Paradigmas de Sistemas Distribuídos
PeerLending: serviço de empréstimo entre pares

Daniel Fernandes, Maria Helena Poleri, and Mariana Miranda

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mail: {a78377,a78633,a77782}@alunos.uminho.pt

Resumo O presente documento tem o intuito de descrever as principais decisões de implementação do sistema PeerLending, um serviço de intermediação de empréstimos a empresas por parte de investidores. Trata-se de uma implementação multi-paradigma e multi-linguagem, usando diversas ferramentas e linguagens como Java, Erlang, ZeroMQ, Dropwizard e Protocol Buffers.

1 Introdução

O presente documento tem o intuito de descrever as principais decisões de implementação do serviço PeerLending. Este consiste num serviço de intermediação de empréstimos a empresas por parte de investidores.

O sistema deverá possibilitar às empresas a criação de leilões e emissões de taxa fixa, e aos investidores a possibilidade de licitar nestes. Para esse efeito, o sistema será constituído pelos clientes do serviço (que podem ser investidores ou empresas); um servidor *frontend* ao qual estes últimos se ligam, responsável por autenticar clientes, receber os seus pedidos e encaminhá-los para as *exchanges*; as *exchanges*, responsáveis por processar os pedidos recebidos e o diretório, onde fica armazenada a informação do sistema, tanto atual como histórica (leilões e emissões a decorrer, leilões e emissões passadas, etc.). Para além disso, é ainda esperado que os investidores possam subscrever informação em tempo real sobre os leilões a decorrer de uma determinada empresa (ou várias).

Para a implementação do sistema, serão usados Java (todos os componentes exceto o *frontend*), Erlang (no *frontend*), Protocol Buffers (para as mensagens entre componentes de linguagens diferentes), ZeroMQ e Dropwizard (para implementação da *interface* RESTful do diretório).

2 Arquitetura da Solução

Tal como era proposto no enunciado os componentes desenvolvidos, que compõem o sistema, são: o cliente, onde são realizados os pedidos; o *frontend*, que recebe esses mesmos pedidos e os faz encaminhar às *exchanges*; as *exchanges*, que são os consumidores dos pedidos e incluem toda a lógica da aplicação; o diretório, onde fica armazenada a informação mais histórica da aplicação, bem como o que se está a passar de momento, para consulta dos clientes.

De forma a atender a todos os requisitos estipulados para o projeto, foi necessário definir os padrões de comunicação a utilizar, isto é, que componente comunica com que componente, e de que forma é que essa comunicação é efetuada.

O *Client* necessita de conhecer as empresas existentes, bem como os leilões e emissões ativas, portanto, tem de consultar o diretório, onde tal informação se encontra. A comunicação é realizada utilizando *http*, disponibilizando este uma *API Restful*. Essa mesma API é utilizada na atualização dos seus dados por parte das *Exchanges*.

Independentemente do tipo de cliente, seja Investidor ou Empresa, terá de se autenticar no *Frontend*. Uma vez autenticado, todos os pedidos realizados às *Exchanges*, são intermediados, uma vez mais, pelo *Frontend*, que se encarrega de lhe fazer chegar a resposta.

Para a comunicação *Client-Frontend* utilizamos sockets tradicionais. Já na comunicação do *Frontend* com as *Exchanges*, por comodidade, e tendo em conta a forma com que organizamos o *Frontend*, utilizamos o padrão *push*/*pull*. O *Frontend* faz *push* de pedidos para a *Exchange* a que o pedido se destina, necessitando para tal um socket *push* por *Exchange*. As respostas síncronas aos pedidos, e outras mensagens assíncronas, tais como resultados de leilões e emissões, são recebidas no socket *pull* correspondente a essa *Exchange*.

