Mestrado Integrado em Engenharia Informática

Paradigmas de Sistemas Distribuídos

Peerlending - Relatório de Desenvolvimento

Grupo 1

Jorge Oliveira (A78660)
José Ferreira (A78452)
Nuno Silva (PG38420)

20 de Janeiro de 2019

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Apresenta-se de seguida o relatório de desenvolvimento do trabalho prático de Paradigmas de Sistemas Distribuídos. O objetivo do trabalho foi a elaboração de um protótipo de um serviço de intermediação de empréstimos a empresas por parte de investidores. Neste relatório é apresentada e fundamentada a arquitetura desenvolvida, a constituição e caraterísticas de cada um dos componentes que fazem parte do sistema assim como a forma como estes comunicam entre si.

Capítulo 2

Arquitetura

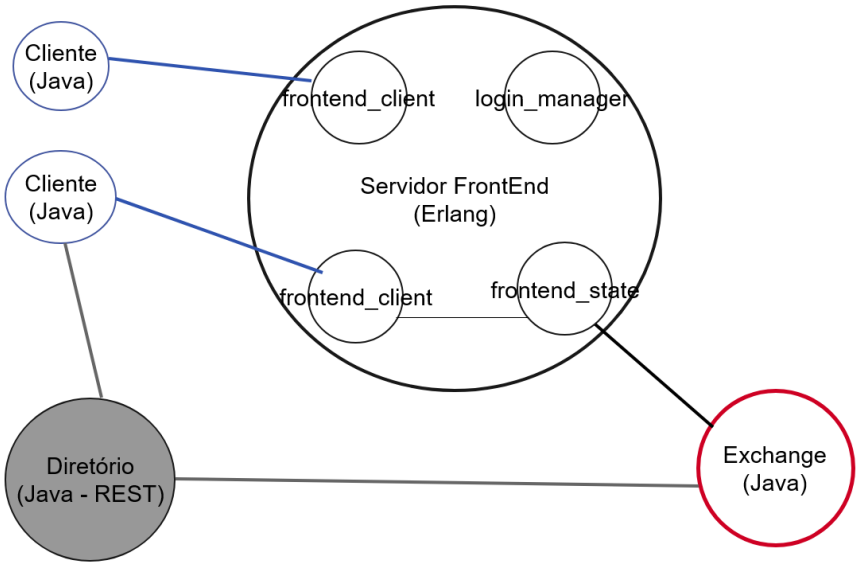

Figura 2.1: Arquitetura do sistema

A arquitetura desenvolvida para o sistema encontra-se descrita na figura 2.1. Existem quatro componentes principais: o Cliente, o *FrontEnd*, a *Exchange* e o Diretório, que serão descritos em maior detalhe nos próximos capítulos do presente relatório. Os Clientes comunicam com o *FrontEnd* via *sockets TCP* e com diretório usando pedidos *HTTP (API REST)*. Já a *Exchange* comunica com o Diretório da mesma forma que o Cliente (*HTTP com API REST*) e com o *FrontEnd* usando *ZeroMQ*. Cada ator *FrontEnd State* possui sockets *ZeroMQ* de *Push* e *Pull*, tal como cada uma das *Exchanges*. Os atores presentes no servidor de *FrontEnd* comunicam entre si usando o mecanismo de mensagens nativo do *Erlang*. De referir que só existe um Diretório e um servidor de *FrontEnd* no sistema, sendo que o número de Clientes e *Exchanges* é variável. Também o número de atores presentes no servidor de *FrontEnd* é variável, aumentando consoante o número de Clientes e *Exchanges* ligadas ao mesmo. De referir que na imagem não é apresentada uma ligação entre o *Cliente* e a *Exchange*, mas que a mesma existe sendo com *sockets ZeroMQ* do tipo *PUB/SUB* para o serviço de notificações.

Para permitir uma maior heterogeneidade entre os componentes, e visto que alguns não são escritos na mesma linguagem, utilizamos protocolos de serialização de dados usando *Protocol Buffers* para permitir uma ligação entre os diferentes componentes. Esta serialização é utilizada em todas as comunicações excetuando as relativas ao **Diretório** e internas.

