FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Informação Clínica em Tempo Real

António Joaquim Ribeiro Garcez

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Prof. João Correia Lopes

28 de julho de 2015

Informação Clínica em Tempo Real

António Joaquim Ribeiro Garcez

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Prof. Cristina Ribeiro
Arguente: Prof. Benedita Malheiro
Vogal: Prof. João Correia Lopes
28 de julho de 2015

Resumo

A *Web*, nos seus primeiros tempos, era basicamente constituída por páginas *Web* cujo conteúdo era estático (algumas páginas *Web* recorriam a *software* externo para alterar o conteúdo). Contudo, com o processo de migração das aplicações, até aqui, *stand-alone* para a *Web* houve a necessidade de desenvolver técnicas e tecnologias que, nativamente, conseguissem alterar o conteúdo da página *Web* sem ter de recarregar manualmente e, assim, poder mostrar informação em tempo real.

Devido à facilidade com que é implementada e aos excelentes resultados que conseguem obter, as empresas começaram a utilizar *polling* para atualizar a informação nas aplicações *Web* que desenvolviam. O *polling* é uma técnica que, graças a um grande número de pedidos HTTP feitos pelo cliente num pequeno intervalo de tempo, permite dar a ideia ao utilizador de que é o servidor que está a informar o cliente sobre as atualizações que estão a acontecer. O aumento de utilizadores e a fiabilidade da informação exigida em algumas aplicações *Web* fizeram com que as limitações do *polling* ficassem expostas. Os principais problemas associados ao *polling* estão relacionados com uma diminuição do desempenho do servidor, com o aumento no tráfego da rede e com problemas no *rendering* da informação nas aplicações *Web*. Estas duas limitações podem fazer com que a informação chegue ao cliente com atraso significativo ou, em casos mais extremos, que acabe por não chegar ao cliente. Numa área tão sensível como é a área da saúde, não pode haver falhas de informação, uma vez que pode induzir em erro os profissionais de saúde, acabando por comprometer a saúde dos doentes internados.

Para este trabalho, o primeiro objetivo passa pela elaboração de uma pesquisa sobre todas as técnicas que permitam a uma aplicação *Web* mostrar informação em tempo real. A partir da pesquisa efetuada e analisando as vantagens e desvantagens das várias técnicas, o segundo objetivo consiste na escolha de uma técnica que resolva os problemas encontrados. O terceiro passa, com base na técnica escolhida, por implementar um módulo que permita às aplicações *Web* mostrar informação em tempo real de uma forma mais eficaz e eficiente, sem que com isso o sistema atual sofra grandes alterações. Por fim, o último objetivo passa pela adaptação de uma aplicação *Web*, previamente desenvolvida, que vai permitir avaliar e testar o módulo implementado.

Com base nas técnicas pesquisadas e analisando as vantagens e as desvantagens de cada uma, chegou-se à conclusão que a melhor técnica seria a tecnologia WebSocket. Para que as aplicações *Web* consigam receber as atualizações da informação através de WebSockets é necessário registar as alterações da informação. Como tal não existe, o registo das alterações é feito na base de dados que, após uma alteração, envia a indicação do que foi alterado para o módulo. Posteriormente, o módulo acede à API para ir buscar informação mais detalhada sobre a indicação recebida e envia a informação, através de WebSockets, para as aplicações *Web* interessadas em receber esse tipo de informação. Os testes efetuados têm por base duas métricas: a latência e o tráfego gerado na transmissão da informação. Com base nestes testes, pode-se observar que os WebSockets necessitam de menos tempo e de menos *bytes* para atualizar a informação das aplicações *Web*. Isto vem comprovar que é possível manter a informação atualizada minimizando a carga dos servidores, da rede e dos clientes.

Abstract

The web, in its early days, was mostly comprised of pages whose content was static, although the content of some web pages was dynamically generated by external tools to the web server. Moreover, the page content was updated only when the user reloaded the web page. However, in the process of migration of standalone applications to the web, there was the need to develop techniques and technologies that, originally, could change the web page content without having to reload it manually and thus showing real time information.

Because of the ease with which it is implemented and the excellent results it obtains, the companies, in the most diverse areas, began using polling to update the information in the web applications they developed. Polling is a technique that, thanks to a large number of HTTP requests made by the client in a short time, gives the idea to the user that is the server that informs the client about the updates that are happening. The increase in users and the reliability of information required in some web applications made the polling limitations being exposed. The main problems associated to the use of polling are connected to a decrease in the server performance and the increase in the network traffic. These two limitations may lead to the information reaches the client with significant delay or, in extreme cases, end up not reaching the client. In an area as sensitive as is the area of health, there cannot be information gaps because it can mislead health professionals, which can end up compromising the health of hospitalized patients.

For this work, the first goal is to research all the techniques that allow a web application showing real-time information. Starting with the research carried out and analyzing the advantages and disadvantages of the various techniques, the second goal consists in choosing a technique to solve the problems encountered. Taking for base the chosen technology, the third objective is to implement one module that will allow web applications show real time information in a more effective and efficient way without making lots of changes in the current system. Finally, the last goal is the adaptation of an existing web application to be used to evaluate and test the deployed module.

Based on the research techniques and analyzing the advantages and disadvantages of each one, it has been concluded that the best technique would be the technology WebSocket. In order to the web applications to receive the information updates trough the WebSockets it is necessary to have a way of recording the changes of information. As such doesn't exist, the changes are recorded in the data base which, after a change is made, sends the indication that something has changed to the module. Lately, the module access API to collect more detailed data about the received indication and sends the information, trough WebSockets, to the web applications interested in receiving such kind of information. The worked tests are based on two metrics: the latency and traffic generated in the transmission of information. Based in the tests we can see that the WebSockets need less time and less bytes to update the information from web applications. This proves it is possible to keep the information updated without overloading the servers, the network and the clients.

Agradecimentos

Em primeiro lugar, queria agradecer ao meu pai Joaquim e à minha mãe Albina pelo apoio incondicional que me deram e pelos muitos sacrifícios que fizeram ao longo destes últimos anos para que conseguisse chegar aonde cheguei.

Queria agradecer ao Professor João Correia Lopes por ter aceitado ser o meu orientador, pelo apoio que prestou durante a dissertação e pela sua prontidão em resolver os problemas que foram surgindo e em responder aos e-mails que ia enviando.

Queria agradecer ao Engenheiro Pedro Rocha e ao Engenheiro Francisco Correia pela disponibilidade que sempre tiveram em responder às minhas dúvidas mesmo quando ia "chateá-los" umas cinco ou seis vezes por dia.

Queria agradecer à minha namorada, Ana Deus, pelo apoio e pela força que me deu ao longo deste último semestre mesmo que, e peço desculpa por isso, muitas das vezes não tenha respondido aos teus SMS por estar a trabalhar.

Queria agradecer ao Fábio Santos pelo seu apoio, pela sua preocupação e pelas sugestões dadas para que o meu trabalho corresse da melhor forma possível.

Queria deixar um agradecimento muito especial à equipa do SIG por me acolherem durante o tempo em que estive a desenvolver esta dissertação, pelo apoio dado e pelos bons momentos que faziam com que o trabalho corresse melhor.

Queria agradecer à Professora Eulália Barbosa pela disponibilidade em verificar se os textos que escrevi estavam de acordo com as regras da língua portuguesa.

Queria agradecer a toda a minha família que diretamente, ou indiretamente, me apoiou ao longo destes últimos anos.

Por fim, queria deixar um agradecimento muitíssimo especial aos meus verdadeiros amigos que me apoiaram, que me aturaram e que ouviram os meus problemas porque sem a vossa ajuda teria sido muito mais difícil chegar aonde cheguei. E, já agora, peço-vos desculpa pelo facto de a minha retribuição ao vosso apoio ser umas caras esquisitas que eu costumo fazer mas, enfim, é a vida!

A todos vocês, muito obrigado por tudo!

António Joaquim Ribeiro Garcez

"The Web as I envisaged it, we have not seen it yet.
The future is still so much bigger than the past."

Tim Berners-Lee

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
BOSH	Bidirectional-streams Over Synchronous HTTP
CGI	Common Gateway Interface
CPU	Central Processing Unit
ERP	Enterprise Resource Planning
Glintt HS	Glintt Healthcare Solutions
HTML	HyperText Markup Protocol
HTTP	HyperText Transfer Protocol
HTTPS	HTTP over TLS
I/O	Input/Output
IETF	Internet Engineering Task Force
IP	Internet Protocol
IT	Information Technology
JSON	JavaScript Object Notation
NTP	Network Time Protocol
RFC	Request For Comments
RSS	Rich Site Summary
TCP	Transmission Control Protocol
TLS	Transport Layer Security
SCALA	Sistema de Comunicação Alternativa para o Letreamento de pessoas com Autismo
SIP	Session Initiation Protocol
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
STOMP	Streaming Text Oriented Messaging Protocol
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
TNS	Transparent Network Substrate
TTR	Time To Refresh
UDP	User Datagram Protocol
URL	Uniform Resource Locator
UTF	Unicode Transformation Format
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XHR	XMLHttpRequest

Capítulo 1

Introdução

Este primeiro capítulo está dividido em quatro secções. A primeira secção explica o enquadramento desta dissertação. Uma vez que esta dissertação foi feita num ambiente empresarial, a segunda secção consiste numa breve apresentação da empresa onde estive a trabalhar neste último semestre. A terceira dá a conhecer a motivação e os objetivos que estão por detrás desta tese. E, por fim, a última secção faz um breve da estrutura deste documento.

1.1 Enquadramento

A *World Wide Web*, vulgarmente conhecida por *Web*, foi criada há cerca de 20 anos e pode ser vista como um sistema que, através da Internet, permite aceder a uma grande quantidade de informação [Edi15].

Nos primeiros anos de vida, a *Web* era constituída por páginas estáticas, isto é, o conteúdo da página só era atualizado quando a página *Web* fosse recarregada – ação que tem de ser desencadeada pelo próprio utilizador. Contudo, nos meados da década passada, vários investigadores começaram a questionar a possibilidade de migrar aplicações, até aqui, *stand-alone* – tais como, jogos, sistemas de mensagens instantâneas (*chat*), ferramentas de trabalho colaborativas, etc. – para os navegadores *Web* [HÏ1].

Na teoria, esta migração traria imensas vantagens tanto para os programadores como para os utilizadores, como por exemplo [MT11]:

- A aplicação *Web* desenvolvida, a partir do momento em que é colocada na *Web*, pode ser acedida por qualquer utilizador;
- Não é necessária a existência de um manual de instruções;
- Os utilizadores não têm que ter a preocupação de manter a aplicação atualizada visto que se presume que a versão disponível é a mais recente – atualmente, as aplicações *Web* elaboradas

com recurso a *HyperText Markup Protocol* versão 5 (HTML5) podem ser utilizadas em modo *offline*;

- As aplicações *Web* podem ser executadas em qualquer sistema operativo desde que este tenha um navegador *Web* instalado;
- Graças às aplicações *Web*, é possível aceder a uma enorme quantidade de dados. A *Web* permite que haja uma maior partilha de dados comparativamente a outros sistemas.

Contudo, verificou-se que, na prática, a migração não seria assim tão simples como se poderia pensar, uma vez que grande parte das aplicações *stand-alone* eram capazes de atualizar a informação em tempo real, o que obrigava a que o conteúdo das páginas *Web* fosse dinâmico e não estático como vinha a acontecer até ao momento. O problema é que, nativamente, a *Web* não conseguia alterar o conteúdo de uma página *Web* – algumas páginas recorriam a *software* externo, como por exemplo, a *Common Gateway Interface* (CGI), a Java Servlets ou a Active Server Pages para alterarem o conteúdo [GLG11].

Devido ao facto de, nativamente, a *Web* não conseguir alterar o conteúdo de uma página *Web*, vários programadores *Web* tentaram desenvolver, recorrendo a técnicas de comunicação assíncrona, uma forma de contornar esta limitação. Até que em 2005, surgiu o *Asynchronous JavaScript and XML* (AJAX) que veio revolucionar a *Web* uma vez que, pela primeira vez, era possível mostrar conteúdo dinâmico sem recurso a *software* externo. Graças à XMLHttpRequest API, o AJAX deu a possibilidade aos navegadores *Web* de criarem pedidos *HyperText Transfer Protocol* (HTTP) para acederem à informação guardada no servidor sem que a página *Web* tenha de ser recarregada. A partir do momento em que se tornou possível alterar, de forma dinâmica, o conteúdo das páginas *Web*, percebeu-se que era possível, com recurso à técnica de *polling*, a criação de aplicações *Web* que mostrassem informação em tempo real [HÏ1].

Apesar de ser uma técnica bastante eficaz, o *polling* apenas permite simular o processo de atualização, uma vez que, graças ao elevado número de pedidos feitos num curto espaço de tempo, consegue captar todas as atualizações que ocorram e, assim, fazer com que pareça que foi o servidor a informar o cliente sobre a existência de novos dados [ZS13]. Apesar de ser uma técnica que apenas simula o processo de atualização, o *polling* é uma técnica muito utilizada visto que, na maioria dos casos, obtém bons resultados, é fácil de implementar e é compatível com a grande maioria dos navegadores *Web*. Todavia, o aumento do número de aplicações *Web* que utilizam o *polling* como forma de atualizar o seu conteúdo e o aumento do número de utilizadores que acedem às aplicações fizeram com que surgissem problemas relacionados com o desempenho dos servidores e com o tráfego da rede.

Com o aparecimento dos problemas relacionados com a utilização do *polling*, houve a necessidade de desenvolver outras técnicas como por exemplo, o *long polling*, o *streaming*, o *Bidirectional-streams Over Synchronous HTTP* (BOSH), o PubSubHubbub e os WebSockets que permitem atualizar o conteúdo de uma aplicação *Web* em tempo real sem afetar tanto a rede como o servidor.

1.2 Contexto

Atualmente, as aplicações *Web* são utilizadas nas mais diversas áreas. Na área da saúde, uma das entidades que mais se destaca no desenvolvimento de aplicações *Web* é a empresa onde desenvolvi esta dissertação, a Glintt Healthcare Solutions (Glintt HS). A Glintt HS é uma empresa, integrada no grupo Glintt, que conta com mais de 20 anos de experiência no setor da saúde. A Glintt HS dedica-se à prestação de serviços na área dos *Information Technology* (IT), tendo como área principal o desenvolvimento de projetos na área da gestão de saúde.

Atualmente, a Glintt HS marca presença em mais de 200 hospitais e clínicas que contam com projetos relacionados desde a:

- Gestão do doente;
- Gestão do circuito do medicamento;
- Gestão de meios complementares de diagnóstico e terapêutica;
- Processo clínico eletrónico;
- Robótica hospitalar;
- Prescrição eletrónica do medicamento;
- Implementação integrada de *Enterprise Resource Planning* (ERP);
- Manutenção de ativos;
- Gestão de recursos humanos;
- Captura e gestão documental;
- Reconhecimento de voz.

1.3 Motivação e Objetivos

A Glintt HS é uma empresa que desenvolve aplicações *Web* que auxiliam os profissionais de saúde em hospitais e clínicas de todo o país. Com as aplicações *Web* desenvolvidas pela Glintt HS, os profissionais de saúde têm acesso à informação sobre o estado dos doentes internados, o número de doentes internados em cada serviço, a medicação que cada doente tem de tomar, etc.

A informação disponibilizada, através das aplicações *Web*, tem de ser atualizada em tempo real para que não induza os profissionais de saúde em erro, uma vez que numa área tão sensível como a da saúde, um pequeno erro na informação disponibilizada pode levar a que o estado de saúde de um doente piore ou, em casos extremos, cause a morte do doente.

O primeiro objetivo deste trabalho é a elaboração de uma pesquisa sobre todas as técnicas que permitam a uma aplicação *Web* mostrar informação em tempo real. A partir da pesquisa efetuada e

analisando as vantagens e desvantagens das várias técnicas, o segundo objetivo passa por escolher a técnica que melhor resolve os problemas encontrados. O terceiro passa, com base na técnica escolhida, por implementar um módulo que permita aos sistemas Glintt mostrar informação em tempo real de uma forma mais eficaz e eficiente, sem que com isso o sistema atual sofra grandes alterações. Por fim, o último objetivo passa pela adaptação de uma aplicação *Web*, previamente desenvolvida, que vai permitir avaliar e testar o módulo implementado.

1.4 Estrutura da Dissertação

Para além da introdução, este relatório contém mais 6 capítulos. No Capítulo 2, são apresentadas um conjunto de técnicas, com as respetivas vantagens e desvantagens, que permitem a uma aplicação *Web* mostrar informação em tempo real. No Capítulo 3 é apresentada uma descrição mais pormenorizada do problema, a arquitetura das aplicações *Web* da Glintt HS e a forma com que o *polling* é feito. O Capítulo 4 vai apresentar a solução que vai permitir solucionar o problema descrito no Capítulo 3, a nova arquitetura das aplicações da Glintt HS e as tecnologias e a técnica que vão ser utilizadas na solução. O Capítulo 5 irá servir para explicar de que forma é que a solução, descrita no Capítulo 4, vai ser implementada. O Capítulo 6 vai apresentar os testes que irão permitir validar, ou não, a solução implementada. Por fim, o Capítulo 7 vai apresentar as conclusões do trabalho e as principais recomendações para trabalho futuro.

Capítulo 2

Informação em Tempo Real nas Aplicações Web

O segundo capítulo mostra um conjunto de técnicas que podem ser utilizadas para mostrar informação em tempo real. Este capítulo é constituído por 8 secções. A primeira secção dá a conhecer um pouco da história da *Web* e a forma como ela, na perspetiva da apresentação da informação em tempo real, evoluiu até aos dias de hoje. Da segunda até à sétima secção são explicadas um conjunto de técnicas, tais como o *polling*, o *long polling*, o *streaming* ou os WebSockets, que permitem mostrar informação em tempo real. Por fim, a última secção é um breve resumo do que foi escrito nas primeiras sete secções.

2.1 Introdução

Foi há pouco mais de 20 anos que a *World Wide Web*, vulgarmente conhecida por *Web*, surgiu. O conceito de *Web* assenta no modelo cliente-servidor [BL90]. No modelo cliente-servidor, o cliente faz um pedido ao servidor, onde está armazenada a informação, e este devolve ao cliente a informação pretendida (mais à frente, este processo vai ser explicado com mais detalhe). A *Web*, ao assentar no modelo cliente-servidor, permite que:

- A informação esteja num único local;
- A manutenção se torne mais fácil;
- Se dê suporte a vários clientes simultaneamente;
- A gestão e segurança da informação seja feita com mais eficácia;
- Se agrupem serviços e partilhem recursos.

Para que o modelo cliente-servidor pudesse funcionar, foi necessário criar um protocolo que permitisse a comunicação entre clientes e servidores: *HyperText Transfer Protocol*, mais conhecido por HTTP. Contudo, para que este protocolo sirva de meio de comunicação entre o cliente e o servidor, é necessário que exista uma ligação *Transmission Control Protocol* (TCP) previamente estabelecida. A partir da Figura 2.1, é possível identificar todas as trocas de mensagens que ocorrem ao longo deste processo.

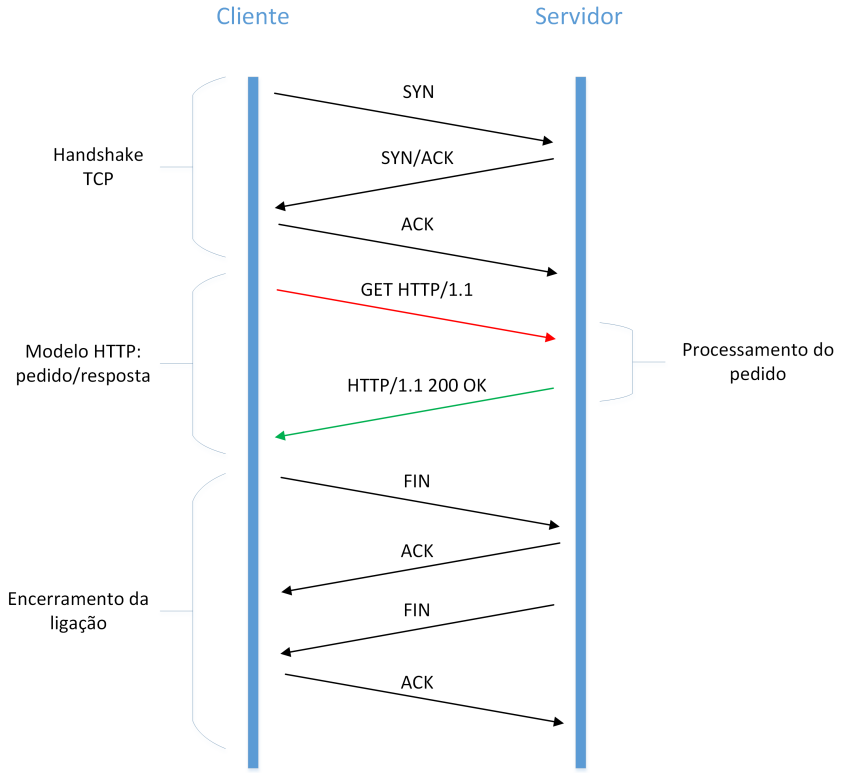

Figura 2.1: Sessão HTTP

Para que uma ligação TCP possa ser estabelecida, o servidor terá de abrir um *socket* e esperar que cheguem pedidos de ligação por parte dos clientes. Clientes esses que, para iniciar uma ligação, terão de enviar um pacote TCP com a *flag* SYN ativa. Depois de enviar o pacote, o cliente fica à espera, durante um determinado período de tempo, para perceber se recebe, ou não, um pacote SYN + ACK proveniente do servidor – o envio deste pacote por parte do servidor significa que este quer estabelecer uma ligação TCP com o cliente. Caso o período de tempo em que o cliente está à espera seja ultrapassado, o cliente dá *timeout* e reenvia um novo pacote SYN. Por fim, para que a ligação entre o cliente e o servidor possa ser estabelecida, o cliente terá de responder ao servidor enviando-lhe um pacote ACK.

Um *socket* é um mecanismo de comunicação que permite a transmissão de dados através de um determinado protocolo de transporte, seja ele TCP ou *User Datagram Protocol* (UDP). Os *sockets* permitem a troca de informações entre duas ou mais aplicações estejam elas no mesmo

computador ou em computadores diferentes. Um *socket* é a combinação entre um endereço *Internet Protocol* (IP) e a porta de um determinado processo a ser executado nesse mesmo endereço IP [Cal81].

Depois de cliente e servidor estarem ligados por TCP, o cliente, com recurso a um endereço URL, pode enviar o seu primeiro pedido HTTP ao servidor. Depois do pedido ter sido enviado, o cliente fica à espera da resposta enviada pelo servidor. A Figura 2.2 mostra o formato de um endereço *Uniform Resource Locator* (URL). No pedido HTTP está indicado o método, a identificação do recurso pretendido e os parâmetros do pedido. Num pedido HTTP, os métodos que podem ser utilizados são: GET, POST, HEAD, DELETE, PUT, OPTIONS e CONNECT [FGM+99].

Figura 2.2: Formato de um endereço URL [The14]

Do outro lado, o servidor, que depois de se ligar ao cliente, fica à espera de receber pedidos HTTP, recebe o pedido, processa-o e envia a resposta para o cliente. A resposta que é enviada ao cliente inclui informações sobre a forma como o pedido foi processado (por exemplo, 200 – Bem-sucedido, o servidor conseguiu processar o pedido; 404 – Não encontrado, o servidor não encontrou o recurso pedido; 403 – Proibido, o servidor recusa-se a processar o pedido; 500 – Erro interno do servidor, o servidor encontrou um erro e não pode completar o pedido [Goo15, FGM+99]). Caso o pedido consiga ser processado com sucesso, a informação pedida pelo cliente vai no corpo da mensagem.

Por fim, o processo que levará ao fim da ligação entre o cliente e o servidor envolve quatro fases em que tanto o cliente como o servidor têm a responsabilidade de encerrar o seu lado da ligação. Caso seja o cliente a iniciar o processo de finalização da ligação, terá de enviar ao servidor um pacote com a *flag* FIN ativa, o qual deverá esperar um pacote ACK oriundo do servidor. Após o processo de conclusão no lado do cliente estar concluído, é a vez de o servidor iniciar o processo que irá levar à cessação da ligação TCP no seu lado. Tal como o cliente, o servidor terá de enviar um pacote FIN e esperar pela receção de um pacote ACK para encerrar a ligação [Cal81].

Como qualquer tecnologia, também o protocolo HTTP evoluiu desde a sua criação, em 1990, até aos dias de hoje. A primeira versão do protocolo HTTP, HTTP/0.9 era uma versão bastante simples em que o único recurso que podia ser enviado do servidor para o cliente era um documento HTML. Porém, no período compreendido entre 1991 e 1995, a criação dos navegadores *Web* e o surgimento da Internet mais direcionada para o consumidor fez com que muitas das limitações do protocolo HTTP/0.9 fossem expostas, o que levou à criação do protocolo HTTP/1.0. Com a implementação desta versão, o cliente pôde começar a pedir, contrariamente ao que acontecia na versão anterior, qualquer tipo de conteúdo, o que conjugado com os melhoramentos feitos na comunicação entre os clientes e os servidores, fez com que várias limitações do HTTP/1.0 acabassem por

ficar visíveis. Uma das limitações consistia no facto de, por cada ligação TCP, só poder haver um pedido do cliente e a respetiva resposta do servidor (esta limitação já se verificava no protocolo HTTP/0.9). Até que em 1999, foi lançada uma nova versão do protocolo HTTP, o HTTP/1.1, que tinha como objetivo resolver muitas das ambiguidades deixadas pelas anteriores versões e introduzir melhorias significativas no desempenho da comunicação entre clientes e servidores. Uma das melhorias serviu para corrigir a limitação acima identificada, uma vez que, a partir de uma ligação TCP, se tornou possível fazer vários pedidos HTTP [Inc13].

Com a implementação do protocolo HTTP/1.1, muitos dos problemas pareciam resolvidos e aqueles que iam surgindo eram facilmente corrigidos [KLI00, NHL10, RG11, FANR12]. Todavia, as coisas começaram a mudar quando as aplicações que, até aqui, eram *stand-alone* – como por exemplo: os jogos, os sistemas de mensagens instantâneas (*chat*), as ferramentas de trabalho colaborativas, etc. – começaram a ser transportadas para os navegadores *Web* [HÏ1]. A migração das típicas aplicações *stand-alone* para a *Web* trariam imensas vantagens, já referidas na introdução, tanto para os utilizadores como para os programadores *Web*. No entanto, a migração para a *Web* também acaba por trazer algumas desvantagens, como por exemplo [Smi15, She02]:

- As aplicações *stand-alone* conseguem tirar partido de todas as capacidades que o *hardware* do computador oferece. Ao contrário do que acontece com as aplicações *Web* uma vez que estas estão limitadas à forma com que os navegadores *Web* aproveitam o *hardware* do computador;

- As aplicações *Web* estão expostas a mais problemas de segurança do que as aplicações *stand-alone*. Como é possível ter o controlo total sobre as aplicações *stand-alone*, todas as vulnerabilidades que possam existir são facilmente corrigidas;

- A integração com dispositivos externos, tais como as impressoras ou os *scanners*, é mais fácil nas aplicações *stand-alone* do que nas aplicações *Web* uma vez que os navegadores *Web* dificultam o acesso aos dispositivos externos.

Apesar de todas as vantagens que existiam na migração das aplicações *stand-alone* para a *Web*, as aplicações *Web*, tal como acontecia com as aplicações *stand-alone*, teriam de ter a capacidade de atualizar a informação em tempo real e isso era algo que o protocolo HTTP não permite. Até esta altura, muitas páginas *Web* eram estáticas (algumas conseguiam mostrar algum conteúdo dinâmico graças ao CGI, aos Java Servlets ou ao Active Server Pages [GLG11]) – o conteúdo da página *Web* só podia atualizado manualmente – o que ia ao desencontro das necessidades deste novo tipo de aplicações *Web*. Esta incapacidade de, nativamente, conseguir mostrar informação em tempo real prendeu-se com o facto de o protocolo HTTP ser um protocolo *half duplex*, isto é, só permite enviar pedidos numa direção (do cliente para o servidor) e, para se poder manter a informação a ser atualizada em tempo real, seria necessário um sistema que permitisse o envio de pedidos em ambas as direções [SM11].

Devido à limitação da *Web* em conseguir mostrar informação em tempo real, os programadores *Web* desenvolveram várias técnicas que permitem contornar esta limitação e que vão ser descritas nas secções que se seguem.

2.2 *Polling*

Polling é uma técnica que permite ao cliente fazer pedidos HTTP ao servidor em intervalos de tempo constantes. O *polling* seria uma excelente técnica para aceder a informação em tempo real caso se conhecesse a frequência exata com que os dados são atualizados no servidor. O problema é que na maioria dos casos isso não acontece, o que leva a que a frequência com que o cliente faz os pedidos ao servidor tenha de ser elevada para conseguir captar todas as atualizações que ocorreram [Mul14]. A Figura 2.3 mostra o funcionamento do *polling* ao longo do tempo. Contudo, fazer vários pedidos HTTP, durante um curto espaço de tempo, traz consequências para o servidor, para a rede e para o próprio cliente [Jo11].

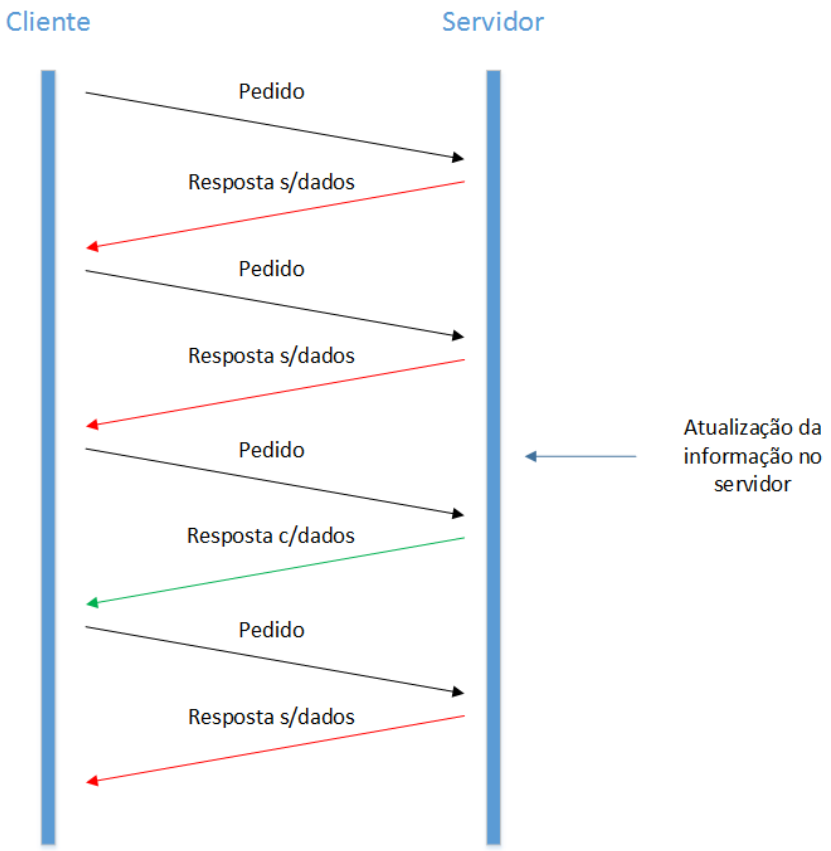

Figura 2.3: *Polling*

No servidor, a grande quantidade de pedidos pode fazer com que o desempenho do servidor diminua sendo que, muitas das vezes, as respostas não transmitem informação válida para o cliente. As consequências para o cliente estão relacionadas com o número de respostas que recebe,

isto é, devido ao elevado número de pedidos que envia, o cliente tem de receber as respostas a esses mesmos pedidos o que pode vir a comprometer o *rendering* da página [Mul14] e dificultar a interação com o utilizador. No caso da rede, a troca de muitos pedidos/respostas entre o cliente e o servidor faz com que o tráfego da rede aumente.

