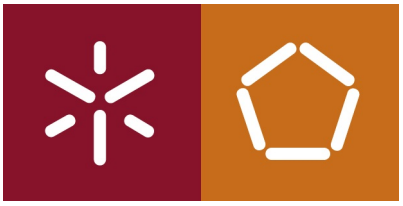

Universidade do Minho
mestrado integrado em engenharia informática

Tolerância a Faltas

(2º Semestre / 4º Ano)

Perfil de Mestrado: Sistemas Distribuídos

Relatório do Trabalho Prático

InTFmarché

Grupo 10:
Henrique Pereira (a80261)
João Silva (a82005)
Miguel Carvalho (a81909)

Conteúdo

1 Introdução

No âmbito da Unidade Curricular Tolerância a Faltas do perfil de Sistemas Distribuídos do Mestrado Integrado Integrado em Engenharia Informática foi-nos proposta a implementação de uma aplicação para uma supermercado online, sendo esta tolerante a faltas. Esta aplicação teria que ser escrita em Java e utilizando o protocolo de comunicação Spread.

Esta aplicação precisaria de ter, como funcionalidades, métodos para controlar a replicação (de forma a tolerar faltas), armazenar de forma persistente o estado da aplicação em bases de dados HSQLDB, permitir a recuperação de um servidor que falhe e possuir uma interface simples que permita ao utilizador efetuar compras.

Ora, relativamente ao supermercado online em si, que apelidamos de *InTF-marché*, este deveria possuir mecanismos de permitam guardar um catálogo de produtos e mostrá-lo aos clientes, pesquisar um certo produto, adicionar e remover produtos do carrinho de compras de cada cliente e, por fim, proceder à encomenda destes produtos, caso haja stock disponível e caso o carrinho de compras se encontra ativo no intervalo de tempo permitido (TMAX).

As decisões que tomamos e as abordagens que seguimos de modo a responder a todos os requisitos e funcionalidades requeridos pelos docentes para esta aplicação serão explicitados nas secções que se seguem.

2 Modelo do Sistema

2.1 Arquitetura Física

Desde início para a resolução do problema admitimos uma arquitetura para o sistema onde deveriam estar presentes algumas componentes base. Desde logo, **Load Balancers**, **Servidores**, **Bases de Dados**, uma por servidor, e por fim obviamente os **Clientes**.

Todas estas componentes serão detalhadas ao longo do relatório.

