Universidade do Minho

Tolerância a Faltas

Trabalho Prático

Serviço de Escalonamento de Tarefas Distribuído para um Intermediário de Bolsa de Valores

José Ferreira - a78452 Jorge Oliveira - a78660

Nuno Silva - PG38420

30 de Junho de 2019

Resumo

No presente relatório será feita uma contextualização do problema a resolver e apresentadas as motivações e objetivos para o sistema a implementar. Numa segunda parte será abordado o desenvolvimento aplicacional, detalhando a estratégia de desenvolvimento, a solução proposta e os testes efetuados sobre a mesma. Por fim será feito um balanço do trabalho efetuado e dos desafios encontrados durante a sua realização.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

O mercado da bolsa é bastante atrativo a nível financeiro atualmente e leva a que muitas pessoas procurem mais conhecimento para poder investir corretamente no mesmo. Para corresponder a estas necessidades, tem também havido um crescente de aplicações capazes de tratar de facilitar a compra e venda de ações para os clientes, que devem ser fiáveis e capazes de resistir a falhas.

Hoje em dia há uma enorme preocupação sobre as falhas que um sistema possui, mesmo que as mesmas sejam esporádicas, mas quando acontecem são capazes de danificar por completo uma aplicação. Seria ótimo arranjar um mecanismo para as evitar por completo, mesmo com a escrita de um código cuidadoso e bem estruturado não significa que a falha não possa aparecer já que a mesma está associada a diversos fatores. Mesmo que estejamos perante um código perfeito a falha pode ocorrer até por efeitos ambientais. Podemos fazer de tudo para que a falha não ocorra, mas é necessário implementar um mecanismo que seja capaz de reagir, caso a mesma aconteça, de modo a corrigi-la ou

então a mascará-la dando a entender ao utilizador comum que o sistema continua em perfeito funcionamento.

Quando temos um elemento vital no nosso sistema então a única solução passível para contornar a sua falha replicar esse mesmo elemento, no que diz respeito ao nível funcional do mesmo. Necessitamos de introduzir redundância e esta é conseguida através da **replicação**.

1.2 Motivação e Objetivos

O servidor da nossa aplicação é um órgão vital da mesma, pelo que é necessário introduzir replicação sobre o mesmo, para que caso ocorra a sua falha então o sistema continua a correr como "se nada tivesse acontecido"do ponto de vista do cliente.

Com a introdução da replicação não só tornamos a nossa aplicação tolerante a falhas, como também proporcionamos uma alta disponibilidade da mesma, permitindo assim que esteja *online* e desta forma sempre preparada para a utilização por parte do cliente.

A replicação pode ainda aumentar a performance de uma aplicação, porém, para esta aplicação em concreto, esse fim não era o principal objetivo na sua utilização.

Desta forma os objetivos que pretendemos que a nossa aplicação cumpra são os seguintes:

- Fornecer a possibilidade de um cliente registar os seus *holders*;
- Proporcionar a troca de ações, através da possibilidade do utilizador comprar ou vender ações;
- Garantir uma alta disponibilidade da aplicação;

- Assegurar que a mesma seja tolerante a falhas;
- Tornar a recuperação da informação de um novo servidor o mais eficiente e transparente possível;

Capítulo 2

Desenvolvimento Aplicacional

2.1 Estratégia de Desenvolvimento

2.1.1 Operações

Inicialmente definimos as operações com que a aplicação teria de lidar. É uma tarefa crucial, visto que a seleção de certas operações podem levar a que seja inviável o uso de algumas técnicas de replicação. Desta forma, primeiramente foi necessário determinar as operações do sistema para que depois pudéssemos aplicar uma técnica de replicação compatível.

Como foi referido anteriormente o objetivo do nosso sistema é intermediar a troca de ações entre diferentes acionistas. Para isso definimos que:

- Para que um acionista possa comprar/vender ações deve estar registado no sistema;
- A compra de ações envolve sempre dois acionistas;

- Para que uma ação possa ser comprada o vendedor tem de ter efetuado pedidos para venda no valor da quantidade requerida pelo comprador.

