esta função consiste na obtenção dinâmica do tipo de serviço que pretende executar (HTTP ou *WebSocket*), e consoante o número de pedidos que o utilizador introduziu, adiciona todos os *output* do *endpoint* de obtenção de lista de resultados em forma de *Promise*[86], auferindo no final um valor que será representativo do tempo de execução desde o começo do processo até ao fim (em que se obtém *output* de todos os pedidos).

Se o utilizador tiver escolhido a funcionalidade de criação de utilizador (um método POST), é executada a função *createUserProfile* (Código 7).

```
async function createUserProfile(type = 'rest', times = 1) {
  const app = apps[type]();
  const promises = [];
  const start = new Date().getTime();
  for (let i = 0; i < times; i++) {
    var name = faker.name.findName(); // Rowan Nikolaus
    var email = faker.internet.email(); // Kassandra.Haley@erich.biz
    var card = faker.helpers.createCard(); // random contact card containing many
properties
    const userProfile = { name, email, card };
    promises.push(app.service('users').create(userProfile));
  }
  const result = await Promise.all(promises);

  return { average: new Date().getTime() - start, result };
}
```

Código 7 - Código relativo à criação de perfil de utilizador

A função *createUserProfile* é semelhante à função mostrada anteriormente (Código 6) com o extra de gerar dados de utilizador para enviar para o serviço, com a ajuda da biblioteca externa *Faker*[84].

Por outro lado, se o utilizador escolher a opção de pedidos sequenciais (*timed*), a aplicação executará a função *runTimed* (ver Código 8).

Consoante o número de segundos e o tipo de protocolo que o utilizador introduziu, a função avalia o serviço a utilizar de forma dinâmica e executa um a um, incrementando um contador que representa o número de pedidos já satisfeitos com sucesso.

A forma como a aplicação consegue definir em tempo de execução qual protocolo utilizar resulta de um mapeamento de dados que relaciona o tipo com o módulo a ser utilizado – HTTP ou REST/HTTP.

Após explicada a lógica que se encontra atrás da execução dos vários tipos de operações (em sequência ou paralelo), podemos observar os resultados obtidos com as experiências feitas sobre estes algoritmos.

```
async function runTimed(type = 'rest', time = 10000, liveGamesFeature = true) {
  const app = apps[type]();

  let running = true;
  let counter = 0;

  setTimeout(() => (running = false), time);

  while (running) {
    if (liveGamesFeature) {
      await app.service('livegames').get('getlivegames');
      counter++;
    } else {
      var name = faker.name.findName(); // Rowan Nikolaus
      var email = faker.internet.email(); // Kassandra.Haley@erich.biz
      var card = faker.helpers.createCard(); // random contact card containing man
y properties
      const userProfile = { name, email, card };
      await app.service('users').create(userProfile);
      counter++;
    }
  }

  return counter;
}
```

Código 8 - Algoritmo para execução de pedidos sequenciais

5.5 Sumário

Este capítulo descreve todos os conceitos relacionados com a implementação do protótipo da presente dissertação, incluindo um aprofundamento dos requisitos funcionais que a englobam, algumas descrições da interface de utilizador.

6 Avaliação

Este capítulo tem como objetivo a avaliação da solução proposta. De forma a cumprir esse objetivo serão definidas as grandezas que serão utilizadas na avaliação, a metodologia de avaliação e ainda o teste de hipóteses que será usado. Por fim, são apresentados resultados de *benchmarking* feitos ao protótipo e que responderão à questão **Q2** e **Q3** (Secção 1.3).

6.1 Grandezas a avaliar

De forma a responder à questão final **Q3**, referida na Secção 1.3, e que se destina a responder à necessidade de encontrar uma tecnologia/protocolo que seja significativamente superior à outra é necessário a análise destes através da implementação de aplicações protótipo.

Para esse efeito foi definido um conjunto de grandezas a avaliar, derivando a partir das métricas definidas na Subsecção 4.2.1 e Subsecção 4.2.2:

- **Desempenho das conexões integradas numa aplicação *web***: esta grandeza é bastante importante visto que o desempenho e eficiência de uma aplicação *web* depende obrigatoriamente do desempenho das suas respetivas conexões ao servidor e a outras entidades externas, de serviços extra necessários. Este desempenho pode ser medido através de técnicas de *benchmarks*.
- **Escalabilidade da infraestrutura de comunicação**: esta grandeza é importante para a eficiência da aplicação *web,* visto que será necessário avaliar se esta será capaz de ser autossustentável a fim de suportar um determinado número de pedidos por segundo, mantendo sempre a consistência de dados e auferir a capacidade do servidor em aguentar tal carga.

6.2 Metodologia de avaliação

A metodologia de avaliação a ser utilizada para auferir se as grandezas de avaliação são satisfeitas será a utilização e verificação de métricas relativas a cada grandeza (especificada anteriormente):

6.2.1 Desempenho das conexões integradas numa aplicação web

Neste caso um bom desempenho das conexões integradas numa aplicação *web* pode representar a capacidade e velocidade de resposta que o servidor garante ao cliente, resultando consequentemente numa eficiência ao nível de utilizador final visto que a aplicação cliente será suficientemente fluída de forma a que este interaja sem qualquer problema ou entrave. Serão comparados os tempos de resposta, bem como a utilização da memória pelas aplicações cliente e servidora de ambos os protótipos.

6.2.2 Escalabilidade da infraestrutura de comunicação

Com o intuito de avaliar a escalabilidade de uma infraestrutura de comunicação é necessária a utilização de uma ferramenta que consiga simular a realização de vários pedidos consequentes às várias aplicações de forma a testar a sua sustentabilidade e a sua tolerância a falhas.

6.3 Teste de hipóteses

Um teste de hipóteses representa um teste estatístico que tem o objetivo de facilitar a toma de decisão entre duas ou mais hipóteses existentes.

No caso da presente dissertação pode-se considerar que $\gamma 0$ representa o protocolo de *WebSockets* e $\gamma 1$ representa o protocolo HTTP (utilizado em *WebServices*).

A hipótese 1 é a hipótese que se pretende rejeitar, visto que esta significa que ambos os protocolos são iguais relativamente às grandezas a avaliar.

$$H0 : \gamma 0 = \gamma 1 = 50\% \text{ (Hipótese 1)}$$

$$H1 : \gamma 0 \; != \gamma 1 \text{ (Hipótese 2)}$$

Caso isto não se verifique e existam diferenças entre os protocolos (hipótese 2), é importante avaliar ambas de maneira a que seja percetível a vantagem de utilização de um protocolo em detrimento do outro.

Caso a hipótese 3 se verifique, esta significa que o protocolo *WebSocket* é melhor nos termos avaliados, e caso se verifique a hipótese 4 o protocolo HTTP (utilizado em REST) será melhor.

$$H2 : \gamma 0 > \gamma 1 \text{ (Hipótese 3)}$$

$$H3 : \gamma 0 < \gamma 1 \text{ (Hipótese 4)}$$

6.4 Benchmarking – avaliação de grandezas

Nesta Secção serão apresentados os resultados das experiências às funcionalidades descritas na Secção 5.2, com base no que foi especificado na subsecção relativa aos *benchmarks* (4.2.2) e nas grandezas referidas na Secção 6.1.

De referir que todos os testes efetuados no *browser* não incluem o tempo que leva a estabelecer uma conexão via *WebSocket*, que é, em média, aproximadamente 190 milissegundos.

6.4.1 Resultados funcionais

Esta subsecção vem na medida de avaliação de uma das grandezas que foram referidas na Secção 6.1 – desempenho das conexões integradas numa aplicação *web*.

