Universidade do Porto

Faculdade de Engenharia

FEUP

WeEat

Relatório final

Sistemas Distribuídos

3º ano do Mestrado Integrado em Engenharia Informática e Computação

Elementos do Grupo:

David Azevedo – up201405846 – up201405846@fe.up.pt

João Ferreira – up201404332 – up201404332@fe.up.pt

José Martins – up201404189 – up201404189@fe.up.pt

Marcelo Ferreira - up201405323 - up201405323@fe.up.pt

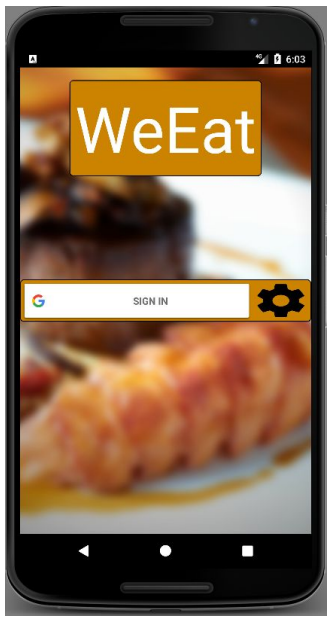
21 de Maio de 2017

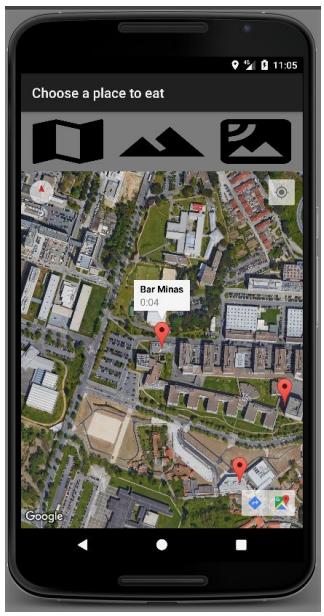
Índice

Introdução

A aplicação WeEat permite que os seus utilizadores encontrem pessoas com quem possam almoçar/jantar caso não tenham companhia para a refeição. Com o WeEat, é possível criar grupos numa determinada localização em que se pretende ir comer, sendo automaticamente criado um chat para essa localização, onde os interessados poderão conversar e combinar todos os detalhes. É também possível visualizar no mapa grupos previamente estabelecidos e juntar-se ao chat desse encontro para discutir os pormenores com os participantes.

Para este serviço ser possível, foi desenvolvida uma API RESTful que oferece as funcionalidades necessárias, relativas à criação de grupos de almoço assim como a dinâmica de uma *chatroom* onde os nossos utilizadores podem interagir em tempo real. A dinâmica e interação dos servidores foi pensada e implementada tendo em conta questões relativas a : tolerância a falhas, segurança, escalabilidade e consistência.