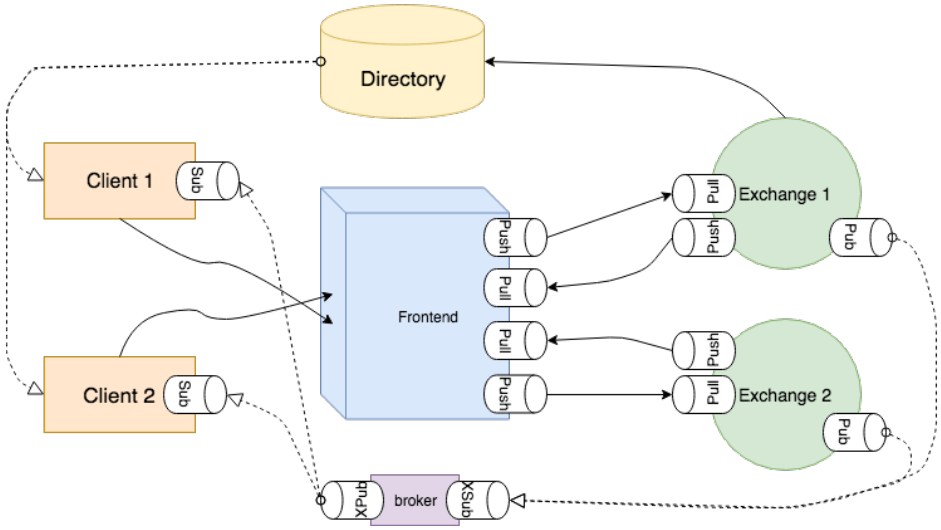

Figura 1. Visão geral da arquitetura do projeto

Esperava-se ainda com este projeto que o cliente recebesse notificações de empresas, caso, por exemplo, esta desse início a um leilão ou emissão, ou ainda que o mesmo terminasse. Tais mensagens não eram de suma importância para a funcionalidade da aplicação, pelo que não precisavam necessariamente de ser fiáveis. Toda esta funcionalidade de subscrição ou remoção de subscrição de empresas é facilmente conseguida através do padrão *pub*/*sub*. Como tais informações são produzidas nas empresas consideramos sensato não utilizar o *Frontend* para transmitir as mensagens, mas criamos um *broker* para ligação entre os sockets do tipo *sub* dos clientes e do tipo *pub* das *Exchanges*.

3 Implementação

3.1 Serialização

Uma vez que o *Client*, *Frontend* e *Exchange*, foram desenvolvidos em linguagens de programação distintas, Java e Erlang, e de forma a contornar possíveis problemas relativos à heterogeneidade das linguagens, decidiu-se utilizar Protocol Buffers como meio de serialização das mensagens entre estes componentes. O formato das mensagens foi algo que, no decorrer do trabalho, foi alvo de várias modificações de forma a atender a novos requisitos. Neste aspeto, Protocol Buffers tornou-se muito útil, porque permitiu tais modificações serem realizadas eficientemente.

As mensagens que circulam entre estes componentes podem ser divididas em dois tipos, síncronas e assíncronas. As mensagens síncronas dizem respeito às mensagens que são enviadas pelos clientes e que espera uma resposta imediata por parte da *exchange*. Efetivamente, todas as mensagens que partem do cliente, por exemplo um pedido de criação de leilão, são síncronas já que requerem algum tipo de confirmação, no exemplo apresentado, de sucesso ou insucesso. Pelo contrário, as mensagens que provém da *exchange*, podem ser

síncronas ou assíncronas. São síncronas quando se tratam de uma resposta a um pedido de um cliente. São assíncronas, quando a iniciativa de enviar a mensagem parte da *exchange*, isto sem que o cliente esteja à espera. Tais mensagens são as que resultam de um leilão ou emissão ter terminado.

Dependendo ainda do tipo de pedido do cliente ou ação da *exchange*, necessariamente as mensagens vão envolver dados diferentes e por isso campos diferentes, pelo que uma solução que surge de forma imediata é a de criar um formato de mensagem para cada tipo de mensagem. Foi isto exatamente o que fizemos, contudo, para facilitar todo o processo de descobrir de que tipo de mensagem se tratava, no momento da descodificação, criamos uma mensagem universal *MessageWrapper*, que apenas pode transportar dentro uma mensagem de um dos diversos tipos de mensagem que definimos. Para além desta mensagem, foi definido ainda no seu interior um campo que faz a distinção da sincronia da mensagem e outro de carácter opcional que representa um *token* de sessão, permitindo, quando incluído na mensagem, identificar o ator do *frontend* associado à sessão do cliente e para o qual a mensagem de resposta deverá ser encaminhada.

Uma vez que o diretório utiliza REST, toda a comunicação realizada com o mesmo é feita através de documentos JSON. Já para as mensagens referentes às notificações, devido à simplicidade do conteúdo envolvido, não foi utilizado qualquer tipo de serialização, pelo que se converte apenas para *bytes* uma *string* começada pelo nome da empresa.