Todas as experimentações têm como base de análise dois grandes fatores essenciais no que diz respeito às aplicações *web* (Secção 4.2) e aos seus mecanismos de comunicação – tempo de resposta que um pedido demora até que termina a sua execução, e o tamanho de *payload* que cada pedido inclui, que pode por sua vez impactar o tempo de resposta.

De seguida irão ser descritos os testes feitos sobre estas duas matérias e os seus respetivos resultados, bem como uma breve explicação de conceitos. Estes foram analisados tendo sido utilizadas sobretudo as ferramentas que o *Google Chrome Developer Tools* providencia (Figura 29), pois esta permite avaliar o tempo de execução, bem como o seu *payload*.

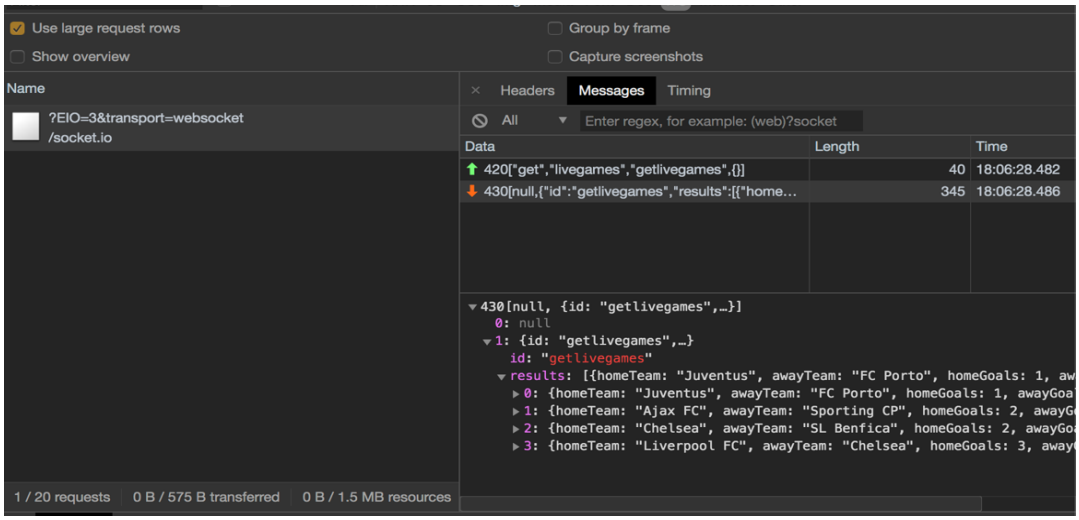

Figura 29 - *Google Chrome DevTools*

O primeiro teste (Figura 30) consiste na análise de tempo de resposta de um pedido GET feito no âmbito do requisito funcional relativo à obtenção de lista de resultados desportivos.

6.4.1.1 Tempo de resposta

Segundo a análise de resultados, em média um pedido HTTP demora cerca de 10.6 milissegundos, enquanto um pedido semelhante em *WebSocket* demora em média 3.4 milissegundos.

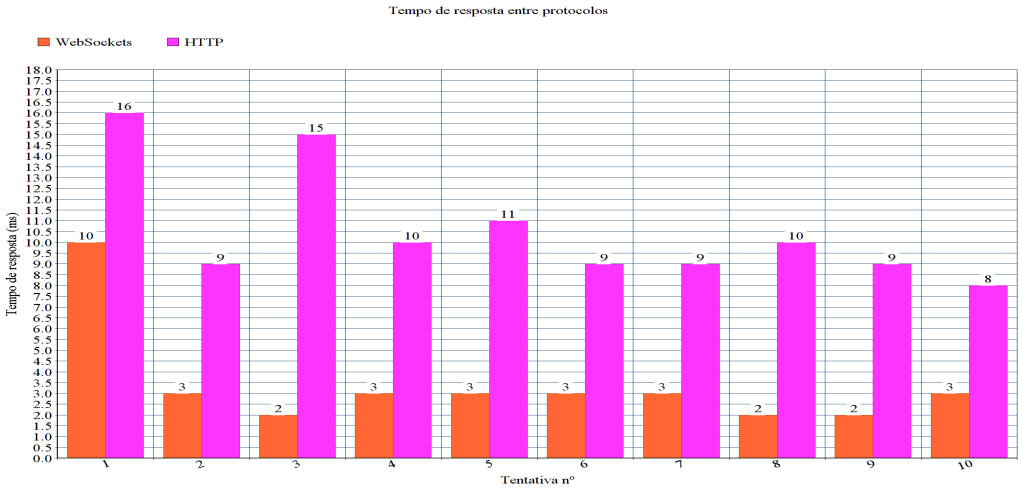

Figura 30 - Tempo de resposta de um pedido, consoante cada protocolo

Foram feitas 100 tentativas de forma a obter uma amostra assertiva e conclusiva sobre o tempo de resposta de cada protocolo, encontrando-se ilustradas no gráfico da Figura 30 apenas as dez primeiras amostras.

O cenário de teste relativo ao número de pedidos em paralelo (Figura 31) manteve-se diferente, com 60 pedidos a serem executados via *WebSocket* após uma média de 38.4 milissegundos, enquanto que via *WebService* (utilizando HTTP) o mesmo número de pedidos demorou cerca de 194.9 milissegundos.

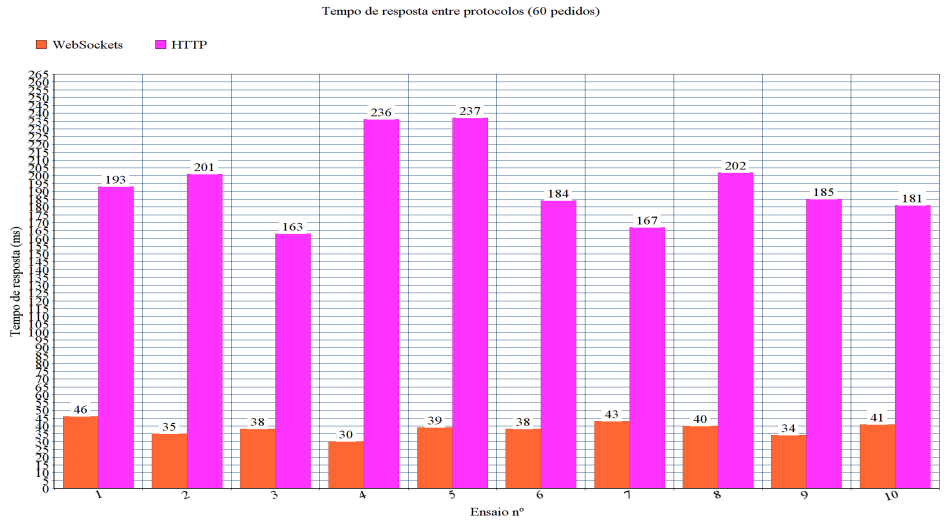

Figura 31 - Tempo de resposta após 60 pedidos em paralelo

Com estes dados pode-se afirmar que em geral o protocolo HTTP permitiu executar 307 pedidos por segundo, enquanto o protocolo *WebSocket* executava 1562 pedidos no mesmo período de tempo. Uma das razões que pode ter impacto nestes resultados é a grande diferença que exista entre a limitação do número de conexões HTTP concorrentes existentes. No *browser* de teste – Google Chrome – este limite é 6 para conexões HTTP[87], enquanto que não existe limite na quantidade de mensagens enviadas e recebidas através de uma conexão *WebSocket*.