Por conseguinte, definimos as seguintes operações:

- Para que um acionista efetue o seu registo no sistema, deve indicar a quantidade de ações disponíveis;
- Um acionista realiza um pedido de venda de ações, passando estas a estarem disponíveis para venda;
  - As ações apenas passam a estarem disponíveis para venda caso o acionista tenha uma quantidade de ações igual ou superior à quantidade definida para venda
  - Quando as ações são colocadas para venda deixam de pertencer diretamente ao acionista, passando a estar no sistema para que outro acionista as adquira;
- Um acionista compra ações a outro acionista;
  - Apenas pode comprar as ações caso exista uma quantidade suficiente disponível para venda. Isto é, caso queira comprar 10 ações mas de momento só existem 5 então não é possível realizar a operação;
- Consulta do estado atual do sistema, expondo os acionistas presentes e as ações disponíveis;

Tendo em conta os requisitos mencionados anteriormente, a *API* desenvolvida é detalhada na figura abaixo.

```
public CompletableFuture<Boolean> venda(String holder,long quantidade) throws SpreadException {...}
public CompletableFuture<Boolean> compra(String holder,long quantidade,String comprador) throws SpreadException {...}
public CompletableFuture<Boolean> regista(String holder, long acoes) throws SpreadException {...}
public CompletableFuture<HashMap<String,Holder>> holders() throws SpreadException {...}
```

Figura 2.1: API da Aplicação

2.1.2 Técnica de Replicação

Depois da determinada a API da aplicação prosseguimos para a escolha da técnica de replicação. Durante o funcionamento da Unidade Curricular estudamos de forma pormenorizada duas técnicas de replicação: **ativa** e **passiva**.

Como todas as operações definidas para o sistema são determinísticas e pelo facto da replicação ativa ter a vantagem de ser transparente para o cliente já que o mesmo não necessita de saber quem é o servidor para o qual deve de enviar pedidos, decidimos utilizar a replicação **ativa**. Para além disto, o modelo de faltas assumido pelo grupo foi o de faltas omissivas, visto ter sido o modelo assumido nesta Unidade Curricular, o que permite a que apenas seja necessário esperar pela resposta de um servidor, aumentando as vantagens da replicação ativa. Obviamente que todas estas vantagens têm alguns pontos negativos, sendo estes discutidos com detalhe posteriormente no presente relatório.

De salientar, que podemos esperar apenas por uma resposta porque, como estamos a considerar faltas omissivas, as respostas das diferentes réplicas vão ser as mesmas já que os pedidos são determinísticos. Se optássemos por um modelo de faltas assertivo, onde as respostas podem ser alteradas por uma entidade exterior, não chegaria esperar apenas por uma resposta, mas sim por uma maioria das mesmas para obter a resposta correta já que em algumas réplicas a mesma possa ter sido alterada por esta entidade maliciosa.

2.2 Solução Proposta e Detalhes de Implementação

2.2.1 Arquitetura

O nosso sistema contém servidores que são as entidades replicadas que guardam o estado de toda a aplicação. Para intermediar o contacto entre o cliente e os servidores criámos um **stub** que recebe os pedidos do cliente e contacta os servidores para que estes os processem.

Para a comunicação entre as várias réplicas e o *stub* decidimos utilizar o *toolkit* de comunicação em grupo *Spread*. Como temos mais do que um servidor, ou o "cliente" teria de contactar cada um individualmente ou então comunicava em *multicast* para o grupo de servidores. A segunda opção é muito mais viável e a utilização do *Spread* facilita a implementação da replicação ativa, aplicando as propriedades corretas nas mensagens, como por exemplo o *setAgreed* que nos dá a garantia de ordem total.

Cada servidor possuí um ficheiro de *log* que contém todas as operações realizadas pelo mesmo. Este ficheiro tem como objetivo o auxílio de recuperação de estado, já que sempre que um servidor se junta ao grupo deve verificar se possui um ficheiro de *log* e em caso afirmativo executar todas as operações e em seguida pedir o estado e só executar as únicas operações que faltam.

Para identificar univocamente todas as mensagens e operações no sistema decidimos utilizar o *UUID* (uma *API* fornecida pelo JAVA) que nos fornece um identificador único universal.

De referir, novamente, que o modelo de faltas assumido para este sistema é o omissivo onde a falha é devido a uma causa natural e não provocada por uma entidade externa maliciosa. Se escolhêssemos outro modelo teríamos uma aplicação mais preparada para a realidade, no

entanto nesta Unidade Curricular apenas nos foi instruído como tratar faltas omissivas, pelo que decidimos então apenas restringir-nos a este modelo.