Capítulo 3

Componentes

3.1 Diretório

O Diretório, codificado em JAVA utilizando a *framework* ***Dropwizard*** é fundamental para manter alguns dados que são carregados pela *Exchange* e acedidos pelo utilizador.

Temos 4 representações, ***Leilao*** contêm dados sobre qual o id, a empresa correspondente, uma lista das propostas apresentadas, entre outras, a ***Emissao*** que é parecida com a anterior, a ***Proposta*** referente às várias propostos (licitador, montante, etc) e ainda a ***Exchange*** onde guarda uma lista de empresas que estão associadas a um endereço de uma *Exchange*.

No entanto temos 3 recursos que é são referentes ao **Leilao**, **Emissao** e **Exchange**. O Leilao e a Emissao são recursos semelhantes pelo que só iremos abordar um, sendo que o outro segue a mesma lógica.

O ***LeilaoResource*** tem dois mapas com o par Empresa-Lista de leilões, um referente aos leilões ativos e outro para os leilões finalizados. Recursos disponíveis:

GET /leilao - pode conter um parâmetro, com a identificação da empresa, ou não. E retorna todos os leilões ativos, ou então só os da empresa especificada

GET /leilao/finalizados - lógica idêntica ao anterior, mas neste caso os leilões terminados

PUT /leilao/empresa/terminado/id - coloca o estado do leilão, especificado pelo *id* da *empresa* requerida, como terminado

POST /leilao - cria um novo *Leilao*

Como referido anteriormente, para o *EmissaoResource* a lógica é de todo idêntica.

A ***ExchageResource*** tem apenas um mapa com o par Endereço da *Exchange*-Lista de empresas associadas. Recursos disponíveis:

GET /exchange - fornece a informação sobre qual o endereço da *exchange* referente à empresa passada como parâmetro. Não retorna apenas a *string* com o valor do endereço, mas sim um objeto que contêm um array com todas as empresas que o estão associadas, desta forma reduzimos a quantidade de pedidos efetuados ao *Diretorio*

GET /exchange/todas - lógica semelhante ao anterior, mas desta vez retorna um array de objetos

POST /exchange - cria uma nova *Exchange*

3.2 Cliente

O cliente, escrito em JAVA, é responsável pela interação com o utilizador, quer seja uma empresa ou então um licitador. Este programa apresenta as ações que cada tipo de utilizador pode tomar (ex: no caso da empresa apresenta uma

opção para criar o leilão), as respostas por parte do servidor, as notificações e ainda as informações atualizadas de todos(as) leilões/emissões. Ao iniciar o programa do Cliente é lido de um ficheiro as informações de configuração: o endereço do Diretorio e também o endereço do FrontEnd.

Os utilizadores podem efetuar um pedido ao servidor e depois obterão uma resposta, para saber o *status*, tendo assim a noção de que a sua ação obteve sucesso ou não. Apesar de ser um par pedido/resposta o nosso programa não poderia ficar simplesmente à espera da resposta do servidor, já que poderão surgir mais do que uma resposta para um mesmo pedido, isto é, dando o exemplo de uma licitação num leilão por parte de um licitador, o mesmo poderá receber uma resposta a informar que teve sucesso, como de seguida que a sua licitação foi ultrapassada, bem como receber uma mensagem a informar que o leilão terminou e qual foi o resultado do mesmo.

Por esta razão foi necessário criar uma *thread* nova para poder correr em *background* e ficar à escuta das mensagens direcionadas para aquele utilizador com as informações para o seu pedido. Desta forma, não impedimos a interação com o utilizador e conseguimos apresentar as mensagens para o mesmo, sem que o programa pare para as receber. De salientar, que esta *thread* apenas é criada quando o utilizador se autentica com sucesso.

Para além, disto o utilizador tem de ser capaz de receber notificações. Mais uma vez, receber as notificações, implica que a *thread* fique bloqueada à espera de uma mensagem, então foi necessário criar uma nova para receber as notificações. Decidimos criar duas classes novas, uma *Notificacoes* e outra *GerirSubscricoes*. Na primeira é onde estabelecemos a ligação com a Exchange, através de *sockets ZeroMQ* e onde apresentámos as mensagens, enquanto que a segunda trata da interação com o cliente para escolher que subscrições deseja. De salientar, que a segunda classe é como se fosse uma extensão da nossa classe principal e que não foi necessário criar uma nova *thread* já que quando o cliente está a gerir as suas subscrições, não poderá efetuar outra ação.