Relativamente à funcionalidade de execução de pedidos em sequência foram também realizadas experiências e analisados os seus resultados, ilustrados na Figura 32.

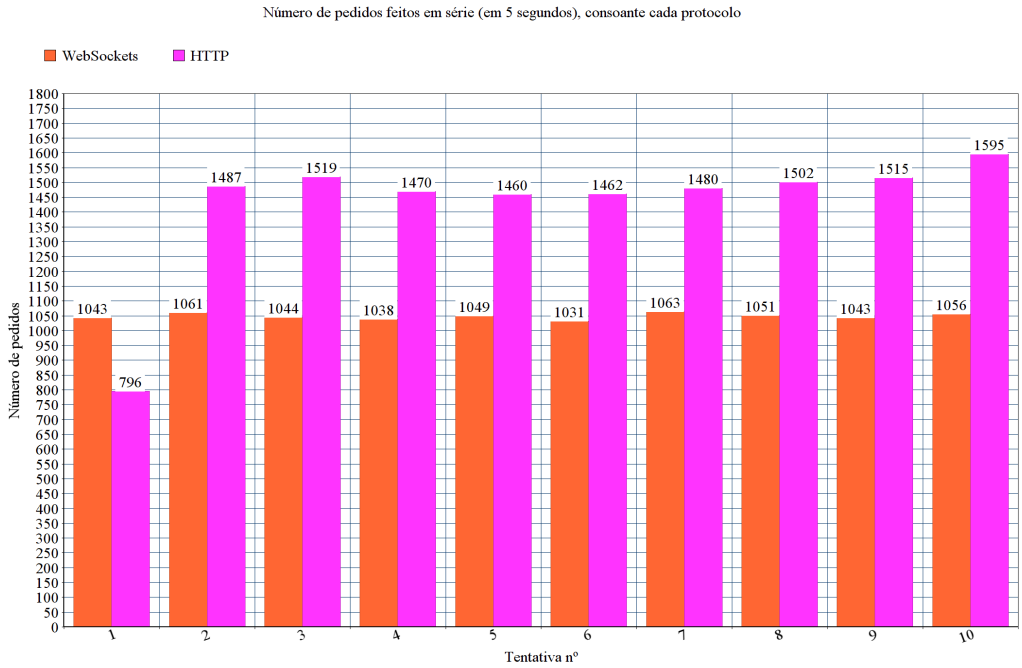

Figura 32 - Número de pedidos feitos em série (em 5 segundos)

Através da Figura 32 observa-se que, em média, são executados 1428 pedidos em série em 5 segundos utilizando o protocolo HTTP, enquanto que no mesmo tempo são executados em média 1047 pedidos através de comunicação via *WebSockets.*

Desta forma pode-se afirmar que o protocolo HTTP consegue realizar mais pedidos em série (em média 400 pedidos a mais), podendo afirmar que se trata do mecanismo de *cache* que este possui, ao invés do protocolo *WebSocket*.

Apesar de ter uma componente técnica diferente de implementação, os resultados auferidos na sequência dos mesmos testes sobre o requisito funcional de criação de perfil de utilizador foram semelhantes, pelo qual optou-se por não replicar demonstração de dados.

Com estes dados pode-se afirmar que uma API REST aumenta o seu tempo de processamento consoante o número de mensagens e isto deve-se ao facto de que as múltiplas conexões HTTP têm de ser iniciadas e terminadas consecutivamente, enquanto que a *WebSocket* API necessita de uma única conexão TCP para transmitir dados, fazendo-o de forma mais direta e eficiente.

6.4.1.2 Tamanho do *payload* relativo a pedidos

Tal como referido anteriormente, uma outra métrica interessante para ser tido em consideração é o *payload* que é transferido entre pedidos (4.2.2), dependendo do protocolo a ser utilizado.

O *payload* representa o conteúdo da resposta resultante de uma comunicação entre aplicação cliente e servidora e pode impactar o tempo de resposta de um pedido, visto que este tem de ser transferido pela aplicação cliente via rede. Ou seja, em teoria, quanto menos *payload* um pedido tiver na sua resposta, menos tempo demorará a que seja completa, visto que significa menos recursos para a aplicação cliente "consumir" através da rede.

Em termos teóricos, uma conexão *WebSocket* quando estabelecida não necessita de enviar os *headers* juntamente com a resposta, logo é expectável que o tamanho de dados transferidos por mensagem enviada seja menor quando comparada com um pedido HTTP.

Os *headers*[88] caracterizam-se por um conjunto de valores *key-value* (par de valores) e que permitem com que exista uma troca de informação extra entre pedidos HTTP. O único *header* que o protocolo *WebSocket* conhece atualmente é o *Sec-WebSocket-Protocol*, que é característico do mesmo.

Num único pedido são transferidos, em média, 354.8 Bytes através de comunicações em *WebSocket* enquanto são transferidos, em média, 584.7 Bytes de informação utilizando o protocolo HTTP. As várias experiências estão ilustradas na Figura 33.

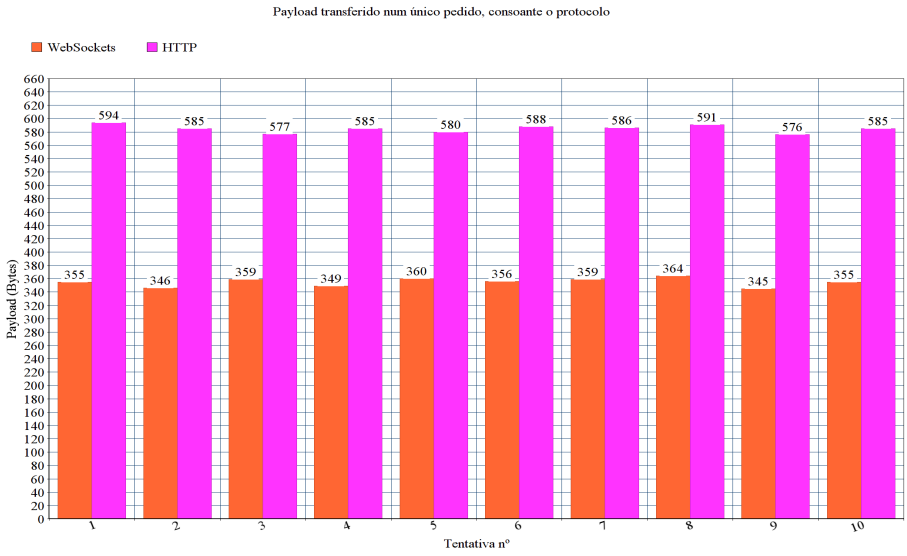

Figura 33 - *Payload* transferido num pedido através dos diferentes protocolos

6.4.1.3 Percentagem de erros

Tal como referido anteriormente, a percentagem de erros representa uma métrica importante na avaliação do desempenho de uma aplicação *web* (Subsecção 4.2.2). No entanto, no âmbito da experimentação do protótipo desenvolvido, não foi possível verificar a existência de erros durante os pedidos executados.

6.4.2 Escalabilidade

Esta subsecção vem na medida de avaliação de uma das grandezas que foram referidas na Secção 6.1 - escalabilidade da infraestrutura de comunicação.

A escalabilidade de uma aplicação é bastante importante na sua existência e manutenção (ver 4.2.1.4), visto que esta contribui para a competitividade, reputação e até qualidade do produto em questão[89].

