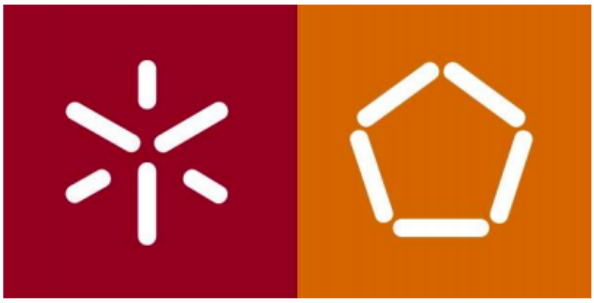

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Tolerância a Faltas

Trabalho Prático

Eduardo Lourenço da Conceição (A83870)
Ricardo Filipe Dantas Costa (A85851)
Rui Nuno Borges Cruz Oliveira (A83610)
Cândido Filipe Lima do Vale (PG42816)

11/06/2021
Braga

Índice

1 Introdução

Para o trabalho da cadeira de Tolerância a Faltas foi-nos proposta a construção de um sistema distribuído de gestão de contas de um banco, utilizando técnicas de replicação passiva e ferramentas utilizadas nas aulas.

Neste relatório iremos explicar as decisões que tomamos para implementar o sistema e todas as valorizações, e terminaremos com uma análise da *performance* do sistema, medindo algumas métricas que nos auxiliam na compreensão do comportamento do sistema.

2 Contextualização

Num qualquer sistema, seja ele distribuído ou não, existe a possibilidade de algo correr mal. A descrição daquilo que pode correr mal e o que pode afetar no sistema é o que se chama um modelo de faltas. Os sistemas a que nos referimos normalmente são compostos por vários componentes distintos, sendo que temos de nos precaver para faltas, erros ou até falhas de pelo menos os componentes mais críticos de ditos sistemas, sendo que a única forma de realmente tornarmos um sistema tolerante a faltas é introduzindo redundâncias dos componentes fulcrais.

Sistemas distribuídos, em geral, são sistemas ideais para serem tolerantes a faltas, devido a várias características que eles apresentam: modularidade, heterogeneidade, encapsulamento, etc.

Assim sendo, faz sentido que, ao criarmos um sistema com informação preciosa, como um Banco, no âmbito deste projeto, devamos tomar em conta aspetos de tolerância a faltas, nomeadamente sabendo que se trata de um sistema distribuído. Como tal, a forma como nós introduziremos mecanismos de tolerância a faltas será através da utilização de técnicas aprendidas nas aulas de replicação, nomeadamente replicação passiva.

Trabalharemos com um modelo de faltas *fail-stop*, em que o *software* de comunicação de grupo informa cada processo da falha de um processo, do ponto de vista do servidor. Não teremos de nos preocupar com clientes ou *backups* bizantinos.

3 Implementação da Solução

3.1 Ferramentas Utilizadas

Para implementar o sistema em questão, utilizamos a linguagem de programação Java, uma linguagem que se adequa bastante ao problema devido à quantidade, qualidade e heterogeneidade de bibliotecas que a linguagem oferece que se adequam a trabalhar com o modelo de cliente-servidor. Em particular, para este aspeto de comunicação com o servidor utilizamos a biblioteca *Atomix*[1], que oferece canais de comunicação assíncronos, o que nos permite criar um modelo de comunicação orientada a eventos.

Como mencionamos anteriormente, o sistema deverá implementar uma forma de replicação passiva. Para este efeito, será necessário termos um serviço de comunicação de grupo que nos ofereça uma primitiva de *view synchronous multicast*. O serviço que utilizamos para este efeito é o *Spread Toolkit*[4], a ferramenta que estudamos nas aulas.

3.2 Estrutura *Bank*

Como mencionamos anteriormente, a aplicação deverá implementar um banco distribuído. Este banco é constituído por um número fixo de contas, sendo que a API não oferece operações de inserção e remoção de contas (o que facilitará algumas porções mais críticas do controlo de concorrência). O serviço do banco deverá oferecer as operações básicas que serão esperadas do banco, sem qualquer autenticação ou autorização, oferecendo operações de:

1. Fazer uma injeção ou remoção de capital na conta (*movement*);
2. Verificar o saldo de uma conta (*balance*);
3. Fazer uma transferência entre duas contas (*transfer*);
4. Verificar o histórico das últimas N operações feitas sobre uma conta (*history*);
5. Adicionar uma taxa de interesse sobre todas as contas do banco (*interest*);

O banco na aplicação em si é representado pela classe interface *BankInterface*, que contém os seguintes métodos abstratos a implementar:

- ***boolean movement(int accountID, float amount)***: função para implementar o ponto 1, adicionando *amount* unidades monetárias à conta de ID *accountID*, devolvendo *true* se a operação foi concluída com sucesso e *false* caso contrário;
- ***float balance(int accountID)***: função para implementar o ponto 2, devolvendo o saldo da conta *accountID*, devolvendo -1 se a conta não existir;
- ***boolean transfer(int from, int to, float amount)***: implementa o ponto 3, retirando da conta *from amount* unidades monetárias e adicionando a *to* a mesma quantidade, e retornando um booleano correspondente a se a operação teve sucesso ou não;
- ***void interest()***: implementa o ponto 4, adicionando a cada conta do banco a taxa de juros que a classe que implementa a interface tem;
- ***void set(int accountID, float amount)***: não implementando nenhum dos pontos anteriores e servindo apenas para operações internas do servidor, coloca o saldo da conta *accountID* a *amount* unidades monetárias;
- ***List<Transaction> history(int accountID)***: implementa o ponto 5, devolvendo a lista de transações da conta *accountID*. Esta classe *Transaction* é fulcral ao trabalho em si e será explicada mais à frente, nesta secção;

Sendo *BankInterface* uma interface a implementar, existem duas classes distintas que implementam a mesma: a classe ***Bank*** e a classe ***ClientStub***. Estas duas classes são bastante distintas e à classe *ClientStub* será atribuída uma secção própria, descreveremos a classe *Bank* aqui.