Era necessário estabelecer então uma forma para que estas duas *threads* comunicassem, o que foi conseguido através de uma ligação *inproc*. Na classe *Notificacoes* temos de receber as mensagens de notificações da *Exchange* e ainda o conteúdo que o utilizador pretende subscrever. Como apenas temos um socket foi necessário efetuar uma análise da mensagem recebida, já que decidimos acrescentar um prefixo às mensagens de subscrição do cliente, *comuSub*, desta forma caso a mensagem começasse por esta *string* é porque é para adicionar/retirar uma subscrição do *socket* caso não comece então é uma mensagem de notificação. De salientar que decidimos utilizar *multi-part message* nestas mensagens, já que nos permitia, na primeira parte colocar os *bytes* utilizados para subscrição no *socket* e a segunda parte da mensagem é o corpo da mesma, ou seja, o conteúdo para ser apresentado.

Para finalizar a explicação sobre o Cliente, para evitar andar a criar sempre todas as conexões com as várias *Exchanges*, sendo que por vezes poderão nem ser utilizadas, decidimos que só efetuávamos *connect* a um endereço quando o utilizador queria subscrever uma nova empresa. Como é evidente, caso quisesse subscrever uma empresa na qual já se conhece o endereço da *Exchange* então não é criada uma nova conexão. A informação sobre qual o endereço correto da *Exchange* para uma determinada empresa é obtida através de um pedido ao diretório.

3.3 *FrontEnd*

O **FrontEnd** é escrito em *Erlang* e a sua responsabilidade é mediar as interações entre os clientes e as *Exchanges*, servindo como um mediador entre as duas entidades. Este é composto por quatro atores principais:

- Frontend
- Login Manager
- Frontend Client : como o próprio nome indica, é o ator responsável por mediar o funcionamento dos clientes do sistema, tendo ações diferentes para licitadores e empresas;
- Frontend State : a sua função é passar a comunicação que lhe chega para a exchange onde se liga. Cada exchange possui um ator

3.3.1 Ator *Frontend*

Este é o primeiro ator em funcionamento. A sua responsabilidade é aceitar as ligações que lhe cheguem via TCP e descobrir o papel do utilizador. Depois disso é este ator quem cria o servidor que vai suportar a interação desse cliente, indicando se se trata de uma empresa ou licitador, passando-lhe o socket de comunicação com o cliente. É também

este ator que comunica com o *Login Manager* de forma a autenticar os clientes. É também neste ator que é feito o parsing dos ficheiros de configuração em *JSON*. Desses ficheiros são retirados os dados das contas de cliente, que são enviadas para o *Login Manager*, e os dados relativamente à divisão das empresas por *Exchanges*, que são usados para saber o número de atores *Frontend State* que vão ser necessários. De referir que este ator *Frontend* é único no sistema, isto é, só existe uma instância dele em funcionamento.

3.3.2 Ator *Login Manager*

Como o próprio nome indica, é o ator responsável pelas tarefas de autenticação dos clientes. Guarda no seu estado um mapa que associa os nomes de utilizador (*username*) a um objeto que contém a sua *password*, se está online e o seu papel no sistema. É colocado em funcionamento mal o sistema arranca, podendo ser contactado pelo ator *Frontend* ou pelo *Frontend Client*.

3.3.3 Ator *Frontend Client*

Este ator é responsável por receber as mensagens vindas do cliente e transmiti-las para o *Frontend State* adequado. Isto é feito com o auxílio de um mapa (Pid =¿ [Empresas]) que associa a cada pid de um ator *Frontend State* a lista de empresas que este trata. Quando é recebida alguma ope

3.3.4 Ator *Frontend State*

Este ator entra em contacto com a *Exchange*, via *ZeroMQ*, de modo a transmitir-lhe as mensagens vindas dos clientes do sistema. Este ator recebe também as mensagens vindas da *Exchange* e propaga-as para o *Frontend Client* adequado, para que o cliente que solicitou uma operação possa saber o seu resultado ou caso tenha acontecido algum evento de relevo na *Exchange*. Existe um ator deste tipo em funcionamento por cada *Exchange*. Esta distribuição deve-se a uma tentativa que o sistema seja mais escalável num uso a grande escala, pois a performance da entrega das mensagens de cada *Exchange* não é afetada caso existam disparidades no número de empresas atribuídas a cada *Exchange*.