2.2.2 Recuperação de estado

Uma parte fundamental da replicação reside na recuperação do estado de um servidor, onde existem 2 casos distintos:

- um servidor que falha, entra novamente e por isso pode ter algum estado em ficheiro;
- um servidor que é totalmente novo e não possui qualquer estado sobre sistema

No caso de ser um servidor que falhou, é efetuada uma leitura no *log* das operações do mesmo. Durante este processo aplicamos as mesmas para que seja possível recuperar o estado, sendo contabilizadas o número de operações lidas e executadas aquando da leitura do ficheiro. De seguida é realizado um pedido de estado às outras réplicas ativas indicando o número de operações lidas. Este número é bastante importante, visto que utilizamos um *LinkedHashMap* para guardar as operações, que nos garante a ordem de inserção, o que permite que as réplicas respondam apenas com as operações que são superiores ao número de operações passadas no pedido. Como o *LinkedHashMap* mantém a ordem de inserção, e o *Spread* nos dá a garantia de ordem total, temos a certeza que todas as operações são mantidas pela mesma ordem. Quando uma réplica indica que possui X operações localmente, então as restantes devem descartar as suas primeiras X operações e respondem com as restantes, garantindo assim a correta recuperação de estado. Ao receber a resposta com as operações restantes, o servidor executa as mesmas para recuperar o estado.

Caso seja um servidor novo a lógica utilizada para recuperar o estado é a mesma. Porém, o mesmo deve indicar no pedido de estado que não executou qualquer operação localmente, ou seja, o número de operações é zero. Assim sendo, a resposta das réplicas contêm todas as operações executadas.

A mensagem de resposta ao pedido de estado efetuado por um servidor é enviada pelo *Spread*, para o grupo privado do mesmo, pelo que foi necessário ter atenção ao limite do tamanho de uma mensagem. Pela investigação efetuada, este limite possui um valor de **100kb**, pelo que decidimos que ao criar a mensagem de resposta, esta deva ser dividida em **chunks** de **80kb** (uma pequena margem de segurança). De seguida, cada parte da mensagem deve de ser enviada por ordem, com a garantia de entrega *FIFO* fornecida pelo *Spread*. Identificamos a ultima com uma *flag booleana* sendo que o servidor que está a recuperar o estado vai armazenando as mensagens recebidas por ordem e quando verifica que é a última, junta todos os *chunks*, reconstruindo a mensagem de resposta.

A mensagem de pedido de estado é enviada em *Total Order Multicast* para o grupo por forma a que o servidor saiba que pode descartar todas as mensagens recebidas antes da receção da sua própria mensagem, já que serão processadas por todas as outras réplicas pelo que já estarão incluídas na resposta de pedido de estado efetuada (o *Total Order Multicast* garante-nos que todas as réplicas vêm as mensagens pela mesma ordem). Após a receção da sua mensagem de pedido de estado, a réplica começa a guardar em fila os pedidos até que tenha o estado recuperado, executando-os todos por ordem.

Para verificar a necessidade de enviar a mensagem de pedido de estado por parte de um servidor utilizamos a ferramenta de *Group Membership* fornecida pelo *Spread*. Com este mecanismo, o servidor verifica a quantidade de membros presentes no grupo e apenas envia uma mensagem de pedido de estado caso existam mais servidores para além dele (ou seja, se o valor for superior a 1). Para ativar esta funcionalidade é necessário colocar a *flag* relativa ao *Group Membership* como

verdadeira no momento da criação da conexão ao *Spread*.

O fato de utilizarmos este mecanismo de recuperação de estado, para diminuir o estado transferido entre 2 servidores, pode levar a que este processo seja pouco eficiente visto ter de executar de novo todos os pedidos. Na secção trabalho futuro referimos uma possível medida com o objetivo de tornar este processo mais eficiente.

2.3 Testes de Avaliação

Por forma a verificar o correto funcionamento do sistema e perceber algumas implicações relativas à falha dos servidores realizamos testes com o propósito de avaliar o sistema. Decidimos realizar testes para averiguar a consistência entre as diferentes réplicas e o desempenho do sistema separadamente.

