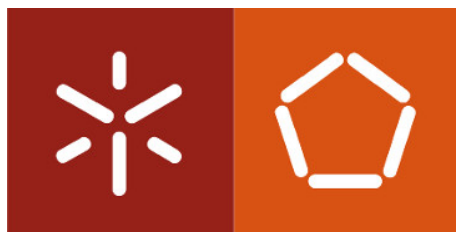


# Tolerância a faltas

<b>Grupo</b>	<b>nr.</b>	<b>5</b>
a84464		Válter Carvalho
a84577		José Pedro Silva
a84783		Pedro Rodrigues
a85006		Hugo Cardoso



Mestrado Integrado em Engenharia Informática  
Universidade do Minho

## Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Arquitetura</b>	<b>4</b>
<b>3</b>	<b>Processamento de um pedido</b>	<b>5</b>
<b>4</b>	<b>Implementação</b>	<b>7</b>
4.1	Replicação Passiva . . . . .	7
4.1.1	Eleição de Líder . . . . .	8
4.2	Base de Dados HSQL Embutida . . . . .	8
4.3	Transferência de Estado . . . . .	8
4.3.1	Estado Incremental . . . . .	9
4.4	Controlo de Acessos Concorrentes . . . . .	10
<b>5</b>	<b>Interface Cliente</b>	<b>11</b>
<b>6</b>	<b>Benchmark</b>	<b>12</b>
<b>7</b>	<b>Requisitos e Valorizações</b>	<b>13</b>
7.1	Requisitos . . . . .	13
7.2	Valorizações . . . . .	13
<b>8</b>	<b>Conclusão</b>	<b>14</b>
<b>9</b>	<b>Anexos</b>	<b>15</b>

# 1 Introdução

No âmbito da Unidade Curricular de Tolerância a Faltas, foi proposta a resolução do projeto prático que consiste na implementação em Java, usando o protocolo de comunicação em grupo Spread, de um serviço tolerante a faltas. Este projeto consiste na criação de um banco que tem  $C$  contas pré-definidas onde existe a possibilidade de executar as seguintes operações:

- **Deposit:** Operação que permite fazer depósito de uma quantidade positiva de dinheiro numa conta.
- **Withdraw:** Operação que permite fazer o levantamento de uma quantidade positiva de dinheiro de uma conta.
- **Transfer:** Operação que permite fazer movimentos entre duas contas.
- **Extract:** Operação que permite pedir um extrato das últimas  $N$  operações de uma conta.
- **Fees:** Operação que aplica uma taxa de juros a todas as contas cujo saldo é superior a 0.

De modo a garantir tolerância a faltas, foi implementado um modelo de replicação passiva onde existe um servidor primário que executa os pedidos e posteriormente envia as atualizações de estado aos servidores secundários. De modo a garantir a resiliência e a persistência de dados cada servidor contém uma Base de Dados HSQLDB embutida.

Operações concorrentes levam muitas das vezes a race conditions. Devido a isso, foi implementada uma interface `ReadWriteLock` que controla as escritas e leituras feitas à base de dados de modo a garantir que tal problema não se verifica.

De modo a interagir com os servidores, foi criada a interface cliente que se conecta por **Atomix**, através do qual tem a possibilidade de enviar comandos para o servidor.

De seguida será explicada em maior detalhe a arquitetura do programa assim como a justificação para todas as escolhas feitas ao longo do projeto.