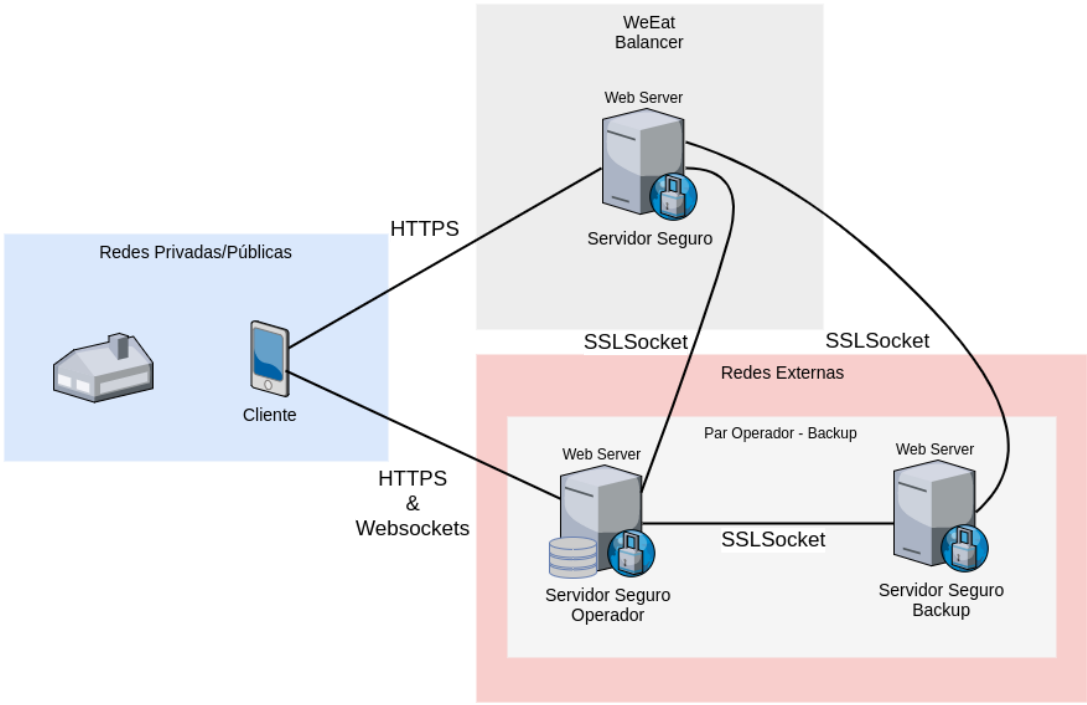


Neste relatório, irá ser descrito o projeto da seguinte forma: inicialmente falar-se-á sobre a arquitecta escolhida, posteriormente serão clarificados os detalhes mais importante da implementação do trabalho, seguido de uma descrição da abordagem seguida no âmbito da segurança, tolerância a falhas, escalabilidade e consistência, por fim, concluiremos este relatório com uma análise crítica do nosso trabalho assim como possíveis melhoramentos que o grupo considera exequíveis e pertinentes.

Arquitetura

Para uma comunicação fluente entre cliente e servidor, foi criado um conjunto de protocolos e conexões, de modo a garantir a disponibilidade do serviço e a respectiva tolerância a falhas.

WeEat Server - Arquitetura de Rede



Com a figura é possível observar de uma forma mais intuitiva a arquitetura do sistema, existem clientes que, com ajuda do DNS, comunicam com o Balancer através de HTTPS. Entre o Cliente e o WebServer, é criada também, uma ligação, com o uso do o protocolo HTTPS e WebSockets devido à necessidade de haver notificações do servidor para com o cliente. O servidor Balancer e os WebServers Operadores/Backups, estão conectados entre si, através de SSLSockets, para sincronização de dados e redirecionamento de utilizadores. Nas máquinas dos WebServers temos também a nossa instância da base dados a correr em postgreSQL, para esta comunicação recorremos à API *jdbc* o que permite compatibilidade em diversos sistemas operativos em Java.

Relativamente à conexão à base de dados postgreSQL, como já referido, com o auxílio da API *jdbc*, o servidor Operador possui uma instância a correr localmente, de notar, que cada instância da base de dados está feita para servir uma região do mapa, a sua implementação está preparada para servir a nível mundial, no entanto, a escalabilidade é garantida horizontalmente através de replicação de nós. O servidor de backup, não possui um servidor postgreSQL a correr enquanto estiver a funcionar como Backup, tem então uma ligação através de SSLSockets com o servidor Operador, o qual é responsável por fazer o backup dos

dados e posteriormente enviar esse backup através da ligação para o seu servidor de backup, cuja informação é enviada através do Balancer. Esta operação, quer o backup quer o envio para o servidor de backup, é realizada de forma repetitiva entre intervalos de tempo específicos na ordem dos segundos. Para garantir a consistência dos dados foi usado o mecanismo de transações de postgreSQL que seguem as propriedades ACID (atomicidade, Consistência, Isolabilidade, Durabilidade), com o recurso a *prepared statements*.
Segue-se uma descrição de cada um dos métodos de comunicação utilizados:

SSLSockets

SSLSockets são utilizados para comunicação entre os diferentes servidores, são estes: balancer, operador, e backup.

A conexão entre o balancer e operador/backup, serve para registar um servidor, armazenando toda a informação relativa às diferentes portas (porta https, porta websocket e porta de backup), IP, modo e localização do servidor.

Um Servidor que se conecte ao balancer, para uma dada localização, caso seja o primeiro, irá passar a ser operador, após o registo no balancer, quando um outro servidor se conectar ao balancer, para a mesma localização, irá ficar registado como servidor de backup. No caso excepcional de existir um terceiro servidor a tentar entrar no balancer para esta localização, este receberá uma resposta de ***FULL***, que indica que esta localização se encontra cheia, e por isso o servidor se deve desligar.