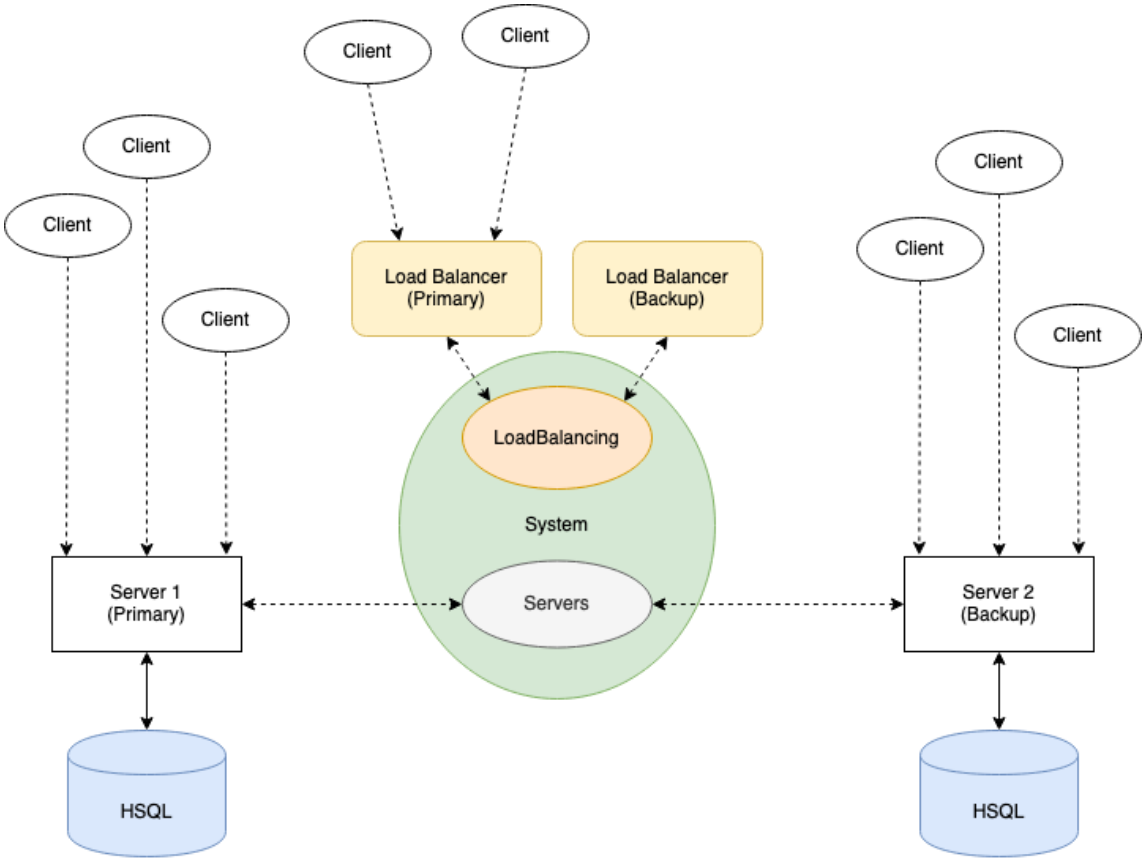

Figura 1: Uma possível arquitetura do sistema

A Figura 1 mostra uma possível configuração do sistema onde existem 2 Load Balancers, 2 servidores, uma Base de Dados por servidor, e múltiplos clientes ligados aos servidores e também a contactarem o Load Balancer primário.

2.2 Comunicação

Todas as componentes do sistema encapsulam e serializam as mensagens a trocar, recorrendo a **Protocol Buffers** que são neutros no que toca à linguagem a usar e à plataforma também, sendo por isso um bom mecanismo de serialização de dados estruturados.

Na ligação de **Cliente - LoadBalancer**, são usados **Sockets TCP** como canais de comunicação.

Nas ligações **Servidor - Servidor** e **Servidor - LoadBalancer**, é utilizada a toolkit **Spread**, que encapsula os aspetos desafiadores do assincronismo e permite a construção de aplicações distribuídss confiáveis e escaláveis.

Dentro do **Spread** utilizamos 3 grupos de comunicação com objetivos diferentes:

- **Servers**: Grupo onde se inserem todos os servidores do sistema. Existe para que os servidores troquem informação e tirem partido da comunicação em grupo para funcionarem num estilo de replicação passiva.
- **LoadBalancing**: Grupo onde se inserem todos os Load Balancers do sistema. Existe para que este tipo de servidores troque informação entre si sobre o Load Balancing e funcione num estilo de replicação passiva.
- **System**: Grupo onde se inserem todos os servidores do Sistema. Existe para que os Load Balancers saibam que servidores aplicacionais estão ativos para lhes poderem encaminhar clientes.

2.3 Replicação por Software

Como indicado na secção anterior, existe uma clara separação de papéis nos servidores existentes no sistema, respetivamente entre os Load Balancers e os Servidores Aplicacionais. Em ambos os casos utiliza-se **Replicação Passiva** para tolerar faltas de componentes.

No caso dos servidores Aplicacionais existe apenas uma pequena variação ao protocolo original, isto é, no protocolo original os clientes contactam o servidor primário para realizarem operações. Neste caso, os clientes podem contactar qualquer um dos servidores do sistema, pois qualquer um deles pode responder a pedidos de leitura. No caso de existirem pedidos de escrita em servidores secundários, estes reencaminham os pedidos para todo o grupo, e apenas o primário os trata, enviando mais tarde as modificações que as operações originaram.

2.3.1 Eleição do Primário

Como utilizamos replicação passiva tanto nos *LoadBalancers* como nos servidores aplicacionais, torna-se necessário escolher um líder para funcionar como **primário**. Existe um líder de *LoadBalancers* e um para os servidores aplicacionais, ainda que o método de seleção seja o mesmo.

Fundamentalmente, o líder é escolhido por ordem de chegada ao grupo, o primeiro a chegar é o líder, se este falhar o líder passa a ser o segundo que chegou. Isto é facilmente conseguido sem que os servidores comuniquem diretamente, basta ao servidor que se junta ver quais os servidores que estavam no grupo aquando da sua entrada, e após a saída de todos esses significa que ele é o líder.

2.4 Arquitetura do Código

De forma a tornar o código mais perceptível e a aplicação modularizada, dividimos o código em sete *packages*.