Bank é a implementação da interface utilizada no *back-end* da aplicação, ou seja, nos servidores, para representar a informação do banco, constituída por dois atributos importantes: um *Map* de *Account*s, classe que representa uma conta do banco, e uma variável *interest*, que representa a taxa de juros que vamos adicionar sobre cada conta quando a operação *interest* é invocada. Como as contas são fixas, não será necessário criarmos um *lock* associado ao *Bank* em si, pois a estrutura não é alterada no seu todo em quase nenhum ponto (mas existem alguns pontos em que teremos de ter controlo de concorrência sobre mais que uma conta ao mesmo tempo). A classe *Account* representa uma conta única com um *lock*, o seu saldo e o seu identificador único, e uma *queue* de *Transaction*s que representa o histórico das N últimas operações feitas sobre esta conta. Para o histórico próprio da conta, distinto do histórico do servidor, este N é igual a 10. É importante notar que, com "operações" apenas consideramos operações de escrita como sendo relevantes de manter no histórico, ou seja, *movement*, *transfer* e *interest*. Operações de *balance* e *history* não são tomadas em consideração para este histórico.

A classe *Transaction* que já várias vezes foi referida é o pacote de informação sobre uma transação que ocorre sobre uma ou mais contas. Este pacote é fulcral em vários aspetos da comunicação entre servidores e para guardar históricos de transações no sistema, guardando os descritores importantes para caracterizar uma transação (descritivo, data de entrada no sistema, valor movido, saldo depois da operação, etc...). De notar que são guardados dois IDs diferentes numa *Transaction*: o *req_id*, que representa o ID interno ao cliente, e o *internal_id*, que representa o ID interno ao grupo de comunicação dos servidores.

3.3 Arquitetura do Sistema

Nesta secção iremos apresentar os dois componentes principais do projeto e de qualquer arquitetura que siga o clássico padrão cliente-servidor: o cliente e o servidor.

3.3.1 Cliente

O cliente da aplicação será uma qualquer classe que chame uma instância da classe anteriormente referida, o *ClientStub*. O *ClientStub* é uma entidade que implementa a interface *BankInterface*, e comunica com o servidor da aplicação utilizando *sockets* assíncronos do *Atomix*. O *Atomix* oferece um tipo de objeto chamado *NettyMessagingService*, que pode registar vários tipos de *handlers*, que correspondem a *callbacks* que serão chamadas quando houver uma resposta de um dado tipo, que segue a regra "<op_type>-res" (p.e., "movement-res").

Sendo que implementa todos os métodos impostos pela interface, eles funcionam de uma forma semelhante. Para cada um, uma mensagem *ReqMessage* é criada, que contém a informação relevante do pedido (o(s) ID(s) da(s) conta(s) envolvida(s), o montante, etc...) e cria um *CompletableFuture* que conterá o resultado do pedido. Por exemplo, para um *movement*, será criado um *CompletableFuture<Long>*. Este

futuro é depois guardado num *Map* específico para cada tipo de pedido, com a chave sendo o ID único desse pedido. De seguida, é enviado um *request* pelos canais de comunicação assíncrona com o tipo do género "<op_type>-req", e é esperado que o futuro seja completado, numa operação bloqueante que envolve a utilização do *get()* do *CompletableFuture*. Uma vez que o seja, o pedido acaba e o resultado da operação é retornado ao cliente.

Mas como é que este futuro pode ser completado? Com os *handlers* que falamos anteriormente. Quando um *handler* é ativado, através da receção de uma mensagem com o seu tipo, é verificado o conteúdo da *ResMessage* que vem na mensagem, que deverá conter o valor de retorno da operação e o ID interno ao cliente da mesma. O que faremos de seguida é verificar se o pedido está no *Map* correspondente, obtendo o *CompletableFuture* com o ID que veio como chave, e completando-o com o *payload* de resposta, e removendo-o da estrutura depois, assim permitindo que um pedido que esteja em espera seja completado.

De modo a ter em conta falhas do servidor primário que não serão transparentes ao cliente, cada um dos *CompletableFuture*s está programado a completar excecionalmente num dado tempo, ou seja, se o futuro não tiver sido completo num espaço de X segundos (por nós definido como 10), esse futuro é completado com *null*. Decidimos implementar isto pois os pedidos podem falhar em certas ocasiões e, como os pedidos são bloqueantes do ponto de vista do cliente, temos de ter algum mecanismo para garantir que o cliente não bloqueia eternamente num pedido que nunca será respondido, fazendo com que ele eventualmente decida que um dado pedido foi perdido e nunca será respondido. Nesse caso, apenas o descartamos, não fazendo com que o pedido seja reenviado. Esta conclusão em *timeout* do futuro é a razão porque temos de distinguir o tipo de futuro do *interest*. Como a operação retorna *void*, o futuro deveria ser do tipo *CompletableFuture*<*Void*>, mas como este tipo de futuro só pode ser completado com *null*, decidimos torná-lo num futuro com outro tipo qualquer de objeto no interior, de modo a distinguir quando ele termina excecionalmente ou quando termina normalmente, tendo ficado como um *CompletableFuture*<*Long*>, mesmo que o conteúdo dele não seja usado.