3.4 *Exchange*

A **Exchange** é escrita em *Java* e tem como objetivo tratar os leilões e emissões de um conjunto de empresas, limitado. Para definir quais as empresas que a mesma pode tratar é passado como argumento o caminho para um ficheiro, em formato *JSON*, que contém as informações necessárias para a exchange, sendo as seguintes:

- Porta na qual vai aceitar "conexões"para receber mensagens do **Front-End**(por *zeroMQ*)
- Porta na qual vai aceitar "conexões"para enviar mensagens para o *Front-End* (por *zeroMQ*)
- Porta na qual vai aceitar conexões para o serviço de notificações por parte dos clientes (usando o padrão *PUB/-SUB* do *zeroMQ*)
- Endereço e porta do **Diretório**
- Lista de empresas que a mesma vai tratar

Depois de carregar as informações acima descritas, necessita de se registar no **Diretório** utilizando para isso o envia de um pedido *http*, já descrito anteriormente. De seguida está pronta a realizar o trabalho para que foi designada, tratar dos leilões e emissões de cada empresa nela registada. Para essa função a empresa pode receber cada uma das seguintes mensagens:

- Licitação de um leilão por parte de um cliente
- Participação numa emisão por parte de um cliente
- Criação de um leilão por parte de uma empresa

- Criação de uma emissão por parte de uma empresa
- Mensagem que indica o término de um determinado(a) leilão/emissão, vindo da comunicação *inproc* permitida pelo *zeroMQ*

De seguida vamos explicar o procedimento realizado para cada tipo de mensagem.

3.4.1 Licitação de um leilão/Participação numa emissão

Decidimos explicar estes 2 casos ao mesmo tempo porque são bastante parecidos, sendo que explicaremos também as diferenças. Ao receber uma mensagem destes tipos, a mesma é descodificada sendo extraídos os seguintes campos:

- Nome da empresa
- Montante
- Taxa (apenas no caso de uma licitação a um leilão)
- Nome do utilizador (licitador)

De seguida é feito um pedido para licitar um leilão à empresa, sendo que caso esta não exista na **Exchange** é devolvida uma mensagem a descrever o mesmo.
Para tratar o pedido são feitas as seguintes verificações:

- Caso não exista um leilão/emissão em curso para a empresa ou o mesmo não seja do tipo do pedido do cliente, é devolvido insucesso.
- Se a proposta não é aceite para a lista de propostas para o leilão em curso (por ser inferior) é devolvido insucesso. O mesmo acontece caso o montante seja negativo.
- Caso a proposta entre na lista de propostas para o leilão e outras sejam ultrapassadas (saiam da lista), é devolvida uma exceção a indicar que propostas foram ultrapassadas com a lista dessas.
- Caso a proposta faça com que a emissão esteja terminada (ultrapasse o montante pedido) é devolvida a Emissão a indicar que a mesma foi terminada.

De referir apenas que no caso de uma emissão as propostas são sempre aceites.
Caso o resultado do pedido feito pelo cliente seja:

sucesso
: É devolvida uma mensagem a indicar o sucesso no pedido

insucesso
: É devolvida uma mensagem a indicar insucesso no pedido

Emissao finalizada
: É devolvida uma mensagem a indicar sucesso e outra a indicar o fim da emissão. É também enviado para o diretório o resultado da emissão.

Propostas de clientes ultrapassadas(apenas no caso do leilão)
: É devolvida uma mensagem a indicar o sucesso ao cliente que realizou a licitação. É também enviada uma mensagem, para os clientes relativo às propostas ultrapassadas, a indicar que essa proposta foi ultrapassada.

Convém salientar que a comunicação é sempre realizado com o **FrontEnd** e não com o cliente diretamente.