Para reduzir o número de pedidos que são feitos ao servidor, uma das soluções passaria por diminuir a frequência com que os pedidos são feitos, mas isso levanta problemas relacionados com a perda de informação. Quando o intervalo de tempo entre os pedidos é grande, o servidor pode estar a ser atualizado, durante o período em que não recebe pedidos, com informação relevante para o cliente que, devido à ausência de pedidos, vai chegar com atraso ao cliente.

Uma outra forma de otimizar o processo de *polling* é através do mecanismo *Adaptive Time To Refresh* (TTR). Este mecanismo consiste na alteração da frequência com que o cliente envia pedidos HTTP ao servidor. O cliente, com base nos últimos pedidos HTTP enviados e cujas respostas trouxeram informação nova, estima qual a frequência com que a informação no servidor é alterada [DKP+01]. Para que o cliente consiga estar sempre a enviar pedidos HTTP com uma frequência semelhante àquela que é utilizada para atualizar a informação no servidor, o processo de estimação tem de ser contínuo.

Adaptive TTR é um mecanismo que, apesar de oferecer melhores resultados comparativamente ao *polling* de frequência fixa, nunca conseguirá evitar que haja pedidos HTTP em que as respostas não tenham informação relevante para o cliente [BMD08].

2.3 Comet

Comet é uma técnica que permite enviar dados do servidor para o cliente sem que este tenha feito, de forma explícita, algum pedido. Esta técnica permite, com recurso a ligações HTTP persistentes que estejam previamente criadas, reduzir a latência com que as mensagens são trocadas entre cliente e servidor. Graças a uma arquitetura assente em eventos, o cliente não necessita, ao contrário do que acontece com o *polling*, de estar a enviar, constantemente, pedidos HTTP para saber se houve alguma alteração no servidor. Em vez disso, o servidor tem uma linha de comunicação aberta que permite, a qualquer momento, enviar dados para o cliente sempre que houver alguma alteração [Rus06, Gra14].

Por fim, a implementação desta técnica numa aplicação *Web* pode ser feita com recurso ao *long polling* ou ao *streaming*.

2.3.1 *Long polling*

Long polling, também conhecido por *Asynchronous-Polling*, é uma técnica, em parte similar ao *polling*, que dá, ao servidor, a possibilidade de manter um pedido HTTP ativo durante um período de tempo. Por omissão, o servidor mantém um pedido ativo durante 45 s. No entanto, o servidor pode alterar o período de tempo em que os pedidos se mantêm ativos [BMD08].

A Figura 2.4 ilustra o processo de atualização da informação com recurso ao *long polling*. No início, o cliente começa por enviar um pedido HTTP ao servidor. Depois, se, no período de

tempo em que o pedido está ativo, houver alguma alteração na informação guardada no servidor, é enviada uma resposta ao cliente e a ligação entre cliente e servidor é encerrada. Contudo, caso a informação se mantenha inalterada, o servidor responde ao cliente notificando-o de que vai terminar o pedido e a ligação será terminada. Depois de receber a resposta ao seu pedido, o cliente tenta estabelecer nova ligação com servidor enviando-lhe um novo pedido HTTP [Aga12].

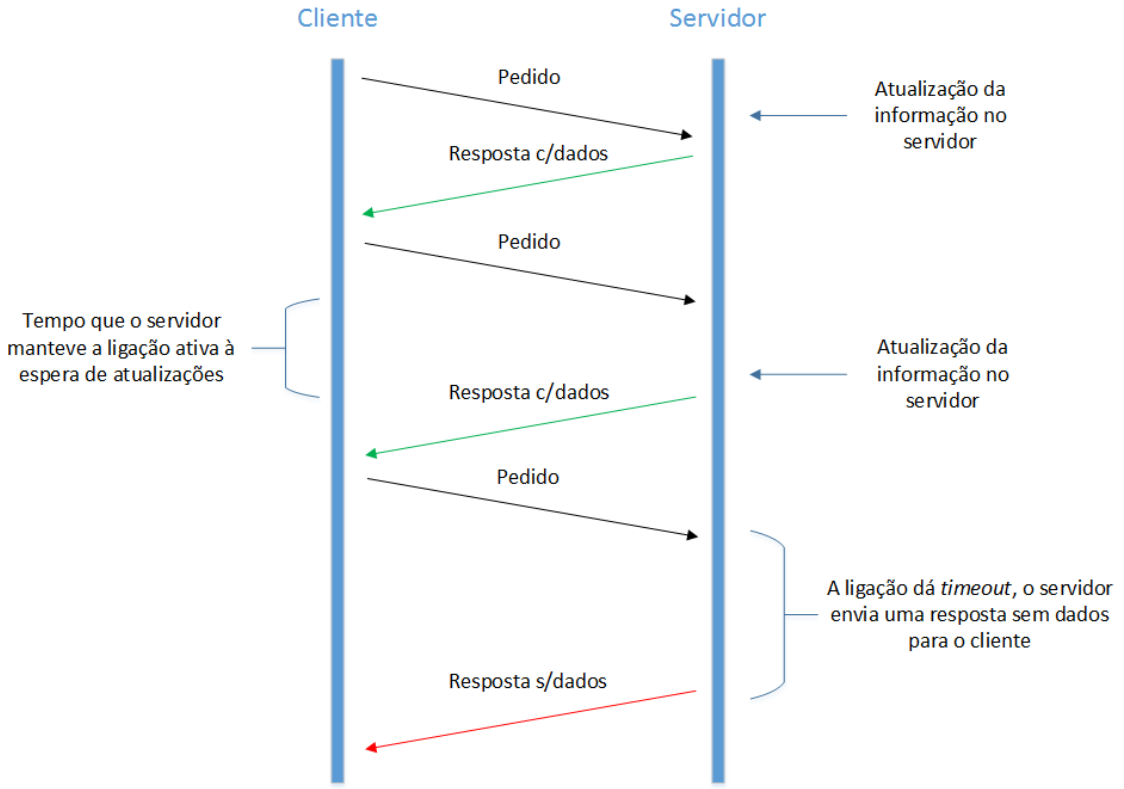

Figura 2.4: *Long Polling*

A principal vantagem do *long polling*, face ao *polling*, é a redução, em média, do número de pedidos que são feitos ao servidor [ZS13]. Contudo, o *long polling* não é uma técnica que possa ser utilizada em qualquer aplicação *Web* uma vez que não deve de ser implementada em aplicações que necessitem de uma elevada frequência de atualização. Esta situação produz uma grande quantidade de mensagens trocadas entre o cliente e o servidor e obriga a que o servidor tenha de gerir, ao mesmo tempo, uma grande quantidade de ligações abertas [The14].

2.3.2 *Streaming*

O *streaming* é baseado numa ligação HTTP persistente. Tal como o *long polling* e o *polling*, esta técnica começa com o envio de um pedido HTTP por parte do cliente, no entanto, a diferença em relação às outras técnicas reside na forma como a resposta é dada. Enquanto nas outras técnicas, o servidor responde ao cliente numa única resposta, no *streaming*, o servidor nunca informa o cliente de que a resposta está completa o que lhe permite manter uma linha de comunicação aberta

e pronta para enviar novos dados. Na Figura 2.5 é percetível a forma como o *streaming* consegue obter informação em tempo real.

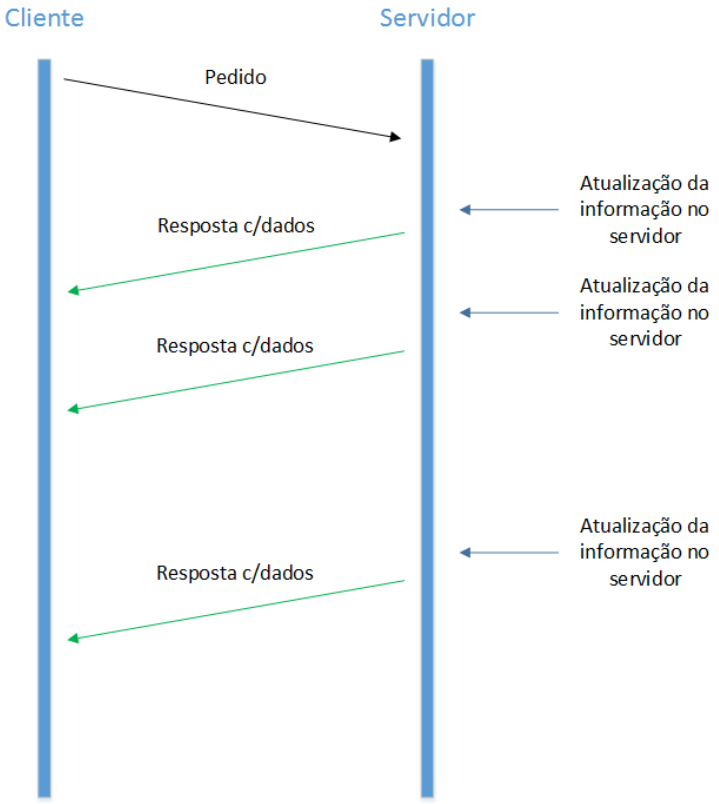

Figura 2.5: *Streaming*

Se compararmos o *streaming* ao *long polling* e ao *polling*, o *streaming* resolve os problemas relacionados com as elevadas frequências com que o cliente envia pedidos ao servidor. Contudo, a transmissão de dados através de *streaming* pode ser prejudicial para o desempenho do servidor. O funcionamento de *streaming* implica que haja uma linha de comunicação aberta entre o cliente e servidor, o que leva a que, quando existam muitos clientes ligados a um servidor em que é frequente a ocorrência de eventos, a carga do servidor aumente, podendo causar problemas de desempenho ou, em casos extremos, causar o colapso do próprio servidor [LG13]. Para além dos problemas com o desempenho dos servidores, não se deve utilizar *streaming* quando se sabe, *a priori*, que na rede existem *proxies* e *firewalls*, uma vez que fazem atrasar a resposta, o que resulta num aumento da latência na entrega das mensagens [WSM13].

Atualmente, a técnica de *streaming* pode ser aplicada através de XMLHttpRequest (XHR) *Multipart Streaming* ou através de XHR *Iframe Streaming*.

2.3.2.1 XHR *Multipart Streaming*

Quando um servidor responde a um pedido de um cliente, a resposta contém, para além dos *headers* necessários, os dados pedidos pelo cliente. Os dados enviados pelo servidor podem estar

num dos seguintes formatos: *application*, *audio*, *example*, *image*, *message*, *model*, *multipart*, *text* ou *video*.

O XHR *Multipart Streaming* tira proveito das vantagens que o *multipart* oferece, uma vez que, devido ao facto de ser um formato desenhado para o envio de grandes quantidades de dados, dá a possibilidade ao servidor de enviar os dados repartidos em várias partes. Este formato pode ser utilizado numa resposta XHR, na medida em que as mensagens enviadas pelo servidor são interpretadas pelo cliente como se fossem uma parte da resposta. Isto permite manter a ligação aberta para as mensagens que irão ser enviadas. Na prática, o servidor, ao enviar mensagens como se fossem partes da resposta ao pedido efetuado, engana o cliente fazendo com que este mantenha a ligação aberta. A vantagem de haver uma única ligação entre o cliente e o servidor é a redução no número de *headers* HTTP enviados nos pedidos e nas respostas.

No XHR *Multipart Streaming*, o cliente começa por enviar um pedido ao servidor. Do outro lado, o servidor indica que o conteúdo vai no formato *multipart*, o que faz com que o cliente mantenha a ligação aberta enquanto está à espera de receber todas as partes que compõem a resposta do servidor. Enquanto o cliente está à espera, todas as mensagens que o servidor envia é como se fizessem parte da resposta que tem de ser enviada. Por fim, quando a aplicação estiver pronta para ser encerrada, o servidor envia a "última" parte da resposta e a ligação é terminada.

Uma limitação do XHR *Multipart Streaming* é que, em algumas situações, o cliente guarda todas as mensagens que recebe em memória. Para reduzir a memória carregada quando se utiliza o XHR *Multipart Streaming* para outros propósitos, a ligação entre cliente e servidor tem de ser fechada e aberta periodicamente [GLG11].

2.3.2.2 XHR *Iframe Streaming*

XHR *Iframe Streaming* é uma técnica que, tal como acontece no XHR *Multipart Streaming*, permite manter uma ligação aberta entre o cliente e o servidor, só que ao contrário do que acontece no XHR *Multipart Streaming*, esta técnica usa *hidden iframe* como mensagem. O atributo "source" é definido com recurso a CGI, Java Servlets ou a Active Server Pages que transmitem o código JavaScript gerado dinamicamente para o navegador *Web*.

A primeira mensagem enviada pelo servidor fornece as *tags* HTML apropriadas enquanto as restantes mensagens enviadas incluem a *tag* <script> com uma chamada a uma função JavaScript que passa, como parâmetro, a mensagem que contém a informação relevante para o cliente. O navegador *Web* executa a função e, em seguida, processa a mensagem.

O XHR *Iframe Streaming* só pode ser utilizado em navegadores *Web* que suportem *iframes*. A sua principal vantagem é a simplicidade de implementação [Car11]. Contudo, a sua implementação apresenta algumas desvantagens como, por exemplo, a inexistência de uma forma para tratar os erros ou para controlar o estado da ligação – a ligação e os dados são manipulados pelo navegador *Web* através de *tags* HTML – e, tal como o XHR *Multipart Streaming*, o XHR *Iframe Streaming*, em algumas situações, guarda todas as mensagens que recebe em memória. Para evitar esta situação, a solução passa por, periodicamente, encerrar e abrir a ligação entre cliente e servidor [GLG11].

2.4 *Plug-ins*

Durante muito tempo, a única maneira de enviar e receber dados em tempo real foi através de *plug-ins*, tais como Java Applets ou aplicações Flash.

Um *plug-in* é um módulo de *software* que adiciona um recurso específico a um sistema. Os navegadores *Web* têm utilizado *plug-ins* para diversas finalidades onde se inclui a visualização de documentos, a reprodução de vídeos, ou o acesso a dispositivos (por exemplo, câmaras) que estão fora do modelo de segurança da *Web*.

Existem vários *plug-ins* para navegadores *Web* que permitem o desenvolvimento de sistemas que mostrem informação em tempo real. Estes são, normalmente, baseados numa linguagem de programação já existente.

Os três *plug-ins* mais utilizados são Java Applets (baseados na linguagem Java), Adobe Flash e Shockwave (baseado no Flex ou em ActionScript) e Microsoft Silverlight (baseado em .Net). As aplicações *Web* desenvolvidas com estas ferramentas são normalmente executadas num ambiente *byte-code* o que torna a execução mais rápida do que recorrendo à interpretação de JavaScript.

Os *plug-ins* possibilitam um ambiente de trabalho completo e com amplo suporte para lidar com gráficos, interfaces, redes, *threads*, etc. No entanto, as restrições de segurança no navegador *Web* podem levar a que os programadores tenham de trabalhar com uma API restrita e com um modelo de arquitetura limitado para a rede. Apesar destas limitações, os *plug-ins* oferecem um ambiente de desenvolvimento equivalente ao de uma aplicação *stand-alone* [Car11].

Contudo, muitos programadores *Web* têm recentemente rejeitado a abordagem baseada em *plug-ins* devido a três fatores:

- Instalação e disponibilidade — Em muitos navegadores *Web*, os *plug-ins* necessários para executar a aplicação podem não existir ou podem não estar disponíveis. Isto quer dizer que os programadores não fazem a mínima ideia se as suas aplicações serão corretamente executadas e, ao mesmo tempo, significa que os utilizadores têm de perder tempo a encontrar e a instalar os *plug-ins* necessários à execução da aplicação. O problema é maior quando se tenta planear as aplicações para *smartphones* ou *tablets*, uma vez que a maior parte dos navegadores *Web* destes dispositivos não suporta *plug-ins*;

- Dependência de uma tecnologia — Os *plug-ins* reduzem o controlo de um programador sobre os recursos ao seu dispor;

- Segurança — Os *plug-ins* são muito vulneráveis a ataques e podem ter graves problemas de segurança. Por exemplo, o *plug-in* da Adobe Flash foi o segundo sistema mais atacado em 2009, sendo que muitos especialistas em segurança não aconselham o uso de Flash durante a visita a páginas *Web* que não sejam confiáveis [Inc10].

2.5 BOSH

Bidirectional-streams Over Synchronous HTTP (BOSH) é uma técnica que simula uma ligação bidirecional entre o cliente e o servidor. Baseada em *long polling* (ver Secção 2.3.1), BOSH é uma técnica que apresenta um modelo de funcionamento com bastantes semelhanças ao *long polling*. A única diferença reside no facto de que no BOSH é possível, graças à criação de um segundo *socket* entre o cliente e o *connection manager* (elemento localizado entre o cliente e o servidor que converte os pedidos HTTP enviados pelo cliente em mensagens que consigam ser interpretadas pelo servidor), o envio de pedidos por parte do cliente durante o período em que outro pedido, anteriormente enviado, está ativo. Quando o *connection manager* verifica que existem dois pedidos ativos, responde ao pedido que estava ativo há mais tempo – na maioria das vezes, a resposta não leva informação relevante para o cliente – e mantém ativo o pedido mais recente.

Uma vez que o *long polling* e o BOSH são técnicas com um modelo de funcionamento muito parecido, as vantagens e as desvantagens associadas à técnica de *long polling* são as mesmas do BOSH (ver Secção 2.3.1) [PSSA+10].

2.6 PubSubHubbub

O protocolo PubSubHubbub baseia-se no modelo de comunicação *Publish-Subscribe* que utiliza um elemento intermediário chamado *hub*. O *hub* é o elemento que permite fazer a ponte entre os *publishers* (interessados em distribuir a informação) e os *subscribers* (interessados em receber a informação atualizada proveniente dos *publishers*).

2.6.1 Modelo *Publish-Subscribe*

O modelo *Publish-Subscribe* é um modelo que permite lidar com mensagens assíncronas, em que os *publishers* são responsáveis pelo envio de mensagens que são, posteriormente, recebidas pelos *subscribers*. Uma grande vantagem deste modelo é a dissociação dos seus elementos uma vez que os *publishers* não têm a necessidade de conhecer os *subscribers* que subscrevem as suas mensagens. No entanto, os *subscribers* podem escolher as mensagens que querem receber. Para além disso, os *subscribers* recebem apenas uma parte das mensagens publicadas. O processo de seleção das mensagens que são recebidas pelos *subscribers* chama-se *filtering*, e pode ser *topic-based* ou *content-based*.

Num sistema *topic-based*, as mensagens são publicadas nos *topics*, que funcionam como repositórios de informação, o que permite aos *subscribers* receber todas as mensagens publicadas num determinado *topic* subscrito por eles. Nos sistemas *content-based*, os *subscribers* definem as restrições sobre as mensagens que querem receber. As restrições às mensagens incidem sobre os seus atributos ou sobre o seu conteúdo.

Em alguns sistemas, existe um elemento intermédio chamado *broker* (no protocolo PubSubHubbub, o *broker* é o *hub*) que armazena e encaminha as mensagens. Neste tipo de implementação, os *publishers* publicam as mensagens no *broker*, que são posteriormente encaminhadas para os *subscribers* registados no *broker*. Também existem sistemas que não utilizam o tal elemento intermédio, para que o *publisher* e o *subscriber* tenham de partilhar informações (metadados) sobre eles próprios, o que leva a concluir que as mensagens são encaminhadas com base na descoberta um do outro [DPC+13].

2.6.2 Protocolo PubSubHubbub

O protocolo PubSubHubbub trabalha com recurso a três operações básicas:

- *Discovery* — O *subscriber* pede, ao *publisher*, informação sobre um determinado *topic* (pode ser acedido através de um endereço URL, onde, com recurso à tecnologia Atom ou *Rich Site Summary* (RSS), a informação atualizada é publicada). Depois, o *publisher* envia ao *subscriber* a informação que este pretende. Com base no que foi enviado, o *subscriber* verifica se existe algum endereço associado ao *hub*, utilizado pelo *publisher* na publicação das atualizações no *topic*, ou se existe outro tipo de informação que seja relevante. Caso exista alguma referência ao *hub*, o *subscriber* pode subscrever-se no *hub* e assim continuar a receber informação atualizada. Por outro lado, caso o *subscriber* não consiga encontrar um endereço para o *hub*, o *subscriber* nunca mais poderá usar o protocolo PubSubHubbub e terá que usar uma técnica alternativa para continuar a receber a informação atualizada;

- Subscrição — O *subscriber* pede ao *hub*, previamente encontrado, para subscrever um determinado *topic*. No processo de subscrição é necessário que o *subscriber* passe o endereço do *topic* que pretende subscrever e a informação necessária para que o *hub* consiga enviar ao *subscriber* a informação atualizada. Em seguida, o *hub* tem de enviar uma mensagem ao *subscriber* a confirmar a subscrição;

- Publicação — Nesta operação, o *publisher* publica no *topic* e de imediato notifica o *hub* sobre a existência de informação atualizada (o *publisher* passa ao *hub* o endereço do *topic* atualizado). Em seguida, o *hub* consulta o endereço enviado pelo *publisher* e obtém a informação para reencaminhar para os *subscribers* que subscreveram o *topic* onde a informação foi atualizada. A Figura 2.6 ilustra a operação descrita neste ponto.

Este protocolo evita que os clientes tenham de estar a verificar constantemente o servidor à procura de atualizações e elimina a comunicação direta entre o cliente e o servidor, a comunicação passa a ser cliente-*hub*-servidor. Contudo, as três operações acima referidas fazem com haja um aumento no tráfego de rede e consomem, desnecessariamente, recursos computacionais [DPC+13].

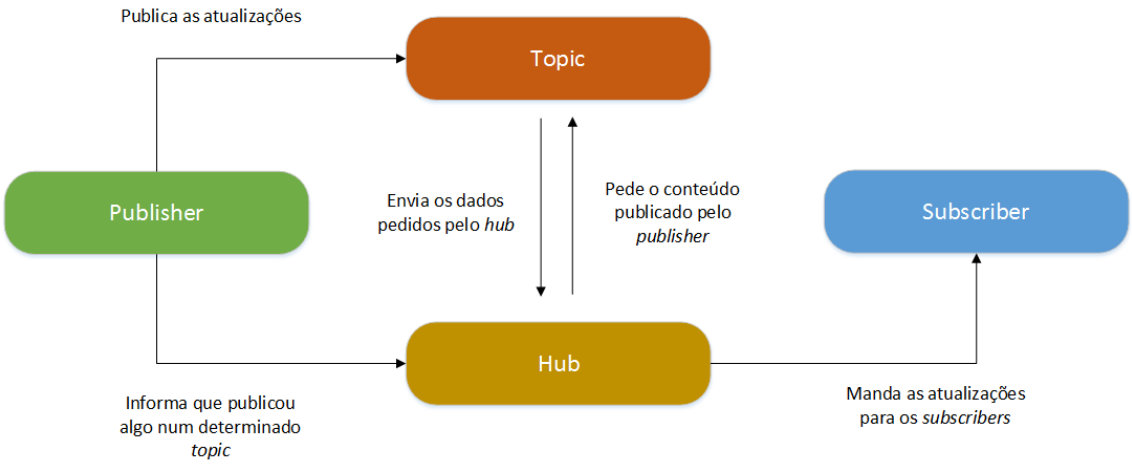

Figura 2.6: PubSubHubbub

2.7 WebSockets

Incluído na especificação do HTML5, o protocolo WebSocket permite, com recurso a um único *socket* TCP, criar uma ligação bidirecional e *full-duplex*[1] entre clientes e servidores [WSM13].

Como pode ser observado na Figura 2.7, para que uma ligação WebSocket possa ser estabelecida, o cliente tem de enviar um pedido HTTP ao servidor. O pedido enviado tem um cabeçalho diferente do habitual, uma vez que vai permitir ao servidor responder com a informação sobre se suporta, ou não, WebSockets. Caso o servidor suporte WebSockets, o cliente pode estabelecer ligação com o servidor. Este processo permite, através da criação de um canal de comunicação direto entre o cliente e o servidor, evitar que *proxies*, *routers* e/ou *firewalls* condicionem a troca de mensagens [SM11].

A tecnologia WebSocket fornece uma API que é acessível a partir do JavaScript, permitindo que os programadores possam abrir um *socket* que ligue o navegador *Web* ao servidor onde é possível enviar e receber dados. A WebSocket API apenas fornece as funcionalidades básicas e não oferece o mesmo grau de controlo sobre os *sockets* que é oferecido por muitas outras linguagens de programação como por exemplo, o Java ou o C++ [GLG11].

Os WebSockets permitem a redução da latência e do tráfego de rede uma vez que, ao contrário do que acontece, por exemplo, no *polling*, o cliente não necessita de estar a verificar constantemente se ocorreu alguma alteração no servidor. Em vez disso, o servidor envia a informação ao cliente sempre que esta seja atualizada. Outra característica que diferencia o protocolo WebSocket das restantes técnicas é a estrutura das mensagens trocadas entre clientes e servidores, visto que a quantidade de dados utilizados nos cabeçalhos das mensagens dos WebSockets é muito menor comparativamente à aos cabeçalhos das mensagens HTTP que são utilizadas pelas restantes técnicas (a diferença cifra-se na ordem de 1:1000 a favor dos WebSockets) [LG13].

[1] Uma ligação *full-duplex* permite que as mensagens sejam trocadas do cliente para o servidor e do servidor para o cliente ao mesmo tempo e assincronamente.

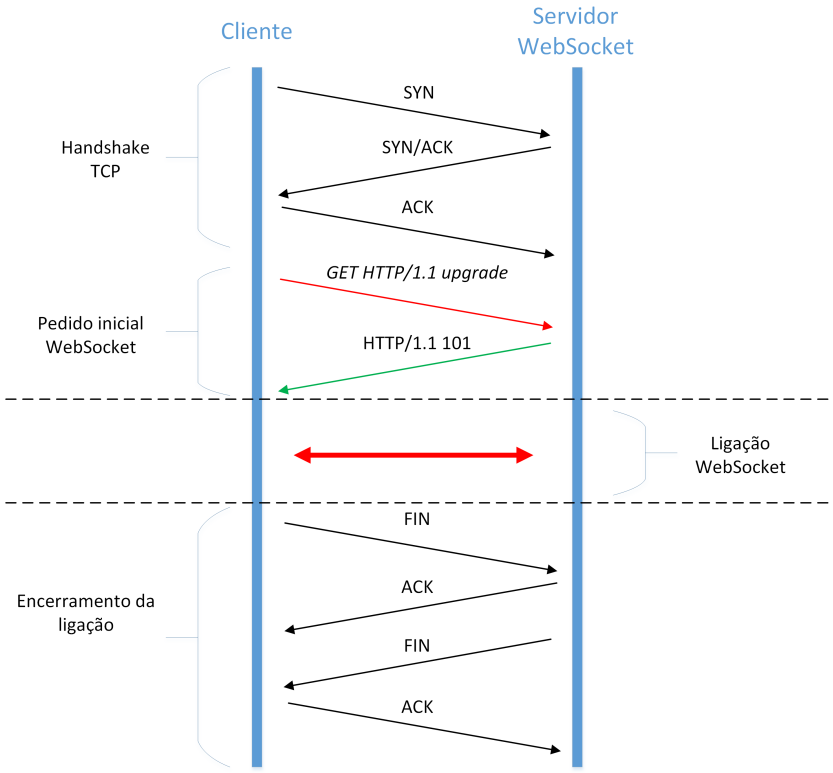

Figura 2.7: WebSockets

> "Reducing kilobytes of data to 2 bytes...and reducing latency from 150 ms to 50 ms is far more than marginal. In fact, these two factors alone are enough to make Web Sockets seriously interesting to Google." [LG13]

2.8 Resumo

Desde a sua criação até aos dias de hoje, a *Web* está em constante evolução. Se compararmos a *Web* na época em que surgiu com o que temos agora, é fácil de ver que a *Web* teve uma evolução espantosa.

Um dos fatores que mais fez crescer a *Web* foi espoletado pelo início do processo de migração das aplicações *stand-alone* para aplicações *Web* uma vez que obrigou a que fossem desenvolvidas técnicas que permitissem alterar dinamicamente o conteúdo das páginas *Web* para mostrar informação em tempo real.

Desde as primeiras tentativas, de migrar uma aplicação *stand-alone* para a *Web*, até aos dias de hoje, foram muitas as técnicas desenvolvidas para que a informação pudesse ser mostrada em tempo real. Atualmente, as técnicas mais utilizadas são o *polling*, o *long polling*, o *streaming*, o uso de *plug-ins*, o BOSH, o protocolo PubSubHubbub e os WebSockets. Desta lista destacam-se três técnicas:

- *Polling* — Atualmente, o *polling* é a técnica mais utilizada na obtenção de informação em tempo real apesar de ser uma técnica que, recorrendo a múltiplos pedidos HTTP feitos pelo cliente num curto espaço de tempo, apenas simula o processo de atualização da informação em tempo real;
- *Plug-ins* — É a única técnica que necessita de *software* externo para conseguir mostrar informação em tempo real. Apesar de ser uma técnica que permite trabalhar num ambiente de desenvolvimento muito parecido com o das aplicações *stand-alone*, é uma técnica com várias desvantagens ao nível da utilização (em navegadores *Web*) e segurança;
- WebSockets — Das técnicas acima mencionadas, o protocolo WebSocket é a técnica mais recente. Esta técnica permite aplicar à *Web* o mesmo modelo de funcionamento dos *sockets* que são utilizados na Internet.

Capítulo 3

Definição do Problema

O Capítulo 3 é constituído por 4 secções. A primeira secção serve para explicar detalhadamente qual o problema que tem de ser resolvido. A segunda secção dá a conhecer a arquitetura que está por detrás das aplicações *Web* da Glintt HS. A terceira explica de que forma é que o *polling* é feito nas aplicações *Web* da Glintt HS. Por fim, a última secção é um breve resumo sobre o que foi escrito nas primeiras três secções.

3.1 Descrição do Problema

Atualmente, as aplicações *Web* desenvolvidas pela Glintt HS, que já estão em funcionamento em várias instituições hospitalares, utilizam *polling* (ver Secção 2.2) para atualizar a informação que é mostrada no ecrã. O *polling*, apesar de obter excelentes resultados e de ser fácil de implementar, é uma técnica que simula o processo de atualização da informação em tempo real e, como tal, acaba por exigir bastante da rede, dos servidores e dos próprios clientes, o que pode prejudicar o desempenho de todo o sistema de informação.

No servidor, o número elevado de pedidos HTTP, que são enviados pelos vários clientes, faz com que o servidor consuma muitos recursos computacionais para conseguir dar resposta a todos os pedidos, degradando o seu desempenho. Com a utilização de *polling*, até os próprios clientes acabam por sofrer as consequências do grande número de pedidos HTTP que enviam, uma vez que, devido ao elevado número de respostas que chegam, os clientes têm dificuldades em processar todas as respostas, afetando o *rendering* da página *Web* e, consequentemente, a interação com o profissional de saúde. Por fim, no caso da rede, a grande quantidade de dados das mensagens HTTP, conjugado com o número de mensagens que são trocadas entre os clientes e os servidores, fazem com que o tráfego da rede aumente.