Um aspeto importante a notar é que todo o processo é feito numa única *thread* no lado do cliente, pelo que não é necessário implementar nenhum mecanismo de controlo de concorrência.

Para que possamos fazer pedidos manualmente, uma pequena GUI foi implementada no terminal com a classe *BankClientGUI*, que apenas recebe *input* do servidor pela linha de comandos e envia os pedidos utilizando um *ClientStub*. Uma instância da classe é chamada no *BankClient*, onde se encontra o método *main*. A classe utiliza um objeto de leitura de *input* do utilizador desenvolvida pelo professor Fernando Mário Martins e que foi disponibilizada na cadeira de Laboratórios de Informática III da Licenciatura.

3.3.2 Servidor

Em contraste à relativa simplicidade do cliente da aplicação, o servidor é um programa consideravelmente mais complexo e com muitas partes em movimento. A classe que implementa o servidor é o *ServerSkeleton*, do *package Server.AtomixServer*. Este

ServerSkeleton é uma entidade *multithreaded* que contém dentro de si uma instância da classe *Bank* (a implementação específica, não a interface), e uma instância da classe *SpreadMiddleware*, que constitui os pontos mais fulcrais e complexos deste projeto. O funcionamento desta classe é complexo, e lidará com toda a comunicação ente servidores que está subjacente a replicação passiva. A separação clara entre os componentes que lidam com o *Spread* e os que lidam diretamente com os clientes foi feita para facilitar a compreensão do código através da apresentação de um fluxo de controlo mais compreensível e de modo a cumprir o ponto de valorização número 2 do enunciado.

Olhando em primeiro lugar para o *ServerSkeleton*, ele possui um *NettyMessagingService* como o cliente, que terá vários *handlers* registados para poder lidar com os pedidos dos clientes, sendo que estão registados de forma a lidarem com os vários tipos de pedidos da forma "<op_type>-req". Este serviço de mensagens, por regra, está desligado, sendo só iniciado quando a instância do servidor em questão é o líder do grupo de comunicação (*Primary*) e quando o mesmo está atualizado (a ver mais tarde como isto é decidido), e está desligado caso contrário (*Backup*). Todos os servidores de um dado grupo partilham o mesmo endereço do *Atomix*, sendo este *localhost:10000*, de modo a que, quando o cargo de líder passa de um servidor para o outro a operação é parcialmente transparente para o cliente, mas isto obriga a que apenas uma instância do *NettyMessagingService* possa estar ativa a um dado momento do tempo, senão incorreremos numa exceção pois o *port* a que nos quereremos ligar já está ocupado.

Sendo que decidimos envergar pela implementação de um sistema tolerante a partições de rede, não críamos uma base de dados embutida, pelo que, para que possa haver persistência de estado, quando o servidor é *shutdown*, ou seja, quando recebe um sinal *SIGINT*, ele escreve o seu estado num ficheiro objeto, com o nome "server_state_-<port_number>.obj". Este estado é constituído pelo *Bank* que possui, serializado num formato binário, e pelo número de mensagens que ele viu e processou devidamente (ou seja, o número de transações que completou menos o número de transações sobre as quais informou os outros servidores). Quando uma instância nova é inicializada, ela procurará pelo ficheiro com o seu *port* no nome e, caso o encontre, inicializará o *Bank* com a informação nesse ficheiro e, caso contrário, apenas inicializará o *Bank* com as N contas a 0 unidades monetárias cada e com historial vazio.

O *SpreadMiddleware* é a classe que lida com a comunicação no grupo de comunicação, e terá de ser ela a informar ao *ServerSkeleton* se ele é ou não o líder e se está ou não atualizado. Como é feita a decisão veremos mais à frente, mas a passagem de informção do *SpreadMiddleware* para o *ServerSkeleton* é problemática, e feita através de um *CompletableFuture*, que é passado ao *SpreadMiddleware* quando este é inicializado, e será completado quando este achar necessário informar o esqueleto do seu estatuto como líder, completando o futuro com o número da última mensagem que viu.

Dois elementos que referimos foram o número de mensagens vistas e o número de transações completas. Estes dois são dois *longs* atómicos (implementados com a classe *AtomicLong* do *java.utils.concurrent*, e correspondem ao número de mensagens de clientes que o servidor recebeu e ao número de mensagens dos clientes que realmente já terminou o processamento, respetivamente. Numa análise cuidada, o espaço de tempo em que estes dois valores são diferentes é relativamente pequeno, mas em que muitas ações podem ocorrer, e os cuidados que tomamos para que não haja problemas com isto serão descritas mais tarde.

3.3.3 Fluxo de Comunicação

Nesta secção será explicado de um modo mais geral o protocolo de comunicação do sistema, nomeadamente qual o fluxo das mensagens no sistema e a interação entre os componentes do grupo dos servidores.

Dentro do grupo de comunicação dos servidores, cada instância do servidor estará ligada a um *daemon* do *Spread* (sendo que estamos a utilizar as imagens do *Docker* fornecidas pelo corpo docente da cadeira, estamos limitados à utilização de apenas 3 servidores, que se ligarão ao *localhost* aos *ports* do *alpha* (4803), *bravo* (4804) e *charlie* (4805)). São estes *daemons* que estão responsáveis pela comunicação no grupo dos servidores.