Uma questão pertinente a destacar é que apesar de terem sido utilizados *WebSockets* no desenvolvimento do protótipo, a comunicação é feita tendo por base um contexto HTTP, visto que a interface que a ferramenta *Feathers* providencia consiste na invocação de uma *RESTful* API. Relativamente à escalabilidade, uma das vantagens da utilização da arquitetura REST é o facto de esta ser considerado *statelessness*, ou seja, qualquer servidor consegue responder a qualquer pedido e não existe necessidade de sincronização de qualquer estado partilhado (além da própria base de dados, caso se aplique).

A ferramenta *Feathers* utiliza o mesmo conceito para as suas conexões *WebSocket*, sendo que a única informação enviada através do *socket*[24] é o perfil do utilizador, de forma a que o sistema decida que eventos em tempo real enviar. Desta forma é possível concluir que desde que estes eventos estejam sincronizados entre múltiplas instâncias, a arquitetura híbrida é escalável por diversos servidores, à semelhança da infraestrutura em HTTP que é utilizada atualmente.

Visto que as experiências relativas a uma só instância (*browser*) já foram detalhadas na Subsecção 6.4.1, falta testar o comportamento do protótipo quando submetido a várias instâncias.

[24] Canal de comunicação utilizado em *WebSockets*

6.4.2.1 Resultados de testes de carga

Na sequência dos testes de carga efetuados sobre o protótipo desenvolvido, foi crucial a utilização de várias bibliotecas externas que pudessem atuar como agentes ativos, de forma a simularem a existência de múltiplas instâncias.

Para efeitos de *benchmark* relativos a pedidos HTTP foi utilizada uma biblioteca externa denominada *Autocannon*[90], enquanto que para *WebSockets* foi utilizada uma outra denominada de *ws-benchmark*[91]. Ambas incluem comandos que permitem configurar o número de pedidos, número de clientes (concorrentes ou não), etc. De referir que as configurações consideradas são iguais para a execução de ambas bibliotecas, para que se obtenham resultados num ambiente de teste o mais semelhante possível entre as duas opções.

Os resultados dos testes de carga foram configurados para 1, 10 e 100 conexões concorrentes, respetivamente, e estão representados num gráfico (Figura 34) em que se pode observar e comparar o número de pedidos por conexão em função do número de pedidos por segundo.

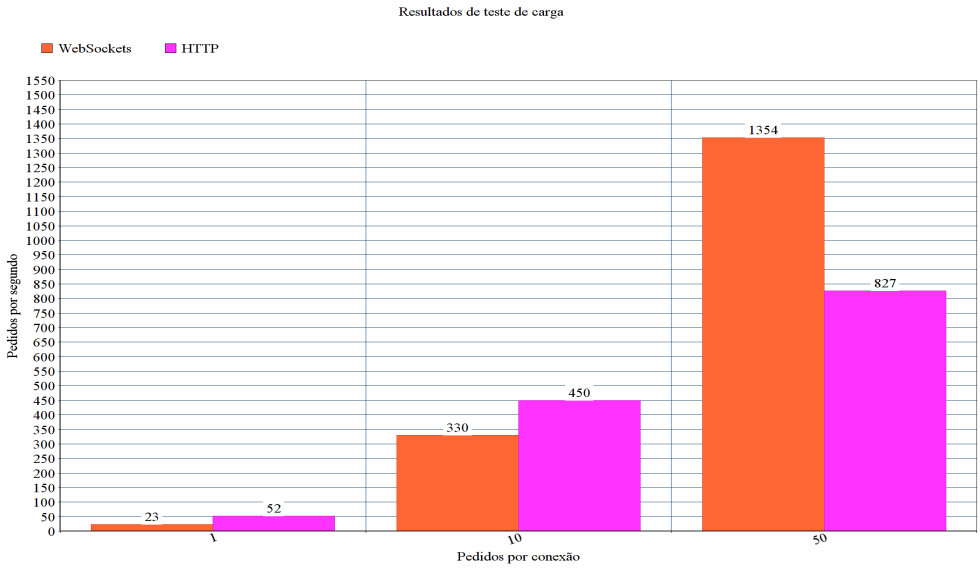

Figura 34 - Resultados teste de carga, com múltiplas instâncias

Como se pode observar no gráfico acima, a execução de um pedido por conexão é cerca de 50% mais lento ao utilizar *WebSockets*, visto que a conexão tem de ser estabelecida primeiro. Este fator é notoriamente visível na secção referente aos 10 pedidos. Porém, na secção relativa aos 50 pedidos utilizando a mesma conexão, a comunicação via *WebSockets* é aproximadamente 50% mais rápida do que a comunicação via HTTP.

De forma a obter resultados mais coerentes e confirmar este princípio, foram executados os mesmos tipos de testes, mas com números de pedidos por conexão superiores aos anteriormente configurados – 500, 1000 e 2000 – ver Figura 35.

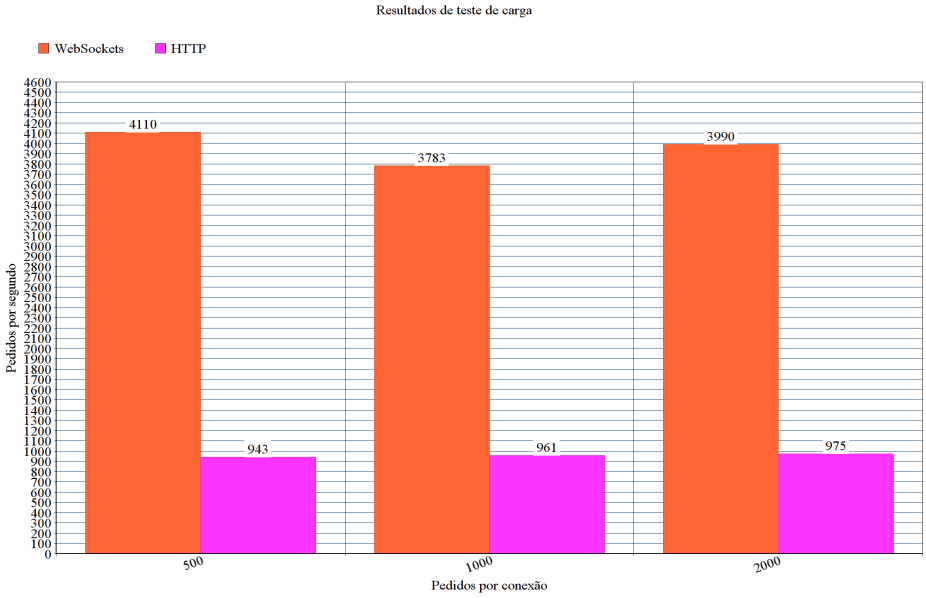

Figura 35 - Resultados teste de carga, com múltiplas instâncias em números superiores

Neste último gráfico observa-se que a comunicação via HTTP tem um pico máximo de aproximadamente 959 pedidos por segundo, enquanto que através de *WebSockets* o pico máximo atingido é em média de 3961 pedidos por segundo.

Estes valores vêm comprovar que para um grande conjunto de pedidos por conexão, o protocolo *WebSockets* é mais eficiente pois consegue executar mais pedidos por segundo do que HTTP.

Segundo o teste de hipóteses definido na Secção 6.3, avalia-se que a hipótese que se verifica com base nos resultados descritos nas subsecções anteriores é a hipótese 3, que significa uma melhoria de características dos *WebSockets* relativamente aos *WebServices* (HTTP).