A diminuição do desempenho do servidor, o aumento do tráfego de rede e os problemas relacionados com o *rendering* da informação nas páginas *Web* podem fazer com que a informação seja

disponibilizada aos profissionais de saúde com algum atraso significativo ou que, em casos mais graves, acabe por não ficar disponível.

3.2 Arquitetura das Aplicações Glintt

Tal como pode ser observado através da Figura 3.1, a arquitetura das aplicações *Web* desenvolvidas pela Glintt HS é constituída por:

- Um ou mais computadores, com sistema operativo Windows[1], que acedem às aplicações *Web* através dos navegadores *Web*. O navegador *Web* mais utilizado pelas instituições hospitalares é o Internet Explorer – as versões utilizadas são a 9 e a 11 (a mais recente) – contudo, existem instituições que utilizam o Google Chrome;
- Um ou mais servidores das aplicações *Web*. Em cada instituição hospitalar, todas as aplicações *Web* têm o seu "próprio" servidor. O sistema operativo instalado nos servidores das aplicações *Web* é o Windows Server;
- Um ou mais servidores da API. Estes servidores, com Windows Server instalado, permitem fazer a ponte entre os servidores das aplicações *Web* e os servidores aonde estão localizadas as bases de dados;
- Um ou mais servidores das bases de dados. Equipados com Windows Server, estes servidores guardam a informação, em bases de dados Oracle[2], que vai ser disponibilizada às aplicações *Web*.

O facto de os servidores das aplicações *Web* poderem aceder a mais do que um servidor da API deve-se à forma como a API está instalada nas diferentes instituições hospitalares. A API, que a Glintt HS instala nas instituições hospitalares, está dividida em vários módulos que, devido às necessidades da instituição hospitalar, podem, ou não, ser instalados. Quando uma determinada instituição hospitalar exige que se instale uma grande quantidade de módulos, pode ser necessário dividir a API por vários servidores, o que obriga a que os servidores das aplicações tenham de aceder aos vários servidores da API. Para além da possível existência de mais do que um servidor da API, também existe a possibilidade de existir, por instituição hospitalar, mais do que um servidor para as bases de dados. Contudo, isto vai depender da complexidade das bases de dados utilizadas. Inclusive, existem instituições hospitalares cujas bases de dados estão inseridas no mesmo servidor enquanto existem outras instituições em que uma base de dados está repartida em vários servidores.

Na Figura 3.1, a relação entre o computador e o servidor da aplicação *Web* tem uma multiplicidade de 1..* uma vez que, a partir de um computador, é possível aceder a várias aplicações

[1] O sistema operativo dos computadores das instituições hospitalares é Windows uma vez que antigamente as aplicações *Web* só podiam correr no Internet Explorer. Apesar de em algumas instituições o navegador *Web* ter mudado para Google Chrome, os computadores continuaram a utilizar Windows como sistema operativo.

[2] As versões instaladas são a 10 e a 11.

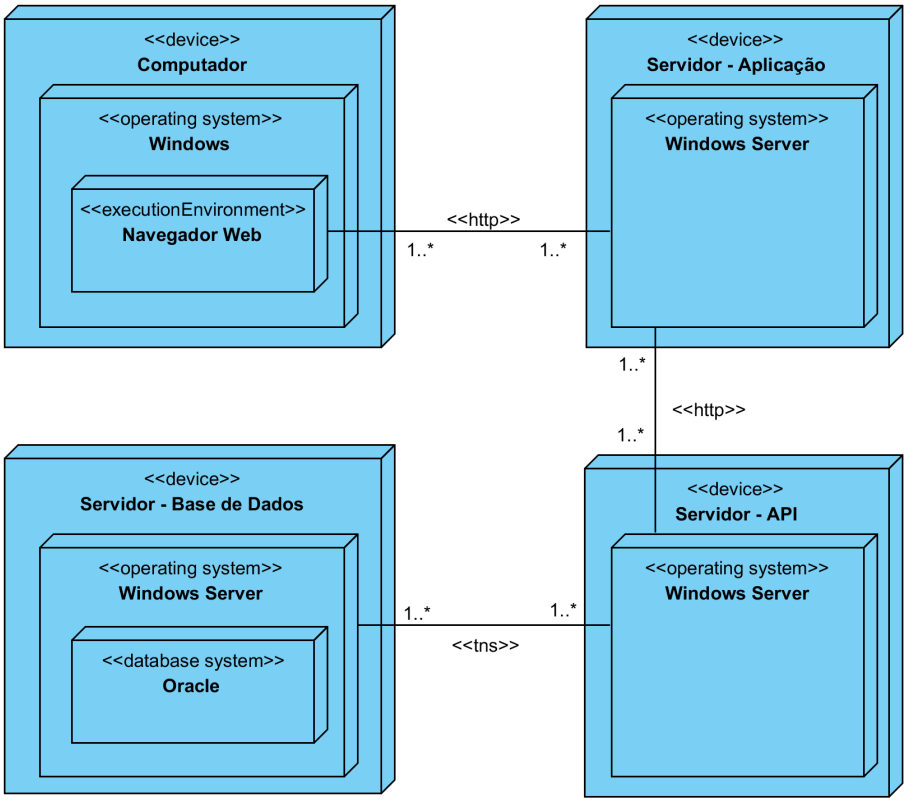

Figura 3.1: Arquitetura das aplicações Glintt

Web e por conseguinte, aceder a vários servidores. A relação inversa à relação acima descrita também apresenta uma multiplicidade de 1..* porque, na mesma instituição hospitalar, um conjunto de computadores pode aceder à mesma aplicação *Web* o que significa que os vários computadores estão a comunicar com o mesmo servidor. A relação entre o servidor da aplicação *Web* e o servidor da API tem uma multiplicidade de 1..*, dado que o servidor da aplicação *Web* vai reencaminhar o pedido HTTP, que recebeu da aplicação *Web*, para o servidor da API que consiga dar resposta ao pedido. Isto quer dizer que, dependendo do tipo de pedido, o servidor da aplicação pode recorrer a vários servidores da API para conseguir satisfazer todos os pedidos feitos por uma aplicação *Web*. No entanto, a relação inversa também tem uma multiplicidade de 1..*, visto que um servidor da API pode ser acedido por diferentes servidores de aplicações *Web* (acontece quando diferentes aplicações *Web* necessitam de fazer o mesmo pedido HTTP). A relação entre o servidor da API e o servidor da base de dados tem uma multiplicidade de 1..* porque o servidor da API, para conseguir obter a informação necessária, pode ter que recorrer a mais do que um servidor de base de dados. A relação inversa também tem uma multiplicidade de 1..*, uma vez que um servidor de base de dados pode ser acedido pelos vários servidores da API instalados na instituição hospitalar.

Atualmente, a forma de comunicação utilizada entre a aplicação *Web* (através do navegador *Web* instalado no computador) e o servidor da aplicação *Web* e entre o servidor da aplicação *Web* e o servidor da API é através de mensagens HTTP. A comunicação entre os servidores da API e

os servidores das bases de dados é feita através de *Transparent Network Substrate* (TNS). O TNS é um protocolo de comunicação, desenvolvido pela Oracle, que, tal como o HTTP, trabalha sobre o protocolo TCP/IP. Este protocolo tem como objetivo facilitar a ligação de um cliente Oracle (neste caso, serão os servidores da API) aos servidores de bases de dados Oracle, sendo que, entre os dois *endpoints*, é possível a chamada de funções que permitam abrir ou fechar uma sessão e enviar ou receber pedidos ou respostas [ABC+00].

3.3 *Polling* nas Aplicações Glintt

Tendo por base a arquitetura acima especificada, é possível elaborar um diagrama de sequência que explica de que forma é que o processo de *polling* funciona nos sistemas da Glintt HS. Segundo a Figura 3.2, o processo de *polling* é composto por cinco etapas:

1. A aplicação *Web* envia um pedido HTTP ao servidor da aplicação (transição 1);
2. O servidor da aplicação reencaminha o pedido que recebeu para o servidor da API que consiga dar resposta ao pedido (transição 1.1);
3. O servidor da API vai buscar ao(s) servidor(es) de base(s) de dados, através do protocolo TNS, a informação necessária para responder ao pedido da aplicação *Web* (transição 1.1.1 e 1.1.2);
4. O servidor da API envia ao servidor da aplicação a resposta obtida (transição 1.2). Posteriormente, o servidor da aplicação envia à aplicação *Web* a informação proveniente do servidor da API (transição 1.3);
5. O conteúdo da aplicação *Web* é atualizado de acordo com a informação recebida (transição 2).

Existe uma particularidade na forma com que as aplicações *Web* da Glintt HS aplicam o conceito de *polling*. De cada vez que uma aplicação *Web* envia um pedido HTTP, a resposta que é devolvida não contém apenas os dados que foram alterados durante o intervalo de tempo entre o último pedido enviado e o pedido que está a ser enviado. O que é devolvido à aplicação *Web* são todos os dados, atualizados (se existirem) e não atualizados, que podem ser observados no ecrã do computador. O que significa que, de pedido para pedido, a quantidade de dados que chega às aplicações *Web* varia muito pouco – isto se, durante o período em que não existem pedidos, não mudar o tipo de informação que está a ser mostrado.

3.4 Resumo

O *polling* é uma técnica que, apesar de ser fácil de implementar e de obter excelentes resultados, simula o processo de atualização da informação em tempo real graças ao grande número de pedidos HTTP que são enviados num curto intervalo de tempo.

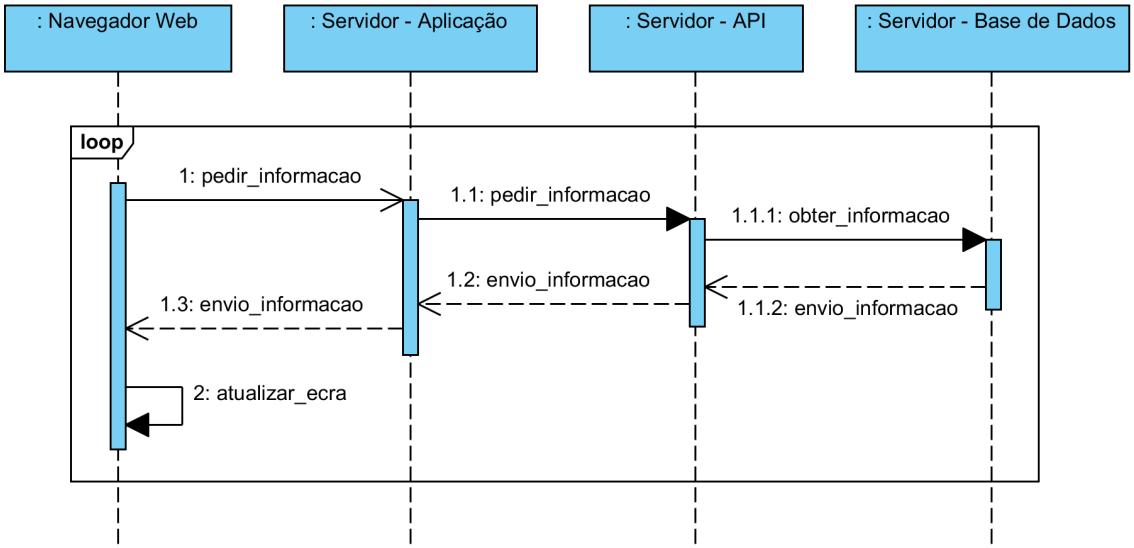

Figura 3.2: Processo de *polling* nas aplicações Glintt

A elevada frequência com que os pedidos HTTP são enviados pode ter fortes impactos no desempenho do servidor, no tráfego da rede e no *rendering* da informação nas aplicações *Web* o que faz com que existam atualizações que cheguem atrasadas ou que acabem por não chegar.

A arquitetura que está por detrás das aplicações *Web* desenvolvidas pela Glintt HS é constituída pelos seguintes elementos:

- Um ou mais computadores;
- Um ou mais servidores das aplicações *Web*;
- Um ou mais servidores da API;
- Um ou mais servidores de base de dados.

Segundo a arquitetura atual, o processo de *polling* começa quando uma aplicação *Web* envia um pedido HTTP ao servidor da aplicação. Depois de ter recebido o pedido, o servidor da aplicação reencaminha-o para o servidor da API que consiga dar resposta ao pedido. O servidor da API vai à base de dados obter a informação necessária para responder ao servidor da aplicação. Quando o servidor da aplicação receber a resposta, ele envia-a para a aplicação *Web*.

Uma particularidade do *polling* nas aplicações desenvolvidas pela Glintt é que a resposta ao pedido HTTP contém toda a informação e não apenas a informação que tiver sido atualizada durante o intervalo de tempo em que não houve pedidos.

Capítulo 4

Solução

O quarto capítulo mostra de que forma é que o problema, descrito no capítulo anterior, vai ser solucionado. Este capítulo é constituído por 4 secções. A primeira secção dá a conhecer a solução que vai ser utilizada para solucionar o problema. A segunda secção explica quais as alterações que irão ser feitas à arquitetura para que a solução descrita na primeira secção possa ser implementada. A terceira secção mostra a técnica e tecnologias que irão ser utilizadas na solução. A última secção é um resumo de tudo o que foi escrito nas restantes secções do capítulo.

4.1 Introdução

Atendendo ao problema descrito no capítulo anterior, à arquitetura das aplicações *Web* da Glintt HS e à restrição que indica que não se pode fazer grandes alterações ao sistema que atualmente existe, a solução encontrada passa pela implementação de um servidor em Node.js. Este servidor Node.js funciona como um módulo que pode ser adicionado em todas as instituições hospitalares uma vez que a sua adição vai fazer com que as aplicações *Web* da instituição hospitalar onde foi colocado o servidor passem a utilizar WebSockets em vez de *polling* para atualizar a informação que mostram no ecrã em tempo real. As ligações WebSocket estabelecidas entre as aplicações *Web* e o servidor Node.js vão ser feitas com o auxílio da biblioteca Socket.io.

4.2 Arquitetura Física

Quando se compara a Figura 3.1 com a Figura 4.1, pode-se observar que a arquitetura do sistema continua a ser constituída por: um ou mais computadores, um ou mais servidores das aplicações *Web*, um ou mais servidores da API e um ou mais servidores das bases de dados.

No entanto, para além destes quatro elementos, irá surgir um novo elemento, um servidor desenvolvido em Node.js, que vai permitir que as aplicações *Web* possam receber as atualizações da informação em tempo real através de WebSockets.

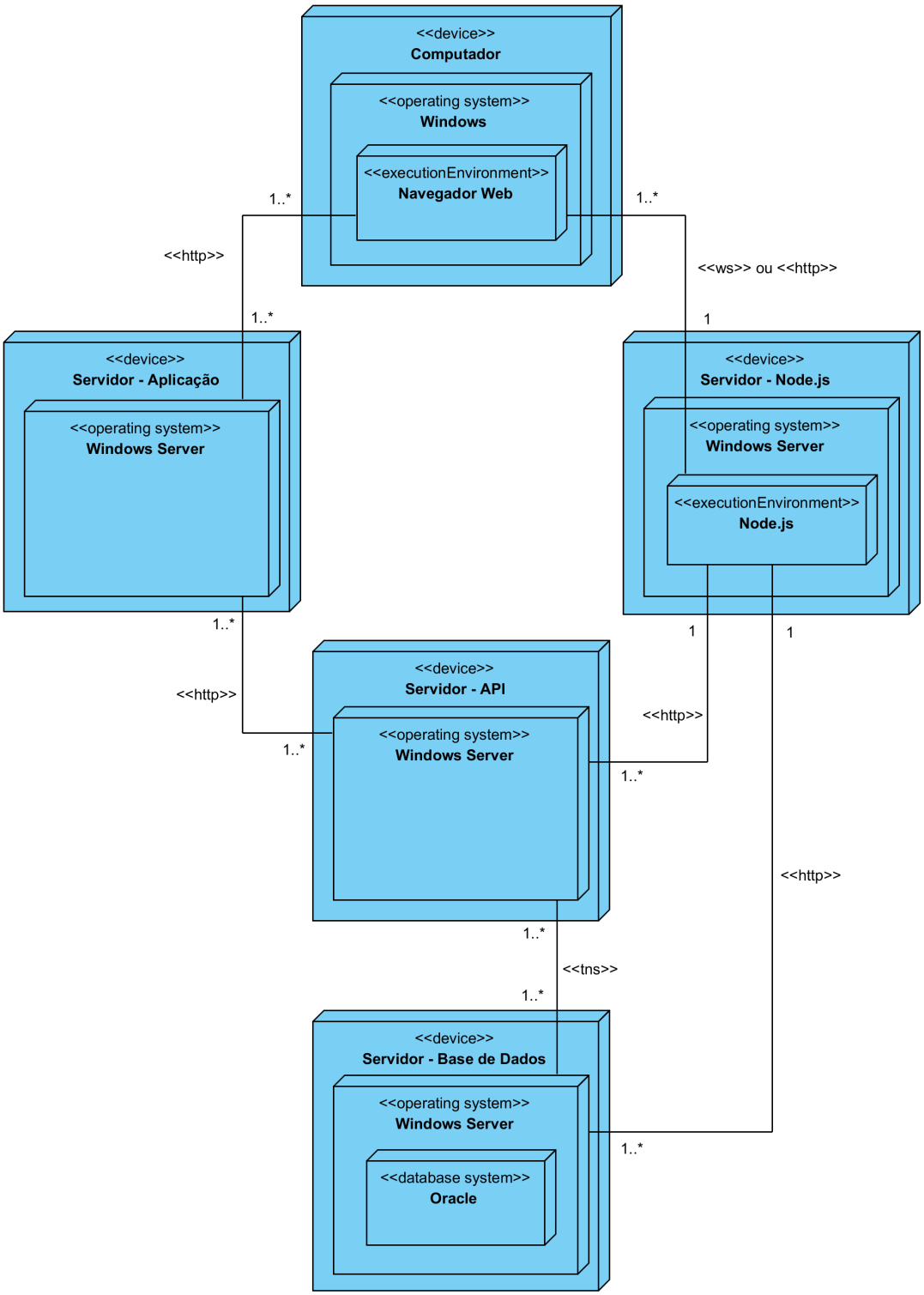

Figura 4.1: Arquitetura da solução

Na Figura 4.1, a relação entre o computador e o servidor Node.js tem uma multiplicidade de 1, uma vez que por mais aplicações *Web* que estejam a correr no mesmo ou em vários computadores, elas acedem sempre ao mesmo servidor Node.js. A relação inversa, a relação entre o servidor Node.js e o computador, tem uma multiplicidade de 1..*, visto que o servidor Node.js pode estar ligado a várias aplicações *Web* que estejam a correr em mais do que um computador. A relação entre o servidor Node.js e o servidor da base de dados apresenta uma multiplicidade de 1..*, uma vez que, dependendo do número de servidores das bases de dados e do número de aplicações *Web* presentes em cada instituição hospitalar, o servidor Node.js pode ter que comunicar com mais do que um servidor da base de dados. A relação entre o servidor da base de dados e o servidor Node.js têm uma multiplicidade de 1 porque, sempre que houver alguma alteração, que seja considerada relevante, na informação guardada nas bases de dados (este processo será explicado com mais detalhe na secção de Implementação), o servidor que aloja a base de dados alterada comunica a alteração ao servidor Node.js. A relação entre o servidor Node.js e o servidor da API tem uma multiplicidade de 1..*, uma vez que o servidor Node.js pode ter que, dependendo do tipo de informação que tem de ser enviada à aplicação *Web*, aceder a mais do que um servidor da API para a conseguir. A relação inversa, a relação entre o servidor da API e o servidor Node.js tem uma multiplicidade de 1, visto que os servidores da API só trocam dados com um único servidor Node.js.

Com esta nova arquitetura, foram adicionadas novas ligações que permitem com que o servidor Node.js consiga comunicar com os restantes elementos do sistema. Com esta alteração, o servidor Node.js comunica com as aplicações *Web* (através do navegador *Web* instalado no computador) através do protocolo WebSocket[1] (*«ws»*). A forma de comunicação utilizada pelo servidor Node.js para comunicar com os servidores da API e os servidores das bases de dados é através de mensagens segundo o protocolo HTTP.

4.3 Arquitetura tecnológica

Esta secção permite explicar detalhadamente qual a nova técnica que vai permitir atualizar a informação em tempo real e as novas tecnologias que vão ser utilizadas para que a arquitetura descrita na secção anterior consiga ser implementada com sucesso.

4.3.1 WebSockets

Esta subsecção permite explicar, com algum pormenor, a nova técnica que vai ser utilizada para atualizar a informação das aplicações *Web* em tempo real e quais as razões que levaram a que fosse escolhida.

[1] Como pode ser observado na Figura 4.1, a troca de mensagens entre as aplicações *Web* e o servidor Node.js pode ser feita através do protocolo WebSocket ou através do protocolo HTTP. Teve de se considerar as duas hipóteses uma vez que o Socket.io utiliza mensagens segundo o protocolo HTTP para atualizar a informação das aplicações *Web* que estejam a ser executadas num navegador *Web* que não suporte WebSockets (ver Subsecção 4.3.3).

4.3.1.1 Definição

A tecnologia WebSocket, especificada no *Request For Comments* (RFC) 6455: The WebSocket Protocol[2], permite a criação de um canal de comunicação bidirecional, persistente e dedicado sobre um único socket TCP. Isto significa que a comunicação entre cliente e servidor pode ocorrer nos dois sentidos, em simultâneo e assincronamente, e que a ligação fica aberta até que uma das partes envie, explicitamente, um pedido de fecho de ligação [FM11]. Graças aos WebSockets, não é necessário o envio de um pedido por parte do cliente para que o servidor envie a informação atualizada para o cliente.

> "The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers." [FM11]

O protocolo WebSocket foi desenvolvido para que clientes (navegadores *Web*) e servidores conseguissem comunicar entre si através de WebSockets. O protocolo WebSocket pode ser considerado como uma camada aplicacional sobre o TCP, adicionando:

- Um modelo de segurança;
- Um esquema de nomes e endereçamento para suportar múltiplos serviços numa única porta e múltiplos nomes de servidores num único endereço IP;
- Um mecanismo de enquadramento;
- Uma nova forma de encerrar ligações.

Atendendo aos requisitos da *Web*, o objetivo é tornar a utilização dos WebSockets o mais parecida possível com o TCP padrão.

Apesar de ser uma tecnologia recente, os WebSockets já são utilizados em vários sistemas e em várias plataformas. De seguida, serão apresentados alguns exemplos da utilização que é dada aos WebSockets:

- Imagens 2D e 3D — Criação de um sistema remoto de visualização interativa 2D e 3D de dados médicos na Internet [MAdSP15]; interação com gráficos 3D num navegador *Web* [KPM13]; criação de um modelo para aplicações com múltiplos ecrãs [BTS+13]; transmissão e visualização de dados vetoriais (SVG) [CMW11]; criação de um sistema de troca de mensagens pictográfico para o *Sistema de Comunicação Alternativa para o Letramento de pessoas com Autismo* (SCALA) [Ram13];

[2]Este texto contém todas as regras que devem ser respeitadas para implementar um servidor ou um cliente baseado no protocolo WebSocket. O RFC 6455 foi publicado em dezembro de 2011, pela *Internet Engineering Task Force* (IETF).

- Acesso remoto — Criação de aplicações *Web* que permitam a monitorização, configuração e atualização remota [FJH11]; visualização remota de informação [WPJR11]; acesso em tempo real a um sensor de vento [PN12]; monitorização, em tempo real, da saúde de um paciente [ZSST13, LD14];
- Prospeção de dados — Suporte a um sistema distribuído de *data mining* [LC11];
- Comunicação entre navegadores *Web* — Implementação de um sistema de comunicação em tempo real, entre navegadores *Web*, segundo as arquiteturas *peer-to-peer* e Session Initiation Protocol (SIP) [Qui13];
- Sistemas baseados em eventos — Sistema de comunicação baseado no modelo *publish-subscribe* [Fel12];
- Interfaces *Web* — Criação de uma plataforma *data binding* que permita a criação de aplicações *Web* [HG12].

4.3.1.2 Protocolo WebSocket

O protocolo WebSocket é definido na camada aplicacional e é baseado no modelo cliente-servidor. Este protocolo suporta o envio de dados binários ou de texto simples.

4.3.1.2.1 Pedido inicial

Todas as ligações WebSocket começam com um pedido HTTP. Este pedido inicial (ou *handshake*) difere dos restantes pedidos HTTP, uma vez que inclui no cabeçalho um pedido de atualização (ou *upgrade*). Esta alteração indica que o cliente pretende atualizar a sua ligação para um protocolo diferente. Neste caso, o protocolo deixa de ser HTTP e passa a ser WebSocket.

Na Figura 2.7 é possível observar as várias fases de uma ligação WebSocket, desde a criação de um canal TCP, passando pelo pedido de atualização do cliente até ao encerramento da ligação WebSocket.

No cabeçalho do *handshake* aparece o campo *origin* que indica o URL do cliente que quer estabelecer uma ligação WebSocket. Este campo pode ser utilizado para diferenciar ligações entre diferentes servidores e ligações feitas a partir de navegadores *Web* ou outro tipo de clientes. Contudo, não convém esquecer que este campo é apenas uma indicação, não muito segura, visto que os clientes que não sejam navegadores *Web* podem colocar qualquer valor neste campo.

A Figura 4.2 mostra um excerto de um pedido inicial feito pelo cliente. Até este pedido estar completo (processo de mudança de protocolo), a sessão WebSocket respeita o protocolo HTTP/1.1.

A Figura 4.3 mostra a resposta do servidor ao *handshake* enviado pelo cliente.

Para que o pedido inicial possa ser completado com sucesso, o servidor de tem provar ao cliente que recebeu o seu pedido de ligação e que aceita ligações WebSocket. O campo do cabeçalho *Sec-WebSocket-Accept* é enviado pelo servidor para o cliente para confirmar que está recetivo em

```
GET ws://echo.websocket.org/?encoding=text HTTP/1.1
Host: echo.websocket.org
Connection: Upgrade
Upgrade: websocket
Origin: http://www.websocket.org
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: nGfr3LO9jRVl23h8SWt7xw==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

Figura 4.2: Pedido inicial enviado pelo cliente

iniciar uma ligação WebSocket. Esta chave, com 20 B e codificados em *base-64*, resulta da concatenação da *Sec-WebSocket-Key* (chave aleatória de 16 B, em *base-64*, enviada pelo cliente) com o Identificador Único Global fixo – "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" – definido na especificação do protocolo WebSocket (os servidores que suportem ligações WebSocket tem a obrigatoriedade de conhecer este identificador). Em seguida, é gerado o *hash* SHA1 da sequência de carateres concatenados com uma posterior codificação em *base-64*. Na Figura 4.4 é possível observar o processo de geração, a partir da *Sec-WebSocket-Key* enviada pelo cliente, da chave guardada no cabeçalho *Sec-WebSocket-Accept*. A Figura 4.5 mostra o código JavaScript que gera a chave guardada no cabeçalho *Sec-WebSocket-Accept*.

```
HTTP/1.1 101 Web Socket Protocol Handshake
Connection: Upgrade
Date: Fri, 12 Jun 2015 12:27:55 GMT
Sec-WebSocket-Accept: nsg+tDxgQD7aBEzLXZCeKAlZN+Q=
Server: Kaazing Gateway
Upgrade: websocket
```

Figura 4.3: Resposta do servidor ao pedido inicial

```
Sec-WebSocket-Key: nGfr3LO9jRVl23h8SWt7xw==
Sec-WebSocket-Key + GUID: nGfr3LO9jRVl23h8SWt7xw==258EAFA5-E914-
                          47DA-95CA-C5AB0DC85B11
Hash SHA-1: 0xB3 0x7A 0x4F 0x2C 0xC0 0x62 0x4F 0x16
            0x90 0xF6 0x46 0x06 0xCF 0x38 0x59 0x45
            0xB2 0xBE 0xC4 0xEA
Sec-WebSocket-Accept: nsg+tDxgQD7aBEzLXZCeKAlZN+Q=
```

Figura 4.4: Geração da chave de resposta (*Sec-WebSocket-Accept*)

Esta troca de chaves não oferece qualquer proteção aos clientes ou aos servidores que estabelecem ligação WebSocket, uma vez que são facilmente intersetáveis. Este mecanismo pretende proteger os servidores que não suportem ligações WebSocket e eliminar a possibilidade de ataques de protocolos cruzados.