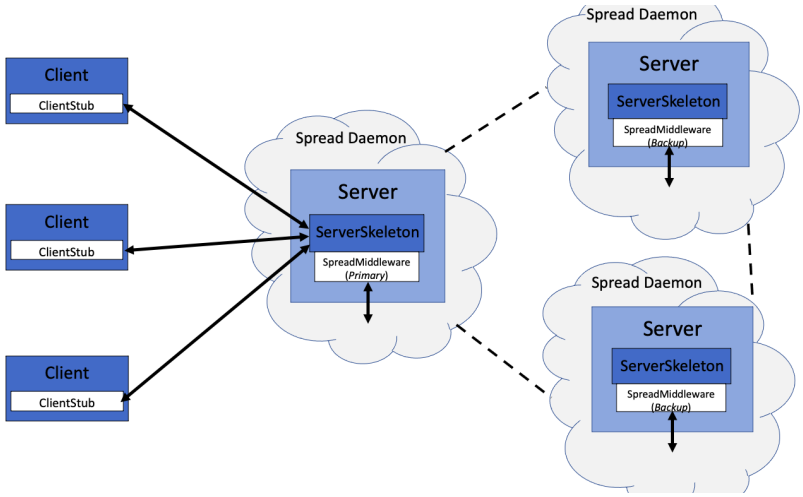

Figure 1: Arquitetura do Sistema

Relativo a aspetos mais gerais da comunicação do sistema (não só dentro do(s) grupo(s) de servidores, mas também entre o cliente e o servidor primário), a comunicação é sempre feita através de canais assíncronos, como mencionamos anteriormente, sejam estes canais do *Atomix* ou dos grupos do *Spread*. O conteúdo que é *streamed* nesses canais são *arrays* de *bytes* serializados pela classe *Serializer* do *Atomix*. A serialização é uma das valorizações pedidas no trabalho, e é sem dúvida um aspeto muito importante na comunicação entre entidades num sistema distribuído. A serialização num formato binário é consideravelmente melhor, nos nossos olhos, do que com um formato de texto com *Strings*, pois é um formato muito mais compacto, genérico, e que não obriga a fazer *parsing* de *strings*, uma vez que a tradução é feita diretamente pela classe em si. Todos este aspetos levaram-nos a escolher serialização com um formato binário, e o *Serializer* do *Atomix* revela-se ideal nisto pois é uma classe de utilização bastante fácil que apenas requer o registo das várias classes com que vai lidar quando este é inicializado.

3.4 Implementação de Mecanismos de Coerência e Tolerância a Faltas

Nesta secção iremos expor como implementamos os vários mecanismos pedidos no enunciado para garantir consistência entre réplicas do sistema utilizando um modelo de replicação passiva.

3.4.1 Replicação Passiva

Redundância é um passo fulcral na construção de um sistema que seja verdadeiramente tolerante a faltas. Um sistema não poderá tolerar a falta de si mesmo, o que nos leva a concluir que deveremos replicar o servidor de modo a que, caso um servidor falhe, outros possam continuar a assegurar o bom funcionamento do sistema. Para isso, podemos envergar por dois caminhos diferentes de replicação: ativa ou passiva.

No âmbito do projeto, foi-nos imposto escolhermos replicação passiva, e esta demonstra várias características interessantes, sendo que a mais vantajosa será que suporta operações não determinísticas. Este aspeto não é o mais relevante neste sistema uma vez que não existem operações estocásticas, mas seria interessante num ambiente diferente. No entanto, exige mais coordenação por parte das réplicas, que têm de ser constantemente atualizadas para que mostrem o estado correto em todos os pontos relevantes do tempo.

Assim, em termos práticos, isto corresponderá a um sistema com uma instância do servidor que é o líder, servindo como *backup* primário, e um ou dois *backups* secundários, limitados pelo número de *daemons* disponíveis.

Sempre que o servidor primário recebe um pedido de um cliente, ele atualiza o seu estado primeiro e eventualmente enviará uma *SpreadMessage* aos *backups* secundários com o conteúdo da atualização sob a forma de uma *Transaction* (a(s) conta(s) atualizada(s) e respetivo(s) saldo(s) após a transação) para o grupo (ou seja, para os constituintes da última vista que ele instalou), de forma a que este atualizem o seu estado, tendo esta mensagem o tipo *state_update*. Esta mensagem é um ponto muito importante, e deverá ser sempre *set* a *safe*. Isto porque, quando uma mensagem é enviada para o grupo, ela é entregue a todos os processos que instalaram a vista em que a mensagem foi enviada (caso não seja entregue a um processo, os outros instalarão uma nova vista), ou seja, é igualmente entregue ao processo que fez *vcast* da mensagem. Quando uma mensagem é marcada como *safe*, o processo que que fez *vcast* da mensagem será o último a fazer *vdelivery* da mesma, o que nos leva à conclusão que todos os processos corretos fizeram *vdelivery* da mensagem antes do primário. E que implicação tem isto? Neste processo de comunicação, nós apenas deveremos informar o cliente que uma transação está completa quando tivermos a certeza que todos os servidores corretos concordam que está completa. Assim, ao garantirmos que o líder é o último a receber o seu próprio *state update*, ele tem uma confirmação que todos os *backups* corretos concordam que a transação existe e aplicá-la-ão eventualmente, e portanto podemos responder ao cliente. Isto não garante, no entanto, que a transação está completa em todos os *backups* quando a resposta é enviada, para isto precisaríamos de receber uma confirmação positiva por parte de cada *backup* e só quando tivermos todas é que podemos responder ao cliente. Esta opção diminuirá, no entanto, o débito da aplicação,

devido ao maior *idle time* de cada pedido no servidor primário, enquanto o mesmo espera pelas respostas dos *backups*, e obrigaria a termos em conta mudanças de *vista* enquanto esperamos pela resposta de cada um dos secundários, uma vez que eles podem falhar enquanto esperamos pela sua resposta, o que levaria a bloqueios em certos pedidos devido a esperas por repostas que nunca virão, caso estas situações não fossem tratadas com o devido cuidado.

