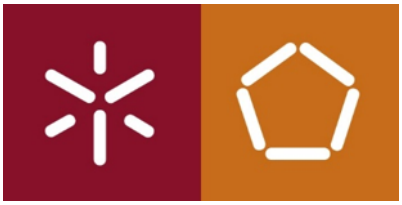

UNIVERSIDADE DO MINHO

PARADIGMAS DE SISTEMAS DISTRIBUÍDOS

Trabalho Prático

RELATÓRIO DE DESENVOLVIMENTO

Mestrado Integrado em Engenharia Informática

Realizado pelo G01:
Hugo André Coelho Cardoso, a85006
José Pedro Oliveira Silva, a84577
Pedro Miguel Borges Rodrigues, a84783
Válter Ferreira Picas Carvalho, a84464

18 de janeiro de 2021

Conteúdo

Lista de Figuras

1 Introdução

No âmbito da Unidade Curricular *Paradigmas de Sistemas Distribuídos*, inserida no perfil de mestrado de Sistemas Distribuídos, lecionado no Mestrado Integrado em Engenharia Informática da Universidade do Minho, foi proposta a resolução do projeto prático **Alarme Covid**.

Este projeto consiste em simular a aplicação **STAYAWAY COVID**, mantendo os utilizadores informados sobre possíveis infeções no seu distrito e cruzamentos com usuários infetados nos vários distritos do país (neste caso é Portugal). Assim, é necessária a utilização de várias componentes para a solução final: um servidor de *front-end*, que é o serviço de autenticação do cliente e intermediário de interrogações por parte do cliente; servidores distritais que mantêm informações sobre cada distrito, enviam notificações públicas e respondem a interrogações do servidor *front-end*; diretório, que dispões uma API REST, onde é guardada informação estatística dos vários distritos.

A solução obtida, com base nesta arquitetura, tira partido das ferramentas lecionadas em contexto de aula, nomeadamente **Erlang** como linguagem para o desenvolvimento servidor de *front-end*; **ZeroMQ** nos servidores distritais e *front-end* para notificações públicas e privadas; **Dropwizard** para expor o servidor REST no diretório; **Java** para a restante lógica nos servidores distritais, clientes e diretório. Foram ainda utilizadas bibliotecas extra, nomeadamente **Protocol Buffers**, para serialização de mensagens trocadas entre utilizadores, *front-end* e servidores distritais e **Apache HttpClient** para realizar pedidos REST, com auxílio de **GSON** para serializar e desserializar JSON.

De seguida será explicitado a arquitetura utilizada, assim como a justificação para todas as escolhas feitas pelos elementos do grupo, de modo a atingir a solução final.

2 Descrição da Solução

2.1 Cliente

A linguagem de eleição para esta entidade é o Java, tal como proposto no enunciado do projeto.

Toda a interface é realizada à base de Menus em formato de texto e *inputs* do cliente para o tipo de operação que deseja realizar, atualizando o seu estado consoante as respostas por parte do servidor, pelo que é gerido sequencialmente com pedidos e respostas todo o fluxo de mensagens que esta entidade produz e recebe.

As mensagens recebidas e enviadas são prototipadas com base nos **Protocol Buffers** da Google, de modo a uniformizar as mensagens trocadas entre o cliente e servidor, facilitando o processo de desserialização e serialização das mesmas.

O cliente, relativamente ao servidor de *front-end*, é composto essencialmente pelas conexões geradas por dois *Sockets* distintos, com as seguintes tarefas:

- Enviar mensagens de autenticação do cliente, enviar interrogações acerca do número de utilizadores numa dada localização e notificar que o cliente está doente. No final de cada pedido, é lido desse *Socket* a resposta do servidor sequencialmente. É responsável, também, pelo envio da porta de notificações privadas do cliente (para o servidor conseguir distinguir vários clientes concorrentes) e a atualização da localização atual do mesmo, pelo que ambas não necessitam de resposta por parte do servidor, logo, não são lidas do *Socket* respostas a estas mensagens;

- Permitir a leitura de notificações privadas do *front-end*, avisando o cliente que esteve em contacto com alguém infetado. Estas notificações são prioritárias, pelo que são mostradas ao cliente quando possível.