```
var KEY_SUFFIX = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
var hashWebSocketKey = function(key)
{
    var sha1 = crypto.createHash("sha1");
    sha1.update(key + KEY_SUFFIX, "ascii");

    return sha1.digest("base64");
}
```

Figura 4.5: Código JavaScript que permite gerar a chave *Sec-WebSocket-Accept* [WSM13]

Durante a troca de mensagens HTTP, entre o cliente e o servidor, que podem levar ao estabelecimento de uma ligação WebSocket, podem surgir os seguintes campos nos cabeçalhos das mensagens [The14, WSM13]:

- *Sec-WebSocket-Extensions* — Este campo, que pode ser utilizado no cabeçalho do pedido inicial e no cabeçalho da resposta do servidor ao pedido inicial, ajuda o cliente e o servidor a definirem um conjunto de extensões que vão ser utilizadas durante a ligação WebSocket;
- *Sec-WebSocket-Protocol* — Este campo pode ser utilizado no cabeçalho do pedido inicial, uma vez que permite, ao cliente, informar o servidor sobre quais os "subprotocolos" – por exemplo *Extensible Messaging and Presence Protocol* (XMPP), *Simple Object Access Protocol* (SOAP) e *Streaming Text Oriented Messaging Protocol* (STOMP) – que suporta. Na resposta ao pedido inicial, o servidor pode utilizar este campo para indicar ao servidor qual o protocolo que vai ser utilizado durante a ligação. Ao contrário do que acontece com as extensões em que podem ser usadas várias extensões por cada ligação, só pode ser utilizado um "subprotocolo" por cada ligação;
- *Sec-WebSocket-Version* — Este campo é utilizado no cabeçalho do pedido inicial, visto que permite, ao cliente, informar o servidor sobre qual a versão do protocolo WebSocket que pretende usar. A versão do protocolo que vem definida no RFC 6455 é a 13. Contudo, se na resposta do servidor ao pedido inicial, o campo *Sec-WebSocket-Version* estiver incluído no cabeçalho significa que o servidor não suporta a versão pedida pelo cliente. Na resposta do servidor, este campo envia ao cliente quais são as versões do protocolo que o servidor suporta. Atualmente, este tipo de disparidades entre versões só acontece se o cliente for mais antigo que o RFC 6455 (dezembro de 2011).

4.3.1.2.2 Formato das mensagens

Durante o tempo em que uma ligação WebSocket se mantém ativa, o cliente e o servidor podem trocar mensagens entre si em qualquer altura.

A Figura 4.6 representa o formato de uma mensagem WebSocket [The14, WSM13]:

- *Opcode* — Indica o tipo da mensagem, só contam os últimos 4 b do primeiro *byte*. O *opcode* pode assumir os seguintes valores:

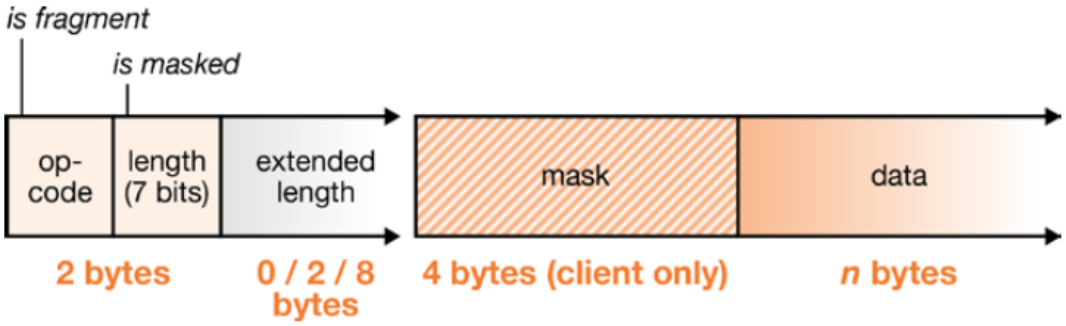

Figura 4.6: Formato de uma mensagem WebSocket [WSM13]

  - Texto:
    - Valor do *opcode*: 1;
    - Descrição: O tipo de dados da mensagem é texto. Quando o tipo de mensagem é texto, a mensagem é codificada em UTF-8 para não haver problemas de codificação entre os clientes e os servidores.
  - Binária:
    - Valor do *opcode*: 2;
    - Descrição: O tipo de dados da mensagem é binário.
  - Terminar ligação:
    - Valor do *opcode*: 8;
    - Descrição: O cliente ou o servidor enviam um pedido ao servidor ou ao cliente para terminar a ligação.
  - *Ping*[3]:
    - Valor do *opcode*: 9;
    - Descrição: O cliente ou o servidor enviam um *ping* ao servidor ou ao cliente.
  - *Pong*[4]:
    - Valor do *opcode*: 10;
    - Descrição: O cliente ou o servidor enviam um *pong* ao servidor ou ao cliente.

- *Mask* — O primeiro *bit* do segundo *byte* indica se a mensagem está *mascarada* (não se consegue perceber qual é o conteúdo da mensagem). Atualmente, o protocolo WebSocket exige que os clientes mascarem todas as mensagens;

- *Length* — Indica o tamanho da mensagem. A forma com que o valor do tamanho da mensagem é guardado no cabeçalho da mensagem depende do seu tamanho:

[3] A troca de *pings* e *pongs* entre o cliente e o servidor permite manter a ligação WebSocket ativa.
[4] Ver *footnote* 3.

  - Menor que 126 B: O tamanho da mensagem está contido num dos primeiros 2 B do cabeçalho;
  - Entre 126 B e 216 B: São utilizados 2 B extra;
  - Maior que 216 B: São utilizados 8 B extra.

A técnica de mascarar as mensagens transforma o conteúdo de cada mensagem enviada pelo cliente fazendo o XOR do *byte n* da mensagem com o módulo entre o *byte n* e 4, de um conjunto aleatório de 32 b. O propósito é facilitar a passagem das mensagens por *proxies* (encobrindo, não cifrando, o conteúdo das mensagens) e evitar ataques de envenenamento da cache HTTP. Neste tipo de ataques, o atacante assume controlo de uma memória cache HTTP, alterando o seu conteúdo.

Geralmente só existe um *frame* por mensagem contudo, podem existir mensagens que sejam compostas por vários *frames*. Apesar de algumas mensagens, na altura do envio, ficarem divididas em vários *frames*, a mensagem só pode ser acedida pelo cliente ou pelo servidor quando estes tiverem recebido todos os *frames* associados à mensagem. Para transmitir uma mensagem, através de vários *frames*, é necessário que todos os *frames*, desde o primeiro até ao penúltimo, tenham o primeiro *bit* do primeiro *byte* a zero. O cliente ou o servidor ficam a saber se um *frame* é o último de uma determinada mensagem quando o primeiro *bit* do primeiro *byte* do *frame* recebido está a um [The14, WSM13].

4.3.1.2.3 Encerramento da ligação

O protocolo WebSocket permite, aquando do encerramento de uma ligação WebSocket, que um cliente ou um servidor informem, através de um código e de uma *string* (a *string* permite dar mais detalhes sobre a razão que levou ao encerramento da ligação), os servidores ou os clientes com quem estavam ligados sobre qual o motivo que levou ao encerramento da ligação.

Para que o cliente ou o servidor, que terminou a ligação, possam informar os servidores ou os clientes terão de enviar um código, representado como um inteiro sem sinal de 16 b, e uma *string*, com uma codificação UTF-8, num *frame* com um *opcode* de 8. A especificação do protocolo WebSocket define um conjunto de códigos que variam consoante o motivo que levou ao encerramento da ligação [The14, WSM13]:

- 1000 (*Normal Close*) — Este código é enviado quando a sessão estiver completa;
- 1001 (*Going Away*) — Este código é enviado quando a ligação é encerrada e não há a possibilidade de haver uma nova ligação. Normalmente, acontece quando o servidor é desligado ou quando a aplicação *Web* é fechada;
- 1002 (*Protocol Error*) — Este código é enviado quando a ligação é encerrada devido a um erro no protocolo;
- 1003 (*Unacceptable Data Type*) — Este código é enviado quando a aplicação *Web* recebe uma mensagem de um tipo que não consegue suportar;

- 1004 (*Reserved*) — Não se pode enviar este código. De acordo com o RFC 6455, este código está reservado e deve de ser definido no futuro;
- 1005 (*Reserved*) — Não se pode enviar este código. A WebSocket API utiliza este código para informar que não foi recebido nenhum código;
- 1006 (*Reserved*) — Não se pode enviar este código. A WebSocket API utiliza este código para informar que houve alguma coisa fora do normal que levou ao encerramento da ligação;
- 1007 (*Invalid Data*) — Este código é enviado quando o cliente ou o servidor recebe uma mensagem que não corresponde ao tipo especificado na mesma;
- 1008 (*Message Violates Policy*) — Este código é enviado quando a aplicação *Web* termina a ligação por alguma razão que não esteja associada a um outro código ou quando não se pretende divulgar a *string* que explica o motivo que levou ao encerramento da ligação;
- 1009 (*Message Too Large*) — Este código é enviado quando a mensagem recebida é tão grande que a aplicação *Web* não a consegue processar;
- 1010 (*Extension Required*) — O cliente (navegador *Web*) envia este código quando a aplicação *Web* necessita de extensões que não foram negociadas com o servidor;
- 1011 (*Unexpected Condition*) — Este código é enviado quando a aplicação *Web* não consegue manter a ligação ativa devido a algum imprevisto;
- 1015 (*Transport Layer Security* (TLS) *Failure (reserved)*) — Não se pode enviar este código. A WebSocket API utiliza este código para informar que o TLS falhou antes do pedido inicial.

4.3.1.3 WebSocket API

A WebSocket API, suportada pela maioria dos navegadores *Web* mais recentes, é uma interface que permite às aplicações *Web* a utilização do protocolo WebSocket. Esta API foi desenvolvida pela *World Wide Web Consortium* (W3C) e vem facilitar a execução de operações como iniciar ou terminar ligação, enviar e receber mensagens e permanecer à escuta de eventos que possam vir a ser espoletados pelo servidor.

4.3.1.3.1 Construtor

Para que uma ligação WebSocket possa ser estabelecida, é necessário, como pode ser observado na Figura 4.7, começar por criar uma nova instância de um objeto WebSocket. O protocolo WebSocket define dois esquemas URL: *ws://*[5] e *wss://*[6], para o tráfego, entre clientes e servidores, não encriptado e encriptado, respetivamente. Quando se pretende utilizar *ws*, o pedido inicial

[5] Protocolo WebSocket.
[6] Protocolo WebSocket com uma camada de segurança que, tal como o HTTPS, utiliza o protocolo SSL/TLS.

```
var ws = new WebSocket("ws://wwww.websocket.org");
```

Figura 4.7: Construtor WebSocket

utiliza o protocolo HTTP. Para utilizar *wss*, o pedido inicial tem de utilizar o protocolo HTTPS para estabelecer uma ligação WebSocket com o servidor.

O construtor recebe o URL do servidor enquanto argumento obrigatório e pode incluir um argumento opcional: uma lista de "subprotocolos" que são suportados pelo cliente. O servidor, na resposta ao pedido inicial, terá de informar qual o "subprotocolo" que pretende utilizar.

Na Figura 4.8, é possível observar um exemplo de um construtor WebSocket que inclui como argumentos o endereço do servidor e uma lista com os "subprotocolos" suportados pelo cliente [The14, WSM13].

4.3.1.3.2 Eventos

A WebSocket API é baseada em eventos, o que significa que as aplicações *Web* apenas têm de se manter à escuta para ver se chegou alguma mensagem nova ou se existiu alguma alteração no estado da ligação.

Os objetos WebSocket reconhecem quatro tipos de eventos [The14, WSM13]:

- *Open* — Informa que a ligação foi estabelecida com o servidor e que a troca de mensagens pode começar;
- *Message* — Este evento é espoletado sempre que uma mensagem é recebida;
- *Error* — Este evento ocorre quando existe alguma falha na ligação (geralmente, a ligação termina depois deste evento);
- *Close* — Este evento é espoletado quando a ligação é terminada.

```
var ws = new WebSocket("ws://wwww.websocket.org",
    ["com.kaazing.echo", "custom.my.protocol"]);

ws.onopen = function(e)
{
    /*
        Quando a ligação WebSocket é estabelecida, o
        cliente verifica o "subprotocolo" escolhido pelo
        servidor
    */

    console.log(ws.protocol);
}
```

Figura 4.8: Construtor WebSocket com uma lista dos "subprotocolos" suportados pelo cliente

```
function setup()
{
    var ws = new WebSocket ("ws://www.websocket.org");

    ws.onopen = function(e)
    {
        console.log("Ligação estabelecida!");
    }

    ws.onclose = function(e)
    {
        console.log("Ligação terminada: " + e.reason);
    }

    ws.onerror = function(e)
    {
        console.log("Erro!");
    }

    ws.onmessage = function(e)
    {
        console.log("Mensagem recebida: " + e.data)
    }
}
```

Figura 4.9: Eventos WebSocket

A Figura 4.9 mostra um exemplo de utilização das funções de retorno correspondentes a cada um dos eventos atrás referidos.

4.3.1.3.3 Métodos

Depois de estabelecida a ligação WebSocket entre o cliente e o servidor, é possível invocar os seguintes métodos [The14, WSM13]:

- *send(data)* — Serve para enviar mensagens de texto ou dados binários para o servidor enquanto a ligação WebSocket está ativa;
- *close()* — É invocado quando o cliente ou o servidor quiserem terminar uma ligação WebSocket.

A Figura 4.10 mostra um exemplo da utilização destes dois métodos.

4.3.1.3.4 Atributos dos objetos WebSocket

A WebSocket API oferece três atributos dos objetos WebSocket que são bastante úteis para se perceber em que estado a ligação se encontra [The14, WSM13]:

- *readyState* — Este atributo retorna um inteiro que indica em que estado se encontra a ligação. Este atributo pode devolver os seguintes valores:

  - WebSocket.CONNECTING
    - Valor do *readyState*: 0;
    - Descrição: A ligação WebSocket ainda não foi totalmente estabelecida com sucesso.
  - WebSocket.OPEN
    - Valor do *readyState*: 1;
    - Descrição: A ligação WebSocket está estabelecida. A partir deste momento, clientes e servidores podem trocar mensagens entre si.
  - WebSocket.CLOSING
    - Valor do *readyState*: 2;
    - Descrição: A ligação WebSocket está a ser encerrada (está a decorrer o processo de encerramento da ligação).
  - WebSocket.CLOSED
    - Valor do *readyState*: 3;
    - Descrição: Significa que a ligação WebSocket já foi fechada ou que a ligação não pode ser aberta.

- *bufferedAmount* — Caso seja necessário enviar grandes quantidades de informação, este atributo indica a quantidade de *bytes* que ainda não foram enviados ao servidor;
- *protocol* — Este atributo devolve o nome do "subprotocolo" escolhido pelo servidor durante o pedido inicial.

```
// Enviar uma mensagem de texto
ws.send("Hello World!");

// Terminar uma ligação
ws.close();
```

Figura 4.10: Métodos WebSocket

4.3.1.4 Motivação

Das técnicas que foram apresentadas no Capítulo 2, a escolha recaiu sobre os WebSockets devido a cinco fatores:

- Desempenho e largura de banda — Comparativamente a outras técnicas, como o *polling* e o *long polling* em que o cliente tem de estar a enviar pedidos constantemente, no protocolo WebSocket, o cliente apenas necessita de abrir uma ligação WebSocket com o servidor e esperar que a informação chegue. Para além disso, quase todas as técnicas referidas no

Capítulo 2 (exceto os *plug-ins*) utilizam o protocolo HTTP nas mensagens trocadas entre o cliente e o servidor. O problema na utilização deste protocolo prende-se com a quantidade de dados que se encontram nos cabeçalhos das mensagens HTTP, que é muito superior à quantidade de dados utilizados nos cabeçalhos das mensagens utilizadas nas ligações WebSocket. Estas duas diferenças fazem com que o protocolo WebSocket, comparativamente às outras técnicas, tenha um melhor desempenho e necessite de uma menor largura de banda [LG13];

- Comunicação bidirecional numa única ligação — Nas outras técnicas, são necessárias duas ligações, uma no sentido cliente-servidor e outra no sentido inverso, para simular uma ligação bidirecional. A existência de duas ligações faz com que haja um aumento do consumo de recursos computacionais e da complexidade na arquitetura do sistema [LG13];
- Eficiência no consumo de recursos computacionais — Quando comparado com *long polling* ou com o *streaming*, os WebSockets fazem uma melhor gestão da memória e da utilização do CPU [She12];
- Suporte nativo — Atualmente, a técnica que os navegadores *Web* mais recentes melhor suportam, de forma nativa, é o protocolo WebSocket [She12];
- Evita *proxies* e *firewalls* — Ao contrário do que acontece, por exemplo, no *streaming*, as mensagens trocadas através de WebSockets não são afetadas pelos *proxies* ou pelas *firewalls* que estejam localizados entre o cliente e o servidor [WSM13].

4.3.2 Node.js

O Node.js é um interpretador de código JavaScript baseado na tecnologia V8 desenvolvida pela Google para interpretar código JavaScript no Google Chrome. O interpretador V8, desenvolvido em C/C++, permite interpretar código JavaScript para código máquina. Apesar de os códigos máquina serem diferentes de processador para processador, o interpretador V8 tem suporte para vários tipos de processadores:

- IA-32;
- x86-x64;
- ARM;
- MIPS.

Tal como pode ser observado nas Figuras 4.11 e 4.12, o Node.js, ao contrário de outras tecnologias como o PHP, o Java ou o .Net, tem uma arquitetura assente em eventos e uma API *asynchronous* I/O, o que permite que, ao contrário do que acontece no *multithreading*, o servidor não fique bloqueado à espera que um processo termine para que um outro possa ser executado.

As vantagens do Node.js que levaram a que o servidor tivesse sido implementado com esta tecnologia são [HCW12]:

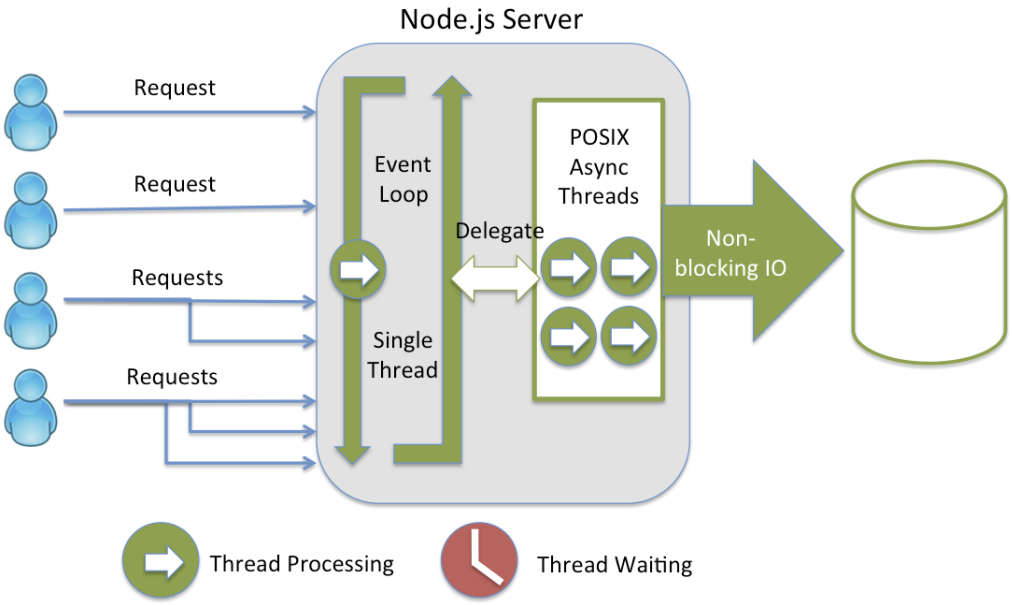

Figura 4.11: Servidor com arquitetura assente em eventos [Rot14]

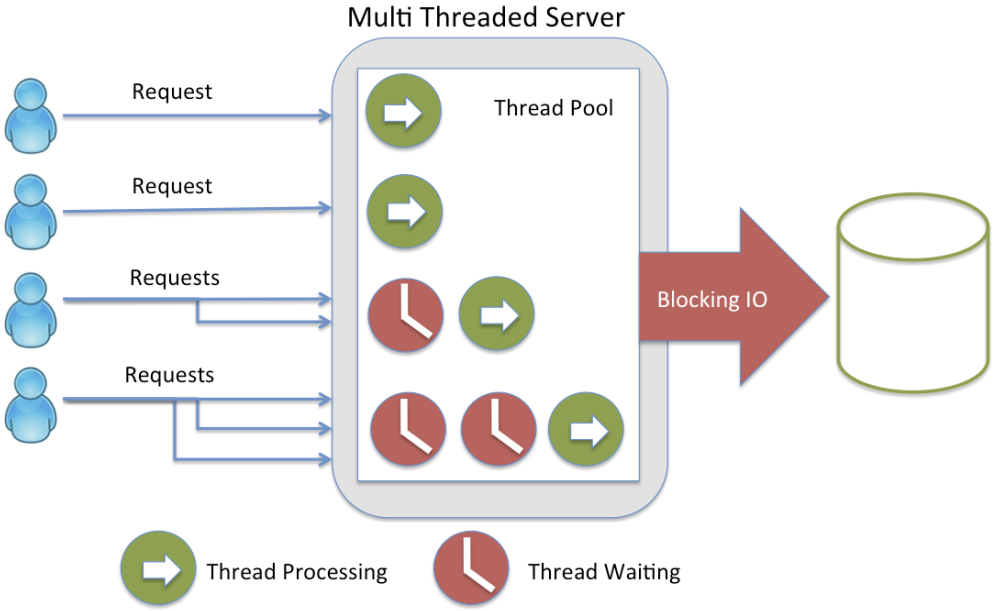

Figura 4.12: Servidor com arquitetura assente em *multithreading* [Rot14]

- A utilização de JavaScript no desenvolvimento de servidores *Web* permite a partilha de código entre o cliente e o servidor;
- Os elevados níveis de desempenho comparativamente a outras tecnologias tais como o PHP, o Java ou o .Net [Fas13, Han14, Rot14, San15, Yer14];
- A criação de servidores altamente escaláveis;
- A redução da complexidade para os programadores, uma vez que com o Node.js não existe o problema de sincronizar processos que estejam a correr em diferentes *threads*;
- A grande quantidade de módulos que existem, incluindo o Socket.io, faz com que o Node.js seja extremamente extensível;
- A comunidade pró-ativa que existe à volta do Node.js, o que faz com que qualquer problema que exista seja prontamente corrigido.

4.3.3 Socket.io

Socket.io é uma biblioteca que, com base num servidor desenvolvido em Node.js, permite que as aplicações *Web* possam criar um canal de comunicação bidirecional (como o que é utilizado pelos *sockets*) com o servidor de forma a poderem receber a informação em tempo real.

Os motivos que levaram à escolha da biblioteca Socket.io foram os seguintes [HCW12]:

- Das bibliotecas que existem, o Socket.io é uma das bibliotecas mais rápidas e mais fiáveis a trocar mensagens entre as aplicações *Web* e o servidor;
- A biblioteca Socket.io é conhecida pela rapidez e fiabilidade das suas ligações WebSocket, no entanto, também, funciona em navegadores *Web* que não suportem WebSockets. Nos navegadores *Web* que não suportam WebSockets, as aplicações *Web* trocam mensagens com o servidor através de AJAX *Long Polling*. Esta característica pesou bastante na escolha desta biblioteca para implementar as ligações WebSocket, uma vez que em muitas instituições hospitalares é utilizado um navegador *Web* que não suporta WebSockets, o Internet Explorer 9;
- A forma com que se utiliza a biblioteca nas aplicações *Web* e no servidor é muito parecida, o que pode acabar por facilitar a partilha de código entre os clientes e os servidores. Através da Figura 4.13 e da Figura 4.14, ficam bem visíveis as similaridades na forma com que o código, relacionado com o Socket.io, é escrito tanto nas aplicações *Web* como no servidor;
- O facto de que para haver comunicação, entre as aplicações *Web* e o servidor, através da biblioteca Socket.io, é necessário, tal como ilustra a Figura 4.15, a criação de um servidor HTTP. A criação deste servidor acaba por ser bastante útil no desenvolvimento do servidor Node.js, uma vez que permite que o servidor para onde a Oracle envia as indicações seja o mesmo servidor que é utilizado para estabelecer as ligações WebSocket com as aplicações *Web*.

```
var socket = io ();

socket.emit('greeting', 'Hello, my name is Billy');
socket.on('message', function(msg)
{
  console.log(msg);
});
```

Figura 4.13: Socket.io – versão cliente

```
io.on('connection', function(socket)
{
  socket.on('greeting', function(text)
  {
      console.log(text);
      socket.emit('message', 'glad to see you');
  });
});
```

Figura 4.14: Socket.io – versão servidor

4.4 Resumo

Com base no problema descrito no capítulo anterior e com a existência da restrição de que não se pode fazer grandes alterações ao sistema que atualmente existe, a solução passa por adicionar um módulo em cada instituição hospitalar. Este módulo vai ser um servidor desenvolvido em Node.js que vai permitir que as aplicações *Web*, que estejam a correr na mesma instituição hospitalar do que o servidor Node.js, possam receber informação atualizada através de WebSockets. As ligações WebSockets que são estabelecidas entre as aplicações *Web* e o servidor Node.js são feitas com o auxílio da biblioteca Socket.io.

Os WebSockets vieram substituir o *polling* como técnica para atualizar a informação em tempo real porque permitem um maior desempenho, a largura de banda que utilizam é menor comparativamente ao *polling*, conseguem criar uma comunicação bidirecional e *full duplex* através de uma única ligação, são mais eficientes no consumo de recursos computacionais, são uma técnica com suporte nativo na grande maioria dos navegadores *Web* atuais e conseguem evitar *proxies* e *firewalls* (são elementos que aumentam a latência das mensagens).

O servidor é desenvolvido em Node.js dado que permite a partilha de código JavaScript entre as aplicações *Web* e o servidor, tem elevados níveis de desempenho quando comparado a outras linguagens, permite criar servidores altamente escaláveis e é bastante extensível devido à grande quantidade módulos que existem, entre eles, o Socket.io.

As ligações WebSockets são estabelecidas com o auxílio da biblioteca Socket.io visto que o Socket.io é uma das bibliotecas mais rápidas a trocar mensagens entre as aplicações *Web* e

```
var http = require('http');
var io_server = require('socket.io');

var server = http.createServer(function(req, res)
{
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World!');
});

server.listen(1337, '127.0.0.1');
console.log('Servidor a correr no http://127.0.0.1:1337');