6.5 Sumário

Neste capítulo foram descritas as grandezas avaliar na solução, bem como a metodologia da sua avaliação e ainda o teste de hipóteses de forma a obter uma conclusão. Por fim, apresentaram-se os resultados de *benchmarking* que resultaram da implementação dos

mesmos (Subsecção 5.4.1) e derivam do estudo feito relativamente a métricas e testes ao desempenho de aplicações *web* (4.2). Estes resultados são importantes no suporte à construção da resposta às questões **Q2** e **Q3** (Secção 1.3), referindo-se ao impacto dos *WebSockets* no mundo das aplicações *web*.

7 Conclusões

Este capítulo contém uma síntese do contexto e do problema, bem como conclusões os objetivos alcançados na presente dissertação, consoante os resultados obtidos na experiência. Estas conclusões serão respondidas de acordo com as questões pertinentes que foram definidas na Secção 1.3.

7.1 Síntese

Nesta secção é apresentada uma síntese do contexto e do problema a que a dissertação pretende responder.

O desenvolvimento de aplicações *web* tem vindo a crescer cada vez mais no mundo da tecnologia, e é considerado hoje em dia[92] como uma das maiores oportunidades de negócio do mundo. Dito isto, é necessária que as aplicações desenvolvidas sejam o mais eficientes possíveis, tendo em conta os vários requisitos funcionais e não funcionais comuns aos bons princípios do desenvolvimento de *software* (tendo como exemplo alguns dos descritos na Subsecção 5.1.2).

Alguns dos principais fatores de desempenho presentes numa aplicação *web* caracterizam-se pela eficiência, segurança e desempenho dos canais de comunicação que existem entre os vários componentes que constituem um sistema *web*, sejam eles o cliente e o servidor. Ambos têm de estar preparados para assegurarem o envio e receção de informação entre eles, de forma a que a aplicação tenha o resultado esperado pelo utilizador final, tolerante a qualquer erro.

Atualmente existem vários protocolos de comunicação no mundo das aplicações *web*, sendo os mais dominantes o HTTP e *WebSockets*. Ambos têm ideais parecidos, visto que a ideia do protocolo *WebSockets* é proveniente de um *upgrade*[25] ao protocolo HTTP, embora o primeiro seja mais focado em ser utilizado em sistemas que necessitem de atualização de informação em tempo real, enquanto o último é mais polivalente na sua utilização. É considerado que o protocolo *WebSockets* e o HTTP estejam em níveis semelhantes no modelo TCP/IP (2.2.1), visto que o primeiro resulta de uma implementação do protocolo TCP (2.2.3).

Atualmente a maioria das aplicações são implementadas utilizando o protocolo HTTP, visto que é o melhor conceituado e praticado desde há algum tempo. No entanto, as vantagens sobre a

[25] melhoria

utilização de novos protocolos também têm surgido e têm sido bastante consideradas na altura de implementar uma nova aplicação *web*.

Tal como referido anteriormente, o problema desta dissertação assenta em perceber o impacto do protocolo *WebSocket* nas aplicações *web*, tendo em conta a escolha da implementação de um protocolo em detrimento do outro, com base nas suas características, vantagens e desvantagens, contextos em que se aplicam, *etc*.

Para decidir qual dos protocolos optar na implementação de uma nova aplicação *web*, decidiu-se comparar ambas as alternativas em termos de desempenho, escalabilidade e complexidade (Subsecção 3.7).

7.2 Objetivos alcançados

Nesta secção serão apresentadas as respostas às questões levantadas na Secção 1.3, bem como uma justificação às mesmas.

7.2.1 Questão Q1

A questão Q1 prende-se com o facto de decidir que *benchmarks* devem ser implementados de forma a obter uma boa avaliação final entre cada protocolo. Através da descrição da Subsecção 4.2.2, consegue-se concluir que existem várias métricas que podem ser consideradas quando feitos testes de desempenho sobre uma aplicação *web*. Os mais importantes e/ou cruciais que determinam uma avaliação completa (no contexto do problema) sobre o desempenho são: o tempo de resposta médio por pedido, o tamanho do *payload* e ainda a percentagem de erros em função do número de pedidos.

7.2.2 Questão Q2

A questão Q2 refere a dúvida quanto ao protocolo de *WebSockets* apresentar melhor desempenho, escalabilidade e confiabilidade, em conjunto com a aplicação desenvolvida para o efeito, com base nas grandezas e metodologias definidas no Capítulo 6.

De forma a avaliar isto analisaram-se sobretudo duas grandezas e/ou métricas para obter uma melhor resposta à pergunta em questão. Estas grandezas caracterizam-se pelo tempo de resposta entre pedidos (consoante o seu protocolo), tendo chegado aos resultados apresentados na Secção 5.4.

Estes dados não permitem obter uma resposta clara à pergunta, visto que ambos os protocolos têm resultados semelhantes e as vantagens na utilização de um protocolo refletem-se nas desvantagens da utilização do outro. Estas vantagens são algumas que podem ser destacadas pelos resultados obtidos na Subsecção 6.4.1.

WebSockets – vantagens:

1. Melhor desempenho relativos a pedidos únicos
    a. Menor tempo de resposta resultante da execução de um pedido feito através da rede
    b. Menor transferência de recursos/informação via rede
2. Melhor capacidade de *overload* de carga, e por isso, melhor capacidade de escalabilidade (Subsecção 6.4.2)

HTTP/REST – vantagens:

1. Apesar de maior transferência de dados sobre a rede, é possível enviar informação extra no pedido, como por exemplo, *cookies*, ou outra informação sobre o utilizador.
2. Os sistemas de proxy, DNS e firewalls (que fazem parte da arquitetura de cliente-servidor – Subsecção 1.1.1) ainda não têm total conhecimento e inteligência suficiente para lidar com tráfego WebSocket, o que pode ser um problema, visto que existe a necessidade de configuração das firewalls. Isto não é um problema no HTTP.
3. O protocolo HTTP possui algumas características que o *WebSocket* não possui tais como:
    a. *Caching*[93] – técnica de armazenamento de cópias de um determinado recurso de forma a poupar *bandwidth*.
    b. *Routing*[94] – processo de seleção de um caminho para tráfego numa rede
    c. *Gzipping*[95] – formato de compressão de dados

Relativamente ao desempenho, apesar da semelhança entre os dois protocolos, na maioria das amostras recolhidas observou-se que o protocolo *WebSocket* tem efetivamente melhor desempenho, visto que, em termos gerais, tem menores tempos de resposta e menores quantidades de transferências de dados que a aplicação cliente tem a fazer, tal como demonstrado na Subsecção 6.4.1.2.

Quanto à escalabilidade e confiabilidade, esta questão foi respondida na Subsecção 6.4.2, tendo sido demonstrado que ambos os protocolos teriam o mesmo nível de escalabilidade e confiabilidade, se configurados de forma apropriada, e se utilizando ferramentas que consigam auxiliar nesse sentido, como é o exemplo do *Feathers*.

7.2.3 Questão Final Q3

A questão Q3 é a questão fulcral da presente dissertação. Tem como objetivo decidir qual o protocolo a ser utilizado nas aplicações *web* desenvolvidas no futuro.

Tal como referido no Capítulo 3, as três grandezas fundamentais para a tomada de decisão eram a escalabilidade, a complexidade e o desempenho.

Relativamente à escalabilidade, foi possível provar que é possível que ambos os protocolos sejam suficientemente capazes de serem escaláveis, se configurados para tal. A capacidade de resposta é semelhante entre os dois protocolos e não irrompe em nenhuma falha em situações de grandes números de clientes e/ou utilizadores.