## 2 Arquitetura

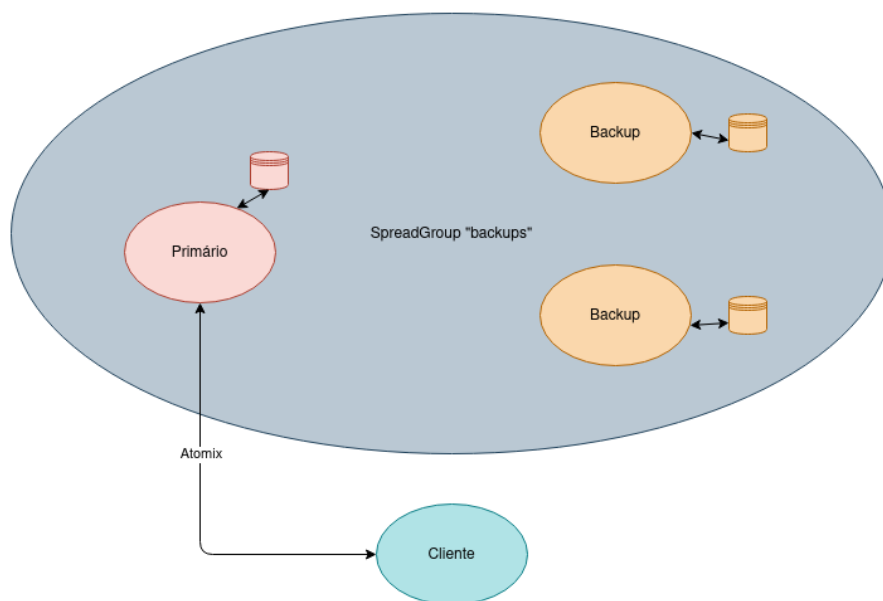


Figura 1: Arquitetura do sistema.

Como previamente descrito, o sistema é composto por N servidores. De modo a estabelecer qual dos servidores é o primário, o critério de decisão utilizado foi o tempo de atividade. Todos os servidores possuem uma base de dados HSQLDB embutida na qual escrevem as atualizações.

O processo de replicação utilizado recorre à replicação passiva, o que significa que apenas o primário irá executar os pedidos e enviar posteriormente as atualizações aos servidores secundários. Estes por sua vez, irão processar esses updates e proceder à atualização das contas implicadas na base de dados.

Por sua vez, a interface **Cliente** interage com o servidor primário através de um socket **Atomix**.

### 3 Processamento de um pedido

Tal como podemos ver pela Figura 1, o cliente conecta-se ao servidor primário através do `atomix` e é por este canal que envia os pedidos.

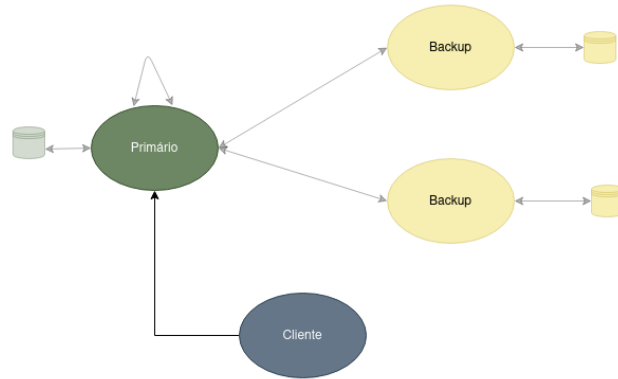
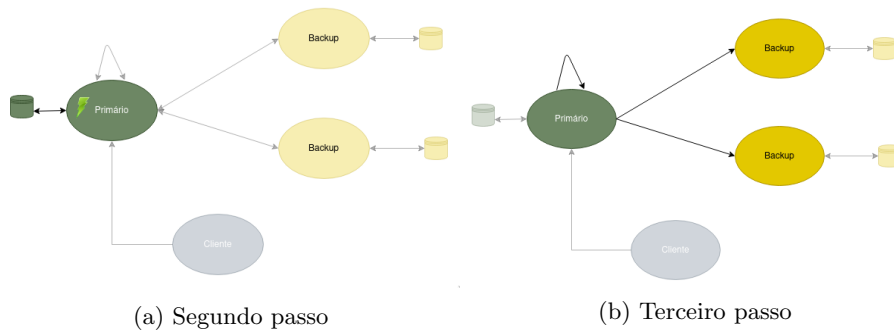