2.3.1 Avaliação da Consistência

Para avaliar o comportamento apropriado do sistema decidimos realizar um teste no qual um cliente realiza um número arbitrário de pedidos. Os pedidos realizados podem ser:

- um registo de um *holder*
- uma compra de ações
- uma venda de ações

Antes de realizar esses pedidos são feitos, previamente, 20 registos de *holders* no sistema. Decidimos realizar estes registos para que o sistema já contenha alguns *holders* presentes visto que existe uma maior

probabilidade de ser realizada uma operação de compra/venda do que um registo na nossa avaliação. De salientar que mesmo com estes registos podem ser selecionados *holders* que não se encontrem presentes no sistema para a compra/venda (onde a operação deverá retornar falso).

No final, esse cliente espera pela resposta de todos os pedidos e localmente executa apenas os que retornaram o valor *true*, já que estes foram válidos e por isso foram executados do lado dos **servidores**. Depois da execução de todos os pedidos corretos, o cliente compara o seu estado local com o estado fornecido pelos servidores, comparando ambos. Em caso de igualdade de estados concluímos que o sistema está a comportar-se como era suposto (dando a ilusão ao cliente que está a contactar com apenas uma réplica).

Sempre que é feito um pedido ao servidor relativamente ao estado do sistema o mesmo escreve o estado para um ficheiro para que seja possível comparar os estados entre as diferentes réplicas.

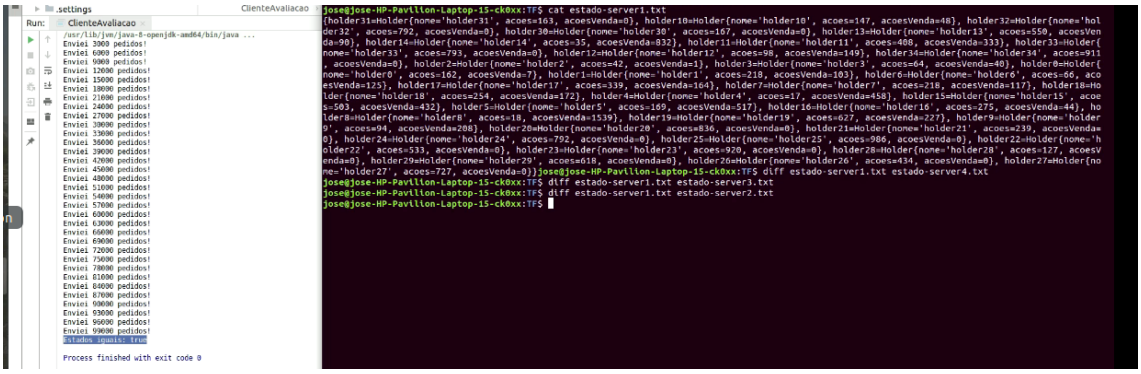

Figura 2.2: Exemplo da Avaliação de Consistência

A figura 2.4, que demonstra um exemplo da avaliação da consistência entre réplicas do sistema, prova o correto funcionamento do sistema uma vez que o estado recebido dos servidores é igual ao estado local no cliente. Para além disso na imagem podemos ainda observar que não

existem diferenças entre os ficheiros de estado das diferentes réplicas, o que nos indica que o estado é igual em todas elas.

2.3.2 Avaliação de Desempenho

Para testar a avaliação de desempenho do nosso sistema decidimos realizar uns testes simples de *benchmark,* utilizando a seguinte abordagem:

- Começamos com 1 cliente e vamos aumentando até 25 clientes concorrentes
- Cada cliente realiza 1000 pedidos
- Depois verificamos as médias do tempo de resposta e o *throughput* relativo a 3 versões:
    1. Com 3 servidores sem falha - 1ª versão
    2. Com 3 réplicas, introduzindo falhas em algumas - 2ª versão
    3. Com 1 servidor - 3ª versão