Os servidores de backup e operador, estão também ligados entre si, ou seja, receberão o IP e porta do outro, para que seja criada uma ligação SSL, de modo a permitir o backup de dados, passando a realizar uma arquitetura Cliente-Servidor, o Operador será o servidor e o Backup será o cliente. Ambos têm conhecimento um do outro por uma medida de segurança, à partida o servidor não iria precisar das informações do cliente, simplesmente abria o socket e “escutava” por conexões, no entanto, e como existia um foco da nossa parte na segurança, decidimos enviar uma mensagem ao Operador quando um servidor de Backup se conectar ao balancer, para que este possa verificar as informações da nova conexão e o Operador só enviará o ficheiro de backup se confirmar, desta forma, a identidade do servidor de Backup.

Caso um dos servidores falhe, o balancer irá notificar o outro servidor, de modo a que caso seja o servidor operador a falhar, o balancer irá tornar o servidor de backup como servidor operador, alterando o seu estado. O backup irá iniciar a base de dados local com o dump mais recente de backup que possui e continuará a servir os pedidos dos utilizadores. Isto garante que exista maior disponibilidade do sistema. Quando, por sua vez, o servidor de backup vai abaixo, o Operador desliga a sua ligação e deixa de enviar backups, até que um novo servidor de backup se ligue.

HTTPS Servers

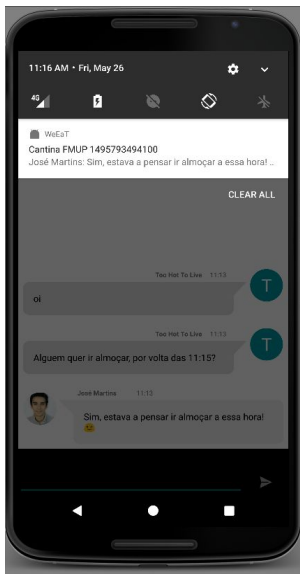
A utilização do protocolo HTTPS é crucial para a criação de aplicações REST seguras, é através desta camada que o utilizador comunica com o balancer e com o servidor operador, ou seja, para uma primeira conexão, o utilizador conecta-se com ajuda do DNS ao balancer, este devolve todas as localizações suportadas, de modo a permitir ao utilizador escolher, qual das localizações a conectar. Após a escolha de uma localização por parte do utilizador, é enviada uma mensagem por HTTPS para o balancer, que retorna informação acerca do servidor operador, ao qual o utilizador se irá conectar, isto inclui informações como, IP, porta do websocket e porta HTTPS.

Com a conexão estabelecida entre o operador e o utilizador, são transmitidas várias mensagens entre estas duas entidades. Numa primeira ligação o utilizador é adicionado à base de dados e são transferidos todos os chats já existentes, para que o utilizador se possa conectar a um deles. Todos os pedidos do utilizador são verificados pelo seu token, de modo a garantir um maior controlo e segurança sobre os acessos.

WebSockets

Devido à necessidade de notificar o utilizador de uma forma rápida, logo que outro utilizador envie uma mensagem para um dos chats em que este se encontra, o WeEat implementa um mecanismos de notificações usando websockets, estes permitem a comunicação bidirecional através de canais full-duplex, sobre um único socket.

Desta forma, no momento em que o servidor é iniciado é criado um websocket que aceitará todas as ligações dos clientes, estas ligações são realizadas cada vez que um cliente entra em um chat. Depois de a ligação estar estabelecida o cliente envia uma mensagem contendo os dados do chat (latitude e longitude) de que pretende receber notificações, o servidor armazena então a sessão do cliente num hashmap em que para cada key (chat representado pelo latitude e longitude) existe um array com todas as sessões de todos os utilizadores que pretendem receber notificações referentes aquele determinado chat.



Assim sempre que o cliente envia uma mensagem para o servidor utilizando https, o websocket notifica todos os clientes que se encontram no hashmap para o chat em questão, enviando um json com todos os dados relevantes para que seja possível reconstruir a mensagem nos outros clientes.

Protocolos

Comunicação por HTTPS

A comunicação por HTTPS utiliza um conjunto de pedidos específicos a cada plataforma, para isso foi criado um distribuidor, que, para uma dada classe Mensagem, a redireciona para o método, onde passa a ser tratada.

Este parser permitiu a criação de um pequeno protocolo em cima do protocolo HTTPS, que facilitou bastante o desenvolvimento e legibilidade de código. Segue-se uma descrição de cada protocolo usado, dependendo do lado da recepção de mensagens.