Figura 2: Primeiro passo

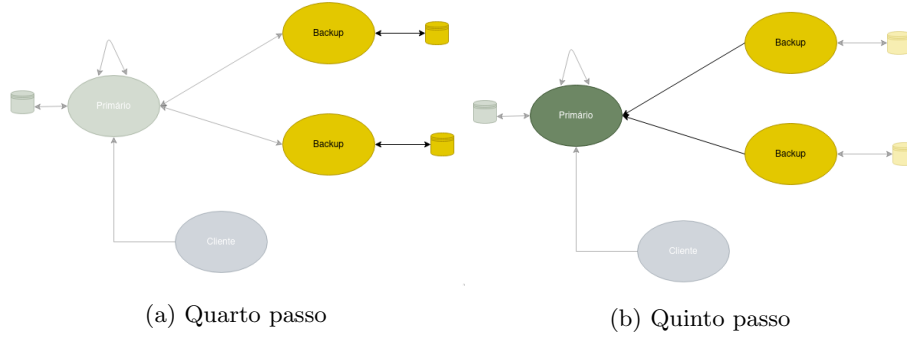
Posteriormente à receção da mensagem, o servidor primário, ligado à sua base de dados que contém o estado da aplicação, executa o pedido do cliente, ilustrado na Figura 3a.

No caso de haverem backups no sistema e o pedido altere o estado da aplicação, é adicionada, a uma tabela, uma entrada que contém o endereço do cliente, uma lista com os backups corretos do sistema naquele momento e a respetiva resposta ao cliente. São posteriormente enviadas as contas atualizadas aos backups (por multicast de grupo, ilustrado na Figura 3b).

Em caso contrário, ou seja, no caso de não haverem backups ou o pedido não alterar o estado da aplicação, é enviada imediatamente a resposta ao cliente, demonstrada na Figura 5



Após a recepção da mensagem multicast, os backups atualizam o seu estado substituindo as respectivas contas pelas recebidas como update (Figura 4a) e enviam um ACK para o servidor primário (Figura 4b).



Posto isto, o servidor primário, após receber um ACK, remove, da tabela referida anteriormente, o backup da lista de backups corretos no momento do pedido, isto faz com que, o servidor apenas envie a resposta ao cliente quando tiver recebido os ACKs de todos os backups a quem enviou o update. Quando, para um determinado cliente, a lista de backups (que faltam enviar ACK) for vazia, então é enviada a respectiva resposta ao mesmo (Figura 5).

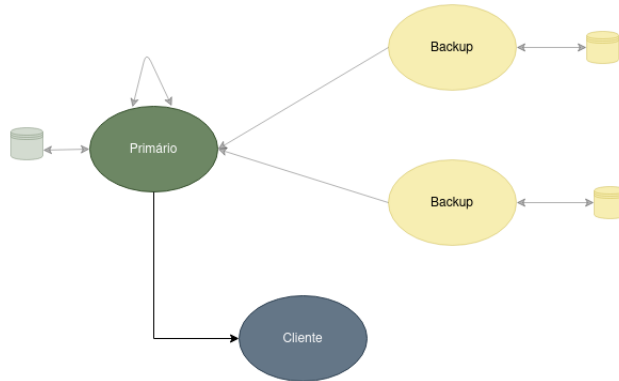


Figura 5: Sexto passo

## 4 Implementação

### 4.1 Replicação Passiva

De acordo com o enunciado do trabalho prático, implementamos o protocolo de replicação passiva. Este protocolo consiste na existência de uma unidade central (servidor primário) para onde os pedidos são enviados e executados. Após a execução destes pedidos, são enviadas atualizações de estado para todas as réplicas do sistema que devem responder com um *ACK*, de forma a que o primário responda ao cliente assim que receber *ACKs* de todas as réplicas corretas (ativas no sistema).

Para tal, foi também implementado um protocolo de eleição de líder, que consiste na eleição do servidor primário. Esta decisão está explicada em detalhe na secção 4.1.1.

Os servidores recebem como argumento o seu identificador/nome (que deve ser único) e a porta por onde devem receber pedidos do cliente. Posto isto, todos os servidores conectam-se ao grupo **backups** e, no caso do servidor primário, conecta-se também na porta referida acima para receber pedidos de cliente através de sockets *atomix*.