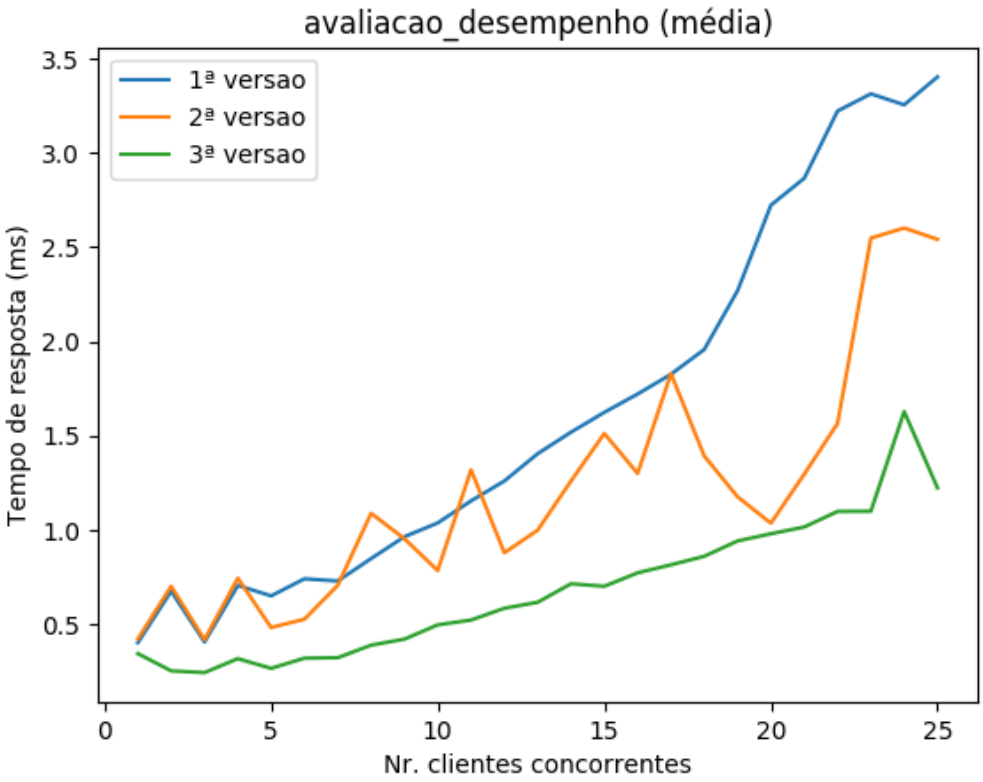

Figura 2.3: Tempo de Resposta Médio

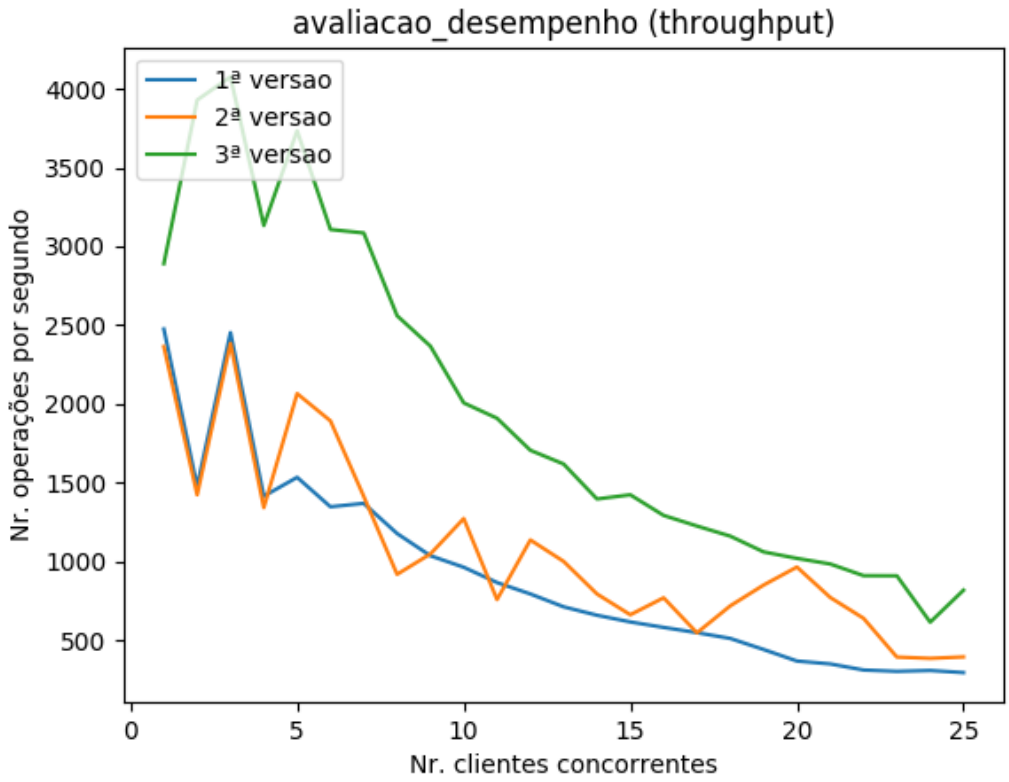

Figura 2.4: Throughput

Como podemos verificar nas figuras acima tanto no tempo de resposta como no *throughput* a versão com falhas apresenta resultados mais eficientes, em média, quando comparada com a versão sem falha. Ao analisar, suspeitamos que esta diferença aconteça porque em alguns momentos possuímos apenas uma ou duas réplicas ativas pelo que o *overhead* criado pela comunicação através do *Spread* seja menor.