Para as notificações públicas, é utilizado um *socket* **ZeroMQ** do tipo **SUB**, dando ao cliente a possibilidade de se inscrever em tempo real a eventos provenientes dos vários servidores distritais, com a limitação imposta de três subscrições no máximo por cliente. Caso pretenda, pode também remover a sua subscrição em qualquer momento. Este *Socket* conecta-se a um **Broker** que será visto na secção dos servidores distritais.

É utilizada uma classe de geração aleatória de posições em grelhas N por N, tendo o cuidado extra de apenas gerar posições adjacentes à atual do utilizador, como método de manter o serviço atualizado sobre a posição do cliente em que, após um intervalo de tempo arbitrário, é enviada uma mensagem ao *front-end* com a sua localização nova gerada por esta classe.

São utilizadas várias *threads* para controlar o fluxo do programa, nomeadamente para leitura dos vários *Sockets*, geração aleatória de posições, entre outros.

Quando o utilizador se declara doente, não volta a poder interagir com o servidor de *front-end* nem receber notificações.

2.2 Servidor Front-End

O servidor *front-end* é implementado em **Erlang** e é responsável pela autenticação dos clientes, mantendo a sua informação em tempo de execução, e pela mediação de toda a troca de informação entre o cliente e o servidor distrital, à exceção das notificações públicas.

A comunicação entre o cliente e a *front-end* é feita exclusivamente em **sockets TCP** por mensagens prototipadas por **protobuffers** - estabelecem-se dois canais de *sockets*, um para envio de informação para o servidor e comportamento **REQUEST REPLY** e outro unidirecional para o envio de notificações privadas para o cliente.

Entre a *front-end* e cada servidor distrital, há também dois canais - um para a troca de informação por mensagens criadas pelos **profobuffers** entre **sockets TCP** e outro com comportamento **PUSH PULL** entre **sockets ZeroMQ**, implementado em **chumak** do lado da *front-end*.

Ao iniciar o servidor, criam-se vários processos: um para gerir as contas, onde é mantida a informação de autenticação dos utilizadores; outro para gerir a conexão aos distritos - liga-se aos **sockets TCP** de todos os servidores distritais e guarda os endereços num mapa; outro ainda que abre um socket **ZeroMQ** do tipo **PULL**, implementado com **chumak**, e se conecta à segunda porta dada por argumento, onde fica em ciclo à espera de receber informação dos servidores distritais para notificações privadas.

Uma vez inicializados todos estes programas, o servidor fica em ciclo a aceitar conexões de clientes, gerando um processo para cada um e redirecionando-os para o módulo da área não autenticada. Este módulo processa apenas pedidos de registo e de *login*, atualizando os dados de estado e, em caso de *login* bem-sucedido, redirecionando o processo do cliente para o módulo da área autenticada.

Na área autenticada, a primeira mensagem que o servidor recebe é a especificar a porta do cliente para as notificações privadas, tratando de abrir um **socket TCP** para a mesma e guardando o respetivo endereço no gestor de distritos, dentro do mapa do distrito em questão.

Este módulo pode processar vários tipos de pedidos, podendo reencaminhar a informação para o respetivo servidor distrital (nos casos da atualização da localização atual e da comunicação de doença) e responder ao cliente com o resultado da operação (nos casos da comunicação de doença, contagem do número de pessoas numa dada localização e *logout*).

Caso o cliente saia do sistema (por *logout* ou quarentena), o programa fecha o seu socket de notificações privadas e remove o registo do respetivo cliente do mapa do distrito.

2.3 Servidores Distritais

Para esta entidade, primeiro foi necessário criar um **Broker**, que atua como **proxy** entre os *publishers* e *subscribers*, permitindo assim que vários clientes se conectem a vários servidores. Para conseguir este funcionamento, foram utilizadas as funcionalidades do **ZeroMQ**, nomeadamente a utilização de *Sockets* do tipo **XPUB**, ao qual os vários clientes se conectam como subscritores e **XSUB**, ao qual os vários servidores se conectam como publicadores.

Este **Broker** é responsável pela disseminação de notificações em tempo real, para os clientes, de eventos ocorridos em todos os servidores distritais, tais como a diminuição ou aumento de concentração numa localização dum distrito, ocorrência de mais um utilizador infetado ou inexistência de utilizadores numa dada localização.