Balancer	**REQUEST_SERVER <LOCATION>**	Pedido do Servidor no qual o cliente se irá conectar.
	REQUEST_LOCATIONS	Pedido das diferentes localizações registadas.
Client	**RESPONSE**	Resposta genérica a um pedido.
	UNLOGGED	Aviso de logout, quando o token de sessão não é confirmado.
	FILL_MAP_MARKERS <JSON_chats>	Recepção das localizações dos encontros de almoço, para povoar o mapa.
	UPDATE_CHAT <JSON_Messages_ChatMembers>	Contém toda a informação necessária para inicializar o chat.
	ADD_SERVER_LOCATIONS <JSON_LOCATIONS>	Todas as localizações registadas no balancer.
	START_SERVER_CONNECTION <JSON_IP_HTTPSPORT_WEBSOCKETPORT>	Contém informação relativa ao IP, à porta HTTPS e à porta do Websocket, de um servidor, de uma dada localização.
Servidor	**ADD_USER <JSON_MAIL_NAME_PICTURE>**	Recebido quando existe uma adição de um cliente novo à base de dados. Contém informação sobre o e-mail, nome e imagem do utilizador.
	ADD_CHAT_GROUP <JSON_latitude_longitude_timestam_title>	Mensagem que contém um request para adicionar um novo chat, num dado tempo, latitude e longitude.
	ADD_CHAT_MEMBER <JSON_latitude_longitude_member>	Responsável por adicionar um utilizador a um dado servidor, localizado numa certa latitude e longitude

ADD_CHAT_MESSAGE<JSON_ content_latitude_longitude>	Permite a adição de uma mensagem a um servidor, em que a sua localização está definida por latitude e longitude.
GET_CHAT_GROUPS	Pedido para retorno de todos os chats de grupo existentes.
GET_CHAT_MEMBERS <JSON_latitude_longitude>	Pedido para retorno dos utilizadores que participam num dado chat.
GET_CHAT_DATA <JSON_latitude_longitude>	Pedido de dados de um determinado chat.

TLS: Servidores e Balancer

Para a comunicação entre Servidores e o Balancer por TLS, foi criado um pequeno protocolo, cada mensagem é recebida como uma linha em string, os primeiros quatro caracteres representam o campo, no qual indica a ação a realizar e a restante string, contém os argumentos, dessa ação, em que cada argumento é separado por um espaço. Com a ajuda de um pequeno distribuidor, as mensagens recebidas são redirecionadas para o método responsável pelo seu tratamento.

Fazem parte deste protocolo as seguintes mensagens:

MODE <MODO>

Responsável por informar o modo no qual o servidor irá operar.
“BACKUP” o servidor passa a modo *backup*.
“OPERATOR” o servidor passa a modo operador.
“FULL” local já contém servidor operador e backup.

BACK <IP>:<PORT>

Responsável por informar o servidor *operador* sobre o IP e porta do servidor de *backup* para que possa aceitar a conexão deste servidor de forma mais segura.

OPER <IP>:<PORT>

Responsável por informar o servidor *backup* sobre o IP e porta do servidor *operador* para que possa conectar a este servidor.

RBAC

Responsável por informar o servidor *operador* sobre a falha do servidor de *backup*.

ROPE

Responsável por informar o servidor de *backup* sobre a falha do servidor *operador*.

WebSockets

Os webSockets são utilizados no momento da receção de um pedido https **ADD_CHAT_MESSAGE** no servidor, enviando assim a mensagem para todos os seus clientes que “subscreveram” o chat em questão.

Implementação

Para implementação da aplicação distribuída foi utilizada a linguagem Java. Para o desenvolvimento dos servidores utilizou-se, como ambiente de desenvolvimento, o IDE IntelliJ IDEA, de modo a facilitar a implementação de alguns conceitos, num menor tempo, utilizaram-se as seguintes APIs descritas abaixo, associadas a uma pequena descrição:

GoogleAPI : De modo a verificar os dados do cliente, do lado do servidor.

json : Capaz de fazer parse/geração de strings em formato json.

postgresql: Comunicação e interação mais eficiente, para com a base de dados.

jetty: Para a criação de websockets e notificações ao cliente, de forma eficiente, o que permite um número elevado de utilizadores simultâneos.

Para o desenvolvimento da aplicação móvel foi utilizado AndroidStudio, com versão SDK 16 mínima compatível. Tal como no servidor, também foram utilizadas algumas APIs focadas na produtividade dos developers, segue-se abaixo uma pequena descrição das APIs:

Google API Auth: Para fazer login e confirmação da identidade do utilizador a partir da sua conta do google.

Google API Maps: Permite a representação dos chats, nos seus locais de encontro.

OkHttp: Utilizado como cliente websocket, para que seja possível a recepção de notificações por parte do cliente.