Todos os servidores contém também uma base de dados embutida, por forma a garantir a persistência do estado do mesmo. Esta implementação está explicada na secção 4.2. De forma a controlar as operações da base de dados, foram criados protocolos de controlo de acesso a zonas críticas (4.4) e um controlador com todas as operações possíveis:

```
getRequest()  
replaceAccounts(Map<Integer, BankAccount> m)  
getBank()  
movement(int id, String desc, float value)  
transfer(int src, int dst, String desc, float value)  
fees()  
getHistory(int id)
```

Estas operações permitem ao servidor primário executar todos os pedidos dos clientes na base de dados retornando sempre a respetiva resposta e todas as contas que foram alteradas, por forma a ser possível enviar essas contas para que todas as réplicas do sistema atualizem o seu estado.

#### 4.1.1 Eleição de Líder

Tal como mencionado acima, o processo de eleição de líder tem como critério o *uptime* do servidor.

A implementação desta funcionalidade é simples, quando um servidor entra no grupo **backups** verifica se existe mais alguém no grupo, em caso negativo, o servidor considera-se líder.

Caso contrário, o servidor guarda uma lista com todos os elementos que estavam no grupo no momento em que ele entrou e, sempre que outro servidor falhe, este é retirado desta lista. O servidor considera-se líder quando esta lista for vazia.

## 4.2 Base de Dados HSQL Embutida

Como acima referido cada servidor tem uma base de dados HSQLDB embutida. Uma base de dados HSQL é denominada de catálogo. Existem três tipos de catálogos HSQL, nomeadamente, *mem*, *file* e *res*.

Dos diferentes catálogos apresentados, foi escolhido o armazenamento num sistema de ficheiros (*file*), isto porque, no caso do *mem*, caso houvesse falha do servidor, perderíamos os dados uma vez que o catálogo é guardado inteiramente na RAM e não qualquer persistência de dados fora da máquina virtual do Java. Por outro lado, no catálogo *res*, apesar do catálogo ficar guardado num **Java resource** (e.g. jar), estes catálogos são apenas read-only.

Com isto, concluímos que a melhor opção que nos garante uma maior persistência de dados é o armazenamento num catálogo do tipo *file*. Neste, caso ocorra uma falha, o servidor pode sempre voltar a ler o seu último estado.

## 4.3 Transferência de Estado

Esta funcionalidade é necessária para garantir a consistência das réplicas. De modo a garantir tal consistência, aquando a entrada de uma réplica no grupo, é efetuada uma transferência de estado (podendo ser uma transferência completa ou parcial dependendo da versão do estado da réplica).

Inicialmente implementamos uma transferência de estado total, em que o primário ao receber a mensagem que informa que a réplica entrou no grupo, enviava-lhe o estado todo e no período entre a receção da mensagem de entrada e o final da reposição do estado, todas as atualizações recebidas pela réplica eram guardadas numa queue, uma vez que o estado iria apenas conter execuções efetuadas antes da receção da mensagem de entrada.

Posto isto, para evitar que o serviço ficasse interrompido, após guardar o pedido na queue, a réplica envia o **ACK** para o primário, confirmando a receção da atualização (apesar de ainda não ter atualizado o seu estado), por forma a que o primário não tenha de ficar à espera de **ACKs** que apenas iria receber após a transferência de estado.



Após a inserção de persistência de dados, implementamos a possibilidade da transferência de um estado incremental, ou seja, apenas eram enviadas as partes do estado que estariam diferentes às da réplica, evitando que fossem feitas atualizações redundantes, diminuindo assim o tamanho da mensagem e consequentemente o tempo de transferência de estado.

#### 4.3.1 Estado Incremental

A transferência total do estado implica o envio de um `Map <Integer, BankAccount>` contendo todas as contas do banco. No entanto, caso um backup falhe e consiga reiniciar após serem executadas poucas operações no sistema, apenas seria necessário enviar nesse `Map` as contas que tivessem sido mudadas nessas operações em que esteve "em baixo".

Para implementar isso, foi acrescentada uma tabela à base de dados que contém uma única entrada com um inteiro que representa o número da versão do estado. Este número aumenta de forma diretamente proporcional ao número de updates feitos. Foi também criada uma `LimitedQueue` no servidor primário que vai guardando os últimos `K` updates enviados, ou seja, as últimas alterações de versão.