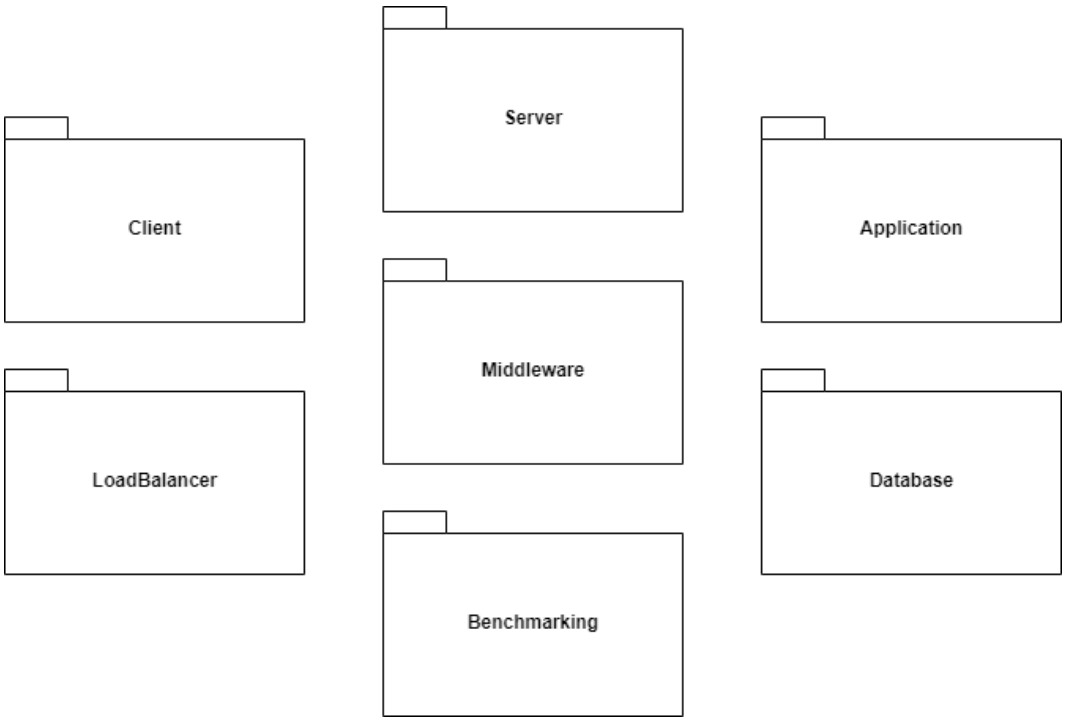

Figura 2: Estrutura do código

Os packages são, então, os seguintes:

- **Client**: pacote composto por todas as classes necessárias ao funcionamento dos clientes da aplicação (interface, driver e stub)

- **Server**: pacote composto pelas classes utilizadas na construção dos servidores (skeleton e managers de replicação e pedidos, por exemplo)
- **Application**: pacote com as classes necessárias à implementação da lógica de negócio da aplicação
- **Database**: classes respetivas às manipulações e controlo das bases de dados
- **Middleware**: classes utilizadas na construção do middleware da aplicação (nomeadamente *protobufs* e funções de *spread*)
- **LoadBalancer**: pacote com as classes utilizadas na construção dos *Load Balancers*
- **Benchmarking**: classes utilizadas para simular clientes e avaliar a performance do sistema (com adaptações do driver e stub do pacote Client)

2.5 Deployment do Sistema

Durante todo o trabalho recorremos ao IDE do *IntelliJ* para dar deploy dos *Load Balancers*, Servidores, e Clientes. Para fazer um *deployment* da arquitetura presente na figura 1, pode-se criar Servidores em qualquer porta, por exemplo na porta 9998 e 9999. No caso dos *Load Balancers* é requerido que estes iniciem na porta 10000, pois é um parâmetro que está *hardcoded* e, por isso, é necessário respeitar. Criando 5 *Load Balancers* todos têm que iniciar na porta 10000.

Quanto aos Clientes, basta correr a sua interface que ele trata de encontrar um servidor.

Para correr o Benchmark, pode-se simplesmente iniciar a main da classe *ClientBot*.

3 Modelo de Faltas

Dado o problema da construção dum supermercado *online*, definimos o modelo em que o sistema irá funcionar, para que seja possível construir uma solução dado o problema e um modelo. Assim sendo, a partir deste momento iremos assumir um modelo de sistema onde:

1. Existe um conjunto finito de servidores;
2. Nodos estão completamente conectados e comunicam através de passagem de mensagens;
3. O sistema é totalmente assíncrono;
4. Os servidores podem falhar por crash e não recuperar;
5. Os canais de comunicação podem experenciar faltas do tipo *Crash-Link* e omissões *Receive/Send/General*;

Dado o problema e o modelo definido acima, existirão vários problemas a serem tratados, como por exemplo, tratamento da recuperação de servidores, transferência de *Timers* para resolver o TMAX, assumir os pedidos não tratados quando o servidor primário falha, etc.

4 Load Balancers

4.1 Propósito

Os *Load Balancers*, como o nome indica, têm o propósito de balancear a carga entre vários intervenientes. No nosso caso, os nossos *Load Balancers* irão balancear os clientes entre os vários servidores, de modo a que estes clientes tenham direito a um serviço similar.

No entanto, pelas limitações da execução passiva, as modificações ao sistema irão todas passar pelo servidor primário, sendo este balanceamento apenas eficaz para as operações de consulta.

4.2 Comunicação

Os *Load Balancers* fazem recurso de duas tecnologias diferentes de comunicação diferentes: **TCP** e **Spread**.

O TCP é o método usado para comunicar com os clientes. Os clientes conhecem apenas um endereço que é utilizado pelo *Load Balancer* primário para indicar os endereços dos servidores (que serão também contactados por TCP).

O Spread é o método utilizado para estabelecer a comunicação em grupo entre os *Load Balancers* e os servidores. Cada *Load Balancer* pertence a dois grupos de Spread, sendo um reservado apenas para os *Load Balancers* e outro para ambos os intervenientes no sistema (*i.e.* os servidores e os *Load Balancers*).