/*
    Criação do servidor para funcionar com a biblioteca
    Socket.io
*/
var io = new io_server(server);
```

Figura 4.15: Criação de um servidor HTTP com Socket.io

os servidores, a forma com que se processa a troca de mensagens nas aplicações *Web* é igual à forma como as mensagens são processadas no servidor e o Socket.io garante portabilidade com navegadores *Web* que não suportem WebSockets – a comunicação é feita através de *long polling*.

Capítulo 5

Implementação

Este capítulo explica de que forma foi implementada a solução descrita no capítulo anterior. A primeira secção serve para explicar quais as alterações que têm de ser feitas na base de dados Oracle para que a solução possa ser concretizada. A segunda secção explica de que forma é que o servidor Node.js tem de funcionar para receber as indicações da Oracle e para enviar a informação atualizada para as aplicações *Web*. Na terceira secção, estão escritas as alterações que têm de ser feitas às aplicações *Web* para que estas consigam comunicar com o servidor Node.js. Na última secção, é feito um pequeno resumo acerca das três secções anteriores.

5.1 Componente Oracle

Tal como foi referido na última secção do último capítulo, durante o processo de *polling*, o servidor da API devia devolver às aplicações *Web* os dados que foram alterados durante o intervalo de tempo que existe entre dois pedidos HTTP, contudo não é isso que acontece. Em vez disso, de cada vez que uma aplicação *Web* envia um pedido HTTP, o servidor da API devolve toda a informação quer esta esteja, ou não, atualizada.

Dado que o *polling* envia toda a informação às aplicações *Web*, ao sistema da Glintt HS, faltava uma forma de conseguir registar as alterações que estão a acontecer. A fim de registar as alterações da informação, a primeira hipótese passou pela colocação de código na API que permitisse saber o que foi alterado para que se pudesse enviar a informação atualizada para as aplicações *Web*. A API onde o código iria ser colocado não é a mesma API que as aplicações *Web* utilizam para obter a informação, esta API recebe pedidos HTTP para adicionar, alterar ou eliminar a informação. Contudo, esta hipótese teve de ser descartada, uma vez que não foi possível ter acesso a essa mesma API, o que fez com que se tivesse de pensar noutra solução.

A solução encontrada passa por inspecionar, ao nível das bases de dados, as tabelas – tem de ser em tabelas cujo conteúdo seja considerado relevante para as aplicações *Web* – à procura de alterações que possam ocorrer no seu conteúdo. Posto isto, a primeira ideia foi a colocação

de *triggers* nas várias tabelas que interessava inspecionar, uma vez que permitia ter acesso ao conteúdo que fosse adicionado, alterado ou eliminado. O passo seguinte seria a transmissão, para o servidor Node.js, de uma indicação sobre o conteúdo alterado. Assim, o servidor Node.js conseguiria, com base na indicação que recebeu, ir buscar informação mais pormenorizada ao servidor da API – a informação viria de acordo com as regras de negócio da Glintt HS. Depois de obtida a informação, o servidor Node.js apenas teria que enviar a informação para as aplicações *Web*. Contudo, esta ideia não seguiu para a frente devido a três fatores:

- Desempenho da base de dados — O elevado número de tabelas que iriam ser inspecionadas conjugado com a alta frequência com que o conteúdo das tabelas é modificado podia fazer com que o desempenho da base de dados acabasse por diminuir;
- Alterações de código — Devido ao elevado número de tabelas que iriam ser inspecionadas, iriamos acabar por ter um grande número de *triggers* que, apesar de serem todos diferentes, têm a mesma estrutura. O que significa que, para alterar alguma coisa na estrutura ou para corrigir algum erro, que seja comum a todos os *triggers*, é necessário corrigir, um a um, todos os *triggers*;
- *Commit* das alterações — A passagem da informação da Oracle para o servidor Node.js é feita através de mensagens HTTP. Durante o intervalo de tempo que vai desde a altura em que a Oracle envia o pedido HTTP – com a indicação acerca do que foi atualizado no corpo do pedido – ao servidor Node.js até à receção da resposta do Node.js por parte da Oracle, a transação (*insert*, *update* ou *delete*) fica pendente. Isto significa que, durante este período, todas as consultas à tabela da base de dados retornam a informação sem a alteração que espoletou o *trigger*. Caso a troca de mensagens HTTP entre a Oracle e o servidor Node.js dê algum erro, pode fazer com que a alteração, que espoletou o *trigger*, acabe por nem ser registada.

Apesar da utilização dos *triggers* ser uma péssima ideia, a ideia de inspecionar algumas tabelas das bases de dados, para encontrar alterações ao conteúdo das mesmas, continua a ser válida, uma vez que, tal como já foi referido, não é possível aceder à API onde chegam os pedidos HTTP para alterar a informação.

A ideia passa pela utilização do *package* DBMS_CHANGE_NOTIFICATION. Este *package*, disponibilizado pela Oracle, tem uma interface onde é possível registar as tabelas que interessam inspecionar e onde é possível definir o procedimento que vai ser chamado quando existir alguma alteração no conteúdo das tabelas anteriormente registadas. As principais vantagens da utilização deste *package*, em comparação com a utilização dos *triggers*, estão relacionadas com as alterações de código e com os *commits* das alterações. Com este *package*, é possível inspecionar várias tabelas com apenas dois procedimentos, ao contrário do que acontecia com os *triggers* em que era necessário haver um *trigger* por cada tabela. Qualquer alteração de código, que necessite de ser feita à forma com que as tabelas estão a ser inspecionadas, só necessita de ser feita em dois procedimentos e não em dezenas de *triggers* que se encontrem espalhados pelas bases de dados. A outra

vantagem na utilização deste *package* está relacionada com o *commit* das alterações, uma vez que, com este *package*, as alterações que forem feitas às tabelas das bases de dados ficam salvaguardadas independentemente da existência, ou não, de erros que possam ocorrer nos procedimentos gerados pelo *package* DBMS_CHANGE_NOTIFICATION [Ora15, Inf13, Hal15].

A componente Oracle da solução implementada conta com o seguinte *package*:

- PCK_NOTIFICATION_PROCESS — Este *package* agrupa uma série de procedimentos que permitem a manipulação do *package* DBMS_CHANGE_NOTIFICATIONS, a gestão da informação alterada e a construção de um objeto *JavaScript Object Notation* (JSON) que será enviado, no corpo de um pedido HTTP, para o servidor Node.js – o objeto JSON enviado contém a indicação do conteúdo que foi alterado.

Nesta fase inicial, o *package* PCK_NOTIFICATION_PROCESS contém cinco procedimentos:

- *register_notification_code* — Uma vez que os nomes das tabelas que interessam inspecionar vão ficar guardados numa tabela da base de dados, este procedimento vai, dinamicamente, criar um outro procedimento (este procedimento vai estar fora do *package* PCK_NOTIFICATION_PROCESS) que, quando chamado, vai criar o processo que vai inspecionar as tabelas que estão parametrizadas nessa tabela da base de dados;

- *register_notification* — Quando este procedimento é chamado, ele começa por chamar o procedimento *register_notification_code*. Depois de o procedimento *register_notification_code* ter sido executado, o *register_notification* vai chamar o procedimento gerado pelo *register_notification_code*, o procedimento chama-se *register_notification_code_pr*, para que as tabelas registadas no procedimento possam começar a ser inspecionadas. Para além do registo das tabelas, no *register_notification_code_pr*, é indicado qual o procedimento que vai ser chamado quando existir alguma alteração no conteúdo das tabelas registadas. Quando o procedimento *register_notification_code_pr* é executado, um ID é associado ao registo acabado de criar. Este identificador permite, ao administrador da base de dados, gerir quais os registos que se devem, ou não, manter ativos;

- *remove_notification* — Este procedimento permite, passando como parâmetro o ID de um registo, eliminar o registo. Isso significa que as tabelas associadas ao registo eliminado deixaram de estar a ser inspecionadas;

- *procedure_notification* — Este procedimento é o procedimento que é chamado quando o conteúdo de alguma das tabelas que se está a inspecionar é alterado. O parâmetro deste procedimento é um objeto do tipo SYS.chnf$_desc que guarda toda a informação acerca da tabela alterada, da linha alterada e do tipo de alteração que ocorreu (adição, atualização ou remoção);

- *process_handler* — Dos cinco procedimentos que compõem o *package* PCK_NOTIFICATION_PROCESS, este é o que está menos relacionado com o *package* DBMS_CHANGE_NOTIFICATION. Este procedimento está mais relacionado com a gestão da informação

alterada e com a criação do objeto JSON que contém a indicação do que foi alterado e que vai ser enviado para o servidor Node.js. Este procedimento recebe como parâmetro o identificador da alteração que espoletou a chamada deste procedimento. Este procedimento, com base no que foi alterado e na forma com que aconteceu (se através de um *insert*, *update* ou *delete*), começa por verificar se, na instituição hospitalar em que a aplicação *Web* está presente, a aplicação *Web* pode receber a informação alterada. Caso a aplicação *Web* possa receber a informação alterada, este procedimento cria o objeto JSON com a indicação do que foi alterado e chama a função *send_update_node_http* que permite enviar, através de um pedido HTTP, o objeto JSON para o servidor Node.js. Este procedimento só é terminado quando a função *send_update_node_http* tiver recebido a resposta ao seu pedido HTTP.

Para além deste *package*, é necessário criar tabelas para parametrizar informação e para que, no futuro, seja possível ter uma ideia do que foi enviado, quando foi, se na época foi enviado, ou não, com sucesso, etc.. No total, e como pode ser observado na Figura 5.1, foram criadas as seguintes tabelas:

- WEBS_CHANGE_ROWS_NOTIFICATIONS — Esta tabela, chamada no procedimento *procedure_notifications*, vai guardar os detalhes sobre a alteração que fez com que o procedimento *procedure_notifications* fosse chamado. A tabela WEBS_CHANGE_ROWS_NOTIFICATIONS é constituída pelas seguintes colunas:
  - ID (PK)
    - Tipo: NUMBER;
    - Descrição: Obtido através da sequência *webs_seq_rows_notifications*, este valor vai permitir identificar as alterações que estão a acontecer.
  - ROW_ID
    - Tipo: VARCHAR2 (50);
    - Descrição: Esta coluna vai guardar os *row_id* – internamente, a Oracle atribui um identificador único a todas as linhas que componham as bases de dados – das linhas que sofreram uma alteração (um *insert*, um *update* ou um *delete*).
  - TABLE_NAME
    - Tipo: VARCHAR2 (60);
    - Descrição: Esta coluna guarda o nome das tabelas cujo conteúdo foi alterado.
  - COD_APPLICATION
    - Tipo: VARCHAR2 (50);
    - Descrição: Esta coluna é preenchida com um código da aplicação referente a uma das aplicações *Web* em funcionamento. A associação da tabela alterada com o código da aplicação inserido nesta coluna está guardada na tabela WEBS_PARAM_TABLE_APPLICATION.

  - COD_DATA
    - Tipo: VARCHAR2 (100);
    - Descrição: Devido ao facto de numa aplicação *Web* podermos ter uma grande variedade de dados (médicos, pacientes, enfermeiros, entre outros), esta coluna guarda o código do tipo de dados associados à tabela alterada. A associação da tabela alterada com o código do tipo de dados inseridos nesta coluna está guardada na tabela WEBS_PARAM_TABLE_APPLICATION.
  - ACTION
    - Tipo: VARCHAR2 (20);
    - Descrição: Esta coluna guarda a ação que fez com que o procedimento *procedure_notification* fosse chamado. Esta coluna só pode tomar três valores: *insert*, *update* ou *delete*.
  - SEND_INFO_NODE
    - Tipo: VARCHAR2 (1);
    - Descrição: *Flag* que permite saber se a indicação do conteúdo do que foi alterado foi enviado com sucesso para o servidor Node.js ou se houve algum erro (S – enviou com sucesso; N – não enviou; O – o pedido foi enviado sem dados; E – aconteceu um erro).
  - DATE_CRI
    - Tipo: TIMESTAMP;
    - Descrição: Esta coluna guarda as datas em que as alterações ocorreram. O tipo de dados desta coluna é TIMESTAMP e não DATE, uma vez que, com TIMESTAMP, é possível saber, até à ordem dos milissegundos, a data em que o conteúdo de uma tabela foi alterado. A possibilidade de saber as datas até à ordem dos milissegundos permite calcular o tempo que demora desde a atualização na base de dados até à alteração estar visível no ecrã.

- WEBS_PARAM_APPLICATION_ACTION — Utilizada no procedimento *process_handler*, esta tabela guarda os dados que permitem fazer a filtragem das alterações (*insert*, *update* ou *delete*) que são, ou não, válidas para uma determinada aplicação *Web*. A filtragem vai depender, para além do tipo de alteração, da aplicação *Web* para onde a informação atualizada pode ser enviada e da tabela onde a alteração ocorreu. A tabela WEBS_PARAM_APPLICATION_ACTION é constituída pelas seguintes colunas:
  - TABLE_NAME
    - Tipo: VARCHAR2 (60);
    - Descrição: Esta coluna guarda os nomes das tabelas que vão entrar no processo de filtragem.
  - COD_APPLICATION

    * Tipo: VARCHAR2 (50);
    * Descrição: Esta coluna guarda os códigos das aplicações *Web*.
  - ACTION
    * Tipo: VARCHAR2 (20);
    * Descrição: Esta coluna guarda as ações (*insert*, *update* ou *delete*) que são válidas para uma determinada aplicação *Web* e para uma determinada tabela.

- WEBS_PARAM_TABLE_APPLICATION — Utilizada no procedimento *procedure_notification* e no procedimento *process_handler*, esta tabela indica o COD_APPLICATION e o COD_DATA associados à tabela cujo conteúdo foi alterado e que fez com que o procedimento *procedure_notification* fosse espoletado. A tabela WEBS_PARAM_TABLE_APPLICATION é constituída pelas seguintes colunas:
  - ID (PK)
    * Tipo: NUMBER;
    * Descrição: Este valor vai permitir identificar as linhas desta tabela. Os valores desta coluna são muito utilizados, no procedimento *process_handler*, para aceder às linhas das tabelas WEBS_PARAM_TABLE_COL_ELEMENT e WEBS_PARAM_TABLE_COL. As chaves primárias destas duas tabelas são chaves estrangeiras desta coluna.
  - TABLE_NAME
    * Tipo: VARCHAR2 (60);
    * Descrição: Esta coluna guarda os nomes das tabelas que interessam inspecionar.
  - COD_APPLICATION
    * Tipo: VARCHAR2 (50);
    * Descrição: Esta coluna guarda os códigos das aplicações *Web*.
  - COD_DATA
    * Tipo: VARCHAR2 (100);
    * Descrição: Esta coluna guarda os códigos dos vários tipos de dados associados às tabelas e às aplicações *Web*.

- WEBS_PARAM_TABLE_COL_ELEMENT — Utilizada no procedimento *process_handler*, esta tabela guarda o nome da coluna, por cada tabela, que contém o identificador que vai ser utilizado pelo servidor Node.js para verificar se a aplicação *Web* quer receber as atualizações de informação relacionadas com o elemento que tenha esse identificador. A tabela WEBS_PARAM_TABLE_COL_ELEMENT é constituída pelas seguintes colunas:
  - ID_TABLE_APLICATION (PK)
    * Tipo: NUMBER;

    - Descrição: Chave estrangeira da coluna ID da tabela WEBS_PARAM_TABLE_-APPLICATION, este valor permite identificar a que tabela o nome da coluna guardado na coluna COLUMN_NAME está associado.
  - COLUMN_NAME
    - Tipo: VARCHAR2 (100);
    - Descrição: Esta coluna guarda os nomes das colunas que contêm os identificadores que vão ser utilizados no servidor Node.js.

- WEBS_PARAM_TABLE_COL — Utilizada no procedimento *process_handler*, esta tabela guarda os nomes das colunas, associadas às tabelas registadas, que fornecem os parâmetros necessários para que se possa ir buscar informação mais detalhada à API. A tabela WEBS_PARAM_TABLE_COL é constituída pelas seguintes colunas:
  - ID_TABLE_APLICATION (PK)
    - Tipo: NUMBER;
    - Descrição: Chave estrangeira da coluna ID da tabela WEBS_PARAM_TABLE_-APPLICATION, este valor permite identificar a que tabela o nome da coluna guardado na coluna COLUMN_NAME está associado.
  - COLUMN_NAME
    - Tipo: VARCHAR2 (100);
    - Descrição: Esta coluna guarda os nomes das colunas que contêm os parâmetros necessários para que o servidor Node.js possa aceder ao servidor da API à procura de mais informação.

- WEBS_PARAM_REGISTER_TABLE — Utilizado no procedimento *register_notification_-code*, esta tabela guarda os nomes das tabelas que têm de ser inspecionadas. A tabela WEBS_PARAM_REGISTER_TABLE é constituída apenas por uma coluna:
  - TABLE_NAME
    - Tipo: VARCHAR2 (60);
    - Descrição: Esta coluna guarda os nomes das tabelas que irão ser inspecionadas.

Colocou-se a hipótese de juntar a tabela WEBS_PARAM_TABLE_COL_ELEMENT com a tabela WEBS_PARAM_TABLE_COL para formar uma única tabela. Esta nova tabela iria ter mais uma coluna do que a WEBS_PARAM_TABLE_COL, uma *flag* que iria marcar as colunas que, para além de fornecerem os parâmetros que permitem buscar informação mais detalhada, iriam conter os identificadores que seriam utilizados pelo servidor Node.js. No entanto, a criação desta tabela iria partir do pressuposto de que a coluna que fornece os identificadores também fornece os parâmetros que permitem aceder à informação mais pormenorizada, o que nem sempre acontece.

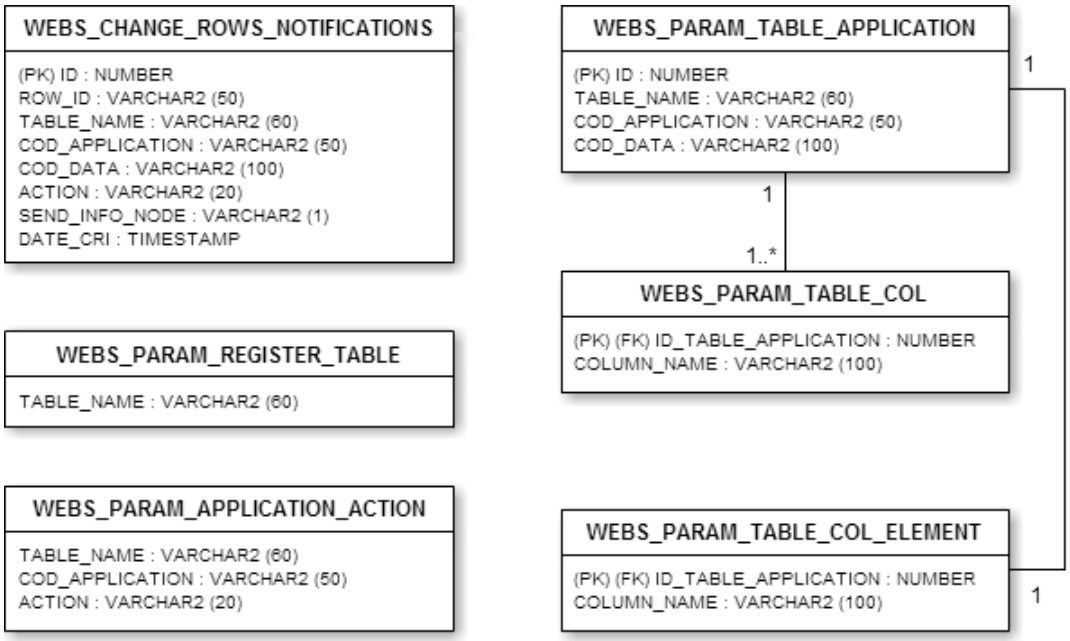

Figura 5.1: Tabelas utilizadas na componente Oracle

O objeto JSON, que é passado como parâmetro à função *send_update_node_http* e que é enviado, através de um pedido HTTP, por esta função para o servidor Node.js, é constituído pelos seguintes atributos:

- *elemento_id* — Este atributo guarda uma *string* com um determinado identificador associado ao conteúdo que foi adicionado, alterado ou eliminado;
- *nome_tabela* — Este atributo guarda uma *string* com o nome da tabela cujo conteúdo foi alterado;
- *parametros_colunas_json* — Este atributo guarda um objeto JSON com os parâmetros que a Oracle passa ao servidor Node.js para que este consiga ir buscar informação mais detalhada sobre a indicação que está a ser enviada;
- *codigo_aplicacao* — Este atributo guarda uma *string* com o código da aplicação definido na coluna COD_APPLICATION da tabela WEBS_CHANGE_ROWS_NOTIFICATIONS;
- *codigo_dados* — Este atributo guarda uma *string* com o código do tipo de dados definido na coluna COD_DATA da tabela WEBS_CHANGE_ROWS_NOTIFICATIONS;
- *acao* — Este atributo guarda uma *string* com a ação que espoletou o envio deste objeto JSON;
- *data_notificacao* — Este atributo guarda uma *string* com a data em que o conteúdo da tabela foi alterado.

5.2 Componente Node.js

A componente Node.js da implementação é constituída por um servidor desenvolvido em Node.js. A função do servidor Node.js é a de enviar, através de ligações WebSocket, às aplicações *Web*, a informação atualizada em tempo real. A implementação das ligações WebSocket entre as aplicações *Web* e o servidor Node.js é feita com recurso à biblioteca Socket.io.

Para que uma ligação WebSocket possa ser estabelecida entre uma aplicação *Web* e o servidor Node.js, a aplicação *Web* tem de enviar um pedido de conexão que tem de ser aceite pelo servidor. Caso a ligação seja aceite, o servidor Node.js cria um objeto JSON e guarda-o num *array* onde vão estar outros objetos JSON referentes a ligações WebSocket que o servidor mantém ativas.

Os atributos do objeto JSON criado são os seguintes:

- *id* — Através da biblioteca Socket.io, o servidor Node.js sabe que estabeleceu uma ligação WebSocket com uma aplicação *Web* quando o evento "connection" é espoletado. Quando o evento "connection" é espoletado, a função associada ao evento recebe como parâmetro um objeto JSON com detalhes acerca da aplicação *Web* que acabou de estabelecer uma ligação WebSocket com o servidor. No objeto recebido, um dos atributos que compõem o objeto é um identificador único que o Socket.io associa a cada ligação. Este identificador único vai ser guardado neste atributo;

- *obj* — Neste atributo vai ser guardado o objeto JSON que é recebido quando, no servidor Node.js, o evento "connection" é espoletado. Este objeto é muito importante para a comunicação WebSocket entre o servidor e as aplicações *Web*, uma vez que é com recurso a este objeto que o servidor consegue enviar mensagens para as aplicações *Web*;

- *cod_application* — Este atributo guarda uma *string* com o código da aplicação relativo à aplicação *Web* que se encontra ligada ao servidor. A forma como os códigos das aplicações são definidos nesta componente é igual à forma como os códigos das aplicações são definidos na Oracle;

- *cod_data* — Este atributo guarda uma *string* com o código do tipo de dados que interessa à aplicação *Web*. A terminologia utilizada na definição dos códigos dos tipos de dados é a mesma terminologia utilizada pela Oracle para definir os códigos dos tipos de dados;

- *list_elements* — Este atributo guarda uma lista de valores que vai especificar, dentro do contexto de uma aplicação *Web* e segundo um tipo de dados bem definido, quais os elementos que a aplicação *Web* tem interesse em receber atualizações. Na maior parte das vezes, esta lista é constituída por identificadores de médicos, pacientes, enfermeiros, etc.

Dos cinco atributos que constituem o objeto JSON utilizado para representar uma ligação WebSocket entre o servidor Node.js e uma aplicação *Web*, só dois é que ficam definidos na altura da criação do objeto: o *id* e o *obj*. Os restantes atributos são inicializados na altura da criação do objeto, contudo, é a aplicação *Web*, depois da ligação WebSocket estar estabelecida, que tem

a responsabilidade de enviar a informação que vai preencher os atributos do objeto JSON. As aplicações *Web* enviam as informações que vão constar nos vários atributos através dos seguintes eventos:

- "set_cod_application" — Quando este evento é espoletado, a função associada ao evento tem como parâmetro uma *string* que vai ser guardada no atributo *cod_application*;
- "set_cod_data" — Quando este evento é espoletado, a função associada ao evento tem como parâmetro uma *string* que vai ser guardada no atributo *cod_data*;
- "set_list_elements" — Quando este evento é espoletado, a função associada ao evento tem como parâmetro uma lista que vai ser guardada no atributo *list_elements*.

No código das três funções associadas aos três eventos acima referidos, antes de associar o parâmetro recebido ao atributo em questão, é necessário saber qual o objeto JSON, representativo da ligação WebSocket entre a aplicação *Web* e o servidor Node.js, cujos atributos vão ser alterados. Para isso, teve de se colocar o código relativo aos três eventos dentro da função espoletada pelo evento "connection", uma vez que só assim é que possível conhecer o identificador da ligação WebSocket por onde os eventos foram chamados. O evento "connection" só é espoletado quando uma ligação WebSocket for estabelecida entre a aplicação *Web* e o servidor Node.js. Após a obtenção do identificador da ligação, basta encontrar, no *array* que guarda todos os objetos JSON que representam as várias ligações WebSocket ativas, o objeto JSON com o atributo *id* igual ao identificador da ligação para lhe alterar os atributos.