Posto isto, quando uma réplica entra no grupo, envia uma mensagem multicast indicando a versão do seu estado. Após a receção desta mensagem por parte do primário, este verifica se a diferença entre o número da versão que ele contém e o número da versão recebida é menor ou igual a `K`, ou seja, verifica se os últimos pedidos guardados chegam para recuperar o estado da réplica que quer entrar.

Se isto se verificar, a queue é transformada numa lista e envia uma sublista contendo todos os updates necessários. Na réplica, estes updates são realizados pela ordem correta (mesma de que o primário) e a versão do estado é atualizada.

Caso a diferença entre as versões seja maior do que o número de pedidos guardados, então procede-se à transferência total do estado.

## 4.4 Controlo de Acessos Concorrentes

O sistema deve permitir operações concorrentes, ou seja, operações que não estejam em conflito têm a possibilidade de serem executadas em paralelo. Para implementar esta funcionalidade, decidimos criar uma classe **BankLock** que controla os acessos ao estado do servidor. Esta classe depende de uma outra **AccountLock** que implementa o controlo de acessos a uma conta.

A classe **AccountLock** contém as operações de `writelock()`, `writeunlock()`, `readlock()` e `readunlock()`.

O processo de decisão é simples e segue apenas duas regras:

1. Se já houverem processos a ler e outro processo tentar adquirir o lock para leitura, é-lhe concedido acesso à zona crítica (caso não haja outro processo na lista de espera), uma vez que leituras concorrentes não apresentam problemas de conflitos. No entanto, caso um processo tente adquirir o lock para escrita, este é colocado numa queue e deixado num estado adormecido, do qual será acordado quando tiver permissão para o fazer.
2. Caso haja algum processo a escrever, nenhum outro processo pode aceder a esta zona crítica, o que faz com que qualquer que seja o tipo de lock que outro processo tente adquirir, é sempre colocado na queue e adormecido.

Sendo assim, a classe **BankLock** apresenta a seguinte API:

```
public void readlock();
public void readunlock();
public void writelock();
public void writeunlock();

public void readlock(int account);
public void readunlock(int account);
public void writelock(int account);
public void writeunlock(int account);

public void readlock(int account1, int account2);
public void readunlock(int account1, int account2);
public void writelock(int account1, int account2);
public void writeunlock(int account1, int account2);

public void readlock(List<Integer> accounts);
public void readunlock(List<Integer> accounts);
public void writelock(List<Integer> accounts);
public void writeunlock(List<Integer> accounts);
```

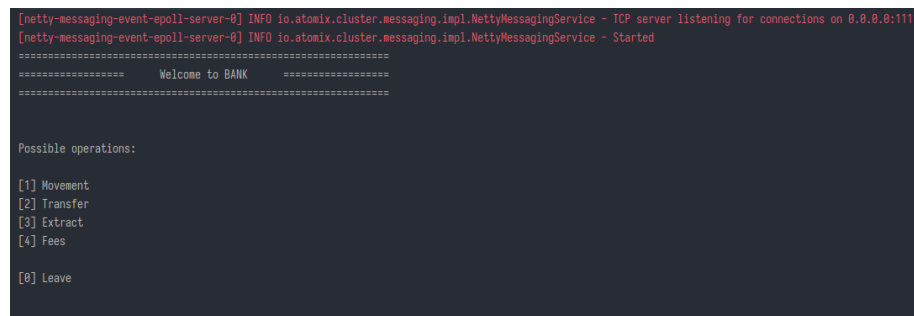
## 5 Interface Cliente

De modo a interagir com o servidor, foi criada uma API para o cliente:

```
public interface IClient {
    boolean withdraw(int amount, int account, String desc)
        throws TimeoutException;
    boolean deposit(int amount, int account, String desc)
        throws TimeoutException;
    boolean transfer(int amount, int src, int dest, String desc)
        throws TimeoutException;
    List<Extract> extract(int account) throws TimeoutException;
    void fees() throws TimeoutException;
}
```

Todos os métodos disponíveis para o cliente tem possibilidade de dar throw de uma `TimeoutException`, isto porque decidimos definir um tempo limite de resposta por parte do servidor. Após esse valor ser ultrapassado, o cliente assume que ocorreu algum problema e dá throw da exceção. Isto permite-nos lidar com um dos principais problemas do protocolo de replicação passiva, que é a falta de total transparência em caso de falha do servidor primário.