4.3 Balanceamento

Para efetuar o balanceamento de carga, o *Load Balancer* faz recurso de um mapa com as cargas de cada servidor. No momento em que o *driver* do cliente contacta o *Load Balancer*, este indicar-lhe-á o endereço do servidor que se encontrar com menos carga.

Estas cargas são atualizadas quando os servidores aceitam clientes ou quando detetam a saída de um cliente da sua conexão e comunicam tal mudança aos *Load Balancers*.

4.4 Adição/Remoção de Servidores

Um bónus que a utilização de *Load Balancers* oferece é a possibilidade de adição ou remoção de servidores ao sistema sem a necessidade de atualizar o cliente.

Como apenas os *Load Balancers* conhecem os endereços dos servidores e esse conhecimento é baseado totalmente na comunicação feita através dos grupos de

Spread, a lista de endereços dos servidores ativos mantém-se sempre atualizada e as cargas dos mesmos sempre corretas.

4.5 Tolerância a Faltas

A adição de *Load Balancers* introduz no entanto um possível ponto de falha no sistema. Caso os *Load Balancers* não estejam operacionais, os novos clientes não conseguem utilizar o sistema, pois não conhecem os endereços dos servidores a contactar.

Para anular este problema, permitimos que o sistema tenha vários *Load Balancers* preparados a executar, sendo que apenas um responde de facto aos pedidos dos clientes.

Deste modo, os outros *Load Balancers* mantêm as cargas dos servidores replicadas e em caso de falta, um dos *Load Balancers* tomará o cargo de primário para que o serviço do cliente se mantenha ininterrupto.

5 Servidores

Os servidores são a peça fulcral do sistema pois contêm toda a lógica de operação. As subsecções seguintes retratam as decisões importantes que tiveram que ser tomadas relativamente a cada tópico.

5.1 Tratamento de Pedidos

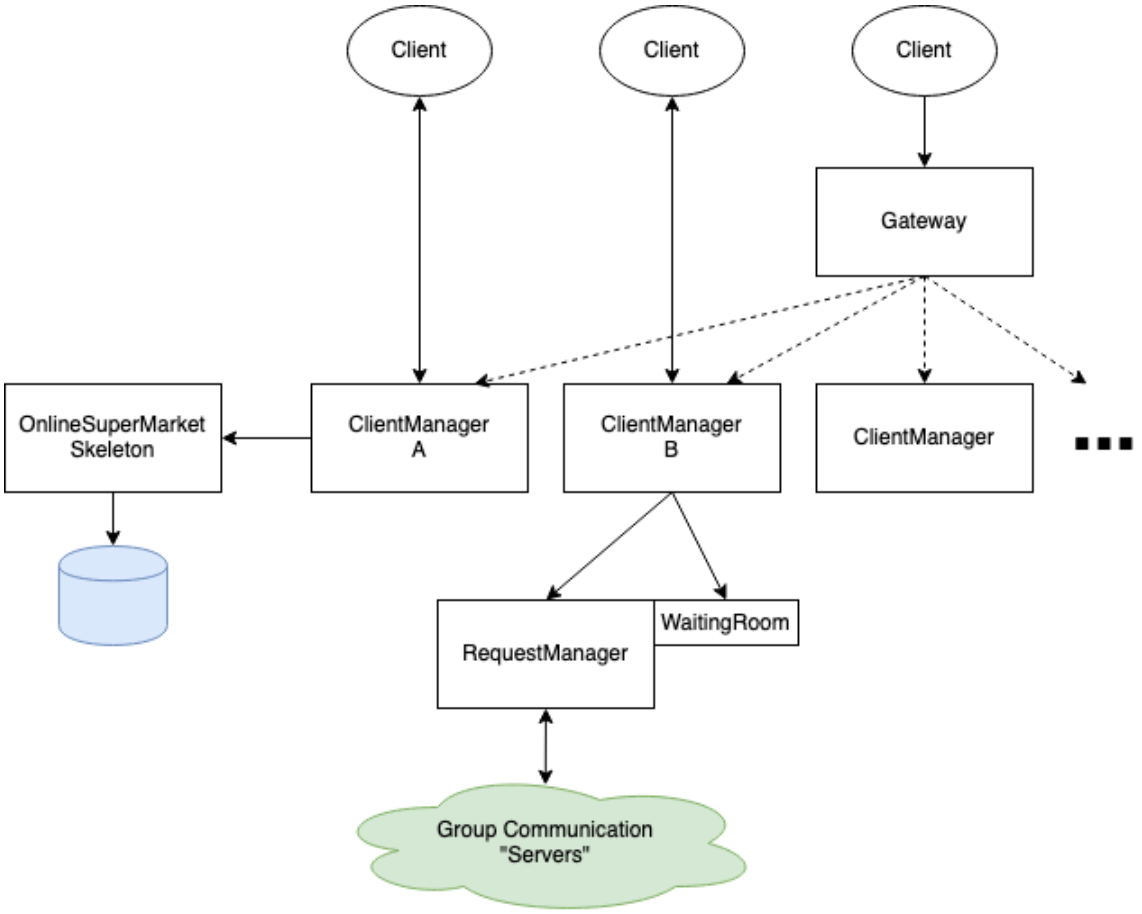

Figura 3: Tratamento de Pedidos

No que toca a tratamento de pedidos estes podem ser divididos por pedidos de **leitura** e pedidos de **escrita** (em detalhe na secção 6). Desta forma, podemos otimizar o tratamento de pedidos do sistema, onde os pedidos de leitura podem ser respondidos por qualquer um dos servidores, sejam eles primário ou secundário. Este caso pode ser retratado pelo *ClientManager A*, na figura 3, onde o cliente invoca