O tópico de complexidade é subjetivo, visto que ambos os protocolos se complementam um ao outro, e conseguem ser facilmente configurados, se for encontrada a ferramenta correta. *Feathers* parece ajudar nesse sentido visto que a sua documentação é bastante completa e permite iniciar o *Setup* de um sistema híbrido em questão de minutos. Não existe complexidade associada à definição de vários serviços que podem ser utilizados em ambos os protocolos (*WebSockets* e HTTP).

A nível de desempenho foi comprovado que o protocolo *WebSockets* tem relativamente melhor desempenho no sentido de ser mais rápido a responder aos pedidos que são feitos ao servidor, gastando também menos recursos de rede (*bandwidth*).

Em suma, a resposta à questão passa por incluir os diferentes protocolos de comunicação (uma solução híbrida), caso haja necessidade, e ser suficientemente capaz de habilitar e/ou desabilitar um ou outro sem que exista a obrigação de alterar a lógica aplicacional. Por exemplo, provavelmente não faz sentido incluir *WebSockets* na criação de utilizadores, mas pode fazer sentido se existirem dados que padeçam de informação em tempo real (como por exemplo as apostas online). Visto que através da presente dissertação podemos afirmar que é possível, uma combinação dos dois mundos faz com que a aplicação seja dinâmica o suficiente para conseguir fazer uma gestão do que utilizar, e quando o fazer. Para isso, é altamente aconselhada a utilização de ferramentas que utilizem tais mecanismos, de forma a que a eficiência e manutenção da aplicação *web* sejam asseguradas. O protótipo desenvolvido no âmbito desta dissertação vem demonstrar que tal é possível, pois existem circunstâncias em que se podem usar um dos dois protocolos, sem que haja qualquer impacto no desempenho da aplicação.

7.3 Limitações e trabalho futuro

A principal limitação no desenvolvimento da dissertação foi a não identificação de funcionalidades que seriam pertinentes para o âmbito do documento, o que fez com que houvesse algum tempo perdido na implementação de funcionalidades que não serviram um propósito. Com isto, uma das principais funcionalidades que seria interessante de incluir neste estudo- desenvolvimento de um *chat* - não foi concluída na totalidade, logo não foi contabilizada nas experiências. Porém, toda a base de um *chat* consiste em vários pedidos GET e POST, os quais foram incluídos no leque de cenários de teste, apesar de em contextos diferentes.

Num possível trabalho futuro seria interessante desenvolver as funcionalidades que faltaram incluir no estudo, pois iria também ser alvo de dados que poderiam possivelmente levar à alteração das conclusões apresentadas. Aumentar os testes efetuados ao código desenvolvido de forma a cobrir todos os algoritmos de *benchmark* seria também um fator importante na manutenção do projeto. Alargar o tipo de testes efetuado no protótipo (bem como incluir mais métricas) seria também um fator benéfico na avaliação do desempenho do protocolo *WebSocket*. Por fim, de forma a obter um ambiente de teste mais realista, a hospedagem da solução num servidor remoto também seria algo a considerar para comprovar todos os valores obtidos nas experiências descritas na Secção 5.4.

7.4 Apreciação Final

Os objetivos que foram mencionados no início da dissertação foram atingidos, tendo chegado a uma conclusão que representa um consenso, e que pode ajudar na decisão da implementação do protocolo *WebSocket* numa aplicação *web*.

No desenvolvimento do protótipo verificou-se que existe uma grande comunidade com muita informação online sobre as características técnicas, resoluções de muitos problemas originados, e até formas de contornar mecanismos que que ainda não existe em *WebSockets*, mas sim em HTTP.

De referir que o desenvolvimento da presente dissertação teve por base as boas práticas de Engenharia de *Software*. Os módulos que foram lecionados no contexto da unidade curricular de Tese de Mestrado de Engenharia de *Software*, especificamente a pesquisa e escrita técnico-científica, análise de valor de negócio contribuíram para o bom desenvolvimento da dissertação e enriquecimento pessoal.

Referências

[1] "Web Application Architechture – Client / Server Architecture | WEB ENGINEERING." [Online]. Available: https://webingineer.wordpress.com/2010/07/24/web-application-architechture-client-server-architecture/. [Accessed: 08-Oct-2019].

[2] "Introduction to Monolithic Architecture and MicroServices Architecture." [Online]. Available: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63. [Accessed: 08-Oct-2019].

[3] "Difference Between API and Web Service - anwar samer - Medium." [Online]. Available: https://medium.com/@programmerasi/difference-between-api-and-web-service-73c873573c9d. [Accessed: 11-Oct-2019].

[4] "What Is an API and Why Should I Use One? – Tyler Elliot Bettilyon – Medium." [Online]. Available: https://medium.com/@TebbaVonMathenstien/what-is-an-api-and-why-should-i-use-one-863c3365726b. [Accessed: 17-Feb-2019].

[5] "World Wide Web Consortium (W3C)." [Online]. Available: https://www.w3.org/. [Accessed: 23-Feb-2019].

[6] Kenchiku Setsubi Iji Hozen Suishin Kyōkai. and 建築設備維持保全推進協会, *Gaiheki no jishin ni taisuru anzensei no hyōka hōhō dō kaisetsu.* Kenchiku Setsubi Iji Hozen Suishin Kyōkai, 2004.

[7] "Which API Types and Architectural Styles are Most Used? | ProgrammableWeb." [Online]. Available: https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26. [Accessed: 17-Feb-2019].

[8] "Website Design Industry Statistics - The Ultimate Collection." [Online]. Available: https://techjury.net/stats-about/website-design-industry/. [Accessed: 12-Oct-2019].

[9] "Digital trends 2019: Every single stat you need to know about the internet." [Online]. Available: https://thenextweb.com/contributors/2019/01/30/digital-trends-2019-every-single-stat-you-need-to-know-about-the-internet/. [Accessed: 12-Oct-2019].

[10] "HTTP/1.1: Status Code Definitions." [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. [Accessed: 23-Feb-2019].

[11] "What is the Internet Protocol? | Cloudflare." [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/internet-protocol/. [Accessed: 23-Feb-2019].

[12] "The TCP/IP Model and Protocol Suite Explained for Beginners." [Online]. Available: http://www.steves-internet-guide.com/internet-protocol-suite-explained/. [Accessed: 08-Oct-2019].

[13] S. R. Singh and S. Kumar, "An Overview of World Wide Web Protocol (Hypertext Transfer Protocol and Hypertext Transfer Protocol Secure)," 2016.

[14] "An Introduction to API's – The RESTful Web." [Online]. Available: https://restful.io/an-introduction-to-api-s-cee90581ca1b. [Accessed: 17-Feb-2019].

[15] L. Masinter, T. Berners-Lee, and R. T. Fielding, “Uniform Resource Identifier (URI): Generic Syntax.”

[16] “MIME types - HTTP | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types. [Accessed: 23-Feb-2019].

[17] “HTTP request methods - HTTP | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods. [Accessed: 23-Feb-2019].

[18] “Pull vs Push Technology - Simplicable.” [Online]. Available: https://simplicable.com/new/pull-vs-push-technology. [Accessed: 11-Oct-2019].

[19] E. Estep, “Mobile HTML5: Efficiency and Performance of WebSockets and Server-Sent Events,” p. 109, 2013.

[20] “Short Polling to HTTP Server with Underlying SQL Database (not to scale) | Download Scientific Diagram.” [Online]. Available: https://www.researchgate.net/figure/Short-Polling-to-HTTP-Server-with-Underlying-SQL-Database-not-to-scale_fig1_311289465. [Accessed: 11-Oct-2019].

[21] “Stream Updates with Server-Sent Events - HTML5 Rocks.” [Online]. Available: https://www.html5rocks.com/en/tutorials/eventsource/basics/. [Accessed: 23-Feb-2019].