3.4.2 Eleição de Líder

Ao longo deste documento mencionamos várias vezes um servidor primário, o dito líder. Mas como é que este líder é eleito? O processo de eleição de líder teve duas versões diferentes ao longo do projeto, sendo que cada uma assumia pressupostos diferentes. Explicaremos primeiro a primeia abordagem que tivemos, e depois explicaremos a versão tolerante a partições de rede.

Assim sendo, nesta secção explicaremos a primeira versão do processo, sendo que na secção 3.4.5 explicaremos a versão tolerante a partições de rede.

A primeira versão do processo de eleição do líder era relativamente simples e assume o seguinte: uma vez que um processo decide que é o líder, a liderança não muda até que o processo líder falhe e um processo que não seja o líder não precisa de saber quem é o líder, apenas precisa de saber quem são os possíveis líderes. Como tal, o processo de escolha é baseado na ordem por que os processos entram no grupo de comunicação.

Do ponto de vista de um processo *P*, a primeira siuação em que precisamos de fazer alguma decisão quanto à liderança do grupo é quando *P* entra no grupo. Aqui, ele recebe uma mensagem que o informa da constituição da vista. Esta mensagem contém no *header* uma lista de grupos *singleton*, um *SpreadGroup[]*. Se a cardinalidade do grupo for unitária, significa que o único elemento do grupo é o próprio processo, pelo que o processo decide então que ele é o líder. Caso contrário, um dos elementos da *view* já é o líder, e não nós, pelo que não haverá nenhuma razão para trocar o líder. Sendo que não sabemos exatamente quem é o líder, guardamos a lista de *SpreadGroup*s, em que um destes é o líder, numa variável chamada *leader_queue*.

Continuando no ponto de vista de *P*, a outra situação em que teremos de verificar se somos o líder é quando recebemos uma mensagem de mudança de vista causada pela desconexão ou pela saída de um processo *P'* do grupo. Nesta situação, deveremos verificar se *P'* está na *leader_queue*. Se estiver, quer dizer que era um potencial líder e, visto que saiu do grupo, deixa de o ser. Logo de seguida, verificamos se na *leader_queue* apenas está o processo *P*. Se estiver, então *P* é o líder. Caso contrário, ainda não será a vez de *P* de ser o líder e ainda existe um processo *Q* distinto de *P* tal que *Q* pertence à *leader_queue* e é o líder.

Todo este processo é realizado pelo *SpreadMiddleware*, e, de modo a informarmos o *ServerSkeleton* que o mesmo passou a ser líder, completamos o futuro que falamos anteriormente. Do lado do *ServerSkeleton*, existirá uma *thread* que espera que o futuro seja completado e, quando vir que foi, inicializa o *NettyMessagingService* e prepara o mesmo para receber pedidos do cliente.

3.4.3 Atualização de Estado dos *Backups*

Para que os *backups* secundários estejam sempre com o estado atualizado, é necessário que o servidor primário envie mensagens de *update* do estado sempre que for necessário ou possível, aquando de operações de escrita no *Bank* com sucesso. É importante distinguir que operações que falhem (isto é, *movements* que envolvam remoção de quantias maiores do que a que uma conta tem) não necessitam de ser guardadas no histórico de operações do servidor ou da conta nem precisam de ser enviadas aos *backups* secundários, e mal vejamos que uma operção falha, respondemos ao cliente.

Para atualizar os *backups*, fazemos *broadcast* de uma *Transaction* serializada numa mensagem do tipo ***state_update*** que contém a informação toda da transação, e os *backups* secundários, ao receber esta *Transaction*, irão atualizar a(s) conta(s) envolvida(s), dando *set* do seu valor igual ao valor que vem na *Transaction* (daí a necessidade da implementação da operação *set*). Como o *broadcast* é feito numa dada vista, o líder também irá receber a mensagem de atualização de estado mas irá ignorá-la, no sentido que não vai aplicar a atualização que está no *payload*, mas a mensagem será útil para sinalizar ao líder que pode responder ao cliente.