Quando o servidor de *front-end* se conecta aos vários servidores distritais, é gerado um *Socket* **ZeroMQ** do tipo **PUSH** em cada um deles, utilizadas para o envio de mensagens que contêm os utilizadores possivelmente infetados quando um utilizador declara que está doente, sendo geradas no servidor *front-end* as notificações privadas para esses clientes. Assim, é garantido no *front-end* que há *fair queuing*, já que utiliza o tipo **PULL** complementar. Esta decisão vem em consequência de vários distritos poderem ter ativadade mais intensa que outros, o que pode levar à *starvation* dos restantes, causando impactos na performance e os utilizadores não serem notificados em tempo real de outros distritos.

Para além deste *Socket* cujo único propósito é o envio de informação para o *front-end*, é necessário utilizar outro que serve como método de comunicação entre ele e o servidor distrital, mas desta vez é apenas um *Socket* normal do Java. Este novo *Socket* é utilizado para receber informação quando um cliente muda de localização, um utilizador se declara doente ou quando é realizada a interrogação do número de utilizadores numa dada localização para o distrito em questão, pelo que é preciso enviar de volta esse valor ao *front-end* no caso da última situação.

Todas as mensagens que recebe e envia (exceto as notificações públicas) são prototipadas com base nos **Protocol Buffers** da Google, que facilita o processo de serialização para o envio ao servidor de *front-end* e desserialização das mensagens que recebe do mesmo.

Todos os servidores distritais mantêm a informação dos vários clientes pertencentes ao seu distrito, como o histórico, os utilizadores que ficaram doentes e a concentração nas várias localizações,

permitindo assim responder eficientemente ao *front-end* quando interrogado, enviar as notificações privadas e públicas quando necessário e manter atualizado o servidor estatístico.

Por fim, de modo a manter atualizado o servidor de consulta estatística, o diretório, é utilizada a biblioteca **Apache HTTP Client**, uma vez que este só dispõe uma interface REST, utilizando a serialização e desserialização do formato JSON em Java através da ferramenta **GSON**, também da Google, para o envio da informação pertinente nos vários pedidos.

2.4 Diretório

O diretório é o local de consulta estatística, implementado utilizando a *framework* **Dropwizard** para Java, onde expõe uma API REST para qualquer utilizador externo.

Este é um servidor totalmente separado das restantes entidades, pelo que foi necessário desenvolver vários tipos de pedido, presentes na tabela seguinte:

Pedido	URL	Descrição
DELETE	http://localhost:8080/api/districts/{id}/users/{user}	Elimina um utilizador (user) de um distrito (id)
GET	http://localhost:8080/api/districts/{id}/users	Retorna um JSON com os utilizadores de um dado distrito (id)
GET	http://localhost:8080/api/districts/{id}/infected	Retorna um JSON com os utilizadores infetados de um dado distrito (id)
GET	http://localhost:8080/api/districts/avg	Retorna o nº médio global de utilizadores que se cruzaram com alguém infetado
GET	http://localhost:8080/api/districts/top5infected	Retorna um JSON com os 5 distritos que registam o maior rácio de utilizadores infetados por número de utilizadores
GET	http://localhost:8080/api/districts/top5locations	Retorna um JSON com as 5 localizações e o distrito respetivo onde se registou a maior concentração de utilizadores
POST	http://localhost:8080/api/districts/{id}/users	Armazena o utilizador e a localização respetiva no seu histórico, enviados na forma de JSON, no respetivo distrito (id)
POST	http://localhost:8080/api/districts/{id}/-concentration	Armazena a localização e a respetiva concentração atual, enviadas na forma de JSON, no respetivo distrito (id)
PUT	http://localhost:8080/api/districts/{id}/infected/{user}	Atualiza o utilizador existente (user) colocando-o como infetado no respetivo distrito (id)

Este servidor recebe informação enviada sobre o formato da tabela acima, pelo que guarda para todos os distritos o histórico de posições de cada utilizador, as concentrações máximas até ao momento de cada localização e os utilizadores infetados.