[22] “Long Polling to HTTP Server with Underlying SQL Database (not to scale) | Download Scientific Diagram.” [Online]. Available: https://www.researchgate.net/figure/Long-Polling-to-HTTP-Server-with-Underlying-SQL-Database-not-to-scale_fig2_311289465. [Accessed: 11-Oct-2019].

[23] P. Lubbers and F. Greco, “HTML5 Websockets: A Quantum Leap in Scalability for the Web,” *SOA World Mag.*, 2010.

[24] G. Sadasivan, J. Brownlee, B. Claise, and J. Quittek, *Architecture for IP flow information export*. RFC Editor.

[25] “What is the Asynchronous Web, and How is it Revolutionary?” [Online]. Available: https://www.theserverside.com/news/1363576/What-is-the-Asynchronous-Web-and-How-is-it-Revolutionary. [Accessed: 11-Oct-2019].

[26] “TCP/IP Overview - Cisco.” [Online]. Available: https://www.cisco.com/c/en/us/support/docs/ip/routing-information-protocol-rip/13769-5.html. [Accessed: 17-Feb-2019].

[27] “What is an Acknowledgement Code (ACK)? - Definition from Techopedia.” [Online]. Available: https://www.techopedia.com/definition/781/acknowledgement-code-ack. [Accessed: 23-Feb-2019].

[28] R. Braden, “Requirements for Internet Hosts - Communication Layers.”

[29] “The WebSocket API (WebSockets) - Web APIs | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Accessed: 17-Feb-2019].

[30] “Protocol upgrade mechanism - HTTP | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Protocol_upgrade_mechanism. [Accessed: 23-Feb-2019].

[31] “OWASP Top 10 Details About WebSocket Vulnerabilities and Mitigations - SecureLayer7.” [Online]. Available: https://blog.securelayer7.net/owasp-top-10-details-websocket-vulnerabilities-mitigations/. [Accessed: 05-Sep-2019].

[32] “#1 Rated Customer Relationship Management Software | SugarCRM.” [Online]. Available: https://www.sugarcrm.com/. [Accessed: 23-Feb-2019].

[33] “Mobile Data Plan Sharing API | Google Developers.” [Online]. Available: https://developers.google.com/mobile-data-plan/. [Accessed: 23-Feb-2019].

[34] “Web Audio API.” [Online]. Available: https://www.w3.org/TR/webaudio/. [Accessed: 23-Feb-2019].

[35] “Web Audio API - Web APIs | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API. [Accessed: 23-Feb-2019].

[36] “What is Remote Procedure Call (RPC)? - Definition from WhatIs.com.” [Online]. Available: https://searchmicroservices.techtarget.com/definition/Remote-Procedure-Call-RPC. [Accessed: 23-Feb-2019].

[37] “SOAP Specifications.” [Online]. Available: https://www.w3.org/TR/soap/. [Accessed: 23-Feb-2019].

[38] “GraphQL | A query language for your API.” [Online]. Available: https://graphql.org/. [Accessed: 23-Feb-2019].

[39] “What are Web APIs - By.” [Online]. Available: https://hackernoon.com/what-are-web-apis-c74053fa4072. [Accessed: 12-Oct-2019].

[40] “What is an API Endpoint? | SmartBear Software Resources.” [Online]. Available: https://smartbear.com/learn/performance-monitoring/api-endpoints/. [Accessed: 12-Oct-2019].

[41] “Netscape.” [Online]. Available: https://isp.netscape.com/. [Accessed: 23-Feb-2019].

[42] “JavaScript Definition.” [Online]. Available: https://techterms.com/definition/javascript. [Accessed: 17-Feb-2019].

[43] “java.com: Java + You.” [Online]. Available: https://www.java.com/en/. [Accessed: 23-Feb-2019].

[44] “Pros and Cons of JavaScript - Weigh them and Choose wisely! - DataFlair.” [Online]. Available: https://data-flair.training/blogs/advantages-disadvantages-javascript/. [Accessed: 05-Sep-2019].

[45] “Node.js Introduction.” [Online]. Available: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm. [Accessed: 17-Feb-2019].

[46] “V8 JavaScript engine.” [Online]. Available: https://v8.dev/. [Accessed: 23-Feb-2019].

[47] “The MIT License | Open Source Initiative.” [Online]. Available: https://opensource.org/licenses/MIT. [Accessed: 23-Feb-2019].

[48] “Socket.IO — Docs | Socket.IO.” [Online]. Available: https://socket.io/docs/. [Accessed: 17-Feb-2019].

[49] “The three main steps of the innovation process (According to Koen et... | Download Scientific Diagram.” [Online]. Available: https://www.researchgate.net/figure/The-three-main-steps-of-the-innovation-process-According-to-Koen-et-al-2002-p-6-and_fig9_303025021. [Accessed: 24-Feb-2019].

[50] “The New Concept Development (NCD) Model according to Koen et al. [1]... | Download Scientific Diagram.” [Online]. Available: https://www.researchgate.net/figure/The-New-Concept-Development-NCD-Model-according-to-Koen-et-al-1-Image-reproduced-by_fig3_281199416. [Accessed: 17-Feb-2019].

[51] “The New Concept Development (NCD) Model according to Koen et al. [1]... | Download Scientific Diagram.” [Online]. Available: https://www.researchgate.net/figure/The-New-Concept-Development-NCD-Model-according-to-Koen-et-al-1-Image-reproduced-by_fig3_281199416. [Accessed: 24-Feb-2019].

[52] “Total number of Websites - Internet Live Stats.” [Online]. Available: http://www.internetlivestats.com/total-number-of-websites/. [Accessed: 17-Feb-2019].

[53] “Verna Allee | KMOL.” [Online]. Available: https://kmol.pt/entrevistas/2001/04/01/verna-allee/. [Accessed: 23-Feb-2019].

[54] “Using the analytic hierarchy process (ahp) to select and prioritize projects in a portfolio.” [Online]. Available: https://www.pmi.org/learning/library/analytic-hierarchy-process-prioritize-projects-6608. [Accessed: 24-Feb-2019].

[55] “Saaty scale for factor ranking Figure 1 describes the different factor... | Download Scientific Diagram.” [Online]. Available: https://www.researchgate.net/figure/Saaty-scale-for-factor-ranking-Figure-1-describes-the-different-factor-ranking-based-on_fig5_299837810. [Accessed: 11-Oct-2019].

[56] “AHP calculator - AHP-OS.” [Online]. Available: https://bpmsg.com/academic/ahp_calc.php?n=3&t=AHP+priorities&c[0]=Desempenho&c[1]=Escalabilidade&c[2]=Complexidade. [Accessed: 24-Feb-2019].

[57] “HTTP vs Websockets: A performance comparison – The Feathers Flightpath.” [Online]. Available: https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77. [Accessed: 24-Feb-2019].

[58] “Software Engineering | Architectural Design - GeeksforGeeks.” [Online]. Available: https://www.geeksforgeeks.org/software-engineering-architectural-design/. [Accessed: 24-Feb-2019].

[59] “4. Process Modeling.” [Online]. Available: https://www.itl.nist.gov/div898/handbook/pmd/pmd.htm. [Accessed: 24-Feb-2019].

[60] “5 principles to follow when designing the architecture of your application.” [Online]. Available: https://niteco.com/blogs/5-principles-for-software-architecture/. [Accessed: 24-Feb-2019].