um pedido de leitura e este invoca um método do *OnlineSuperMarket* que acede por sua vez à Base de Dados.

Porém, como um cliente se pode ligar a qualquer servidor, este também terá que fazer pedidos de escrita ao servidor que está ligado. Quando um pedido de escrita é feito por um cliente, desta vez o *ClienteManager B* não pode simplesmente fazer alterações diretamente na Base de Dados, pelo que o *ClientManager B* tem que enviar esse pedido para o *RequestManager*, que por sua vez recorrendo às primitivas de comunicação em grupo enviará o pedido para todo o grupo. Após isso, coloca o *ClientManager* numa *WaitingRoom*, até que chegue a modificação ao servidor referente ao pedido anterioremente invocado.

Quando uma modificação referente ao pedido chega ao *RequestManager*, ele desbloqueia o *ClientManager B* que está na *WaitingRoom* e este entrega a resposta ao Cliente.

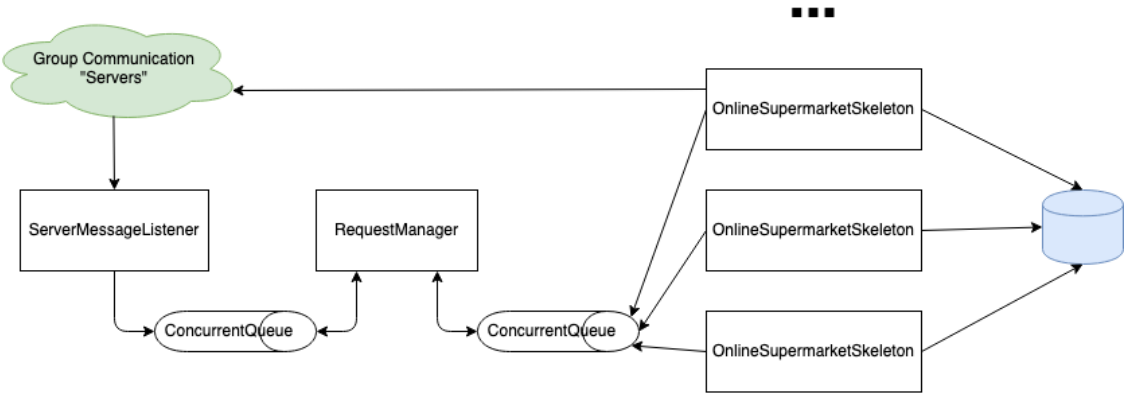

Figura 4: Tratamento de Pedidos Primário

Como os pedidos são partilhados pelo grupo, então todos os servidores ativos irão recebê-los, mas apenas o servidor primário os pode tratar. No caso do servidor ser o primário ele consome o pedido duma *Queue* anteriormente preenchida pelo *ServerMessageListener* e armazena esse pedido caso ele não seja proveniente de si mesmo (nesse caso o *RequestManager* já possui o pedido).

Posteriormente, coloca o pedido numa *Queue* para que as várias Threads do tipo *OnlineSuperMarketSkeleton* pertencentes a uma *ThreadPool*, consumam os pedidos a tratar e os executem na Base de Dados. Após isso, enviam as modificações da execução ao grupo de servidores, para que os restantes se atualizem.

No caso do servidor ser secundário, então o *RequestManager* não coloca o pedido na *Queue*, para que os pedidos não sejam consumidos pela Base de Dados mas coloca-os numa estrutura à parte, para no caso do servidor primário falhar, se possam tratar os pedidos que ainda não obtiveram resposta.