3.2 Notificações

Tal como explicado na secção referente à arquitetura, de forma a permitir que os investidores pudessem subscrever notificações, relacionadas com leilões e emissões a taxa fixa de empresas, recorremos ao padrão *pub/sub* do MOM, tendo-se para isso feito uso de sockets ZeroMQ. Com o intuito de desacoplar os subscritores dos publicadores, no nosso os Clientes das Exchanges, criou-se um *broker* que, através de sockets do tipo *XPub* e *XSub*, age como intermediário nas notificações, evitando que cada cliente conheça e comunique diretamente com as *Exchanges*.

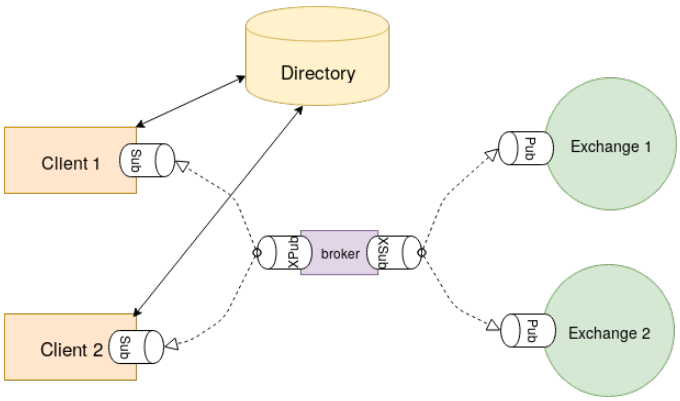

Figura 2. Visão geral da arquitetura de notificações do projeto

Para as notificações não foi utilizado Protocol Buffers, já que consideramos que, pela própria essência das mensagens, não seria necessário a introdução desse *overhead*. As mensagens correspondem, portanto, a strings normais prefixadas com um determinado tópico/assunto no formato $(Leilao|Emissao)\_ < empresa >$.

Tivemos ainda em consideração a manutenção das subscrições caso um investidor se desautenticasse. Para a implementação desta funcionalidade recorremos ao diretório para armazenar as subscrições de cada investidor.

Assim, o diretório passa a possuir toda a informação acerca das subscrições de cada investidor. De cada vez que um investidor pretende subscrever as notificações de leilões ou subscrições de uma empresa, é verificado se este não ultrapassa um limite de subscrições (que estabelecemos ser 5). Caso não ultrapasse, a subscrição é adicionada ao diretório. Desta forma, uma vez executado o programa correspondente a um Cliente, no momento da autenticação, caso este se tratasse de um investidor, eram requeridas ao diretório as informações de subscrição relativas ao investidor em questão, e traduzidas no socket *sub* criado para o mesmo.

Também o cancelamento de uma subscrição implica um `DELETE` do diretório.

3.3 *Client*

O *Client*, foi desenvolvido em Java e atua como interface de interação tanto de investidores como de empresas. Assim que se inicia a execução do programa é devolvido um menu, onde o cliente se autentica. Tal implica o envio de uma mensagem com o pedido de *login* para o *frontend* e a receção de uma mensagem de resposta. Em caso de sucesso, a mensagem de resposta indica, para as credenciais providenciadas, qual o tipo de cliente de que se trata e ainda um *token* de sessão, que corresponde ao `PID` desse cliente no *frontend*. A razão da existência deste *token* na mensagem será explicado mais tarde, quando a implementação do *Frontend* for abordada.

Para ambos os tipos de cliente é criado uma *thread*, que se encarregará de apresentar os menus, recolher a opção pretendida e agir em concordância, por exemplo enviando uma mensagem destinada a uma *exchange*. É também criada uma outra *thread* que apenas está à escuta de mensagens vindas do *frontend*, sejam estas síncronas ou assíncronas. Caso sejam assíncronas são inseridas numa fila, até que o cliente as decidir ler, caso sejam síncronas, muito provavelmente já está a *thread* anterior bloqueada até que esta mensagem seja escrita numa variável criada para o efeito, podendo existir uma única mensagem síncrona em espera a cada momento.

Como os investidores ainda possuem aquela funcionalidade de poderem subscrever empresas e receber as notificações às quais dizem respeito, foi ainda necessário criar mais uma *thread* para este tipo de entidades que estivesse à escuta destas notificações e também as inserisse numa fila.