[61] “The Genius of the Law of Demeter - DZone Java.” [Online]. Available: https://dzone.com/articles/the-genius-of-the-law-of-demeter. [Accessed: 24-Feb-2019].

[62] “P of EAA: Service Layer.” [Online]. Available: https://martinfowler.com/eaaCatalog/serviceLayer.html. [Accessed: 24-Feb-2019].

[63] “Understanding quality of service for Web services.” [Online]. Available: https://www.ibm.com/developerworks/library/ws-quality/index.html. [Accessed: 12-Oct-2019].

[64] “What is bottleneck in performance testing? - Quora.” [Online]. Available: https://www.quora.com/What-is-bottleneck-in-performance-testing. [Accessed: 12-Oct-2019].

[65] “Chapter 2 – Types of Performance Testing | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924357(v=pandp.10)?redirectedfrom=MSDN. [Accessed: 12-Oct-2019].

[66] “What is an SLA? Best practices for service-level agreements | CIO.” [Online]. Available: https://www.cio.com/article/2438284/outsourcing-sla-definitions-and-solutions.html. [Accessed: 12-Oct-2019].

[67] “Web service performance testing - tips and tools for getting started : Assertible.” [Online]. Available: https://assertible.com/blog/web-service-performance-testing-tips-and-tools-for-getting-started. [Accessed: 12-Oct-2019].

[68] “8 Key Application Performance Metrics & How to Measure Them.” [Online]. Available: https://stackify.com/application-performance-metrics/. [Accessed: 12-Oct-2019].

[69] “Server performance metrics: 8 you should be considering · Raygun Blog.” [Online]. Available: https://raygun.com/blog/server-performance-metrics/. [Accessed: 12-Oct-2019].

[70] “New Industry Benchmarks for Mobile Page Speed - Think With Google.” [Online]. Available: https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/. [Accessed: 12-Oct-2019].

[71] “What is a Central Processing Unit (CPU)? - Definition from Techopedia.” [Online]. Available: https://www.techopedia.com/definition/2851/central-processing-unit-cpu. [Accessed: 12-Oct-2019].

[72] “AWS Auto Scaling.” [Online]. Available: https://aws.amazon.com/autoscaling/. [Accessed: 12-Oct-2019].

[73] “Monitoring the health of your application - The upgraded "/ping" route 🏓 - Soham’s blog.” [Online]. Available: https://www.sohamkamani.com/blog/architecture/2018-09-06-application-health-monitoring/. [Accessed: 12-Oct-2019].

[74] “Visual Studio Code - Code Editing. Redefined.” [Online]. Available: https://code.visualstudio.com/. [Accessed: 06-Oct-2019].

[75] “Why Use Node.js? A Comprehensive Tutorial with Examples | Toptal.” [Online]. Available: https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js. [Accessed: 04-Oct-2019].

[76] “Why the Hell Would You Use Node.js - Node.js Collection - Medium.” [Online]. Available: https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e. [Accessed: 06-Oct-2019].

[77] “Feathers | A framework for real-time applications and REST APIs.” [Online]. Available: https://feathersjs.com/. [Accessed: 21-Sep-2019].

[78] “What are Microservices? | API Basics | SmartBear.” [Online]. Available: https://smartbear.com/solutions/microservices/. [Accessed: 06-Oct-2019].

[79] “Sports Betting and Gambling Market/Industry - Statistics & Facts | Statista.” [Online]. Available: https://www.statista.com/topics/1740/sports-betting/. [Accessed: 12-Oct-2019].

[80] “5 ways to build real-time apps with JavaScript – freeCodeCamp.org.” [Online]. Available: https://medium.freecodecamp.org/5-ways-to-build-real-time-apps-with-javascript-5f4d8fe259f7. [Accessed: 24-Feb-2019].

[81] “web services - What is the maximum time a web application (or website) should respond to a request? - Stack Overflow.” [Online]. Available: https://stackoverflow.com/questions/17138585/what-is-the-maximum-time-a-web-application-or-website-should-respond-to-a-requ. [Accessed: 13-Oct-2019].

[82] “Getting Started - Developer guides | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started. [Accessed: 06-Oct-2019].

[83] “HTTP Methods – REST API Verbs – REST API Tutorial.” [Online]. Available: https://restfulapi.net/http-methods/. [Accessed: 06-Oct-2019].

[84] “Marak/faker.js: generate massive amounts of realistic fake data in Node.js and the browser.” [Online]. Available: https://github.com/marak/Faker.js/. [Accessed: 04-Oct-2019].

[85] “What Is a Benchmark? (Definition of Benchmark).” [Online]. Available: https://www.lifewire.com/what-is-a-benchmark-2625811. [Accessed: 04-Oct-2019].

[86] “Promise - JavaScript | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Accessed: 06-Oct-2019].

[87] “Blocking: The Not-So-Hidden Web Performance Villain.” [Online]. Available: https://blog.bluetriangle.com/blocking-web-performance-villain. [Accessed: 05-Oct-2019].

[88] “HTTP headers - HTTP | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers. [Accessed: 04-Oct-2019].

[89] “The Definition and Significance of Scalability.” [Online]. Available: https://www.touchsupport.com/what-is-scalability-and-why-does-it-matter-to-your-business/. [Accessed: 06-Oct-2019].

[90] “mcollina/autocannon: fast HTTP/1.1 benchmarking tool written in Node.js.” [Online]. Available: https://github.com/mcollina/autocannon. [Accessed: 06-Oct-2019].

[91] “ws-benchmark - npm.” [Online]. Available: https://www.npmjs.com/package/ws-benchmark. [Accessed: 06-Oct-2019].

[92] “Internet Statistics & Facts (Including Mobile) for 2019 - HostingFacts.com.” [Online]. Available: https://hostingfacts.com/internet-facts-stats/. [Accessed: 06-Oct-2019].

[93] “What is Caching and How it Works | AWS.” [Online]. Available: https://aws.amazon.com/caching/. [Accessed: 23-Feb-2019].

[94] “What is Routing.” [Online]. Available: https://www.networkkings.org/what-is-routing/. [Accessed: 23-Feb-2019].

[95] “What is Gzip Compression | Enabling Gzip Commands | CDN Guide | Incapsula.” [Online].

Available: https://www.incapsula.com/cdn-guide/glossary/gzip.html. [Accessed: 23-Feb-2019].

[96] “What is MIME (Multi-Purpose Internet Mail Extensions) - Interserver Tips.” [Online]. Available: https://www.interserver.net/tips/kb/mime-multi-purpose-internet-mail-extensions/. [Accessed: 08-Oct-2019].

Anexos

A Vista de implantação

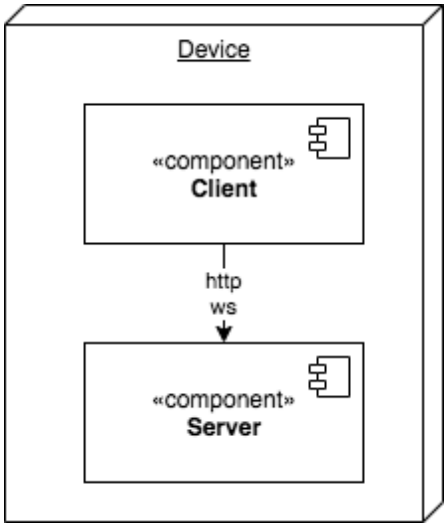

Figura 36 - Diagrama de implantação

O diagrama de implantação (Figura 36) é de fácil compreensão, visto que ambos os componentes cliente e servidor estão hospedados na máquina local e comunicam entre si através de HTTP e/ou *ws* (*WebSockets).*