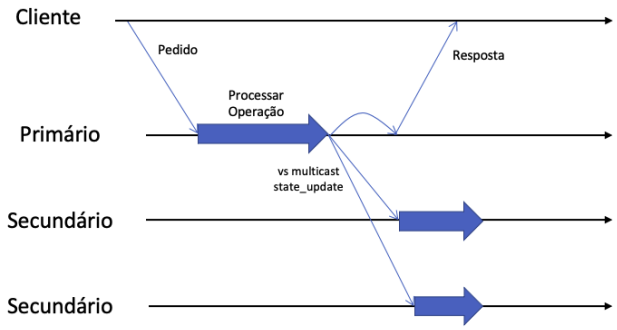

Figure 2: Atualização dos *Backups*

Numa situação de *multithreading*, vai haver alturas em que mais do que um pedido está a ser processado ao mesmo tempo, e pode surgir uma situação como a que vemos na seguinte imagem:

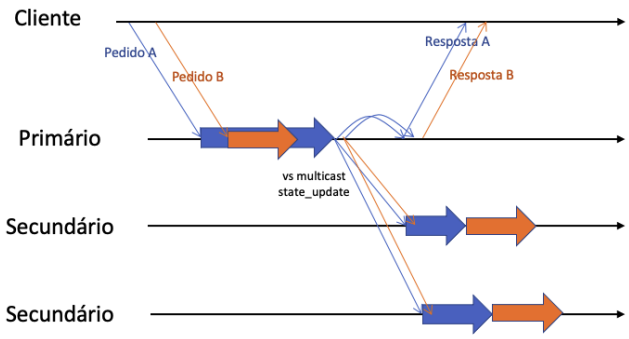

Figure 3: Atualização dos *Backups* com pedidos concorrentes

Nesta situação, em que temos dois pedidos que estão a ser processados ao mesmo tempo, em vez de simplesmente enviarmos o *update* do pedido B quando este acabar, enviamos depois do A acabar, fazendo um *batch update*. Quando vemos que há mais do que um pedido a ser processado ao mesmo tempo, ou seja, quando o número de mensagens vistas é diferente do número de mensagens processadas (*transactions_completed* ≠ *last_msg_seen*), guardamos o pedido numa *queue* de pedidos e, quando não houver mais pedidos concorrentes no sistema, esvaziamos a *queue*, fazendo *multicast* de cada uma das transações que aparecem na *queue*. Não será importante impor uma ordem dentro da *queue* neste caso em particular, pois cada conta é *locked* quando processamos um pedido que a envolva, logo operações que recebam primeiro o *lock* serão as primeiras a acabar. Por parte dos *backups* secundários, os pedidos são processados sequencialmente, pela ordem que recebem, que será uma ordem FIFO, garantida na entrega pelo facto que a mensagem de *state_update* é *Safe*.

3.4.4 Transferência de Estado

A transferência de estado é outro mecanismo de consistência de dados que é importante explicar. O mecanismo consiste num pedido de transferência de dados por parte de um servidor, e é acionado em duas situações:

- Quando um servidor novo se liga;
- Quando é feita *merger* de duas vistas;

No primeiro caso, o mecanismo consiste no envio de uma mensagem do tipo *state_transfer_request* para o grupo, mensagem esta que contém a última mensagem vista pelo novo servidor (que pode ser diferente de 0 visto que nós fazemos persistência de estado através da escrita em ficheiro aquando da falha de um processo). Esta mensagem será respondida de uma de duas maneiras:

- Se a última mensagem que o novo processo viu está no histórico das últimas N mensagens do servidor que recebeu o *state_transfer_request*, então é enviado ao

novo processo uma lista com apenas as transações que ele ainda não viu, iniciando uma ***partial_state_transfer***. Quando o novo processo receber uma mensagem deste tipo, percorre a lista que recebeu sequencialmente, aplicando cada uma das alterações das transações ao seu objeto banco.

- Caso contrário, como não temos em memória o conjunto das últimas operações que levaram ao estado atual do sistema, então criamos uma estrutura *LockFreeBank*, que contém todos os conteúdos da estrutura *Bank*, mas com uma versão das contas, *LockFreeAccount*, que não tem *locks* (pois estes mostraram-se impossíveis de serializar devidamente utilizando o *Serializer* do *Atomix*) e enviamos a estrutura ao novo processo, com uma mensagem ***full_state_transfer***. O novo processo, por sua vez, ao receber esta mensagem, cria uma nova instância do Banco com este *LockFreeBank*.

A distinção destes dois tipos de transferência justifica-se particularmente em sistemas em que o envio da Base de Dados inteira requer a utilização de uma elevada largura de banda. Ao guardarmos as últimas N operações feitas, sendo que, no nosso caso, N é 100, podemos evitar enviar a estrutura inteira, optando por enviar uma mensagem muito mais leve que corresponde apenas às últimas atualizações feitas ao banco. As últimas 100 transações são guardadas num histórico (distinto do das contas) que é um atributo membro do *SpreadMiddleware*.

Aquando de uma *merger* de grupos de comunicação, o processo é semelhante, mas é aplicado apenas aos membros do(s) grupo(s) que viram menos mensagens, que são definidos aquando da eleição do líder nesse caso.

3.4.5 Tolerância a Partições de Rede

Ao abordarmos um dos cenários adicionais que o trabalho enuncia, optamos pelo modelo de grupos particionáveis (EVS) com o *Spread*. Este modelo assume que o sistema pode bloquear processos e que nunca entrega mensagens a vistas minoritárias, sendo que podemos definir a maioria através da diferença existente entre duas vistas consecutivas.

Nesta componente adicional, é introduzido assim o termo vista de transição, isto é, uma vista regular que nos informa sobre a incerteza sobre que mensagens são entregues e a que membros do grupo. Esta incerteza obriga a que, em cada vista de transição, haja uma nova eleição do líder ou a que este deixe de existir no sistema e, consequentemente, a uma desconexão do mesmo relativamente ao cliente. Através do auxílio do *Spread*, que entrega as vistas a todos os membros do grupo particionado, cabe à aplicação o papel de bloquear se e quando necessário.