No servidor Node.js, para além dos quatro eventos relacionados com os WebSockets já referidos ("connection", "set_cod_application", "set_cod_data", "set_list_elements"), existe mais um evento associado aos WebSockets que pode ser espoletado, o evento "disconnect". Este evento é espoletado quando a ligação WebSocket entre uma aplicação *Web* e o servidor Node.js é encerrada. Tal como os eventos "set_cod_data", "set_cod_application" e "set_list_elements", também o código deste evento tem de estar dentro da função espoletada pelo evento "connection", visto que só assim é possível obter o identificador da ligação que acabou de encerrar. Depois de conhecer o identificador da ligação que encerrou, tem de se encontrar, no *array* que guarda todos os objetos JSON que representam as várias ligações WebSocket ativas, o objeto JSON com o atributo *id* igual ao identificador da ligação que acabou de encerrar para que este seja eliminado do *array* das ligações ativas.

Depois de uma ligação WebSocket entre uma aplicação *Web* e o servidor Node.js estar estabelecida e do servidor Node.js ter recebido todas as informações acerca da ligação WebSocket, o servidor Node.js pode começar a enviar informação, com base nas indicações da Oracle, em tempo real para a aplicação *Web*. O processo que vai desde o envio da indicação, por parte da Oracle, para o servidor Node.js até ao envio da informação atualizada para a aplicação *Web* é constituído pelos seguintes passos:

1. O servidor Node.js recebe, sob a forma de um objeto JSON, a indicação da Oracle sobre o que foi alterado;

2. No caso de não existirem ligações WebSocket ativas, o processo de atualização da informação termina aqui. No caso de existirem ligações ativas, verifica quais as aplicações *Web* que estão interessadas em receber a informação indicada pela Oracle. Os objetos JSON das aplicações interessadas são guardados num *array*. Considera-se que uma aplicação *Web* está interessada em receber a informação proveniente de uma indicação da Oracle quando:

    - O *cod_application* do objeto JSON que representa a ligação tem de ser igual ao *codigo_aplicacao* do objeto JSON que a Oracle enviou;
    - O *cod_data* do objeto JSON que representa a ligação WebSocket tem de ser igual ao *codigo_dados* do objeto JSON que a Oracle enviou;
    - A ação que levou a que a Oracle enviasse a indicação é um *insert* ou caso a ação tenha sido um *update* ou um *delete*, o valor do atributo *elemento_id*, que pertence ao objeto JSON que a Oracle enviou, tem de pertencer à *list_elements* do objeto JSON que representa a ligação WebSocket.

3. Caso não exista nenhuma aplicação *Web* interessada na indicação enviada pela Oracle, o processo de atualização da informação termina aqui. Caso exista alguma aplicação interessada, o servidor Node.js tem diferentes comportamentos dependendo da ação que levou ao envio da indicação por parte da Oracle:

    - Se a ação for um *insert* ou um *update*, o servidor Node.js envia um pedido HTTP a um dos servidores da API na tentativa de obter informação mais pormenorizada. Depois de a informação ter, ou não, sido obtida, é criado um objeto JSON que vai ser enviado para todas as aplicações *Web* cujas ligações WebSocket estão guardadas no *array* formado no ponto 2;
    - Se a ação for um *delete*, o servidor Node.js cria um objeto JSON, com a indicação do vai ser eliminado, que vai ser enviado para todas as aplicações *Web* cujas ligações WebSocket com o servidor Node.js estão representadas sob a forma de objetos JSON no *array* formado no ponto 2.

4. Independentemente do objeto JSON gerado, o servidor Node.js envia o objeto para as aplicações *Web* interessadas através do evento "send_information".

Os pedidos HTTP enviados pelo servidor Node.js para tentar obter informação mais pormenorizada têm sempre o mesmo destino. A única coisa que os diferencia é a composição do objeto JSON que transportam. O destino dos pedidos HTTP é uma função, localizada num dos servidores da API, que, com base nos atributos *cod_application* e *cod_data* do objeto JSON, tenta aceder a uma outra função que lhe permita obter a informação que o servidor Node.js necessita. Para além do *cod_application* e do *cod_data*, o objeto JSON envia os parâmetros, parâmetros esses que chegaram ao servidor Node.js através da Oracle e que são guardados no atributo *parameters* do objeto JSON, que permitem utilizar a função que obtém a informação que vai ser utilizada para atualizar as aplicações *Web*. A vantagem desta estratégia é que por muitas aplicações *Web* que estejam

interessadas em receber a informação que é obtida através da indicação enviada pela Oracle, só é feito um pedido HTTP (no caso de a ação que motivou o envio da indicação ter sido um *insert* ou um *update*) à API para obter a informação que vai ser enviada, através das ligações WebSockets, para as aplicações *Web*.

Caso corra tudo dentro da normalidade, o objeto JSON que é enviado para as aplicações *Web* é constituído pelos seguintes atributos:

- *state* — Este atributo guarda uma *string* com o valor "OK";
- *action* — Este atributo guarda uma *string* com a ação (*insert*, *update* ou *delete*) que desencadeou todo este processo de atualização da informação;
- *cod_data* — Este atributo guarda uma *string* com o código do tipo de dados que a aplicação *Web* pediu;
- *obj* — No caso da ação que levou a todo este processo ter sido um *insert* ou um *update*, este atributo guarda um objeto JSON com a informação proveniente dos servidores da API. No caso de a ação ter sido um *delete*, o atributo guarda uma *string* com a indicação do que vai ser eliminado;
- *date* — Este atributo guarda uma *string* com data em que a base de dados Oracle foi alterada.

Contudo, no caso de haver algum erro durante a obtenção de informação mais pormenorizada, o objeto JSON que é enviado para as aplicações *Web* é constituído pelos seguintes atributos:

- *state* — Este atributo guarda uma *string* com o valor "error";
- *error* — Este atributo guarda uma *string* que descreve qual a razão do erro.

5.3 Componente Cliente

A componente do cliente vai estar diretamente relacionada com o funcionamento dos WebSockets nas aplicações *Web* da Glintt HS. Dado que as ligações WebSocket entre as aplicações *Web* e o servidor Node.js vão ser geridas pela biblioteca Socket.io, no lado do cliente é necessário adicionar uma referência à versão para aplicações *Web* da biblioteca Socket.io.

Quando uma aplicação *Web* é iniciada, ela tenta estabelecer de imediato uma ligação WebSocket com o servidor Node.js. Para além do estabelecimento da ligação WebSocket com o servidor Node.js, a aplicação *Web* envia um pedido HTTP ao servidor da aplicação a pedir informação que vai ser mostrada no ecrã. Depois de a resposta ao pedido HTTP chegar, a aplicação *Web* envia ao servidor Node.js, através dos três eventos já referidos na secção anterior ("set_cod_application", "set_cod_data" e "set_list_elements"), o código com o qual a aplicação se identifica, o código do tipo de dados que a aplicação *Web* pediu e uma lista com os identificadores dos elementos que chegaram à aplicação *Web* na resposta ao pedido HTTP enviado ao servidor da aplicação. Estes dados

irão permitir ao servidor Node.js saber se determinada indicação enviada pela Oracle interessa, ou não, à aplicação *Web*.

A função associada ao evento "send_information" tem como parâmetro o objeto JSON que transporta, no caso de o servidor Node.js ter conseguido obter a informação sem qualquer problema, a informação atualizada. A forma como a informação recebida é colocada nas aplicações *Web* vai depender do tipo de ação (este valor está guardado no atributo *action* do objeto JSON) que levou à receção da informação e vai depender da forma como o código que faz o *rendering* da informação na aplicação *Web* esteja organizado.

Uma das vantagens da biblioteca Socket.io é que, caso a ligação WebSocket entre a aplicação *Web* e o servidor Node.js falhe devido a situações que não tenham a ver com o fecho da aplicação *Web* ou com o encerramento do servidor Node.js, o Socket.io tenta restabelecer automaticamente a ligação WebSocket entre os dois *endpoints*. Apesar desta tentativa em restabelecer a ligação WebSocket, no servidor Node.js foi espoletado o evento "disconnect", o que fez com o objeto JSON que representava a ligação WebSocket entre a aplicação *Web* e o servidor Node.js fosse eliminado. O que significa que, com o objeto JSON, toda a informação que o servidor Node.js iria utilizar para verificar se determinada indicação interessa à aplicação *Web* também será eliminada.

No caso de o Socket.io conseguir restabelecer a ligação WebSocket entre uma aplicação *Web* e o servidor Node.js, um objeto JSON é criado e é adicionado ao *array* que contém os objetos JSON que representam as ligações WebSocket ativas contudo, os dados que indicam os interesses da aplicação *Web* não seriam passados, uma vez que a aplicação *Web* não vai voltar a pedir a informação ao servidor da aplicação para depois andar a passar ao servidor Node.js, com base na informação que recebeu, os dados que indicam se a aplicação *Web* está, ou não, interessada nas indicações enviadas pela Oracle. A solução passa por, na primeira vez em que a ligação WebSocket é estabelecida entre a aplicação *Web* e o servidor Node.js, guardar, na aplicação *Web*, a informação que foi enviada ao servidor Node.js para que, sempre que o Socket.io tenha de restabelecer a ligação (quando o evento "connect" é espoletado, a aplicação *Web* sabe que uma ligação WebSocket foi estabelecida com o servidor Node.js), a aplicação *Web* consiga enviar ao servidor Node.js os dados que indicam o interesse da aplicação *Web*.

No entanto, caso a aplicação *Web* tenha de mudar a informação que está a ser mostrada, os dados que descrevem o interesse da aplicação *Web* vão ser diferentes, o que significa que, para além de atualizar a informação, relativa ao interesse da aplicação, no objeto JSON guardado no servidor Node.js, os dados que indicam o interesse da aplicação *Web* que estão guardados na própria aplicação *Web* terão de ser atualizados para que, após o restabelecimento de uma ligação WebSocket, a aplicação *Web* envie os dados que descrevem o interesse da aplicação nas indicações que a Oracle envia.

5.4 Resumo

A implementação da solução pode ser divida em três componentes:

- Componente Oracle;

- Componente Node.js;
- Componente Cliente.

Dado que, durante o processo de *polling*, as respostas aos pedidos HTTP enviados pelas aplicações *Web* contêm toda a informação em vez de ter apenas a informação que foi atualizada durante o intervalo de tempo que existiu entre os pedidos HTTP significa que não existe nada que registe as alterações da informação que estão a ocorrer. A primeira solução seria colocar código na API, que recebe os pedidos HTTP para adicionar, alterar ou eliminar informação, para registar as alterações da informação, no entanto não é possível ter acesso a essa API. Então, a solução passou por inspecionar as tabelas das bases de dados cujo conteúdo fosse considerado relevante para as aplicações *Web*. Assim, por cada alteração que houvesse nessas tabelas (através de um *insert*, *update* ou *delete*), a Oracle enviava uma indicação do que foi alterado ao servidor Node.js para que este pudesse, em seguida, ir buscar ao servidor da API informação mais detalhada sobre a indicação que recebeu.

Para inspecionar o conteúdo das tabelas, pensou-se em utilizar *triggers* nas tabelas que interessava inspecionar, no entanto os *triggers*, face ao grande número de tabelas que teriam de ser inspecionadas, acabariam por ter um forte impacto no desempenho da base de dados. Sendo assim, a solução encontrada passa pela utilização do *package* DBMS_CHANGE_NOTIFICATION. Este *package*, com muito menos impacto na base de dados, permite registar as alterações que estão a ocorrer no conteúdo das tabelas. Depois de haver uma alteração ao conteúdo de uma tabela que esteja a ser inspecionada, para além de ser guardada numa tabela a referência à alteração que ocorreu, é enviado um objeto JSON para o servidor Node.js com a indicação do que foi alterado.

Depois de receber a indicação, o servidor Node.js verifica se existe alguma aplicação *Web* que esteja interessada em receber a informação baseada na indicação que recebeu. Caso exista alguma aplicação *Web* interessada e, caso a ação que espoletou a indicação tenha sido um *insert* ou um *update*, o servidor Node.js acede a uma função num dos servidores da API que, com base no *cod_application* e do *cod_data*, chama uma outra função onde vai ser possível obter a informação mais detalhada. Caso essa informação chegue ao servidor Node.js, o servidor cria um objeto JSON com a informação atualizada e envia-o para as aplicações *Web* interessadas. Caso a ação que espoletou o envio da indicação tenha sido um *delete*, o servidor Node.js não necessita de ir buscar informação mais detalhada uma vez que basta mandar para as aplicações *Web* o identificador do elemento que foi eliminado.

As aplicações *Web* tem uma grande importância no funcionamento da solução, uma vez que, logo após a ligação WebSocket estar estabelecida, as aplicações *Web*, dependendo da informação que estão a mostrar, enviam, para além dos identificadores dos elementos para os quais querem receber atualizações da informação, os dados que permitem ao servidor Node.js saber as características das aplicações *Web* com quem está ligado.

Quando as aplicações *Web* recebem um objeto JSON com a informação atualizada, as aplicações *Web* têm de fazer o *rendering* da informação que receberam. A Figura 5.2 ilustra o processo

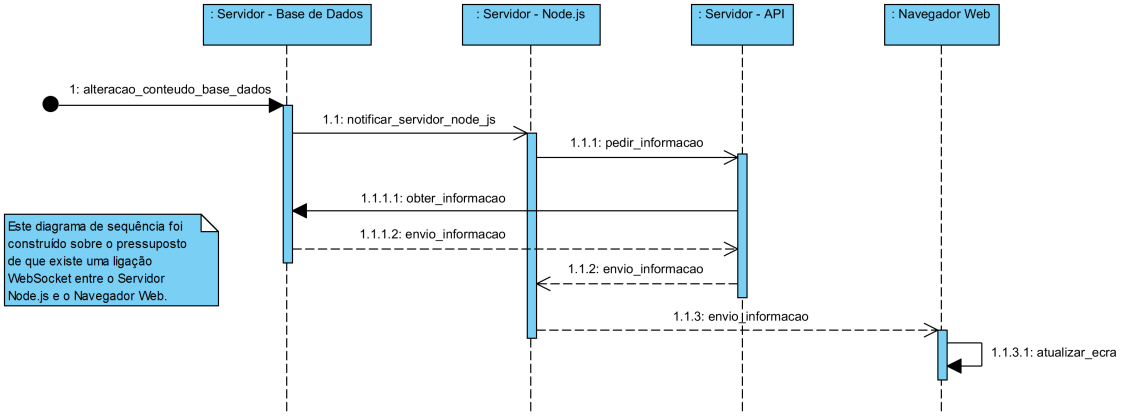

Figura 5.2: Processo de WebSockets nas aplicações Glintt

que vai desde a alteração do conteúdo de uma das tabelas que estava a ser inspecionada até ao passo em que a informação atualizada é mostrada no ecrã da aplicação.

Capítulo 6

Testes e Avaliação

O Capítulo 6 mostra os testes que foram feitos à solução a fim de verificar se a solução obtém melhores resultados em relação ao que estava implementado. A primeira secção serve para explicar quais as métricas que vão ser utilizadas para avaliar a solução e quais as condições em que os testes se realizaram. A segunda secção serve para explicar de forma é que uma das métricas foi medida e quais os resultados obtidos com base nessa métrica. A terceira secção, tal como a segunda, serve para descrever a forma como é que uma das métricas foi medida e quais foram os resultados obtidos. Por fim, na última secção é elaborado um pequeno resumo dos resultados descritos nas duas secções anteriores.

6.1 Introdução

Ao longo deste capítulo, irão ser apresentados vários testes que permitirão comprovar se a solução descrita no capítulo anterior apresenta melhores resultados do que a solução que atualmente existe e que está a funcionar em várias instituições hospitalares.

Os testes que irão ser apresentados neste capítulo podem ser agrupados segundo duas métricas:

- Latência;
- Tráfego gerado na transmissão da informação para a aplicação *Web*.

Apesar de a ideia ser a de comparar o *polling* aos WebSockets com base nos valores da latência e na quantidade de *bytes* utilizados na transmissão da informação, é importante medir estas duas métricas para o *long polling* uma vez que o Socket.io (ver Subsecção 4.3.3) utiliza *long polling* para as comunicações entre os servidores e os navegadores *Web* que não suportam WebSockets. Como o navegador *Web* utilizado em algumas instituições hospitalares é o Internet Explorer 9, é um navegador *Web* que não suporta WebSockets, convém verificar, através dos valores da latência e da quantidade de *bytes* utilizados, se mesmo com *long polling*, a solução continua a ser viável.

A última versão do Internet Explorer 11 tem uma funcionalidade que permite emular o Internet Explorer 9 e assim é possível testar o desempenho do *long polling*.

Para que a solução desenvolvida pudesse ser testada da melhor forma possível, a Glintt HS deu a possibilidade de testar a solução numa aplicação *Web* que já está presente em várias instituições hospitalares, o Painel de Enfermagem. No Painel de Enfermagem, os enfermeiros podem consultar os serviços (obstetrícia, pediatria, cardiologia, neurologia, etc.) que lhes estão atribuídos. Por cada serviço, os enfermeiros têm acesso, por exemplo, à localização dos pacientes (quarto/cama ou enfermaria/cama), à data em que os pacientes foram admitidos, à data prevista para a alta médica, ao motivo do internamento, à informação sobre as alergias dos pacientes internados, etc.. As Figuras 6.1, 6.2 e 6.3 mostram o estado atual da aplicação *Web* que permite monitorizar as tarefas dos enfermeiros. Para que os dados, tanto da latência como da quantidade de bytes utilizados, que são obtidos através do Painel de Enfermagem tenham alguma fiabilidade, é necessário definir duas condições:

- Apesar da grande quantidade de filtros que podem ser colocados no Painel de Enfermagem, estes mantiveram-se inalterados ao longo dos testes;
- Apesar de os pacientes estarem divididos por secções (cada secção agrupa 15 pacientes), a única secção que é visível ao longo dos testes é aquela que é carregada quando o utilizador inicia sessão.

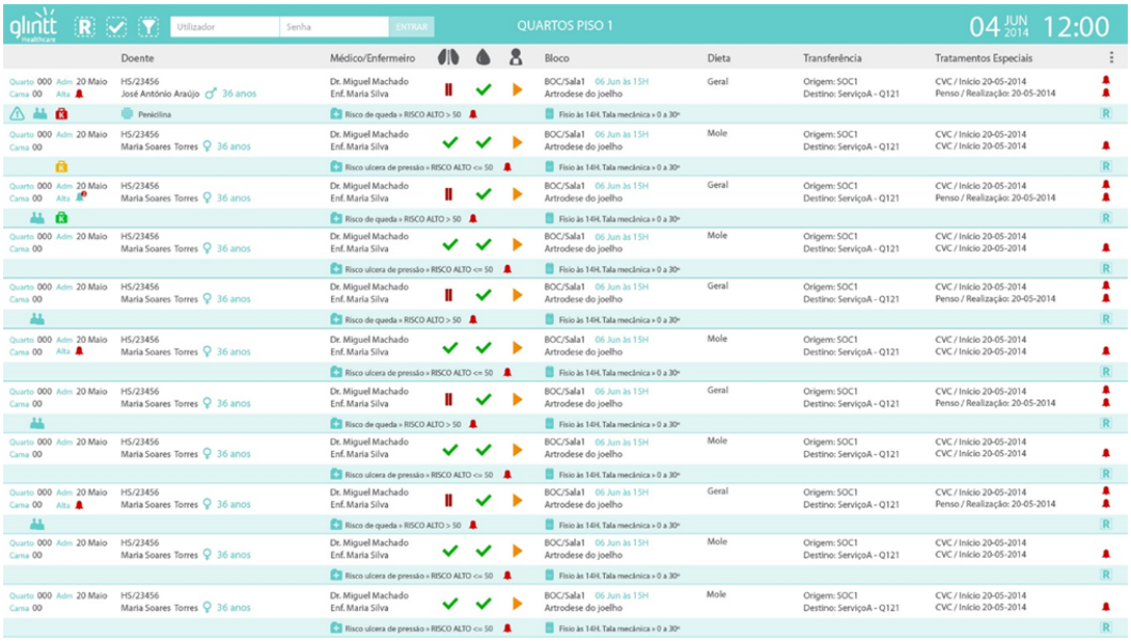

Figura 6.1: *Dashboard* do serviço de internamento

Convém salientar que os testes que irão ser apresentados neste capítulo foram executados no computador fornecido pela empresa para desenvolver esta dissertação. O computador utilizado

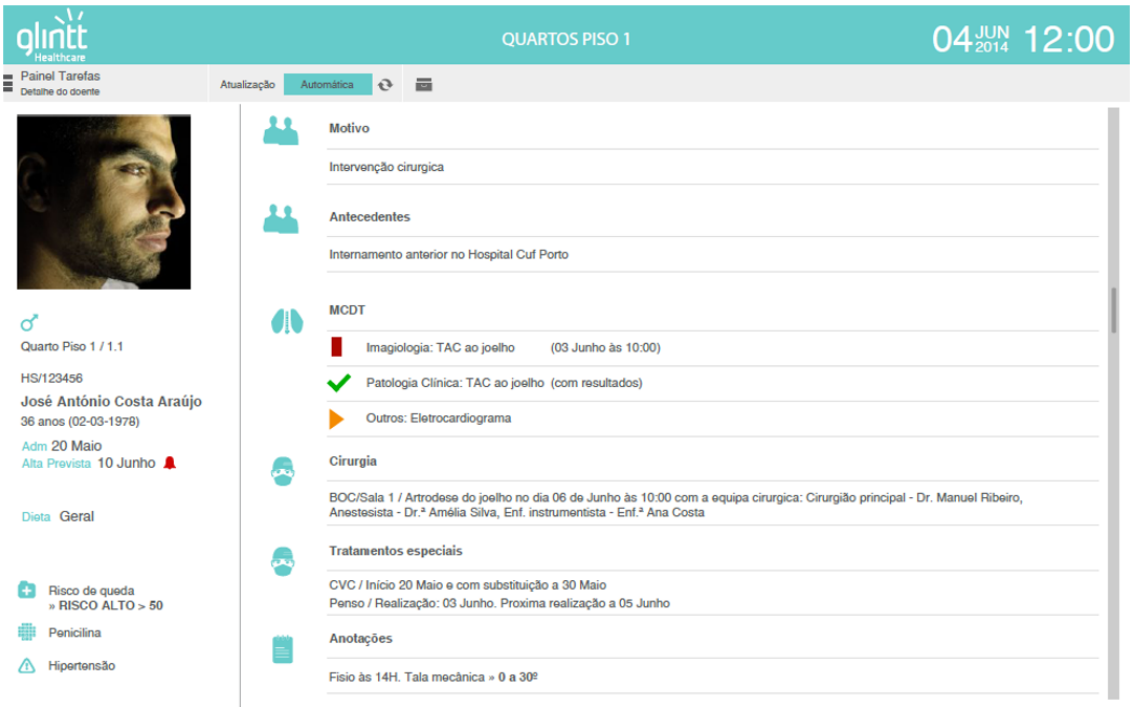

Figura 6.2: Detalhes do doente

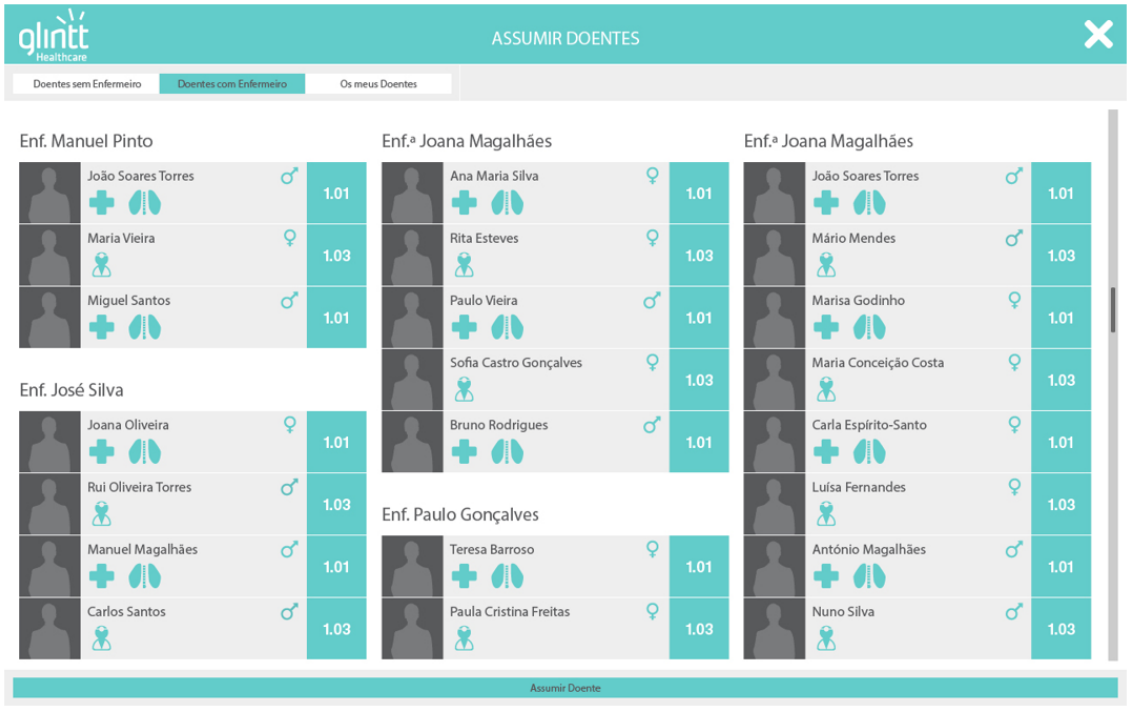

Figura 6.3: Assumir responsabilidade de um doente

tem um processador Intel(R) Core(TM) i3-4000M CPU @ 2.4 GHz e com uma memória RAM de 4 GB. Apesar de nas instituições hospitalares haver um servidor para cada aplicação *Web*, um ou mais servidores da API e, caso a solução descrita no capítulo anterior fosse implementada, um servidor Node.js, para efeitos de teste, no computador fornecido pela empresa vai ser executado o código referente aos três, ou mais, servidores acima referidos. Isto significa que, tirando as comunicações com o servidor da base de dados, todas as comunicações vão ser feitas dentro do *localhost*.

Durante a execução dos testes, para que estes fossem o mais fidedignos possível, os únicos programas que estavam a correr eram o navegador *Web* – no caso do Google Chrome, as extensões eram sempre desligadas – sempre em modo privado, a consola que permitia ligar e desligar o servidor Node.js e um programa que permitia aceder à tabela da base de dados onde as alterações estavam a ser feitas.

6.2 Latência

A latência pode ser definida como o intervalo de tempo entre o início de um evento e o momento em que os efeitos desse evento tornam-se visíveis. Apesar de no *polling*, no *long polling* e nos WebSockets os efeitos do evento inicial serem os mesmos, mostrar a informação atualizada na aplicação *Web*, a forma com que o evento é iniciado no *polling* é diferente da forma com que é iniciado nos WebSockets e no *long polling*.

No *polling*, a latência começa a ser medida a partir do momento em que a aplicação *Web* envia um pedido HTTP ao servidor da aplicação e só é terminado quando a aplicação *Web* recebe e coloca na interface da aplicação o conteúdo recebido. No *long polling* e nos WebSockets, a latência começa a ser medida a partir do momento em que o conteúdo de uma das tabelas que interessa inspecionar é alterado e só é terminado quando a aplicação *Web* recebe a informação atualizada proveniente do servidor Node.js. Para medir a latência no *polling*, basta utilizar um *Web debugger*, como o Fiddler4, ou as ferramentas do programador disponibilizadas pelos navegadores *Web*. Nos WebSockets e no *long polling*, para medir a latência, é necessário seguir os seguintes passos:

1. Guardar a data em que o conteúdo de uma das tabelas que interessava inspecionar foi alterado (através de um *insert*, *update* ou *delete*);

2. Junto com a indicação, a Oracle tem de enviar, num objeto JSON, ao servidor Node.js a data em que a alteração ocorreu;

3. Depois de o servidor Node.js ter obtido a informação mais pormenorizada acerca da indicação que recebeu por parte da Oracle, o servidor tem de enviar, através de um objeto JSON, à aplicação *Web* a informação obtida e a data em que a alteração ocorreu;

4. Quando o objeto JSON é recebido, a aplicação *Web* calcula a diferença de tempo entre a data atual e a data que vem guardada no objeto JSON. A diferença de tempo é calculada em milissegundos.

A forma com que, nos WebSockets e no *long polling*, a latência é medida pode levantar algumas dúvidas acerca da fiabilidade dos valores obtidos, uma vez que estão a ser obtidos com recurso a duas datas de dois dispositivos (o computador onde a dissertação foi desenvolvida e o servidor da base de dados Oracle) diferentes que, muito provavelmente, não têm os relógios sincronizados. No entanto, uma vez que tanto o computador utilizado como o servidor da base de dados estão ligados à rede interna da Glintt e que todos os relógios dos dispositivos ligados à rede interna da Glintt estão sincronizados, não existem problemas relacionados com a fiabilidade dos valores obtidos. Os computadores e os servidores da rede interna da Glintt estão sincronizados através do servidor *Network Time Protocol* (NTP) da Microsoft.

O gráfico da Figura 6.4 mostra a variação da latência ao longo de vinte pedidos HTTP, para o caso do *polling*, e ao longo de vinte alterações às bases de dados, para o caso dos WebSockets e do *long polling*. Para que fosse possível comparar os dados da latência que foram obtidos através da utilização das três técnicas em estudo, o Painel de Enfermagem teve que ser executado no Internet Explorer, uma vez que é o único navegador *Web* onde é possível testar as três técnicas em estudo.

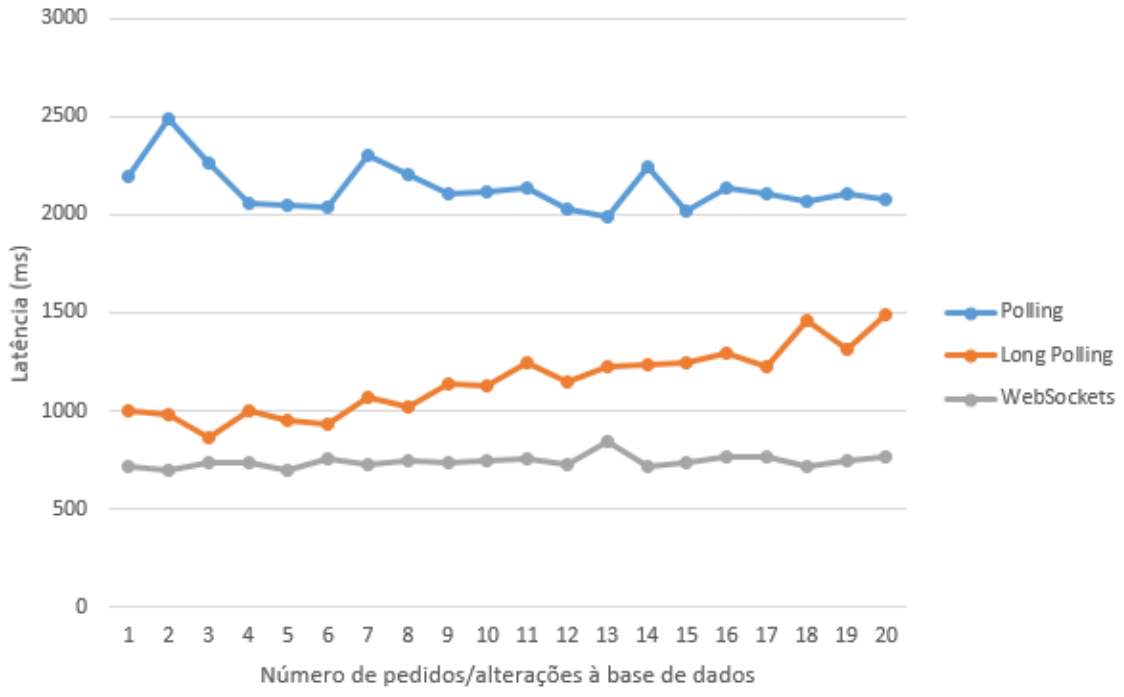

Figura 6.4: Latência do *polling*, do *long polling* e dos WebSockets testada no Internet Explorer

A partir do gráfico da Figura 6.4, é possível observar que a latência com que o *polling* entrega a informação ao Painel de Enfermagem é consideravelmente superior comparativamente ao *long polling* – em média, o *polling* apresenta uma latência de 2 136 ms, enquanto o *long polling* tem

uma latência na ordem dos 1 148 ms (quase metade da latência média do *polling*) – e muito superior comparativamente aos WebSockets. Nos testes efetuados, os WebSockets, no Internet Explorer 11, apresentam uma latência média de 741 ms, o que faz com que o valor da latência média nos WebSockets seja um pouco superior a uma terça parte do valor da latência média no *polling* (2 136 ms).