Com a finalidade de maximizar o desenvolvimento em paralelo, o projeto foi dividido em pequenos módulos, de forma a facilitar a atribuição de tarefas a cada membro do grupo. Assim, o trabalho foi divido em, Servidores HTTPS, Websockets, TLS, acessos e manipulação da base de dados, uso e implementação dos módulos da aplicação android, comportamentos do Cliente e Servidores, Sistema de redirecionamento de mensagens e integração da API do Google.

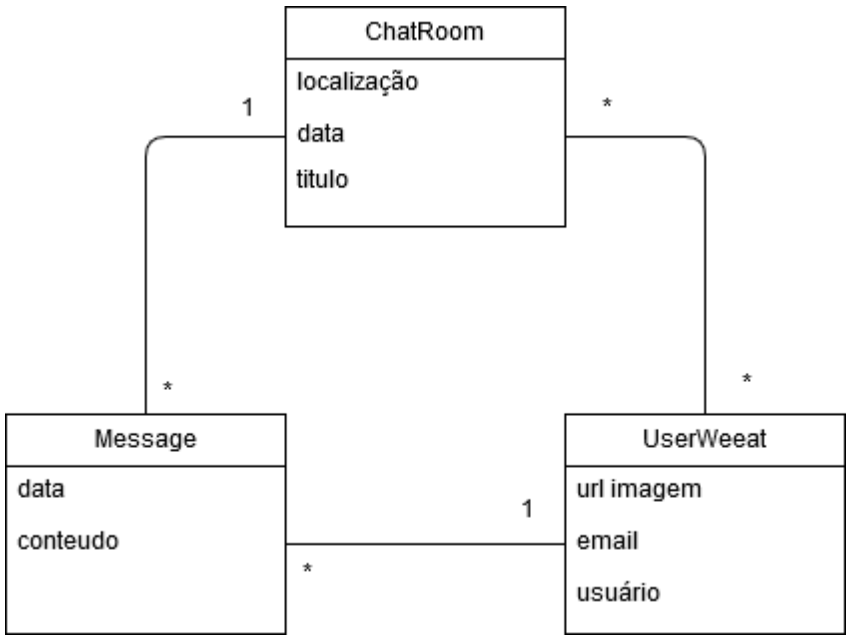
De forma a facilitar o tratamento de mensagens, de uma forma prática, foi criado um simples distribuidor, que mapeia um comando (em formato string ou inteiro), a uma função que mais tarde trata o pedido. Este padrão é utilizado em vários locais, devido à flexibilidade que oferece.

A conexão HTTPS, utiliza internamente o padrão anteriormente referenciado, necessitando apenas de definir os tipos e métodos, responsáveis pelo tratamento dos diferentes tipos de mensagens. O uso de *truststore* e *keystore* permitiu garantir um canal de conexão segura, e é esta conexão HTTPS, por onde o cliente comunica primariamente com os servidores. A conexão por HTTPS ao Balancer, numa primeira idealização, seria realizada a partir de um endereço DNS e uma porta fixa, não permitindo ao utilizador alterar estes dados, mas para efeitos de demonstração e teste, foi incluída uma opção na aplicação android para a possibilidade de especificar o IP e porta do Balancer.

Como referido anteriormente, para a comunicação do par de servidores com o balancer, foi utilizado um canal de comunicação por SSLSockets. Como o Balancer necessita de uma pequena troca de mensagens e de saber o estado de vida de cada servidor conectado, optou-se pela utilização de *threads* para o tratamento destes pedidos, isto porque, o número de mensagens trocadas entre os servidores e o balancer é pequena e as *threads* vão estar bloqueadas o tempo todo após a primeira conexão. Esta ligação é capaz de detetar a existência do servidor e na ocorrência de algum problema, o Balancer identifica qual o servidor que foi abaixo, e irá indicar ao servidor par, do ocorrido, procedendo à alteração do seu comportamento. Esta gestão de *threads* encontra-se na classe /server/src/network/sockets/ConnectionArmy.java. Esta classe inicializa os listeners, nos quais os servidores se conectam para determinar a sua função (operador/backup). Após um servidor se conectar, é criada uma instância da classe /server/src/network/load_balancer/ServerConnection.java para confirmar a ligação (linha 48), armazenar todos os dados vitais do servidor e atribuir uma função ao servidor conectado. Em caso de impossibilidade, é enviada a mensagem MODE FULL, mencionada anteriormente. Todos os servidores conectados com sucesso, são armazenados numa classe /server/src/network/load_balancer/ServerPair.java, que, por sua vez está mapeada pela sua localização, no hashmap da classe ConnectionArmy. Assim, para cada pedido de servidor, por localização, posteriormente feito do cliente ao Balancer é enviado o servidor operador dessa localização ou null em caso de impossibilidade.