5.2 Tratamento de Atualizações

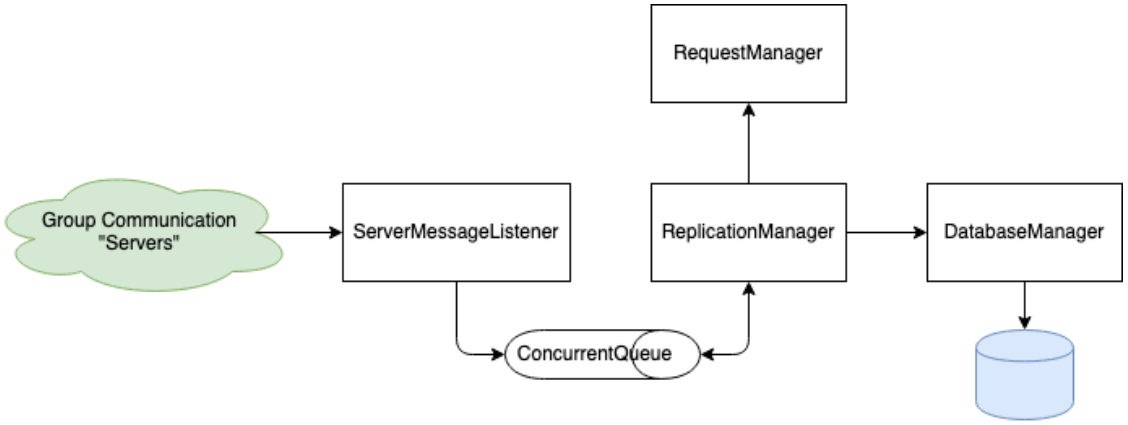

Figura 5: Tratamento de Pedidos

Quando um servidor recebe uma mensagem de atualização este coloca-a numa *Queue* para que mais tarde ela seja consumida pelo *ReplicationManager*. Quando o *ReplicationManager* consome uma mensagem da Queue, ele tem que ver se aquela modificação tem uma origem igual à sua (isto é verdade quando o servidor primário recebe a sua própria mensagem de modificação). Se a modicação for sua então ele não a entrega à Base de Dados pois essa modificação já foi executada anteriormente, mas no caso da modificação não ser proveniente de si mesmo, esta tem que ser entregue à Base de Dados para que esta se atualize (figura 5).

Em qualquer dos casos retratados, o *ReplicationManager* tem que informar o *RequestManager*, para que este possa desbloquear o pedido pendente, ou eliminá-lo caso o pedido não pertença ao servidor atual.

De referir ainda que a execução destas modificações é sequencial pois não se criou maneira de descobrir como executar modifcações concorrentes sem que houvesse conflito. De lembrar ainda que o tratamento de *requests* na Base de Dados é concorrente, apenas as modificações nos servidores secundários é que são sequenciais.

5.3 Tratamento da Recuperação

A decisão do método de recuperação foi talvez aquela que consumiu mais tempo e que mais opções foram tidas em conta para que esta fosse feita tirando partido dos mecanismos da Base de Dados e se diminuisse o volume de dados copiado dum servidor para o outro.

1. **Utilizar o ficheiro .script para recuperação**

Inicialmente assumimos que o ficheiro *script* crescia incrementalmente e albergava apenas as operações interessantes de modificação da Base de Dados, mas isso não se verifica pois o este ficheiro aquando dum *CHECKPOINT* altera as linhas necessárias in-place pelo que não se pode usar este ficheiro apenas numa lógica incremental.

2. **Utilizar o ficheiro .log para recuperação**

   Começamos por assumir que o ficheiro de *log* seria suficiente para realizar a recuperação, mas na realidade, apenas a utilização deste ficheiro não é possível para o efeito. Alguns dos problemas com este ficheiro são indicados a seguir:

   - Se o ficheiro de *log* ficar demasiado grande (10Kb(default) - 4Gb), a Base de Dados fará checkpoint automaticamente, e desta forma não se pode usar apenas o *log* para recuperar um novo servidor pois pode ocorrer um *CHECKPOINT* entre a morte e o renascimento do mesmo;
   - Se fosse possível impedir o *CHECKPOINT* automático, no caso dum servidor morrer durante muito tempo, o ficheiro de *log* dos restantes pode acabar por ser maior do que a própria base de dados, não compensando transferir o *log* mas sim toda a base de dados (ficheiro *script*);
   - O *log* regista todas as conexões à base de dados, e no neste caso qualquer servidor pode responder a pedidos de leitura (ficando as suas conexões registadas) resultando em *logs* distintos entre servidores, tornando a comparação dos mesmos ineficiente e pouco viável.

3. **Realizar a recuperação por software**

   Ainda que após os anteriores problemas sejam complicados, esta solução pareceu viável, mas seria necessário criar um ficheiro de *log* próprio que mantivesse todas as operações importantes de modificações à Base de Dados. Isto levaria a problemas de ficheiros de *log* enormes, que teriam que ser limpos de periodicamente, e no caso dum novo servidor se juntar seria impossível de o recuperar apenas com esse ficheiro, pois não conteria toda a informação importante.

   Por último, com esta solução não tiraríamos partido dos mecanismos da Base de Dados e muito menos estaríamos a poupar na informação enviada.

4. **Utilizar Backups datados da última Atividade**

   Uma solução que foi utilizada por algum tempo, passava pelos servidores fazerem um *Backup* aquando da saída dum servidor do grupo, marcando esse *Backup* como sendo referente ao servidor que saiu. Quando o servidor voltasse,

os servidores fariam um *CHECKPOINT* à sua Base de Dados, comparavam-na com o Backup registado e enviavam as diferenças dos ficheiros, ao servidor acabado de entrar.