Apesar de a grande maioria das instituições hospitalares utilizar o Internet Explorer para correr as aplicações *Web*, existem várias instituições que utilizam o Google Chrome para correr as aplicações *Web*. O gráfico da Figura 6.5 mostra a variação da latência ao longo de vinte pedidos HTTP, para o caso do *polling*, e ao longo de vinte alterações às bases de dados, no caso dos WebSockets.

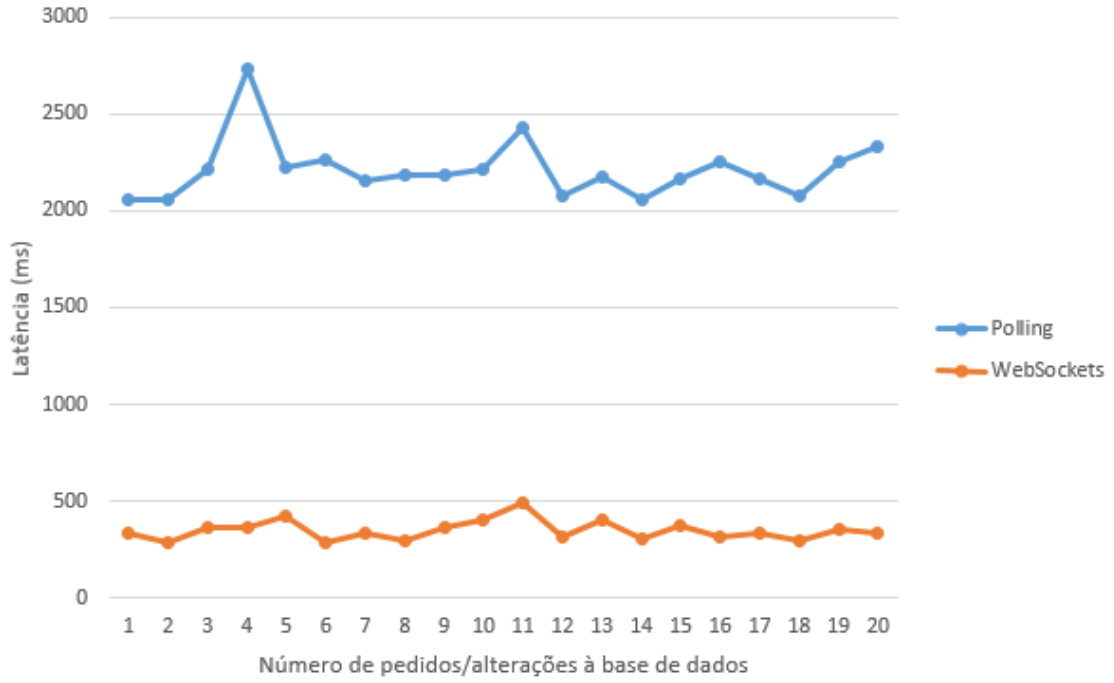

Figura 6.5: Latência do *polling* e dos WebSockets testada no Google Chrome

Neste gráfico não são estudados os valores da latência obtidos através de *long polling* uma vez que todas as versões do Google Chrome suportam WebSockets – as primeiras versões do Google Chrome suportam versões antigas do protocolo WebSocket [Dev15]. Depois de analisado o gráfico da Figura 6.5, é possível verificar que a latência com que o *polling* entrega a informação no Google Chrome é bastante parecida com a latência com que o *polling* fornece a informação no Internet Explorer – ver o gráfico da Figura 6.6. A latência média com que o *polling* entrega a informação no Google Chrome é de 2 214 ms enquanto no Internet Explorer é de 2 136 ms. Outra conclusão que pode ser facilmente retirada a partir do gráfico da Figura 6.5 é a enorme diferença que existe na latência com que o *polling* e os WebSockets enviam a informação ao Painel de Enfermagem. Enquanto a latência média com que o *polling* entrega a informação é de 2 214 ms, a latência média utilizada pelos WebSockets é de 350 ms – a diferença entre os dois valores da latência é superior à sexta parte.

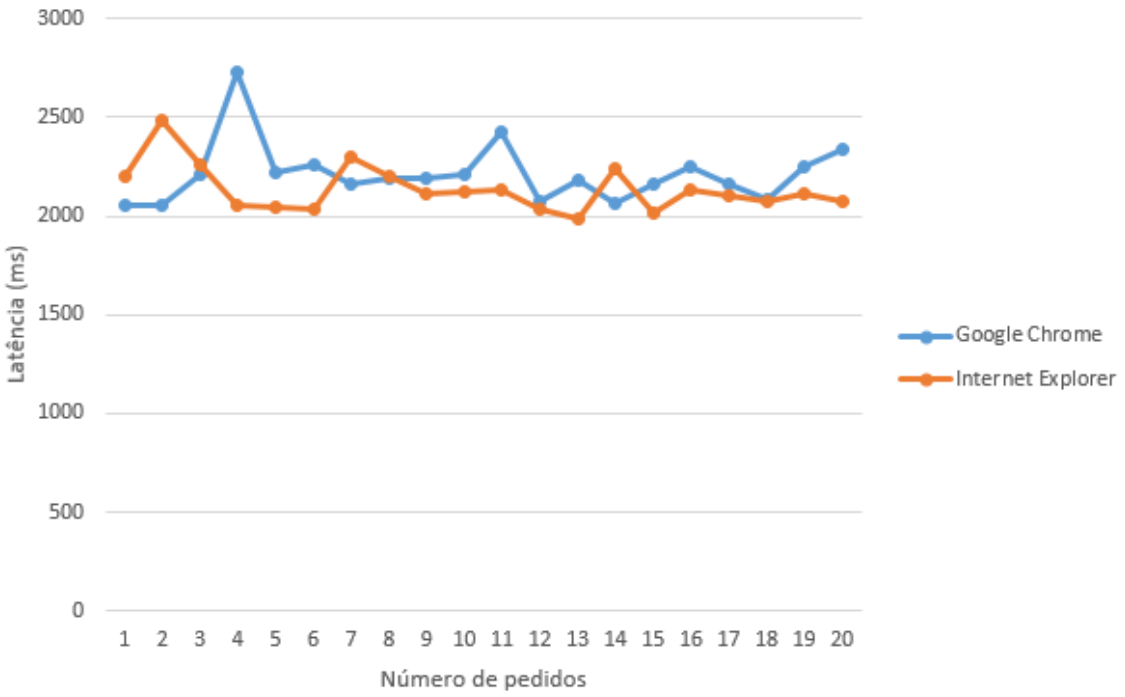

Figura 6.6: Latência do *polling* testada no Internet Explorer e no Google Chrome

Tanto no Google Chrome como no Internet Explorer, o que se verifica, com base nos gráficos das Figuras 6.4 e 6.5, é que existe uma enorme diferença entre os valores da latência no *polling* e os valores da latência nos WebSockets. Esta diferença pode ser explicada com recurso a dois fatores:

- Arquitetura do sistema: O facto de o sistema estar fragmentado em vários servidores (ver Capítulo 3) faz com que haja muito tempo gasto tanto nas ações dos elementos que constituem a arquitetura como no transporte de dados entre os vários elementos. Apesar de a arquitetura associada à solução (ver Capítulo 4) ser, também ela, bastante fragmentada, o tempo gasto a processar as ações dos vários elementos do sistema aquando da utilização de WebSockets são muito inferiores em relação ao tempo gasto a processar as ações dos vários elementos aquando da utilização do *polling*. Um exemplo disso é que no caso do *polling*, a aplicação *Web* envia um pedido HTTP ao servidor de aplicação que, antes de reencaminhar o pedido para o servidor da API, executa um conjunto de ações relacionadas com a validade do pedido que necessitam de algum tempo para serem processadas o que acaba por ter um impacto negativo na latência;

- Quantidade de informação transportada: Conjugado com o facto de a arquitetura do sistema ser bastante fragmentada, sempre que o servidor da API recebe um pedido HTTP do Painel de Enfermagem (reencaminhado pelo servidor da aplicação), o servidor da API vai buscar

a informação, quer esteja, ou não, atualizada, relativa aos detalhes clínicos dos 15 pacientes que estão visíveis na interface da aplicação e que tem de ser enviada para o Painel de Enfermagem.

Ao analisar o gráfico da Figura 6.7, observa-se uma diferença na latência com que os WebSockets entregam a informação no Google Chrome comparativamente à latência necessária para entregar a informação no Internet Explorer – a latência média no Internet Explorer (741 ms) é mais do dobro da latência média no Google Chrome (350 ms). Para tentar perceber o motivo para haver uma diferença tão grande entre os dois navegadores *Web*, foi alterada a altura em que a latência estava a ser calculada.

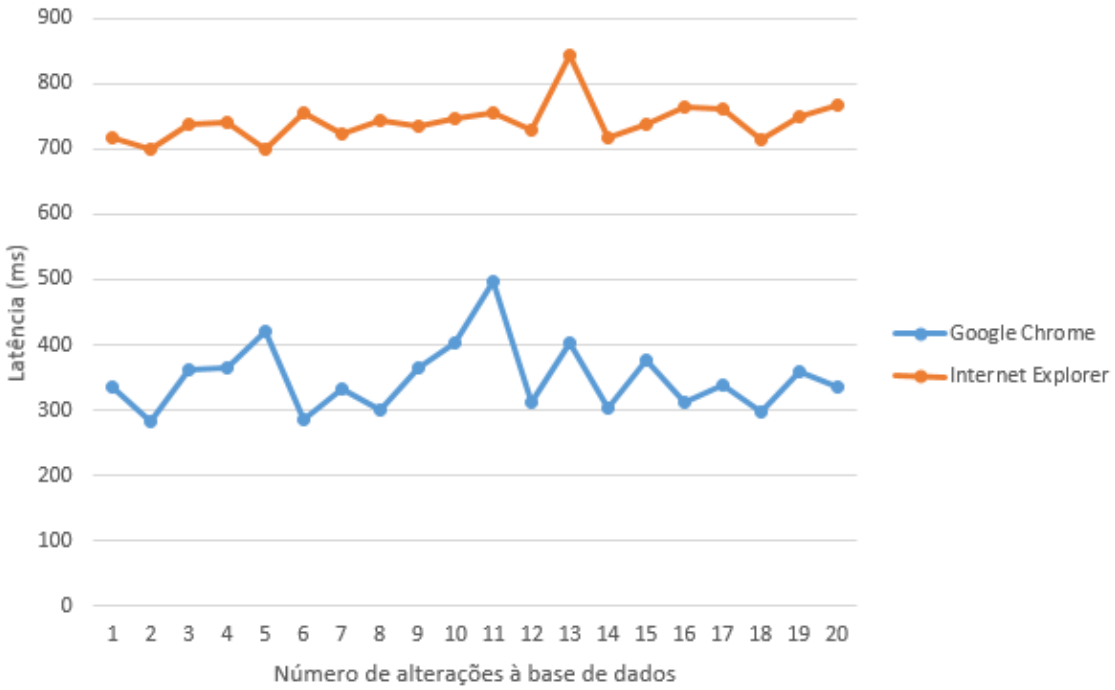

Figura 6.7: Latência dos WebSockets testada no Internet Explorer e no Google Chrome

Até aqui, o cálculo da latência englobava o processo que permitia mostrar a informação atualizada na interface da aplicação no entanto, o cálculo da latência passou a ser calculado quando a aplicação *Web* recebe a informação atualizada proveniente do servidor Node.js. A partir do gráfico da Figura 6.8, é possível observar que, com esta nova forma de cálculo, a latência média obtida no Google Chrome é muito parecida com a latência média no Internet Explorer – 129 ms no Google Chrome e 121 ms no Internet Explorer. Isto significa que o motivo que faz com que na antiga forma de cálculo (após o *rendering* da informação) haja uma discrepância tão grande de valores está relacionado com o interpretador de código JavaScript utilizado pelo Google Chrome e pelo Internet Explorer [Cal15].

A partir dos gráficos das Figuras 6.9 e 6.10 (a Figura 6.10 mostra um gráfico com a média dos valores da latência obtidos no gráfico da Figura 6.9), é possível observar a variação da latência – os

valores da latência foram obtidos com base em dez alterações à base de dados – com o número de aplicações *Web* que se encontrem ligadas ao servidor Node.js (até aqui, os testes tinham sido feitos com apenas uma aplicação *Web* ligada ao servidor Node.js) e que registem interesse no mesmo tipo de informação. Isto permite simular a ideia que estão, por exemplo, vinte navegadores *Web* a correr a aplicação do Painel de Enfermagem. O navegador *Web* escolhido para correr as aplicações *Web* foi o Internet Explorer (versão 11), uma vez que é o navegador *Web* mais utilizado pelas instituições hospitalares. Como pode ser observado através dos gráficos das Figuras 6.9 e 6.10, apesar de termos 40 aplicações ligadas ao servidor, a latência com que as aplicações *Web* recebem as atualizações não sofreu qualquer alteração, quando comparando com os valores da latência em que apenas uma aplicação *Web* estava registada no servidor Node.js. Estes dois gráficos acabam por comprovar a ideia de que o Node.js é uma excelente plataforma para desenvolver servidores que sejam altamente escaláveis.

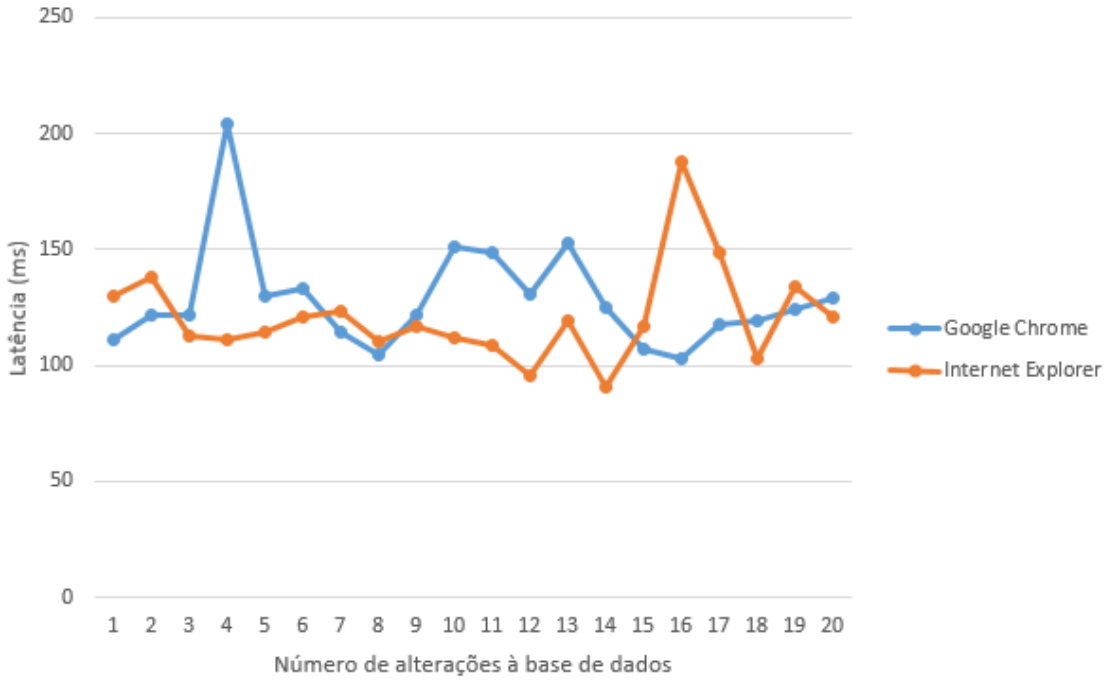

Figura 6.8: Latência dos WebSockets (sem *rendering* da informação) testada no Internet Explorer e no Google Chrome

Por fim, e segundo os dados apresentados nas Tabelas 6.1, 6.2 e 6.3, é possível verificar, em quatro pontos de avaliação, a variação da latência relacionada com o número de alterações por segundo que são feitas ao conteúdo da base de dados. Para que os dados obtidos consigam espelhar a realidade, é necessário criar dois cenários:

- Cenário 1: A latência é calculada logo após o Painel de Enfermagem ter recebido uma mensagem WebSocket com a informação atualizada;
- Cenário 2: A latência é calculada após o *rendering* da informação no Painel de Enfermagem.

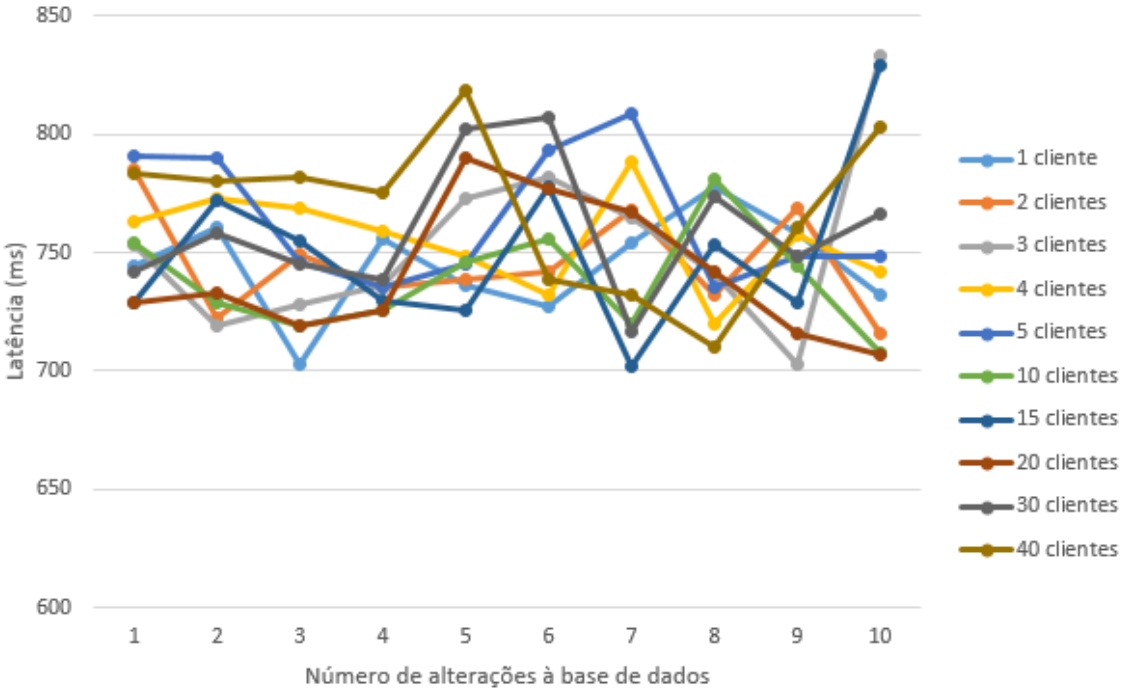

Figura 6.9: Evolução da latência relacionada com o número de clientes conectados ao servidor Node.js (valores absolutos)

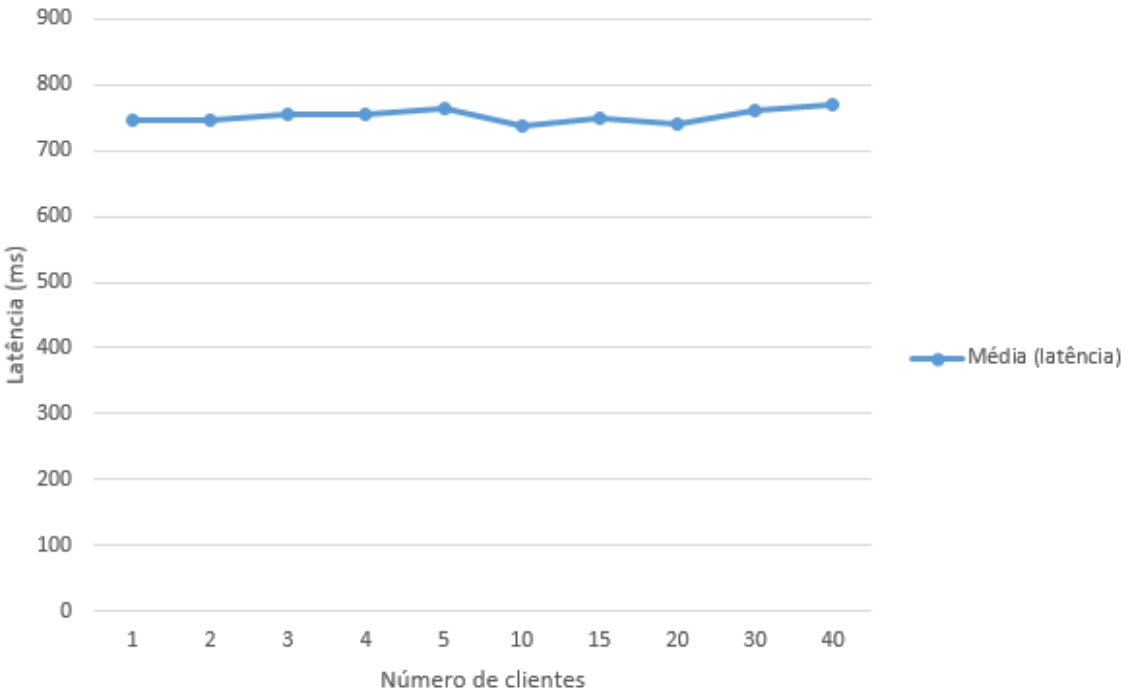

Figura 6.10: Evolução da latência relacionada com o número de clientes conectados ao servidor Node.js (média)

Tabela 6.1: Valores obtidos no cenário 1

Tempo decorrido (ms)	4 alterações/s	8 alterações/s	12 alterações/s	16 alterações/s
250	111 ms	141 ms	91 ms	80 ms
500	149 ms	108 ms	90 ms	103 ms
750	127 ms	108 ms	95 ms	110 ms
1 000	111 ms	122 ms	111 ms	106 ms

Tabela 6.2: Valores obtidos no cenário 2 (Internet Explorer)

Tempo decorrido (ms)	4 alterações/s	8 alterações/s	12 alterações/s	16 alterações/s
250	850 ms	882 ms	998 ms	1 020 ms
500	1 284 ms	2 010 ms	2 720 ms	3 662 ms
750	1 695 ms	3 072 ms	4 458 ms	6 108 ms
1 000	2 135 ms	4 147 ms	6 235 ms	8 684 ms

Tabela 6.3: Valores obtidos no cenário 2 (Google Chrome)

Tempo decorrido (ms)	4 alterações/s	8 alterações/s	12 alterações/s	16 alterações/s
250	345 ms	387 ms	394 ms	429 ms
500	370 ms	791 ms	1 033 ms	1 398 ms
750	407 ms	1 027 ms	1 474 ms	2 091 ms
1 000	436 ms	1 225 ms	1 898 ms	2 744 ms

O que se pode observar com o primeiro cenário é que, apesar do número de alterações que podem ser feitas à base de dados num curto espaço de tempo, a latência com que as mensagens WebSocket chegam ao Painel de Enfermagem não sofre alterações[1], quando comparado com outros valores que foram obtidos da mesma forma e que já foram especificados ao longo deste capítulo, ver Figura 6.8. Neste primeiro cenário, era indiferente se a aplicação corria no Internet Explorer ou no Google Chrome, uma vez que, devido à forma com que os valores são obtidos, a diferença de valores entre os dois navegadores é praticamente nula. No entanto, o segundo cenário já é um pouco mais complexo do que o primeiro, uma vez que passa a ser necessário fazer testes que cubram as situações em que o Painel de Enfermagem está a correr no Google Chrome e no Internet Explorer. Com o cálculo da latência a ser feito após o *rendering* da informação, verificou-se que os valores da latência são diretamente proporcionais ao número de alterações por segundo que são feitas ao conteúdo da base de dados. No caso mais extremo, com 16 alterações por segundo, a latência máxima atingida no Google Chrome foi de 2 744 ms enquanto no Internet Explorer foi de 8 684 ms. Com estes resultados, é possível comprovar a ideia, anteriormente referida, de que o Google Chrome é melhor do que o Internet Explorer ao nível da interpretação de código

[1] As oscilações que se verificam nos valores da Tabela 6.1 devem-se ao tempo que a API demora a obter a informação atualizada que vai ser enviada para o servidor Node.js.

JavaScript. Outra conclusão que se pode tirar é que não basta otimizar as bases de dados ou os servidores para mostrar informação em tempo real, também é preciso ter em atenção a forma com que é feito o *rendering* da informação.

6.3 Tráfego gerado na transmissão de informação

Esta métrica permite quantificar o número de *bytes* que são necessários para que a informação chegue ao Painel de Enfermagem. Nesta métrica não serão quantificados apenas os *bytes* utilizados para transmitir a informação desde o servidor da aplicação (no caso do *polling*) ou desde o servidor Node.js (no caso dos WebSockets e do *long polling*) para a aplicação *Web*. Esta métrica contabiliza todos os *bytes* que são enviados e recebidos durante o processo que trata da atualização da informação presente no ecrã da aplicação *Web*.

Tal como acontecia com a latência, a forma como os *bytes* utilizados vão ser quantificados vai depender da técnica que estiver a ser utilizada. No *polling* vão ser quantificados os *bytes* utilizados nas seguintes ações:

- No pedido HTTP enviado pelo Painel de Enfermagem ao servidor da aplicação;
- No pedido HTTP enviado pelo servidor da aplicação ao servidor da API. Tal como já foi referido, o servidor da aplicação, depois de receber um pedido HTTP, reencaminha-o para o servidor que melhor responda ao pedido;
- Na resposta que o servidor da API envia ao servidor da aplicação. A informação que o Painel de Enfermagem pretende está contida no corpo da resposta;
- Na resposta que o servidor da aplicação envia para o Painel de Enfermagem.

No caso do *polling*, não são contabilizados os *bytes* que são trocados entre o servidor da API e o(s) servidor(es) da base de dados uma vez que não é possível quantificar com exatidão o número de *bytes* que percorrem a ligação TNS.

Nos WebSockets vão ser quantificados os *bytes* utilizados nas seguintes ações:

- No estabelecimento da ligação WebSocket entre o Painel de Enfermagem e o servidor Node.js;
- Através da ligação WebSocket, no envio das informações (*cod_application*, *cod_data*, *connection_obj*, etc.) que permitem ao servidor Node.js aceder a um dos servidores da API para ir buscar informação pormenorizada relacionada com uma indicação do servidor da base de dados;
- No pedido HTTP que o servidor da base de dados envia ao servidor Node.js com a indicação, caso o conteúdo de uma das tabelas que estavam a ser inspecionadas ter sido alterado, do que foi alterado;

- Na resposta que o servidor Node.js envia ao servidor da base de dados onde informa se a indicação foi, ou não, recebida e processada com sucesso;
- No pedido HTTP que o servidor Node.js envia, após receber uma indicação de algo que foi alterado, ao servidor da API a pedir informação pormenorizada acerca da indicação recebida;
- Na resposta do servidor da API enviada ao servidor Node.js com a informação pormenorizada pedida pelo servidor Node.js;
- Através da ligação WebSocket, no envio de um objeto JSON, do servidor Node.js para o Painel de Enfermagem, que contém a informação pormenorizada que o servidor Node.js recebeu do servidor da API;
- Através de ligação WebSocket, na troca de pequenas mensagens – troca de *pings* e *pongs* (ver Subsecção 4.3.1) – entre a aplicação *Web* e o servidor Node.js que permite que a ligação se mantenha ativa.

No *long polling*, a quantificação dos *bytes* utilizados é muito parecida com aquela que é feita nos WebSockets. A única diferença prende-se com o facto de que como no *long polling* não podem existir ligações WebSockets, todas as ações que necessitem de uma ligação WebSocket têm de ser feitas com recurso a mensagens segundo o protocolo HTTP.

Para que se consiga conhecer a quantidade de *bytes* que cada uma das técnicas utiliza, é necessário que o Painel de Enfermagem esteja a correr no Internet Explorer – é o único navegador *Web* em que se consegue ter a aplicação a utilizar *polling*, *long polling* e WebSockets. Enquanto a aplicação está a correr, é necessário ter o WireShark ou o Fiddler4 a capturar a atividade do Painel de Enfermagem e dos restantes elementos (servidor Node.js, servidor da base de dados, etc.) para saber quais as mensagens que são trocadas e assim saber a quantidade de *bytes* utilizada.

A partir do gráfico da Figura 6.11, é possível observar a quantidade de *bytes* utilizados, durante uma hora, pelas três técnicas (*polling*, *long polling* e WebSockets) para atualizar a informação que está visível no Painel de Enfermagem. Para que as medições possam ser mais fiáveis, estimou-se que a frequência com que o conteúdo da tabela – associada à informação que estava a ser mostrada no Painel de Enfermagem – é alterado (só é feita uma alteração de cada vez) seria de um minuto e que a frequência com que a aplicação *Web* envia pedidos HTTP ao servidor também seria de 1 minuto – corresponde à melhor situação uma vez que as respostas aos pedidos do Painel de Enfermagem trazem sempre informação nova. No entanto, o que se observa no gráfico da Figura 6.11 é uma enorme diferença na quantidade de *bytes* utilizados pelo *polling* comparativamente à quantidade utilizada pelo *long polling* e pelos WebSockets. Esta diferença de quantidades é facilmente explicada, uma vez que, tal como já foi dito na Secção 3.3, apesar de a resposta ao pedido trazer informação nova, a resposta traz muita mais informação que não está atualizada – a resposta devolve a informação relativamente a 15 pacientes contudo, só a informação de um paciente é que foi atualizada – ao contrário do que acontece no *long polling* ou nos WebSockets em que a

informação que entra no Painel de Enfermagem está unicamente relacionada com o paciente cuja informação foi atualizada.

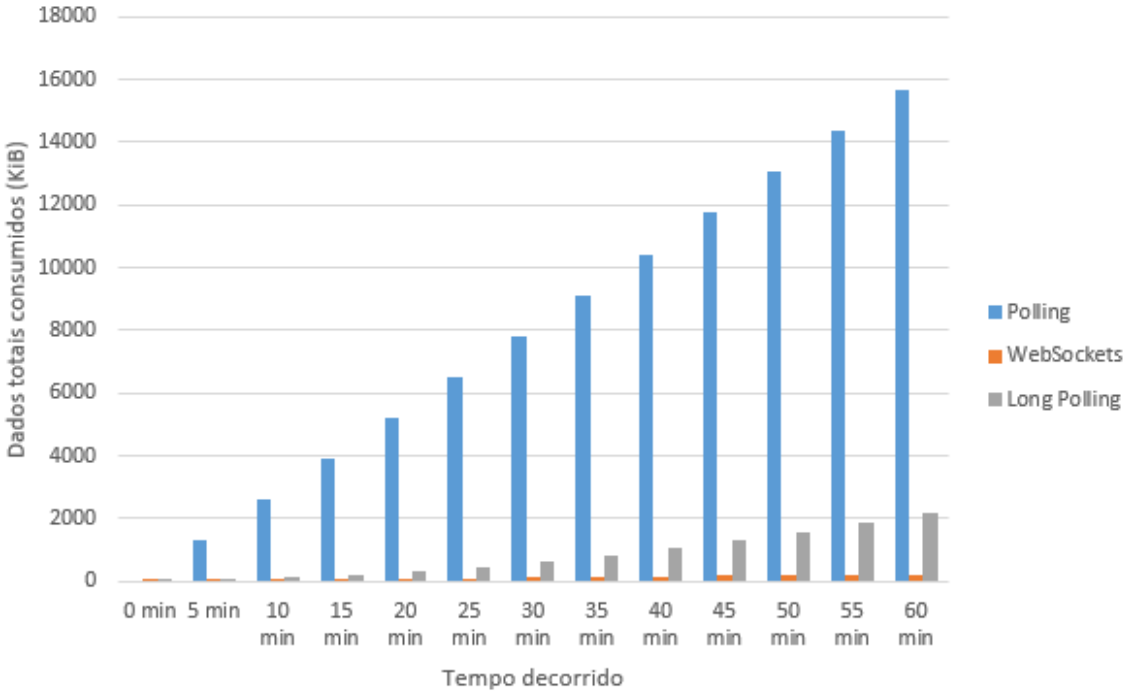

Figura 6.11: Tráfego gerado pelo *polling*, pelo *long polling* e pelos WebSockets com atualizações de informação

O gráfico da Figura 6.12 ainda consegue ser mais "alarmante" que o da Figura 6.11. Com base no gráfico da Figura 6.12, é possível observar a evolução, ao longo de uma hora, da quantidade de *bytes* utilizados pelas três técnicas para atualizar a informação do Painel de Enfermagem. No entanto, ao contrário do que aconteceu na situação que originou os dados do gráfico da Figura 6.11, o conteúdo da tabela associada ao Painel de Enfermagem não vai ser atualizado. A única coisa que se mantém é a frequência com que a aplicação faz pedidos HTTP ao servidor, continua a ser de 1 minuto. O que se observa no gráfico da Figura 6.12 é uma diferença maior em relação àquela que foi registada no gráfico da Figura 6.11 entre a quantidade de *bytes* utilizados pelo *polling* e a quantidade utilizada pelo *long polling* e pelos WebSockets. Esta discrepância é explicada pelo facto de que apesar de o conteúdo da tabela não ser atualizado, o Painel de Enfermagem, de minuto a minuto, envia um pedido HTTP ao servidor e este devolve a informação não atualizada de 15 pacientes. Já no *long polling* e nos WebSockets, a quantidade de *bytes* utilizados é menor quando comparado com a quantidade que é mostrada no gráfico da Figura 6.11 uma vez que na segunda situação, como não existiram alterações no conteúdo da tabela associada ao Painel de Enfermagem, os *bytes* só foram utilizados para estabelecer a ligação entre o Painel e o servidor Node.js e para a manter ativa. As diferenças na quantidade de *bytes* utilizados pelos WebSockets e pelo *long polling* estão visíveis nos gráficos das Figuras 6.13 e 6.14.

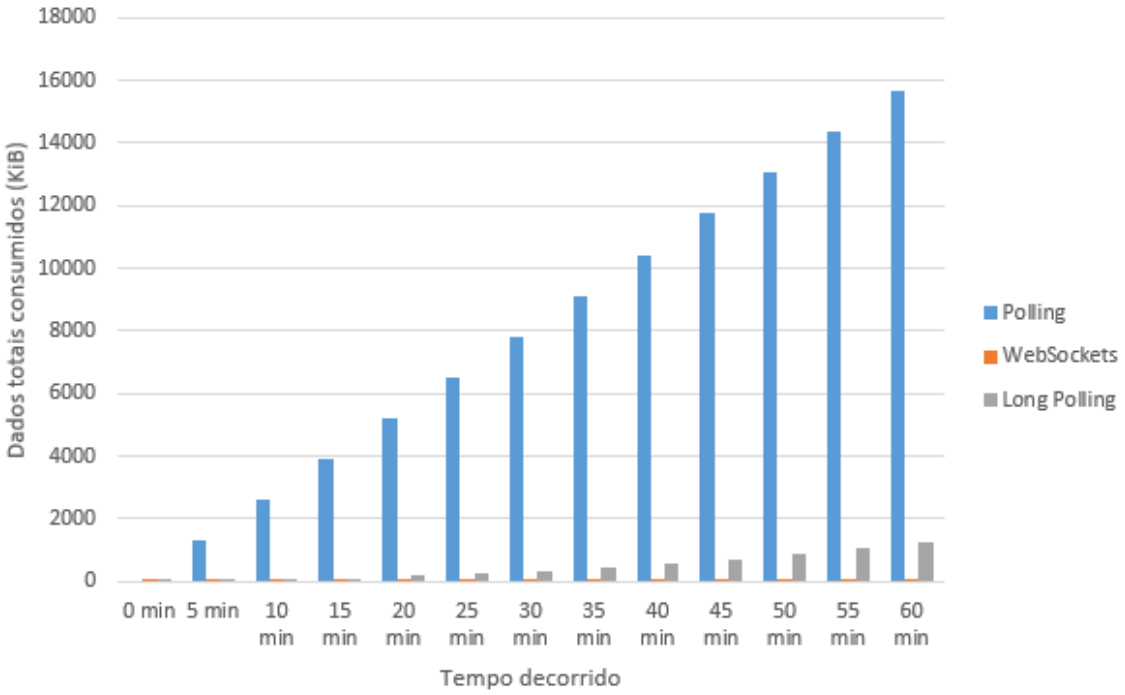

Figura 6.12: Tráfego gerado pelo *polling*, pelo *long polling* e pelos WebSockets sem atualizações de informação

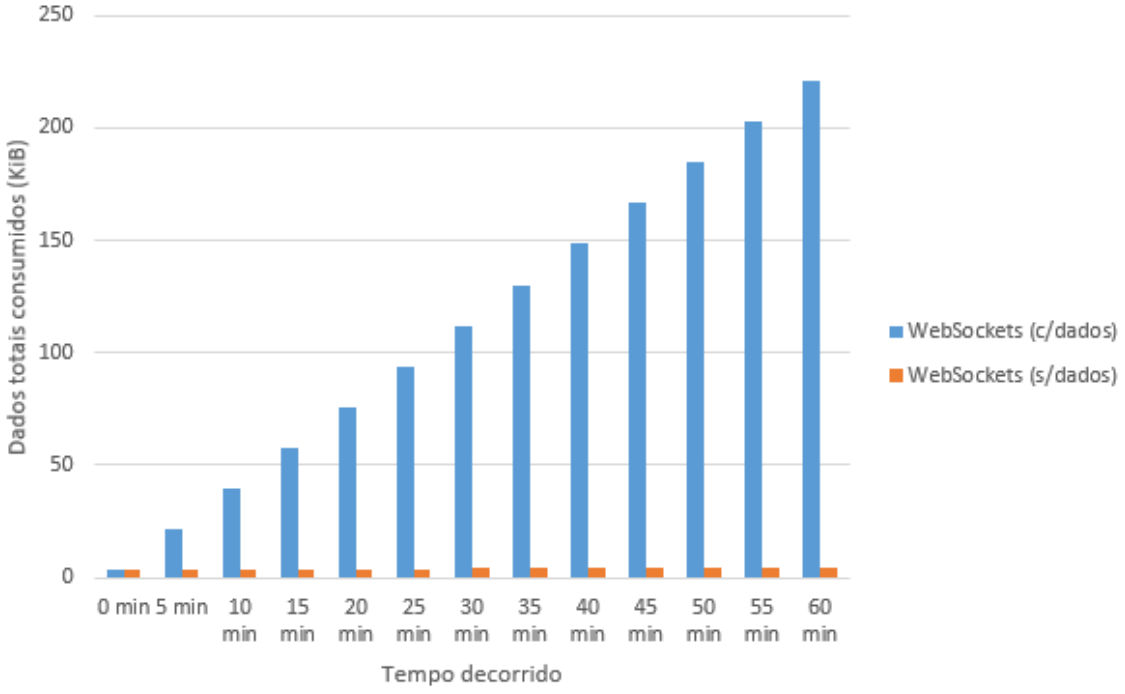

Figura 6.13: Comparação do tráfego gerado pelos WebSockets

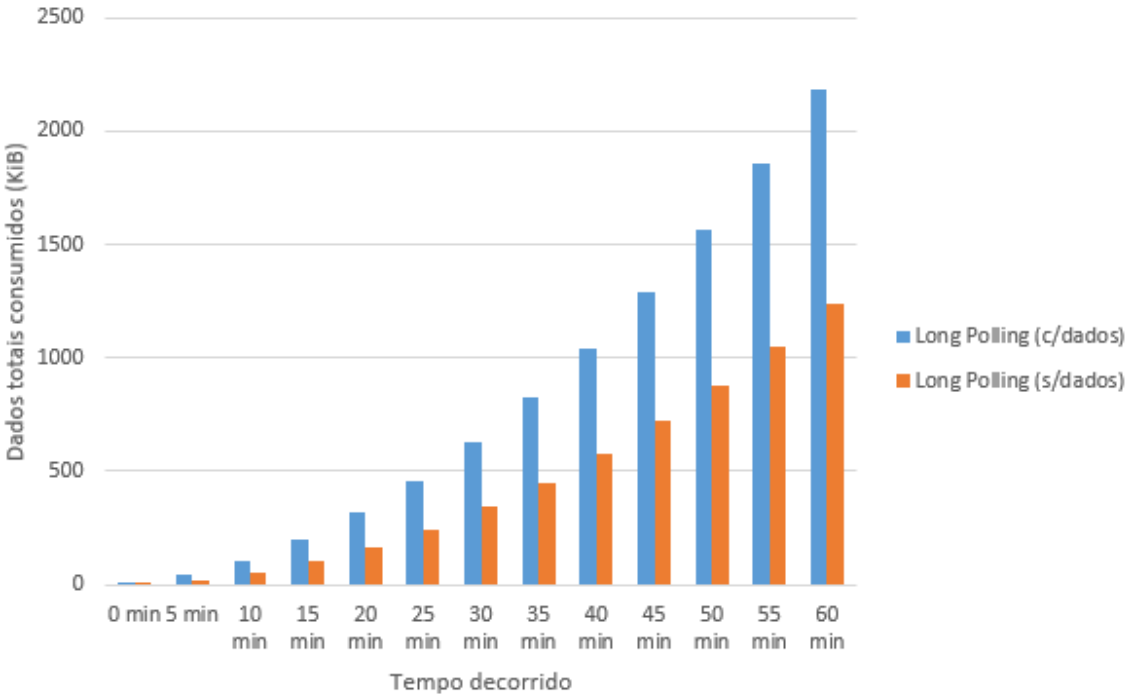

Figura 6.14: Comparação do tráfego gerado pelo *long polling*

Pelo que foi visto até agora, os WebSockets e o *long polling* (principalmente os WebSockets) são uma excelente alternativa ao *polling*, uma vez que conseguem, com muito menos *bytes*, manter a informação do Painel de Enfermagem atualizada no entanto, convém explorar uma situação em particular para perceber a viabilidade dos WebSockets na arquitetura das aplicações *Web* da Glintt HS. Admitindo que o processo de *polling* só envia para o Painel de Enfermagem os pacientes que foram atualizados durante o intervalo de tempo entre os últimos dois pedidos, que a informação acerca de cada paciente utiliza cerca de 1,3 KiB[2] e que a frequência com que o conteúdo da tabela associada ao Painel de Enfermagem é alterado (só é alterado um paciente de cada vez) é igual à frequência com que a aplicação envia os pedidos HTTP, pode concluir-se, com base no gráfico da Figura 6.15, que a quantidade de *bytes* utilizados pelo *polling* está muito mais próxima da quantidade de dados utilizada pelos WebSockets do que está, por exemplo, no gráfico da Figura 6.11. Esta redução significativa na quantidade *bytes* utilizados pelo *polling* deve-se ao facto de que, na situação retratada no gráfico da Figura 6.11, era sempre transportada a informação de 15 pacientes enquanto agora só é transportada a informação dos pacientes que viram a sua informação ser atualizada. Uma outra conclusão que se pode retirar do gráfico da Figura 6.15 é que o *long polling* passou a ser a técnica que mais *bytes* utiliza. Esta alteração pode ser explicada pelo simples facto de que o Socket.io, ao ver que o navegador *Web* não suporta WebSockets, tenta, através de *long polling* e com recurso a mensagens HTTP, fazer as mesmas trocas de mensagens que faz com as

[2] 1,3 KiB é o tamanho médio do objeto JSON que o servidor Node.js envia para o Painel de Enfermagem, através de uma ligação WebSocket, com a informação atualizada acerca de um paciente.

aplicações *Web* que estejam a correr num navegador *Web* que suporte WebSockets. Esta troca de mensagens faz com que a quantidade de *bytes* utilizados no *long polling* seja elevada.

Para finalizar, admitindo que o processo de *polling* só envia para o Painel de Enfermagem os pacientes que foram atualizados durante o intervalo de tempo entre os últimos dois pedidos e que a frequência com que a aplicação *Web* envia pedidos HTTP é de minuto a minuto, pode concluir-se, com base no gráfico da Figura 6.16, que a quantidade de *bytes* utilizados pelo *polling* está muito mais próxima da quantidade de *bytes* utilizados pelos WebSockets do que está, por exemplo, no gráfico da Figura 6.12. Esta redução significativa na quantidade de *bytes* utilizados pelo *polling* está relacionada com o facto de que, na situação retratada na Figura 6.12, era sempre transmitida a informação de 15 pacientes mesmo que, como é o caso em estudo, não houvesse atualizações da informação. Nesta situação, os *bytes* utilizados referem-se aos pedidos e às respostas que chegam ao Painel de Enfermagem e que não têm informação relevante para a aplicação. Tal como pode ser observado no gráfico da Figura 6.15, também no gráfico da Figura 6.16 o *long polling* surge como a técnica que mais *bytes* utiliza durante uma hora. As razões para que isso aconteça são as mesmas já referidas no parágrafo anterior.

6.4 Resumo

Para comprovar que a solução implementada obtém melhores resultados, os testes têm de ser agrupados segundo duas métricas:

- Latência;
- Tráfego gerado para transmitir a informação.

A Glintt HS deu a possibilidade de trabalhar com uma das suas aplicações para que pudesse testar a solução: o Painel de Enfermagem. O Painel de Enfermagem é uma aplicação que permite aos enfermeiros conhecer o estado atual dos pacientes. O Painel de Enfermagem foi adaptado para que conseguisse receber informação atualizada através de WebSockets e assim poder testar e avaliar a solução implementada.

A latência pode ser definida como o intervalo de tempo entre o início do evento e o momento em que os efeitos desse evento tornam-se visíveis. No *polling*, a latência começa a ser medida quando a aplicação *Web* envia o pedido HTTP a pedir a informação e só acaba de ser medida quando a informação é mostrada. Já no caso dos WebSockets e do *long polling*, a latência começa a ser medida quando o conteúdo de uma das tabelas que interessa inspecionar é alterado e só é parada quando a informação atualizada é mostrada no ecrã da aplicação *Web*. Os resultados obtidos através desta métrica mostram uma diferença considerável entre os valores obtidos pelo *polling* (2 214 ms no Google Chrome e 2 136 ms no Internet Explorer) e os valores obtidos pelos WebSockets (350 ms no Google Chrome e 741 ms no Internet Explorer) ou pelo *long polling* (1 148 ms no Internet Explorer).

A métrica da quantidade de *bytes* utilizados não conta apenas o número de *bytes* que vão desde o servidor da aplicação para a aplicação *Web* (no caso do *polling*) ou o número de *bytes* que vão

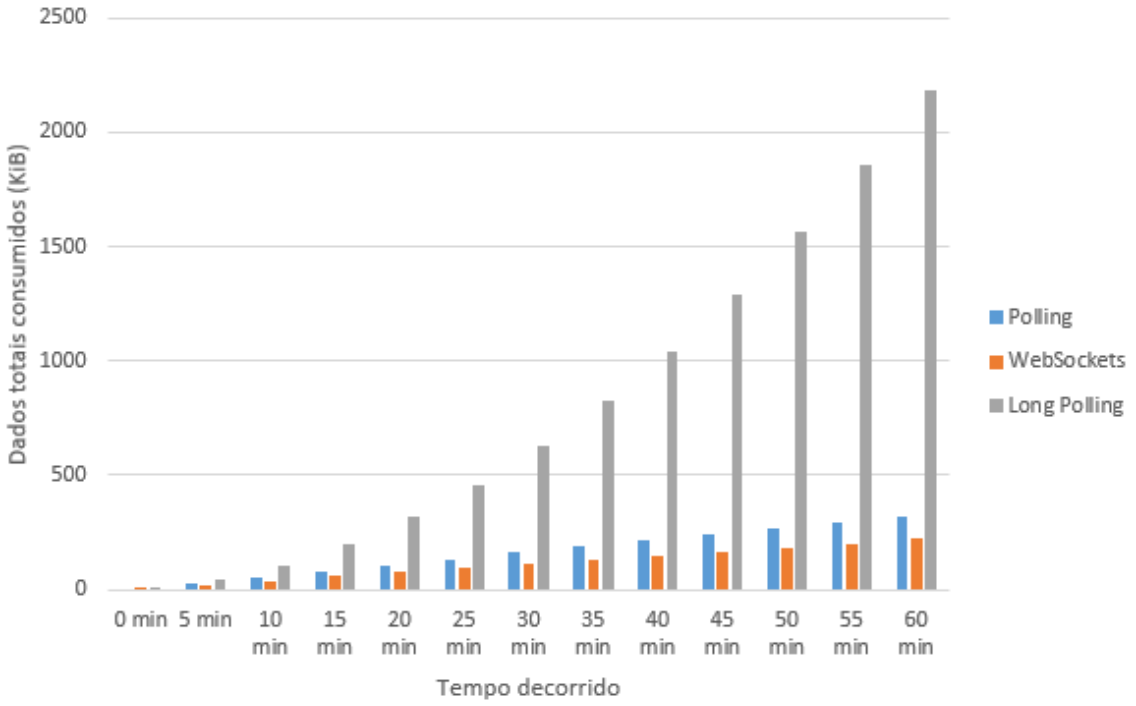

Figura 6.15: Tráfego gerado pelo *polling*, pelo *long polling* e pelos WebSockets com atualizações de informação de um paciente

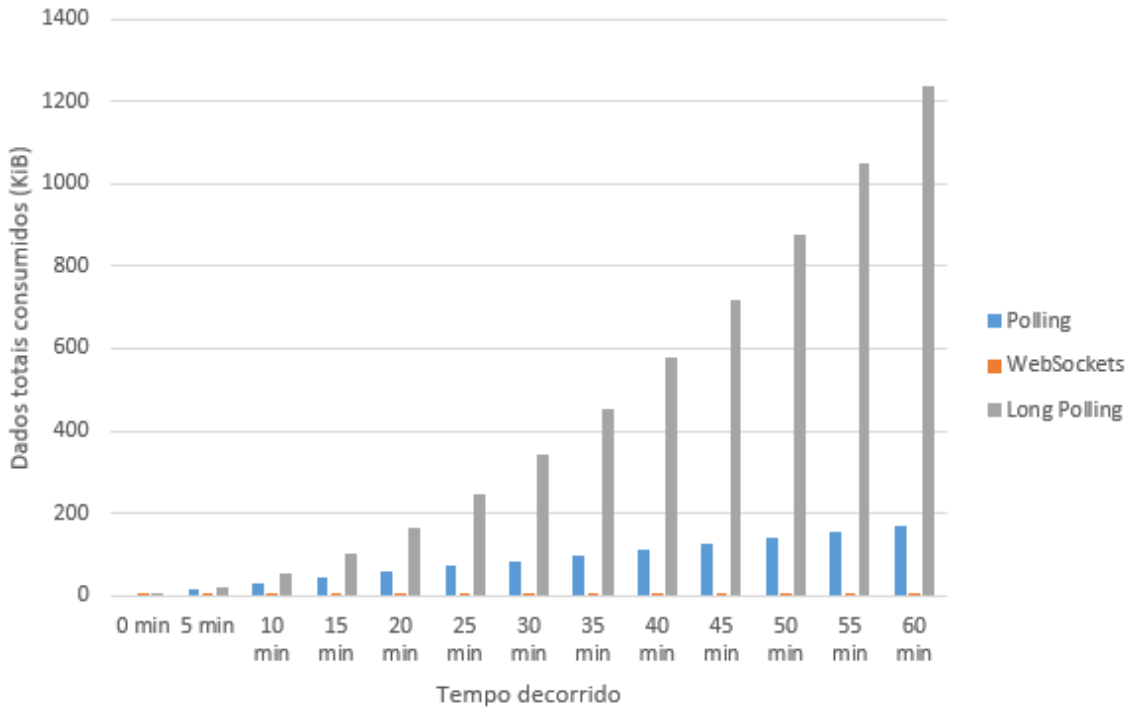

Figura 6.16: Tráfego gerado pelo *polling*, pelo *long polling* e pelos WebSockets sem atualizações de informação de um paciente