Relativamente à comunicação entre os servidores Operador e Backup, a implementação pode ser encontrada nos ficheiros /server/src/network/protocol/SecureClientBermuda.java e /server/src/network/protocol/SecureServerBermuda.java, estas duas classes derivadas de SecureClient e SecureServer respectivamente representam a ligação entre o Operador e o Backup e são responsáveis pelo envio e receção do ficheiro de backup atualizado da base de dados. O servidor tem um mecanismo que o permite aceitar e verificar a conexão e posteriormente iniciar uma TimerTask que enviará o ficheiro de backup de forma repetitiva. Por sua vez o cliente inicia a ligação e fica indefinidamente a receber o novo ficheiro.

A base de dados seguiu uma implementação simplista, como pode ser observada na seguinte imagem :



Consiste em 3 entidades, a ChatRoom que representa um chat/encontro de almoço, este contém as informações relativas a apenas uma refeição, o UserWeeat são os nossos utilizadores com as suas informações pessoais, e por fim, a Message representa uma mensagem que é enviada por um utilizador para uma sala de conversação.

Questões Relevantes

Como foi descrito anteriormente, de modo a tolerar falhas e a tornar o sistema escalável, consistente e seguro, é utilizado:

Um balancer, que contém toda a informação dos servidores atuais, e comunica com os pares de servidores, de modo a garantir um servidor operador em caso de falha. Apesar deste comportamento já ter sido descrito no ponto anterior, vale a pena relembrar que os ficheiros responsáveis por toda esta comunicação são:

- /server/src/network/sockets/ConnectionArmy.java:
  Responsável pelo mapeamento, confirmação e atribuição dos servidores.

- /server/src/network/load_balancer/ServerConnection.java
  Responsável pela gestão da conexão com um servidor operador/backup.

- /server/src/network/load_balancer/ServerPair.java
  Responsável por anotar o par de servidores(operador/backup).

Backups automáticos de forma a garantir a consistência da base de dados em caso de falha, classe principais:

- /server/src/network/protocol/SecureServerBermuda.java

Responsável pelo envio dos backups da base de dados(servidor operador).

- /server/src/network/protocol/SecureClientBermuda.java

Responsável pela receção dos backups da base de dados(servidor de backup).

- /server/src/database/DatabaseManager.java

Responsável pelo inicialização, *restore* e backup da base de dados localmente.

- /server/src/database/DatabaseConnection.java

Responsável pela ligação de cada servidor à instância local da base de dados, garante a consistência dos dados utilizando o mecanismo de transações do postgreSQL.

A segurança no envio da informação, com a utilização de *keystore* e *truststore* está contida nas seguintes classes:

- **Servidor**
  - /server/src/network/socket/SecureServer.java
  - /server/src/network/socket/SecureServer.java
  - /server/src/network/ServerWeEat.java
- **Cliente**
  - /network/Client.java

Conclusão

As aplicações distribuídas estão em todo lado, na sociedade tecnologicamente avançada em que vivemos. Hoje em dia, quase todas as aplicações que usamos seja no browser ou no smartphone são distribuídas, ligando as pessoas e os serviços entre si. Posto isto, o conhecimento e familiarização com este tipo de conceitos e tecnologias torna-se fulcral no enriquecimento técnico e na formação de um Engenheiro Informático atual, pelo que o desenvolvimento deste projeto permitiu ao grupo entrar nesta nova via de desenvolvimento de software tão usada e ainda despertar o interesse de todos os elementos do grupo, ao dar a conhecer, informação vital para a criação de uma aplicação distribuída. Contribuindo para uma visão mais ampla e a colocar em prática algumas das técnicas que são utilizados no mundo real, no nosso dia a dia. Neste projeto o grupo teve contacto com conceitos importantes na implementação de uma aplicação distribuída, nomeadamente : WebSockets, SSL/TLS, RESTful API, entre outros.

Consideramos que alguns aspectos poderiam ainda ser melhorados e implementados por forma a garantir uma maior estabilidade e performance, tanto a nível de comunicação mas principalmente ao nível da ”user experience”, que uma vez que não era o foco deste trabalho acabou por ser um pouco descorada, não tendo sido dedicado o tempo pretendido para realizar o design e conceção das interfaces que serão manipuladas pelo utilizador.