Esta solução parte do pressuposto que o *BACKUP* registado na saída do servidor era de facto igual à Base de Dados desse servidor no momento da sua saída, o que pode ser errado quando o número de operações no sistema é muito grande. Seria necessário guardar as operações que ficaram por fazer antes de atualizar a Base de Dados com a diferença recebida.

5. **Utilização dum Backup inicial**

   Esta é última solução encontrada, e é a solução que está implementada no trabalho prático. Neste caso, cada servidor, após a criação da sua Base de Dados e do povoamento inicial, cria um *BACKUP* que dita que aquele é o estado inicial da sua Base de Dados. Neste momento todos os servidores têm Bases de Dados iguais, então todos têm um *BACKUP* que dita este estado.

   À medida que os servidores vão executando, escrevem no seu *log*, pelo que quando um servidor entra ou reentra no sistema, todos os restantes fazem *CHECKPOINT* à sua Base de Dados e comparam-na com o seu *Backup* inicial, enviando apenas a diferença entre esses dois ficheiros. Na receção dessa diferença, o servidor a recuperar, basta repor, o seu *Backup* inicial e fazer um *patch* desse ficheiro com as diferenças recebidas. Fica desta forma, a sua BD replicada.

Dada esta solução é preciso ainda usar mecanismos de comparação de ficheiros (*diff*), e de modificação dum ficheiro dado um resultado dum *diff*, mais nomeadamente um *patch*.

De realçar ainda, que para poupar comunicação apenas o servidor **primário** envia a mensagem de diferença ao servidor a recuperar, ainda que todos os restantes façam todo o processo de *CHECKPOINT* e cálculo das diferenças. O servidores secundários apenas guardam estas diferenças para o caso do servidor primário falhar durante a recuperação de algum servidor e quem assumir o papel de primário tenha que completar a recuperação.

Quando o servidor a recuperar for de facto recuperado, envia uma mensagem de confirmação ao grupo para que todos possam limpar a diferença de ficheiros guardada.

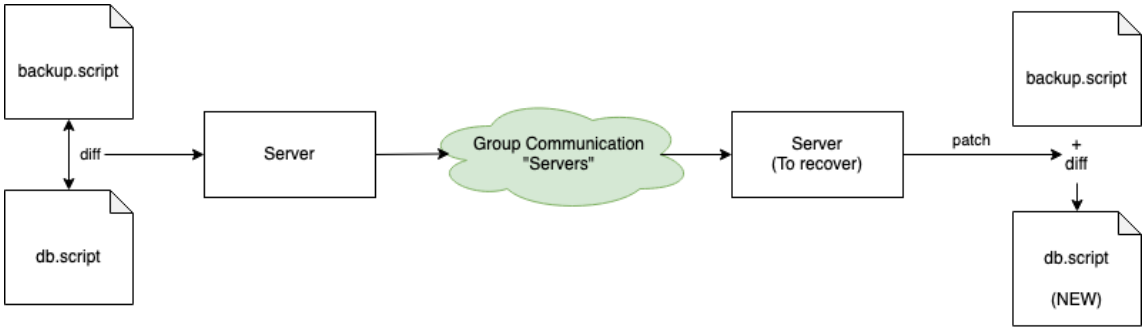

Figura 6: Tratamento de Recuperação

Finalizando esta secção, este método pode não ser o ideal para fazer a recuperação, mas no mínimo consegue comprimir demasiado as operações da Base de Dados, basta imaginar o decremento do stock dum item 100 vezes o que corresponderia a 100 modificações, com este método basta apenas dizer que a linha correspondente a esse item foi alterada. No pior caso envia-se a Base de Dados toda (quando toda sofre alterações), mas no mínimo nada se envia, o que pode ser muito melhor do que todas as soluções expostas anteriormente.

5.4 Tratamento do TMAX

O **TMAX** é, como estipulado nos requisitos, o tempo máximo que o servidor considera que um carrinho deve manter os seus itens. Este tempo serve, essencialmente, para que não sejam guardados conteúdos sobre encomendas que nunca irão acontecer e estão a utilizar recursos do sistema.

Ora, o fixar de um tempo físico entre ações num ambiente sequencial é trivial de programar. No entanto, quando tratamos sistemas distribuídos o tempo físico traz algumas complicações e deve ser evitado.

No nosso caso, dada as faltas possíveis nos servidores, a necessidade de consistência entre as bases de dados e a não sincronização dos relógios de cada servidor, esta tarefa tornou-se bastante complicada.

Começamos por iniciar um temporizador, no servidor primário, de tempo **TMAX** na adição do primeiro item a um carrinho. Após esse tempo, o temporizador irá apagar os itens do carrinho. No entanto, o servidor primário poderá falhar e caso esse temporizador não fosse replicado, nunca se limparia esse carrinho.