Por forma a testar a nossa implementação, a classe `Client` contém uma *Command line interface* com menus que interagem com `stdin/stdout` permitindo a execução das operações apresentadas na API.



```
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - TCP server listening for connections on 0.0.0.0:1111
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - Started
=====
Welcome to BANK
=====

Possible operations:

[1] Movement
[2] Transfer
[3] Extract
[4] Fees

[0] Leave
```

Figura 6: Command line interface.

## 6 Benchmark

De maneira a testar e avaliar o desempenho do nosso programa, foi criada uma classe de teste em `src/test/java/Benchmark`, onde, após receber como argumento o número de clientes ( $C$ ) e o número de execuções ( $N$ ), executa para cada  $C \times 2$  clientes  $3 \times N$  operações. Um cliente deposita 10 euros numa conta, levanta 5 e transfere 5 para um outra, enquanto outro cliente faz o mesmo nessa outra conta.

No final, é impresso no ecrã o tempo de execução total e a média de tempo por pedido, bem como o saldo das duas contas em questão. O valor do saldo das contas deve ser consistente entre os clientes e podemos afirmar que o saldo deve ser igual a  $5 \times N \times C$ .

```
Accounts balance should be consistent and equal to 500.0 €
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - TCP server listening for connections on 0.0.0.0:8881
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - Started
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - TCP server listening for connections on 0.0.0.0:8882
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - Started
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - TCP server listening for connections on 0.0.0.0:8883
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - Started
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - TCP server listening for connections on 0.0.0.0:8884
[netty-messaging-event-epoll-server-0] INFO io.atomix.cluster.messaging.impl.NettyMessagingService - Started
Time elapsed: 43 seconds
Mean time per request: 0.28666666 seconds
[0]: 500.0 €
[1]: 500.0 €
Time elapsed: 43 seconds
Mean time per request: 0.28666666 seconds
[1]: 500.0 €
[0]: 500.0 €
Time elapsed: 44 seconds
Mean time per request: 0.29333332 seconds
[0]: 500.0 €
[1]: 500.0 €
Time elapsed: 44 seconds
Mean time per request: 0.29333332 seconds
[1]: 500.0 €
[0]: 500.0 €
```

Figura 7: Exemplo de execução de benchmark para  $C = 2$  e  $N = 50$

Por forma a ter uma melhor ideia dos tempos de execução, foram executados os testes apresentados na tabela abaixo  $C = 2$ .

# Servers	N	Total Time	Mean Time per Request
3	100	87 s	0.29 s
	500	435 s	0.29 s
	1000	893 s	0.298 s
6	100	100 s	0.33 s
	500	503 s	0.338 s
	1000	1020 s	0.34 s

Como era esperado, o tempo de execução aumenta com o aumento do número de servidores. Isto deve-se ao facto de o servidor primário ter que esperar pelo *ACK* de mais réplicas.

## 7 Requisitos e Valorizações

### 7.1 Requisitos

- *Par cliente/servidor da interface descrita, replicado para tolerância a falhas usando o protocolo de replicação passiva. Exceto na valorização 5(b), pode assumir-se um modelo VS não particionável.*

Justificado nas secções 2 e 4.1.

- *Transferência de estado para permitir a reposição em funcionamento de servidores sem interrupção do serviço.*

Justificado na secção 4.3.

- *Interface do utilizador mínima para teste do serviço. Não se consideram questões de segurança, pelo que não é preciso validar a identidade dos utilizadores.*

Justificado na secção 5.