Para a implementação de grupos particionáveis, foram considerados 3 cenários possíveis:

- O líder é particionado, mas há maioria, portanto, verificamos entre os membros do grupo maioritário qual aquele que tem mais informação relativa às transações que ocorreram no sistema (em caso de empate, o critério de desempate utilizado é o nome);

- O líder é particionado, mas não há maioria, há então uma desconexão entre servidor e cliente, sendo o *atomix* interrompido;

- *Merge* de partições, o protocolo de eleição do líder passa pelo mesmo referido no ponto 1.

Primeiramente, o servidor primário (líder) ao receber uma vista de transição é informado sobre a incerteza, logo deixa de ser líder, garantido assim que não ocorrem pedidos do cliente durante o período da incerteza do sistema, desligando o *NettyMessagingService*. Neste momento do processo não existe nenhum servidor a responder a pedidos. Por isto, olhando para o Teorema CAP passamos a prescindir da disponibilidade (*Availabity*) em favor da consistência de dados (*Consistency*) e da tolerância a partição de rede (*Partition*).

Após receber através do *Spread* uma mensagem de *membership* causada pela partição de rede, cada processo verifica se pertence à maioria. Por um lado, caso pertença, é iniciada uma *leader_election* e cada processo faz a sua candidatura. Por outro lado, caso não pertença, considera que passa a estar *outadated*, precisando, provavelemente, no futuro de uma transferência de estado. Na eventualidade de haver mais tarde junção do(s) grupo(s) minoritário(s) com o maioritário, os membros que não estavam no grupo maioritário irão pedir uma transferência de estado normal.

Sendo efetuadas todas as candidaturas, cada processo verifica quem é o novo líder, caso seja ele retoma a conexão com o cliente e, caso tenha pedidos por responder ao mesmo, fá-lo-á de seguida. No entanto, caso não seja o novo líder, verifica se foi introduzido na maioria através de um *merge* e verificando que sim, inicia um *state_transfer_request*.

3.5 *Multithreading* e Controlo de Concorrência

A maior parte dos itens mais relevantes a ter em conta para podermos ter um servidor *multithreaded* já foram descritos antes e, como tal, não voltaremos a descrever esses aspetos. No entanto, existem ainda alguns considerações que não mencionamos:

- As variáveis que são representadas apenas por tipos primitivos e a que mais que uma *thread* acede ou pode aceder ao mesmo tempo (*last_msg_seen*, *transactions_completed*...) são todas atómicas, utilizando a implementação de variáveis atómicas da biblioteca *java.utils.concurrent*. Isto permite-nos garantir que não existe mais do que uma *thread* a escrever sob a mesma variável ao mesmo tempo, ou seja, a atualização das mesmas é atómica, e permite-nos, mais ainda, diminuir o número de zonas críticas que fazem operações mais pesadas como sincronização ou *locking*;

- Outras estruturas que não sejam constituídas por tipos primitivos, nomeadamente o histórico, o *map* dos pedidos por responder e a *queue* de *updates* por mandar aos *backups* têm todos *locks* diferentes associados, de modo a tornar as várias zonas críticas o mais pequenas possível, para aumentar o nível de paralelismo do programa. Todos os acessos a estas variáveis são feitos com os devidos *locks*.

- O número de *threads* é fixo, sendo ele o mesmo para todas as instâncias do servidor, e tiramos partido de *multithreading* utilizando um *ExecutorService* com uma *thread pool* fixa com esse número de *threads*, que está encarregue de fazer receber os pedidos do *NettyMessagingService*, e todas as *threads* da *thread pool* acedem à mesma instância partilhada do *SpreadMiddleware* (onde está uma grande parte dos mecanismos de controlo de concorrência que descrevemos).

4 Análise de *Performance*

De modo a podermos medir a *performance* do sistema, realizamos alguns testes, avaliando algumas métricas relevantes. Nesta secção iremos explicar as condições dos testes, a metodologia utilizada e os resultados que obtivemos.

4.1 Condições e Metodologia de Teste

Os testes foram realizados num dos computadores pessoais da equipa, um *OMEN by HP 15-ce012np*[3].

Nome	OMEN by HP 15-ce012np
CPU	Intel® Core™ i7-7700HQ [2]
#Cores	4
Frequência	2.8 GHz (base), 3.8 GHz (max.)
Memória	32 GB DDR4-2400 SDRAM

Table 1: Descrição da máquina da equipa

No que toca à metodologia de teste, fizemos testes com números variáveis de clientes (1, 2 ou 4), cada um a enviar até N pedidos, sempre para o mesmo número de *threads*, ou seja, 8. Um número maior de *threads* não iria afetar o *throughput* pois os clientes são bloqueantes, como seriam num contexto de mundo real, pelo que débito dos pedidos vai ser afetado pela latência da rede, uma vez que dificilmente conseguiremos enviar um número muito elevado de pedidos de uma só vez, e os clientes são processos um pouco pesados para o *IntelliJ*, ferramenta que utilizamos para desenvolver o código do projeto. Mais ainda, os clientes foram testados para cada tipo de pedido (N pedidos de *movement*, por exemplo), e por fim testados com um número aleatório de cada tipo de pedido.

Para a caracterização de um sistema de resposta a pedidos com o modelo servidor-cliente as métricas mais relevantes serão o **débito de pedidos**, ou seja, o número de pedidos que o servidor consegue tratar numa unidade de tempo, e o **tempo médio de resposta**, ou seja, o tempo médio que o servidor demora desde que recebe um pedido até que o responde. As duas métricas parecem ser o inverso direto uma da outra, mas não são, uma vez que o débito está muito limitado pelo número de pedidos que conseguimos lançar por unidade de tempo, enquanto que o tempo médio de resposta vai estar limitado pela capacidade do servidor de processamento e de gestão de recursos.