desde o servidor Node.js até à aplicação *Web* (no caso dos WebSockets e do *long polling*). Os *bytes* contados são todos os *bytes* utilizados para que a informação consiga chegar à aplicação *Web* incluindo, como no caso dos WebSockets e do *long polling*, os *bytes* utilizados para manter a ligação WebSocket ativa. Na contagem dos *bytes* utilizados existe uma exceção, devido à pouca precisão na contagem dos *bytes* utilizados entre o servidor da API e o servidor da base de dados (a ligação é feita através do protocolo TNS), os *bytes* utilizados nesta transmissão não são contados em todas as medições. Mesmo com esta exceção, a quantidade de *bytes* utilizados pelo *polling* (neste teste, o *polling* conhecia a frequência com que os dados eram alterados) é muito superior à quantidade de *bytes* utilizados pelo *long polling* e pelos WebSockets. Ao final de uma hora, a frequência com que os dados eram alterados era de 1 minuto, o *polling* tinha utilizado aproximadamente 15 650 KiB, o *long polling* tinha utilizado aproximadamente 2 200 KiB e os WebSockets tinham utilizado aproximadamente 220 KiB. A diferença ainda é maior quando se faz um teste para ver a quantidade de *bytes* utilizados quando, durante uma hora, não existem atualizações de informação uma vez que a quantidade de *bytes* utilizados pelo *polling* continua a mesma (aproximadamente 15 650 KiB) enquanto no *long polling*, o valor desce para aproximadamente 1 240 KiB e nos WebSockets desce para aproximadamente 5 KiB.

Capítulo 7

Conclusões e Trabalho Futuro

A *Web* evoluiu muito nos últimos anos e, com ela, a forma como a informação é mostrada. No início, a *Web* apenas permitia mostrar páginas com texto e com hiperligações para outras páginas. Depois, o conteúdo das páginas *Web* começou a tornar-se mais "rico", uma vez que já era possível mostrar imagens e o texto já podia aparecer com diferentes estilos.

Com o processo de migração das aplicações *stand-alone* para aplicações *Web* foram desenvolvidas várias técnicas e tecnologias que permitiram alterar o conteúdo da página *Web*, até aqui estático, de forma a mostrar informação em tempo real. Graças à facilidade com que se implementa e aos excelentes resultados que demonstrou desde o início, o *polling* tornou-se na técnica mais utilizada para mostrar informação em tempo real.

Desde o aparecimento do *polling* até aos dias de hoje, foram muitas as empresas, incluindo a Glintt HS, que utilizaram esta técnica para atualizar a informação nas aplicações *Web*. Todavia, o aumento do número de utilizadores e a fiabilidade da informação exigida em algumas aplicações *Web* fez com que o número de pedidos HTTP começasse a aumentar, o que acabou por expor as limitações do *polling*. As limitações do *polling* estão relacionadas com a diminuição do desempenho do servidor, o aumento do tráfego de dados na rede e a incapacidade do cliente em fazer o *rendering* da informação recebida na página *Web*. Estes três fatores podem levar a que parte de informação chegue atrasada, ou ainda pior, que a informação não seja disponibilizada ao utilizador final.

No caso das aplicações *Web* desenvolvidas pela Glintt HS, a perda de informação é um cenário que não pode acontecer, uma vez que a Glintt HS desenvolve aplicações *Web* para a área da saúde. Nesta área, onde qualquer erro pode colocar em risco a vida de um doente, a falha de informação pode levar a que o estado de saúde dos doentes piore ou, no pior dos casos, levar à morte dos doentes.

Com base no problema acima descrito, foram definidos quatro objetivos para esta dissertação:

- Pesquisar as técnicas que permitem a uma aplicação *Web* mostrar informação em tempo real;

- Analisar as vantagens e as desvantagens das várias técnicas encontradas para perceber qual a técnica que melhor se adapta ao problema encontrado;
- Depois de escolhida a técnica, implementar um módulo que permita que as aplicações *Web* da Glintt HS mostrem informação em tempo real de uma forma mais eficaz e eficiente sem causar perturbações no sistema existente;
- Para que possa testar e avaliar a viabilidade do módulo implementado, o último objetivo passa pela adaptação de uma das aplicações *Web* desenvolvidas pela Glintt HS – a aplicação *Web* utilizada foi o Painel de Enfermagem – para receber a informação atualizada através do módulo.

A pesquisa às várias técnicas que permitem às aplicações *Web* mostrar informação em tempo real e a análise das vantagens e das desvantagens das várias técnicas permitiu concluir que a melhor técnica é a tecnologia WebSocket. Esta tecnologia adapta o conceito dos *sockets* TCP às necessidades da *Web*. Os WebSockets permitem criar uma ligação bidirecional e *full duplex* entre os clientes e os servidores, o que dá a possibilidade aos servidores de informarem os clientes, sem a existência de um pedido previamente enviado pelos clientes, de que existem atualizações na informação que está a ser mostrada.

No entanto, para que os WebSockets possam funcionar, é necessário a existência de um sistema que permita saber quando é que a informação foi alterada. Uma vez que o *polling* na Glintt HS não envia apenas a informação atualizada (envia toda a informação quer esteja, ou não, atualizada), o primeiro passo passou pela criação de um sistema que conseguisse saber quando é que a informação foi alterada. Contudo, uma vez que não é possível aceder à API aonde as aplicações pedem para adicionar, alterar ou eliminar informação, a solução passou por inspecionar as tabelas das bases de dados, com auxílio do *package* DBMS_CHANGE_NOTIFICATION, cujo conteúdo fosse relevante para as aplicações *Web* para que, quando o conteúdo fosse alterado (através de um *insert*, *update* ou *delete*), a informação atualizada pudesse ser enviada para as aplicações *Web*.

Depois deste sistema estar concluído, teve de se implementar um módulo, por cada instituição hospitalar, capaz de comunicar essas mesmas alterações às aplicações *Web*. O módulo consiste num servidor desenvolvido em Node.js que vai enviar, através de ligações WebSocket previamente estabelecidas com as aplicações *Web*, a informação que acabou de ser atualizada. As ligações WebSocket foram estabelecidas com o auxílio da biblioteca Socket.io. Em relação à arquitetura que existia, as únicas alterações prendem-se com a adição deste novo servidor e com as ligações que este servidor tem de ter com as aplicações *Web*, com os servidores das bases de dados de onde recebe as indicações do que foi alterado e com os servidores da API onde tem de ir buscar a informação mais detalhada, com base na indicação que recebeu, para enviar às aplicações *Web*.

A parte final do trabalho consistiu na adaptação de uma das aplicações *Web* à solução implementada para que fosse possível elaborar testes para comprovar a fiabilidade da solução. A aplicação *Web* utilizada foi o Painel de Enfermagem. Para comprovar se os WebSockets são realmente melhores que o *polling* foram efetuados vários testes tendo por base duas métricas: a latência e o

tráfego gerado na transmissão da informação. Com base nos resultados obtidos, pode observar-se que os WebSockets necessitam de muito menos tempo, quando comparados ao *polling*, para mostrar informação atualizada e que o tráfego gerado pelos WebSockets é muito inferior ao tráfego gerado pelo *polling*. Um pormenor a ter em conta é que, ao contrário do que acontece frequentemente, o *polling*, durante os testes que mediam o tráfego gerado e que implicavam a atualização da informação guardada na base de dados, conhecia a frequência com que a informação era alterada o que permitia que, de cada vez que o Painel de Enfermagem enviava um pedido HTTP, a resposta trouxesse sempre informação nova, reforçando a relevância dos resultados obtidos. Estes resultados sustentam a ideia de que a solução implementada permite melhorar a interação das aplicações *Web* com os profissionais de saúde e minimiza a carga dos servidores e da rede. Isto foi possível graças ao melhoramento dos processos que levam a informação atualizada para as aplicações *Web*.

7.1 Trabalho Futuro

A principal recomendação para o trabalho futuro está relacionada com uma das dificuldades que foram sentidas no desenvolvimento da solução e que está associada com a deteção das alterações da informação. A solução vai ao nível da base de dados para verificar se existe alguma alteração da informação, no entanto, a solução ideal seria ter acesso à API aonde as aplicações pedem para adicionar, alterar ou eliminar a informação. Caso existisse a possibilidade de ter acesso a essa API, a verificação das alterações só era feita quando um determinado conjunto de pedidos fosse feito em vez de verificar, constantemente, se existem, ou não, alterações ao conteúdo da base de dados. O acesso à API, conjugado com o facto de haver implementações de WebSockets em C# (linguagem na qual a API foi desenvolvida), iria permitir que o servidor Node.js deixasse de existir e que, em vez de se utilizarem dois pedidos HTTP (da Oracle para o servidor Node.js e do servidor Node.js para o servidor da API), passar-se-ia a utilizar apenas um (do servidor da API onde a informação é alterada para o servidor da API aonde se vai buscar a informação).

Uma outra recomendação, para o trabalho futuro, prende-se mais com um trabalho de continuidade em relação ao que já foi implementado até aqui. Uma vez que com este trabalho ficou comprovado que os WebSockets apresentam melhores resultados do que o *polling*, a minha sugestão passa por colocar as aplicações *Web* a utilizar o servidor Node.js e os WebSockets para mais funcionalidades do que utilizar apenas para atualizar a informação, uma vez que, deixar o servidor Node.js e os WebSockets encarregados de mais algumas funções, pode trazer melhorias significativas no desempenho do sistema de informação da Glintt. Por exemplo, a hora que aparece no Painel de Enfermagem, ver Figuras 6.1 e 6.2, é alterada com recurso a um pedido HTTP que é enviado de minuto a minuto, em vez disso, o servidor Node.js podia ficar encarregado de, a cada minuto, enviar a data e a hora à aplicação *Web*.

Referências

[ABC+00] Michael Awai, Matthew Bortniker, John Carnell, Sean Dillon, Drew Erwin, Jaeda Goodman, Bjarki Hólm, Ann Horton, Frank Hubeny, Thomas Kyte, Glenn E. Mitchell II, Kevin Mukhar, Gary Nicol, Daniel O'Connor, Guy Ruth-Hammond e Mario Zucca. *Professional Oracle 8i Application Programming with Java, PL/SQL and XML*. Wrox Press, 2000.

[Aga12] Sachin Agarwal. Real-time web application roadblock: Performance penalty of HTML sockets. In *Communications (ICC), 2012 IEEE International Conference on*, pages 1225–1229, 2012. doi:10.1109/ICC.2012.6364271.

[BL90] Tim Berners-Lee. Information Management: A Proposal. Disponível em http://www.w3.org/History/1989/proposal.html, acedido a última vez em 2 de fevereiro de 2015, 1990.

[BMD08] Engin Bozdag, Ali Mesbah e Arie Van Deursen. Performance Testing of Data Delivery Techniques for AJAX Applications. *Journal of Web Engineering JWE*, 0:287–315, 2008. URL: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-009.pdf.

[BTS+13] Louay Bassbouss, Max Tritschler, Stephan Steglich, Kiyoshi Tanaka e Yasuhiko Miyazaki. Towards a Multi-Screen Application Model for the Web. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, pages 528 – 533, 2013. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6605845&tag=1, doi:10.1109/COMPSACW.2013.96.

[Cal81] University of Southern California. RFC 793 - Transmission Control Protocol. page 85, 1981. URL: http://tools.ietf.org/html/rfc793.

[Cal15] John Callaham. Microsoft Edge is already beating IE, Chrome and Firefox in JavaScript benchmarks | Windows Central. Disponível em http://goo.gl/UfvRVA, acedido a última vez em 13 de junho de 2015, 2015.

[Car11] Mathieu Carbou. Reverse Ajax, Part 1: Introduction to Comet. Disponível em http://www.ibm.com/developerworks/web/library/wa-reverseajax1/index.html, acedido a última vez em 1 de fevereiro de 2015, julho 2011.

[CMW11] Padraig Corcoran, Peter Mooney e Adam Winstanley. Effective Vector Data Transmission and Visualization Using HTML5. *GIS Research UK*, pages 3–7, 2011. URL: http://csserver.ucd.ie/~pcorcoran/papers/GISRUK2011_Corcoran_2011.pdf.

[Dev15] Alexis Deveria. Can I use... Support tables for HTML5, CSS3, etc. Disponível em http://caniuse.com/#feat=websockets, acedido a última vez em 13 de junho de 2015, 2015.

[DKP+01] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham e Prashant Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. Disponível em http://www-ccs.cs.umass.edu/~krithi/web/WWW10/www10/, acedido a última vez em 26 de junho de 2015, 2001.

[DPC+13] Porfírio Dantas, Jorge Pereira, Everton Cavalcante, Gustavo Alves e Thais Batista. Light-PubSubHubbub: A lightweight adaptation of the PubSubHubbub protocol. In *Proceedings of the 8th International Conference on Software Engineering Advances (ICSEA 2013)*, pages 432–438, 2013.

[Edi15] Porto Editora. Definição ou significado de web no Dicionário da Língua Portuguesa com Acordo Ortográfico - Infopédia. Disponível em http://www.infopedia.pt/dicionarios/lingua-portuguesa/web, acedido a última vez em 29 de janeiro de 2015, 2015.

[FANR12] Roy Fielding, Adobe, Mark Nottingham e Rackspace. RFC 6585 - Additional HTTP Status Codes. *Internet Engineering Task Force*, pages 1–10, 2012. URL: http://tools.ietf.org/html/rfc6585.

[Fas13] Marc Fasel. Performance Comparison Between Node.js and Java EE For Reading JSON Data from CouchDB. Disponível em http://blog.shinetech.com/2013/10/22/performance-comparison-between-node-js-and-java-ee/, acedido a última vez em 28 de maio de 2015, 2013.

[Fel12] Manuel Vargas Felício. LightStream – Sistema de comunicação Publish / Subscribe com WebSockets. Technical report, Instituto Superior de Engenharia de Lisboa, Lisboa, 2012. URL: http://repositorio.ipl.pt/bitstream/10400.21/2165/1/Dissertaç~{a}o.pdf.

[FGM+99] Roy Fielding, James Gettys, Jeff Mogul, Henrik Frystyk, Larry Masinter, P. Leach e Tim Berners-Lee. RFC2616 - Hypertext transfer protocol–HTTP/1.1. *Internet Engineering Task Force*, pages 1–114, 1999. URL: http://tools.ietf.org/html/rfc2616.

[FJH11] Y Furukawa, Spring Jasri e Sayo-cho Hyogo. Web-Based Control Application Using Websocket. In *Proceedings of ICALEPCS2011*, pages 673–675, Grenoble, France, 2011. URL: http://epaper.kek.jp/icalepcs2011/papers/wemau010.pdf.

[FM11] I Fette e A Melnikov. RFC 6455 - The WebSocket Protocol. *Internet Engineering Task Force*, 1:1–71, 2011. URL: http://tools.ietf.org/html/rfc6455.

[GLG11] Carl Gutwin, Michael Lippold e T. C. Nicholas Graham. Real-Time Groupware in the Browser : Testing the Performance of Web-Based Networking. *Proceedings of the ACM 2011 conference on Computer supported cooperative work CSCW 11*, pages 167–176, 2011. URL: http://dl.acm.org/citation.cfm?id=1958850, doi:10.1145/1958824.1958850.

[Goo15] Google Inc. Códigos de status HTTP - Ajuda do Webmaster Tools. Disponível em https://support.google.com/webmasters/answer/40132?hl=pt-PT, acedido a última vez em 7 de janeiro de 2015, 2015.

[Gra14] Rob Gravelle. Comet Programming: Using Ajax to Simulate Server Push. Disponível de http://www.webreference.com/programming/javascript/rg28/index.html, acedido a última vez em 30 de dezembro de 2014, 2014.

[HÏ1] Harri Hämäläinen. HTML5 : WebSockets. *Communication*, pages 1–9, 2011.

[Hal15] Tim Hall. ORACLE-BASE - DBMS_CHANGE_NOTIFICATION in Oracle 10g Database Release 2. Disponível em https://oracle-base.com/articles/10g/dbms_change_notification_10gR2, acedido a última vez em 22 de junho de 2015, 2015.

[Han14] David Haney. To Node.js Or Not To Node.js | Haney Codes .NET. Disponível em http://www.haneycodes.net/to-node-js-or-not-to-node-js/, acedido a última vez em 28 de maio de 2015, 2014.

[HCW12] Tom Hughes-Croucher e Mike Wilson. *Node Up and Running*. 2012. URL: http://www.oreilly.com/catalog/errata.csp?isbn=9781449398583.

[HG12] Matthias Heinrich e Martin Gaedke. Data binding for standard-based web applications. In *Proceeding SAC'12 Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 652–657, 2012. URL: http://dl.acm.org/citation.cfm?id=2245402, doi:10.1145/2245276.2245402.

[Inc10] Symantec Inc. Symantec Global Internet Security Threat Report: Volume, 2010.

[Inc13] O'Reilly Media Inc. Brief Story of HTTP. Disponível em http://chimera.labs.oreilly.com/books/1230000000545/ch09.html, acedido a última vez em 8 de janeiro de 2015, 2013.

[Inf13] Infosys. Server side cache Invalidation through Oracle Push Notification. Technical report, Infosys, 2013. URL: http://www.infosys.com/manufacturing/resource-center/Documents/oracle-push-notification.pdf.

[Jo11] Mihkel Jõhvik. *Push-based versus pull-based data transfer in AJAX applications*. PhD thesis, Faculty of Mathematics and Computer Science - University of Tartu, 2011.

[KLI00] R. Khare, S. Lawrence e Agranat Systems Inc. RFC 2817 - Upgrading to TLS Within HTTP/1.1. *Network Working Group*, pages 1–13, 2000. URL: http://tools.ietf.org/html/rfc2817.

[KPM13] Kostas Kapetanakis, Spyros Panagiotakis e Athanasios G. Malamos. HTML5 and WebSockets; challenges in network 3D collaboration. In *Proceedings of the 17th Panhellenic Conference on Informatics - PCI '13*, page 33, 2013. URL: http://dl.acm.org/citation.cfm?doid=2491845.2491888, doi:10.1145/2491845.2491888.

[LC11] Saturnino Luz e Oscar Cassetti. The WebSocket API as supporting technology for distributed and agent-driven data mining. In *Transport*, 2011. URL: https://www.scss.tcd.ie/~luzs/publications/CassettiLuz11wap.pdf.

[LD14] Richard K. Lomotey e Ralph Deters. Mobile-Based Medical Data Accessibility in mHealth. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 91–100. Ieee, abril 2014. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6834950, doi:10.1109/MobileCloud.2014.24.

[LG13] Peter Lubbers e Frank Greco. HTML5 Web Sockets: A Quantum Leap in Scalability for the Web. Disponível em http://www.websocket.org/quantum.html, acedido a última vez em 2 de fevereiro de 2015, 2013.

[MAdSP15] Thiago Franco Moraes, Paulo Henrique Junqueira Amorim, Jorge Vicente Lopes da Silva e Hélio Pedrini. Visualização Interativa em Tempo Real de Dados Médicos na Web. Technical report, 2015. URL: http://www.cti.gov.br/invesalius/files/pub/Visualizaç~{a}oInterativaemTempoRealdeDadosMédicosnaWeb.pdf.

[MT11] Tommi Mikkonen e Antero Taivalsaari. Reports of the web's death are greatly exaggerated. *Computer*, 44:30–36, 2011. doi:10.1109/MC.2011.127.

[Mul14] Gabriel L. Muller. *HTML5 WebSocket protocol and its application to distributed computing*. Master thesis, Cranfield University, setembro 2014. URL: http://arxiv.org/abs/1409.3367v1, arXiv:1409.3367.

[NHL10] Mark Nottingham e E. Hammer-Lahav. RFC 5785 - Defining Well-Known Uniform Resource Identifiers (URIs). *Internet Engineering Task Force*, pages 1–8, 2010. URL: http://tools.ietf.org/html/rfc5785.

[Ora15] Oracle Corporation. DBMS_CHANGE_NOTIFICATION. Disponível em http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/d_chngnt.htm#BABEECBE, acedido a última vez em 14 de abril de 2015, 2015.

[PN12] Victoria Pimentel e Bradford G. Nickerson. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing*, 16(4):45–53, 2012. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6197172, doi:10.1109/MIC.2012.64.

[PSSA+10] Ian Paterson, Dave Smith, Peter Saint-Andre, Jack Moffitt, Lance Stout e Winfried Tilanus. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). *XMPP Standards Foundation*, abril 2010. URL: http://xmpp.org/extensions/xep-0124.pdf.

[Qui13] Thomas Quintana. HTML5 WebRTC State of the Art, 2013. URL: http://www.sipforum.org/component/option,com_docman/task,doc_view/gid,622/Itemid,261/.

[Ram13] Leonardo Batecini Ramos. *Um Chat Pictográfico para o SCALA (Sistema de Comunicação Alternativa para o Letramento de pessoas com Autismo)*. PhD thesis, Universidade Federal do Rio Grande do Sul, 2013. URL: http://www.lume.ufrgs.br/handle/10183/86431.

[RG11] J. Reschke e Greenbytes. RFC 6266 - Use of the Content-Disposition Header Field in the Hypertext Transfer Protocol (HTTP). *Internet Engineering Task Force*, pages 1–14, 2011. URL: http://tools.ietf.org/html/rfc5785.

[Rot14] Issac Roth. StrongLoop | What Makes Node.js Faster Than Java? Disponível em https://strongloop.com/strongblog/node-js-is-faster-than-java/, acedido a última vez em 14 de abril de 2015, 2014.

[Rus06] Alex Russell. Comet: Low Latency Data for the Browser – Infrequently Noted. Disponível em http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/, acedido a última vez em 4 de dezembro de 2014, 2006.

[San15] Roberto Sanchez. Comparing Node.js vs PHP Performance. Disponível em http://www.hostingadvice.com/blog/comparing-node-js-vs-php-performance/, acedido a última vez em 28 de maio de 2015, 2015.

[She02] Paul D. Sheriff. Designing for Web or Desktop? Disponível em https://msdn.microsoft.com/en-us/library/ms973831.aspx, acedido a última vez em 20 de julho de 2015, 2002.

[She12] Dmitry Sheiko. WebSockets vs Server-Sent Events vs Long-polling. Disponível em http://dsheiko.com/weblog/websockets-vs-sse-vs-long-polling/, acedido a última vez em 6 de fevereiro de 2015, 2012.

[SM11] Raju Narayana Swamy e G Mahadevan. Event Driven Architecture using HTML5 Web Sockets for Wireless Sensor Networks. *White Papers, Planetary Scientific Research Center*, pages 3–5, 2011. URL: http://isems.org/images/extraimages/3.pdf.

[Smi15] Jeff Smith. Desktop Applications Vs. Web Applications. Disponível em http://www.streetdirectory.com/travel_guide/114448/programming/desktop_applications_vs_web_applications.html, acedido a última vez em 20 de julho de 2015, 2015.

[The14] Vasco Morais Themudo. *Implementação de um servidor de negociação em bolsa baseado em WebSocket*. Tese de mestrado, Faculdade de Ciências da Universidade do Porto, 2014.

[WPJR11] Andrew Wessels, Mike Purvis, Jahrain Jackson e Syed (Shawon) Rahman. Remote Data Visualization through WebSockets. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 1050–1051, 2011. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5945383, doi:10.1109/ITNG.2011.182.

[WSM13] Vanessa Wang, Frank Salim e Peter Moskovits. *The Definitive Guide to HTML5 WebSocket*. Apress, 2013.

[Yer14] Prahlad Yeri. PHP vs node.js: The REAL statistics. Disponível em http://blog.prahladyeri.com/2014/06/php-vs-node-js-real-statistics.html, acedido a última vez em 28 de maio de 2015, 2014.

[ZS13] Lijing Zhang e Xiaoxiao Shen. Research and development of real-time monitoring system based on WebSocket technology. In *Mechatronic Sciences, Electric Engineering and Computer (MEC), Proceedings 2013 International Conference*, pages 1955–1958. Ieee, dezembro 2013. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6885373, doi:10.1109/MEC.2013.6885373.

[ZSST13] Weiping Zhang, Regina Stoll, Norbert Stoll e Kerstin Thurow. An mhealth monitoring system for telemedicine based on WebSocket wireless communication. *Journal of Networks*, 8:955–962, 2013. doi:10.4304/jnw.8.4.955-962.