3.4 Frontend

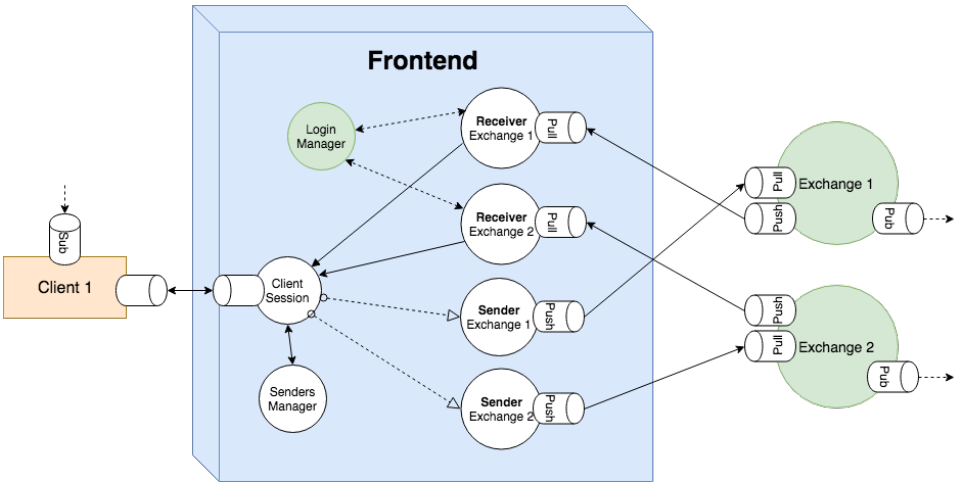

Figura 3. Funcionamento *Frontend* (Atores)

O *Frontend* foi desenvolvido utilizando um paradigma diferente de programação, que é a programação por atores. Este tipo de programação revelou-se muito útil para desenvolver este componente porque nos permite distribuir a carga de vários clientes por diferentes atores de forma fácil, evitando engarrafamentos.

A arquitetura do *Frontend* encontra-se demonstrada na Figura 3.

Tal como podemos verificar cada cliente autenticado dá origem a um ator responsável pela sessão do cliente (*Client Session*). O `PID` dessa sessão é armazenado no *Login Manager*, que conhece todos os clientes autenticados.

Tal ator possui um socket de comunicação com o cliente. Mediante uma mensagem recebida, a primeira coisa a fazer é verificar a empresa envolvida no pedido. Conhecendo a empresa, interroga-se o *Senders Manager*, que possui os mapeamentos de *empresa/exchange* e de *exchange/sender*, de forma a conhecer para qual dos atores *Sender* a mensagem se destina.

Cada *Sender* possui um único socket *push* para a *exchange* à qual está alocado. Através deste o pedido flui então da sessão para a *exchange* suposta. Os últimos atores pelos quais o *Frontend* é constituído são os *Receivers*, um para cada *exchange*. Estes apenas têm o trabalho de encaminhar para a sessão cliente devida as mensagens que lhes chegam vindas da respetiva *exchanges*.

No caso de mensagens síncronas, utiliza-se o *token* de sessão contido na mensagem do pedido para saber a que cliente enviar a mensagem. Isto foi feito desta maneira de forma a evitar a criação de um *bottleneck* no *Login Manager*, não sendo necessário ir lá buscar o `PID` para enviar resposta.

No caso de mensagens assíncronas, no entanto, isto não poderia ser feito desta forma. Como o cliente pode não estar conectado, não podemos utilizar o *token* de sessão, mas estas mensagens são muito mais raras, sendo enviadas apenas quando acaba um leilão ou emissão. Por esta razão, a probabilidade de criar um *bottleneck* é muito mais reduzida. Então para este caso, vê-se o cliente destino na mensagem, depois pergunta-se ao *Login Manager* qual é o `PID` da sessão desse mesmo cliente, e por fim envia-se uma mensagem para essa sessão que o faz chegar ao cliente final.

3.5 Exchange

Toda a lógica do sistema, ou seja, todo o processamento relativo a leilões e emissões é efetuado pelas *exchanges*. Existem múltiplas *exchanges* (no nosso caso estabelecemos que são 3), estando cada uma responsável por um dado subconjunto de empresas.

Estas são responsáveis por receber os pedidos vindos do *frontend*. Escolheu-se usar os *sockets* `PUSH/PULL` do ZeroMQ, por ser necessário que este processo fosse assíncrono e também por ser um padrão que nos proporciona alguma fiabilidade na entrega de mensagens.

Uma vez recebido o pedido e usando Protocol Buffers é determinado de que tipo de pedido se trata. Para o caso dos investidores, a *exchange* só precisa de processar as licitações (ver se são válidas) e responder de acordo.