Métricas retiradas no cliente não serão tão relevantes, uma vez que estarão todas limitadas e ofuscadas pela latência da rede e pela largura de banda disponível.

As medições foram feitas utilizando um servidor líder estático, ou seja, sem instalação de novas vistas enquanto os clientes fazem os pedidos e dois *backups* de modo a obrigar o envio de mensagens de *state_update*.

Para ambas as métricas foi utilizada a média dos valores obtidos, sendo que utilizamos 1000 pedidos para medir o tempo médio de resposta e 100 000 para medir o débito.

4.2 Débito

Começemos por analisar o débito da aplicação:

	movement	***balance***	***transfer***	***interest***	***history***	**todos**
1 cliente	369	6667	396	398	6789	641
2 clientes	444	12500	454	459	11111	766
4 clientes	678	15385	683	656	14286	1084

Table 2: Débito de pedidos, em unidades por segundo

Aqui começamos a ver a diferença que provoca a coordenação extra com os *backups* aquando de uma operação de escrita. As operações de leitura apenas, que, mal o servidor primário acabe de processar, responde ao cliente, têm um débito muito maior do que as de escrita.

Podemos ver, mais ainda, que os valores escalam com o número de clientes ativos, sendo maior quanto mais pedidos o servidor receber por unidade de tempo, o que é de esperar.

Aqui começamos já a ver que a operação de *movement* tem um comportamento estranho quando comparada com as outras operações que envolvem escrita, mas será mais notório quando avaliarmos o tempo médio de resposta.

4.3 Tempo Médio de Resposta

Analisemos agora o tempo médio de resposta dos pedidos:

	movement	***balance***	***transfer***	***interest***	***history***	**todos**
1 cliente	1.443	0	1.172	1.351	0	0.653
2 clientes	1.854	0	1.598	1.835	0	1.669
4 clientes	5.664	0	5.081	5.455	0	3.288

Table 3: Tempo médio de resposta a pedidos (ms)

Há dois aspetos que surgem nestes resultados que merecem uma análise mais cuidada:

1. Em primeiro lugar, os valores que obtivemos para as operações de *balance* e *history* foram nulos. Acreditamos que isto aconteça porque as operações são

bastante rápidas, apenas precisando de fazer *lock* e ler um valor ou uma lista, não necessitando de coordenação com os *backups*, e, mais importante, o mecanismo de instrumentação que utilizamos, com a class *LocalDateTime* não tem precisão suficiente para fazer as medições que precisamos. Em trabalho futuro, deveríamos utilizar classes ou métodos mais robustas e precisos para esse efeito, como *System.nanoTime()*.

2. A operação de *movement* demora mais do que operações de *transfer* ou *interest*, para todos os casos. Este resultado é inesperado, e não encontramos uma justificação válida para isto. Sendo que não aprofundamos nos testes destes aspetos, é possível que isto advenha da utilização de uma condição de verificação de existência de uma chave no *Map* das contas. Isto não justifica, no entanto, pois em teoria isto é uma operação de tempo constante, enquanto que temos várias operações de tempo linear no *interest*, que deveria tornar o método mais lento, mas não se verifica.

No entanto, a maior conclusão a retirar destes testes é que, mesmo num ambiente de concorrência entre clientes, o tempo de execução de um pedido não ultrapassa em média 6 milissegundos, o que ainda é um pouco elevado, mas é de esperar, quando avaliamos a quantidade de coordenação que este tipo de serviço tolerante a faltas apresenta. Também seria de esperar que com um maior número de clientes no serviço maior será a latência, uma vez que a concorrência pelos mesmos recursos tenderá a ser maior e, portanto, vão haver mais *threads* em espera do *lock* da zona crítica que requisitam. A utilização de variáveis atómicas *lock-free* ajuda bastante neste aspeto, diminuindo o número de zonas críticas do código e de operações de zonas críticas.

Quando olhamos para esta métrica por si e para a sua evolução com um número variável de clientes, podemos concluir que o sistema não escala perfeitamente, mas que tenderá a manter um comportamento aceitável, em princípio, se aumentássemos a escala do sistema. O débito de pedidos crescer com o número de clientes também corrobora esta ideia.

5 Conclusão

Neste trabalho conseguimos aplicar técnicas reais de implementação de sistemas tolerantes a faltas num exemplo prático, permitindo-nos consolidar o conhecimento que obtivemos ao longo da cadeira de Tolerância a Faltas, bem como a utilização de técnicas dadas noutras cadeiras do perfil de Sistemas Distribuídos.

Acreditamos que o nosso trabalho foi um sucesso, sendo que cremos ter implementado todos os requisitos e valorizações do enunciado do projeto, utilizando as devidas técnicas. Algum trabalho de otimização e análise mais profunda poderia ser feito no futuro, de modo a tornar este serviço num mais completo e rápido.

6 Referências

[1] *Atomix: A reactive Java framework for building fault-tolerant distributed systems*. URL: https://atomix.io/.

[2] *Intel Core i7-7700HQ Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/97185/intel-core-i7-7700hq-processor-6m-cache-up-to-3-80-ghz.html.

[3] *OMEN by HP 15-ce012np specifications*. URL: https://support.hp.com/gb-en/document/c05631874.

[4] *The Spread Toolkit*. URL: http://www.spread.org/.