Foram implementados métodos extra como valorização e porque o grupo considerou importantes. Assim, para responder às interrogações definidas, podem ser utilizados os vários pedidos **GET**, cujas descrições correspondem à interrogação definida n

3 Arquitetura Final

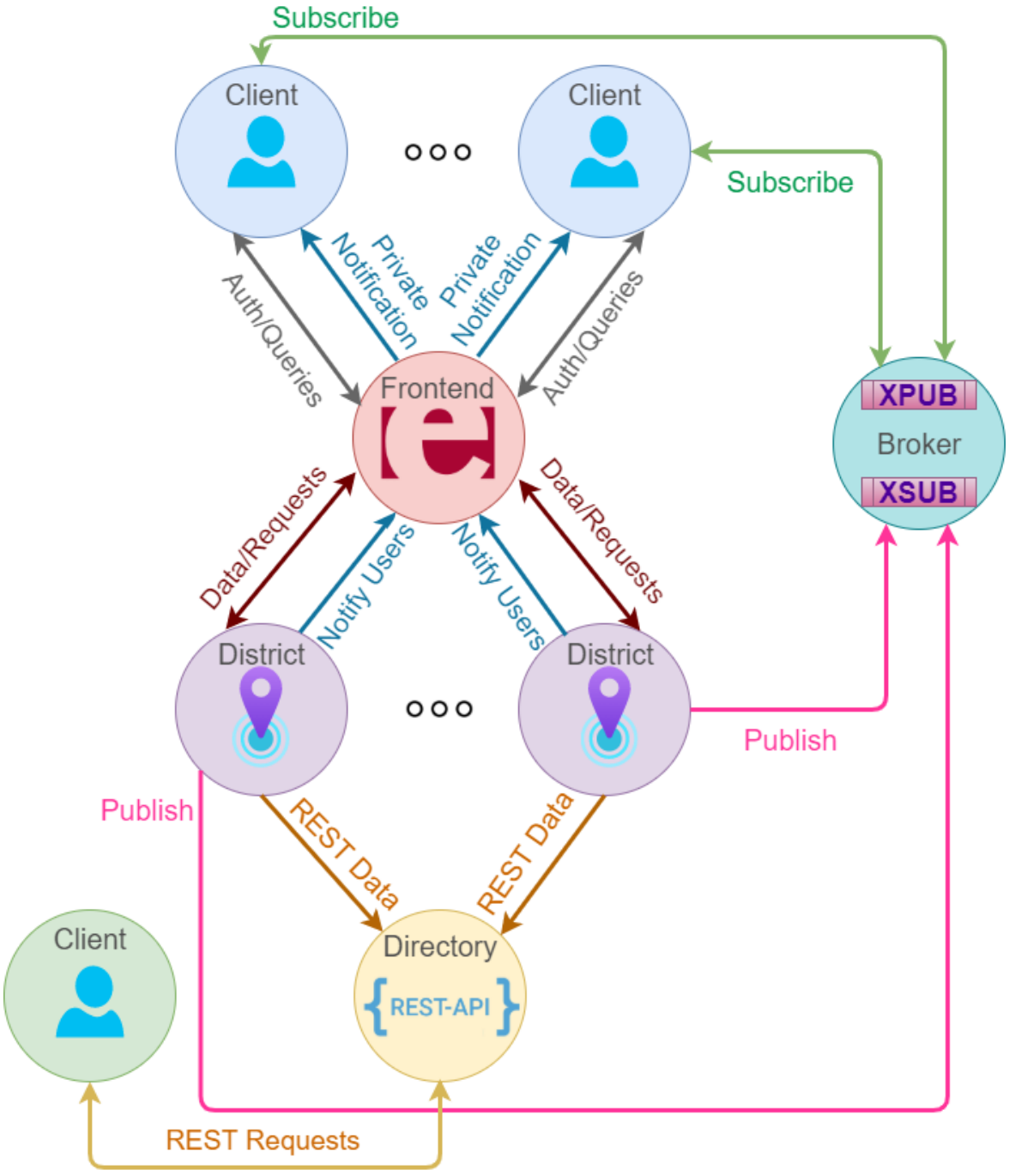

Figura 1: Arquitetura final da aplicação

4 Conclusão

O desenvolvimento deste projeto permitiu a consolidação da matéria teórico-prática lecionada nas aulas da unidade curricular, nomeadamente da utilidade de ferramentas como o ZeroMQ, Dropwizard e Erlang em contextos práticos.

O grupo está bastante satisfeito com o resultado final e considera que não só cumpriu os requisitos propostos no enunciado, como foi para além disso e desenvolveu uma solução escalável e dinâmica, uma vez que os módulos são independentes.

Concluindo, foi possível obter conhecimentos valiosos através da elaboração deste trabalho prático relativamente à programação por atores e troca de mensagens, assim como conhecimentos transversais, nomeadamente trabalho em grupo e utilização de controladores de versões para uma melhor gestão de tarefas.