### 7.2 Valorizações

1. *Utilizar corretamente técnicas genéricas de sistemas distribuídos, tais como serialização, programação concorrente e arquitetura cliente/servidor.*

Justificado nas secções 2 e 4.4

2. *Separar claramente o código entre middleware genérico de replicação e aplicação.*

Visível na figura 8.

3. *Permitir o tratamento de varias operações concorrentemente.*

Justificado na secção 4.4

4. *Fazer uma avaliação experimental de desempenho.*

Justificado na secção 6.

5. *Armazenamento persistente dos dados numa base de dados embedded (e.g., Derby ou o HSQLDB).*

Justificado na secção 4.2.

6. *Permitir transferência de estado incremental.*

Justificação na secção 4.3.1.

## 8 Conclusão

O desenvolvimento deste projeto permitiu a consolidação da matéria teórico-prática lecionadas nas aulas da unidade curricular. Com este projeto conseguimos desenvolver uma solução robusta, que permite a interação entre cliente/servidor, no que diz respeito à tolerância a faltas.

As principais dificuldades deste projeto foram manter a consistência de dados das réplicas, garantindo ao mesmo tempo a possibilidade de execução de operações concorrentes sem conflitos assim como a implementação de uma base de dados HSQL embutida, uma vez que foi a primeira vez que trabalhamos com tal tecnologia.

Em suma, o grupo está bastante satisfeito com o resultado final e considera que cumpriu todos os requisitos assim como todas as valorizações propostas no enunciado.

## 9 Anexos

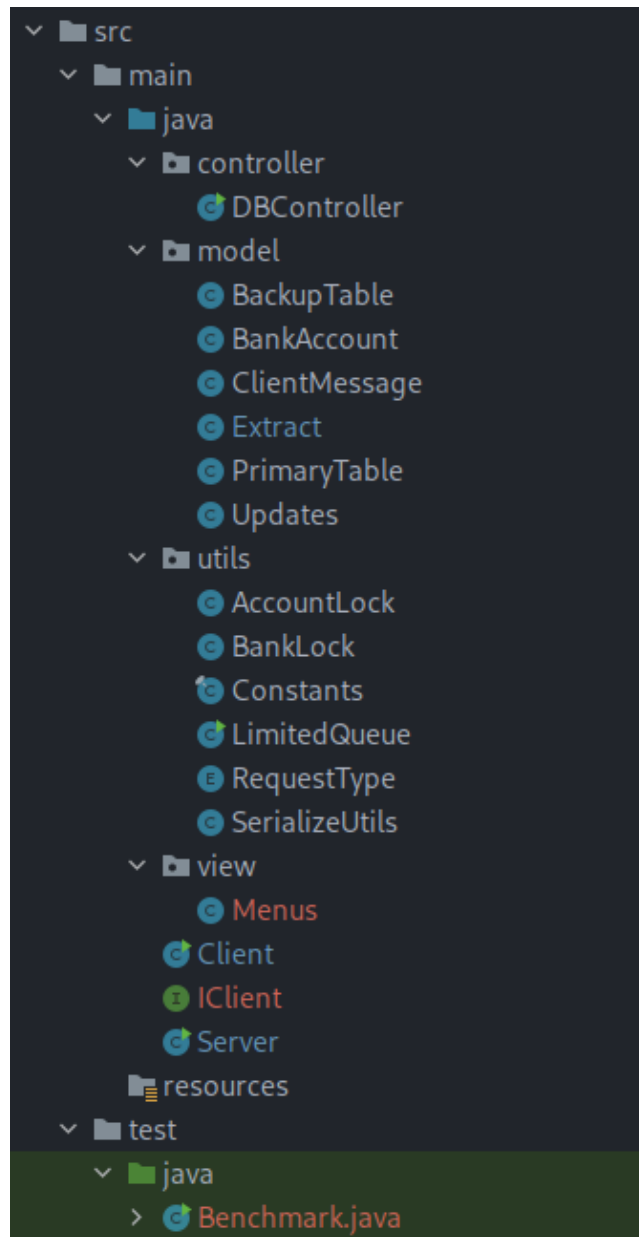


Figura 8: Modularização do código.