Decidimos efetuar o teste com apenas um servidor para verificar se a nossa teoria estaria correta. Ao observarmos os resultados da 3ª versão pudemos concluir que a melhoria advém realmente do aspeto referido, pelo facto dos resultados desta serem bastante melhores. É normal que a falha não tenha impacto do lado do cliente dado que apenas espera-

mos pela resposta de 1 servidor ou seja do mais rápido, levando a que a falha de uma réplica não tenha qualquer impacto.

Como conseguimos verificar, à medida que aumentamos o número de clientes concorrentes existe uma degradação do desempenho nas duas métricas avaliadas. Isto acontece porque apenas executamos um pedido de cada vez (temos apenas uma *thread*). Este não foi de todo o foco do nosso trabalho, no entanto de seguida vamos discutir uma possível abordagem com o propósito de contrariar esta limitação.

Capítulo 3

Conclusão

3.1 Trabalho Futuro e Desafios de Implementação

Um dos pontos que podem ser melhorados é o método de recuperação de estado local. Atualmente para recuperar o estado local um servidor precisa de ler todo o *log* das operações e realizar as mesmas. Caso existam demasiadas operações este mecanismo de recuperação pode ser muito ineficiente visto que tem de as executar todas novamente. Uma possível melhoria pode ser que periodicamente (de X em X operações) o sistema pudesse fazer um *checkpoint* do seu estado para um ficheiro, eliminando as operações do *log* que já estejam presentes nesse *checkpoint*.

Isto permite que ao recuperar seja lido o ficheiro com o *checkpoint*, contendo o estado do sistema nesse momento, e depois seja lido o *log* das operações e apenas executadas as que são superiores ao *checkpoint* atual do sistema. Esta estratégia é mais eficiente que a atual, visto que, necessitávamos ler por completo ficheiro com uma dimensão menor sobre o estado do sistema, o que seria mais rápido, e de seguida ler o *log*

de operações, que é mais rápido do que o atual, já que apenas contém operações superiores ao *checkpoint* atual (as outras eram eliminadas por já estarem contidas no *checkpoint*).

Por fim, como foi referido anteriormente, caso houvesse a necessidade de executar operações de forma concorrente (necessidade essa que poderia advir da aplicação a ser replicada) seria, provavelmente, necessário utilizar um modelo *multi-thread*.

Isto permitiria o aumento da eficiência para tratar pedidos concorrentes. No entanto, seria necessário usar o modelo de replicação passiva, pelo facto de não existirem garantias de determinismo em execuções concorrentes. Esta alteração traria outros impactos, sendo um deles a perda da transparência da falha de um servidor do lado do cliente (se for o primário pode ser necessária retransmissão do pedido), tendo sempre de ter atenção aos *trade-offs* que esta mudança impõe. Como mencionamos, esta execução concorrente não foi o foco do nosso grupo no trabalho pelo que decidimos não implementar a mesma, sendo contudo um assunto a estudar no futuro.

Uma das dificuldades sentidas durante a implementação da aplicação foi termos atingido por diversas vezes o limite de mensagens com que o *Spread* consegue lidar num determinado espaço de tempo, resultando em erros na recuperação de estado. Desta forma, foi necessário atrasar o envio das mensagens aumentando um pouco o tempo de transferência de estado para que a recuperação fosse executada corretamente.

3.2 Conclusão

Em suma, podemos afirmar que conseguimos cumprir todos os objetivos e os requisitos traçados inicialmente tendo uma aplicação totalmente funcional em que o foco foi a replicação dos servidores para que

o sistema se tornasse tolerante a falhas. O processo de codificação da aplicação permitiu-nos melhorar a nossa familiarização com o *toolkit* de comunicação em grupo *Spread,* aprendendo assim quais são as suas vantagens e desvantagens e quando é que a sua utilização deve ser considerada em futuras implementações.