3.4.2 Criação de um leilão

Caso seja recebida uma mensagem para a criação de um leilão para uma empresa é realizado um pedido para a criação de um leilão na **Empresa** respectiva. Esse pedido pode retornar 2 resultados:

- O leilão criado, caso posso ser realizado
- *Null* caso já exista uma emissão/leilão em curso ou o montante e a taxa sejam negativos

Caso o leilão seja criado com sucesso então é enviada uma mensagem para o utilizador (empresa) a indicar o sucesso. São também enviadas mensagens para o direório a indicar a criação do leilão e para o sistema de notificações a indicar o mesmo. O tópico enviado para o sistema de notificações é **leilao::empresa::**.

Caso o leilão não possa ser criado é enviada uma mensagem para a empresa e indicar o insucesso.

3.4.3 Criação de uma emissão

A criação de uma emissão é bastante similar à criação do leilão, pelo que apenas vamos referir a diferença existente. A única diferença existente é que pode não ser possível criar uma emissão, caso não se tenha realizado um leilão com sucesso ainda, pelo que o método para criar uma emissão na **Empresa** pode retornar uma exceção a indicar que não é possível realizar a emissão. De resto segue tudo a mesma lógica da criação de leilão.

3.4.4 Terminar leilão/emissão

Cada *Thread* criada para contabilizar o tempo de vida de um leilão/emissão, adormece esse mesmo tempo de vida, acordando depois e enviando uma mensagem a indicar para terminar o leilão/emissão. Esta comunicação é feita através de um *socket zeroMQ* com comunicação *inproc*.

Ao receber uma mensagem para terminar um leilão/emissão e **Exchange** veirfica todos os leilões/emissões ativos e termina todos os que o tempo de fim já tenha sido ultrapassado. Ao terminar um leilão/emissão é enviada uma mensagem para o sistema de notificações a indicar que a operação terminou, sendo também enviada para o **Diretório**. Para além disto é enviada uma mensagem com o resultado da operação para o **FrontEnd** para este propagar o resultado para os interessados no leilão, ou seja, a empresa e os licitadores que participaram nele.

3.5 *Protocol Buffers*

O *protocol buffers* foi necessário para garantir que os dados, entre as diferentes tecnologias, eram lidos de forma igual. Decidimos criar três estruturas, uma relativa às notificações, comunicação efetuada entre o Cliente e a Exchange, e ainda uma estrutura para a troca de mensagens para a autenticação, comunicação entre Cliente e FrontEnd apenas, e para todas as mensagens relativas à criação/licitação de leilões/emissões, onde a comunicação é efetuada entre as 3 entidades: Cliente, FrontEnd e Exchange.

No que toca às notificações decidimos colocar os campos que nos pareciam ser mais adequados para apresentar toda a informação devida, como o nome da empresa a que está referente a ação, qual o tipo de ação (licitação ou emissão), o montante entre outros. A autenticação é bastante simples e apenas são transportados os dados referentes ao *username* e *password*, a resposta por parte do FrontEnd informa se obteve sucesso ou não, e caso seja conseguido então é enviado o papel do utilizador e ainda as subscrições desse utilizador.

A estrutura mais complexa é referente à comunicação das diferentes ações. Do Cliente é enviada uma mensagem *MensagemUtilizador* que possui informação sobre qual o tipo de mensagem, se é um leilão, emissão, subscrição ou autenticação. No caso de ser um leilão ou emissão, então o destinatário é a Exchange, e vai conter o papel do utilizador, o seu nome e ainda a sua ação pretendida, exemplo: no caso de ser uma empresa a criar um leilão, então terá uma mensagem com os dados referentes à criação do mesmo, e esta estratégia é análoga às restantes ações. Se for do tipo subscrição ou autenticação então o destinatário é o FrontEnd. A subscrição serve apenas para transportar a informação sobre se o utilizador pretende subscrever/vetar todos os leilões ou todas as emissões ou então uma empresa, quando à autenticação tem por fim efetuar um correto *logout* do utilizador. Nestes casos o FrontEnd nunca envia uma resposta.