Por sua vez, o caso dos pedidos das empresas são mais elaborados. Quando uma *exchange* recebe um pedido de criação de leilão ou emissão deve enviar uma mensagem a confirmar se o pedido é válido ou não, mas, caso seja, deve também efetuar o leilão e esperar pelo seu fim para informar os intervenientes do seu desfecho.

Para o efeito, usamos uma classe em Java chamada **ScheduledExecutorService** que permite criar uma tarefa a ser executada após um determinado período de tempo (neste caso, igual à duração do leilão/emissão).

Ao fim deste tempo, o *scheduler* é responsável por avisar o diretório do fim do leilão/emissão, enviar mensagem às empresas a informar acerca do sucesso do leilão, aos investidores mensagens acerca das suas licitações terem ganho ou perdido e ainda avisar os subscritores das notificações daquela empresa acerca do desfecho.

3.6 Diretório

O diretório é um componente que disponibiliza uma interface RESTful, capaz de informar acerca de que empresas existem, que leilões e emissões estão em curso (ativos) e o histórico de leilões e emissões para cada empresa. Após analisar estas necessidades, chegamos à API que se encontra na Tabela 1.

O diretório tem como clientes as *exchanges*, que informam de novos leilões e emissões através dos métodos **POST**. Quando um leilão ou emissão acaba, usam o método **DELETE** para que estes possam passar a ser informação do histórico da empresa respetiva.

Tem ainda como cliente os próprios investidores, que podem no seu menu escolher ver as empresas existentes, os leilões e emissões ativos e o histórico de leilões e emissões para uma empresa.

URI	**Método HTTP**	**Descrição**
/activeAuctions	GET	Lista todos os leilões ativos no sistema.
/activeAuctions	POST	Adiciona um leilão ativo ao sistema. **NOT-FOUND (404)** - caso a empresa não exista. **CONFLICT (409)** - caso já exista um leilão/emissão para essa empresa.
/activeAuctions/<companyId>/<success>	DELETE	Apaga o leilão ativo de uma determinada empresa, identificada pelo seu nome e sucesso do leilão. Move ainda este leilão para o histórico de leilões dessa empresa. **NOT-FOUND (404)** - caso a empresa não exista ou caso não exista um leilão para essa empresa.
/activeEmissions	GET	Lista todas as emissões ativas no sistema.
/activeEmissions	POST	Adiciona uma emissão ativa ao sistema. **NOT-FOUND (404)** - caso a empresa não exista. **CONFLICT (409)** - caso já exista um leilão/emissão para essa empresa.
/activeEmissions/<companyId>/<success>	DELETE	Apaga a emissão ativa de uma determinada empresa, identificada pelo seu nome e sucesso da sua emissão. Move ainda esta emissão para o histórico de emissões dessa empresa. **NOT-FOUND (404)** - caso a empresa não exista ou caso não exista um leilão para essa empresa.
/companies	GET	Lista todas as empresas existentes.
/companies/<id>/auctionHistory	GET	Lista o histórico de leilões para uma determinada empresa, identificada pelo seu nome. **NOT-FOUND (404)** - caso a empresa não exista.
/companies/<id>/emissionHistory	GET	Lista o histórico de emissões para uma determinada empresa, identificada pelo seu nome. **NOT-FOUND (404)** - caso a empresa não exista.

Tabela 1. API RESTful para o diretório

4 Conclusões e Trabalho Futuro

Neste documento discutiram-se todas as decisões de implementação do sistema Peer Lending, um serviço de empréstimo a empresas por parte de investidores. Abordaram-se todos os componentes necessários e ainda as formas de comunicação usadas entre si. Para o seu desenvolvimento foram utilizadas diferentes ferramentas e linguagens de programação, nomeadamente Java, Erlang, ZeroMQ, Dropwizard e Protocol Buffers.

Pelas suas características multi-paradigma (programação baseada em atores, orientada a mensagens, etc) e multi-linguagem (Java e Erlang), foi um trabalho que nos permitiu consolidar todas as áreas aprendidas e também perceber como ligá-las por forma a construir um sistema coeso, com componentes bem delineados e com funções específicas, tendo sido cada um implementado com a tecnologia que achamos ser a mais apropriada.

Como trabalho futuro, gostaríamos de ter adicionado supervisão no *frontend*. tirando partido deste mecanismo do Erlang que permite construir programa mais tolerantes a faltas.