A Exchange pode ainda enviar uma mensagem para o Cliente, e para tal criamos uma mensagem *RespostaExchange*. Poderá ser de 3 tipos:

Resposta - é uma resposta a um pedido do utilizador, onde indica se a sua ação teve sucesso ou não;

NotificacaoUltrapassado - quando a licitação de um licitador é batida por uma de outro, então este deverá receber a informação de que a sua licitação para um determinado leilão já não se encontra ativa;

Resultado - quando termina um(a) leilão/emissão é enviada uma mensagem a indicar qual foi o resultado do mesmo(a) e quais as propostas que vigoraram nos mesmos

Capítulo 4

Conclusão

Todos os requisitos da aplicação foram satisfeitos pelo grupo, no entanto, durante o processo de codificação do nosso programa, apareceram alguns obstáculos já que a maioria das tecnologias eram novas para todos os elementos. A primeira dificuldade foi a troca de mensagens entre o Cliente (Java) e o FrontEnd (Erlang) já que a ordenação dos *bits* é diferente, e no inicio não estávamos a perceber o porquê de as mensagens não serem bem interpretadas, algo que foi resolvido aplicando uma conversão da ordem dos *bits*.

A utilização de *Protocol Buffers* e de *ZeroMQ* em Java não apresentou grande dificuldade para o grupo já que trabalhamos com estas tecnologias nesta linguagem no decorrer do semestre, no entanto utiliza-las em *Erlang* já se revelou um desafio maior, o que requereu mais tempo de pesquisa a cada elemento para que soubéssemos utilizar as diferentes bibliotecas em *Erlang*.

Um aspeto que não nos agradou no resultado obtido foi a nossa interface com o utilizador. Não é muito *user friendly* já que todos os comandos são invocados a partir do terminal, algo mais visual e com botões poderia dar outra "vida"à nossa aplicação e seria algo que o grupo num futuro próximo melhoraria para garantir uma melhor experiência de uso ao utilizador. Para que o utilizador não recebesse um "spam"de notificações na nossa interface, decidimos criar um ficheiro para cada um e apresentar todas as notificações, ordenadas por ordem cronológica, nele.

Para que a nossa aplicação fosse mais geral decidimos criar ficheiros (JSON) para configurar todos os endereços dos diferentes componentes da nossa aplicação. Desta forma caso quiséssemos mudar o endereço do Diretório bastava modificar o ficheiro e não ter de alterar no código diretamente. Num aspeto que reparámos foi que sempre que o utilizador efetuava *logout* perdia todas as suas subscrições. Para que tal não acontecesse, decidimos guardar no *FrontEnd* a lista de subscrições que um utilizador possuí, sendo ela carregada no ato de autenticação bem sucedida por parte do mesmo.

Uma questão que, na opinião do grupo ficou por resolver mas que pensamos ser importante, é o facto de o utilizador ao se desautenticar e autenticar novamente altera o *PID* referente ao ator que o trata. Isto pode ser um problema visto que o utilizador assim não pode receber as mensagens que ainda não tivesse recebido, para as operações nas quais esteve envolvido. Para resolver este problema pode ser difundido o novo *PID* do ator que trata o cliente para todos os **frontend_state** para que os *PIDs* relativos a esse utilizador sejam atualizados e este passe a receber as mensagens mesmo depois de se ter desautenticado. Outro aspeto que pode ser melhorado é que o utilizador não recebe mensagens ao estar desautenticado, sendo que para resolver este problema poderá ser criada uma fila de mensagens para cada utilizador que se encontre desautenticado, sendo adicionada a essa fila todas as mensagens que cheguem para esse utilizador (que se encontra desautenticado), e no momento da autenticação são apresentadas as mesmas caso existam.

Em suma, foi um trabalho bem conseguido por parte do grupo onde permitiu conciliar os conhecimentos adquiridos na teórica e aplica-los na prática. Ficámos agradados com as tecnologias que trabalhámos e pensámos em utilizar algumas delas nos nossos trabalhos futuros devido à simplicidade das mesmas. Este trabalho serviu ainda como uma pequena simulação de um trabalho aplicado à vida real, onde há uma heterogeneidade de tecnologias e é necessário conseguir conciliar todas e aplicar cada uma no caso correto, tirando partido das melhores funcionalidades de cada uma das tecnologias.