Portanto, usando o mecanismo de atualizações dos servidores, na receção das modificações à base de dados, cada servidor irá verificar se foi adicionado um primeiro item a um carrinho. Em caso afirmativo, cada servidor guardará, associado ao dono do carrinho, um *timestamp* físico local e o tempo que o temporizador terá de esperar (*i.e.*, **TMAX**).

Deste modo, quando um servidor secundário tiver de assumir o cargo de primário, irá percorrer esses *timestamps* e criar um temporizador para cada um. Esses terão tempo de espera igual ao tempo guardado subtraído da diferença temporal entre o *timestamp* atual e o *timestamp* guardado (*i.e.*, timer.schedule(SavedDelay - (CurrentTime - SavedTime))).

Quando um servidor novo entra no sistema ou um servidor que falhou quer recuperar, este tem de conhecer a informação dos temporizadores nos outros servidores pois pode vir a ser primário. Para isso, o primário ao detetar que um servidor entrou, envia ao novo servidor a diferença entre o tempo local atual e o *timestamp* que tem guardado para cada temporizador.

O novo servidor, ao receber esta informação, irá tirar um *timestamp* local e guardará nos temporizadores esse *timestamp* e o tempo que o servidor primário lhe enviou. Deste modo, caso este venha a ser primário, o temporizador que criar deverá acabar aproximadamente ao mesmo tempo que o temporizador do primário iria, caso este não tivesse falhado.

Assim, com o envio entre servidores do tempo decorrido em vez de *timestamps* locais, não precisamos de ter os relógios de cada servidor sincronizados entre si e mantemos os temporizadores de **TMAX** corretos.

6 Base de Dados

Para este projeto, utilizamos o HSQLDB (Hyper SQL Database), que possui um driver JDBC e permite guardar os dados das tabelas em memória e em disco, sendo o seu tratamento simples e rápido.

Sendo esta uma base de dados relacional, desenvolvemos o seguinte modelo lógico para a nossa aplicação:

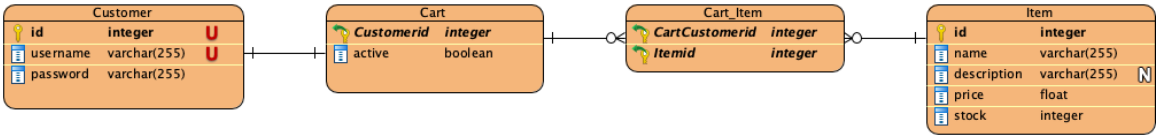

Figura 7: Modelo Lógico da Base de Dados

Ou seja, temos:

1. Clientes caracterizados por um nome de utilizador e uma password
2. Um Carrinho de compras por cada Cliente, com o seu estado (ativo ou inativo)
3. Cada Carrinho tem a identificação de cada Item que contém
4. Um Item é caracterizado pelo seu nome, descrição, preço e quantidade de unidades em stock.

Cada servidor terá uma base de dados própria, gravada em ficheiro, e apenas esse servidor irá aceder a essa base de dados.

Tendo isto em conta, as operações de escrita que definimos irão retornar uma lista de modificações, para que os outros servidores possam processar e atualizar as suas bases de dados com estas modificações, procurando desta forma manter-se a consistência do sistema.

As interações entre o servidor e a base de dados foram realizadas com recurso ao JDBC disponível, utilizando para isso *PreparedStatements*.

Podemos dividir as nossas operações em operações de leitura e de escrita. Como operações de **leitura** temos:

- ***Login*** : verifica se dado username e password constam na base de dados e devolve o ID desse Cliente caso exista.
- **Consultar Catálogo**: obtém os vários Items existentes na aplicação, indicando o seu preço e, mais importante, o seu stock.
- **Consultar Item pelo seu ID**: vai buscar à BD o Item com o ID indicado, para permitir uma consulta mais específica.

- **Consultar Items que estão no Carrinho**: permite verificar quais os items que se encontram no carrinho de compras, de forma a perceber se se podem adicionar mais ou remover.

Como operações de **escrita** temos:

- **Adicionar Item ao carrinho**: adiciona nova linha à tabela Cart_Item, indicando o ID do item a adicionar ao carrinho do Cliente que se encontra *loggado*. Caso o carrinho deste se encontre antes da inserção inativo, a aplicação irá colocá-lo como ativo. Se o Item já existir no carrinho a operação não é realizada pois só pode encomendar uma unidade de cada produto de cada vez.
- **Remover Item do carrinho**: remove item do carrinho com o ID indicado.
- **Encomendar Items presentes no carrinho**: verifica se existem unidades disponíveis no stock dos produtos presentes no carrinho de compras e, caso existam, a encomenda é concretizada com sucesso e é diminuído em uma unidade o stock dos Items do carrinho. Caso contrário, a encomenda não é concretizada e o Cliente terá que remover do carrinho os Items que não possuam stock disponível. Após a encomenda ser completada com sucesso, o carrinho volta novamente ao estado de inativo.

Além das operações referidas, temos também a operação de atualização que, recebendo uma lista de modificações (*Modification*, classe por nós definida que indica a tabela, as colunas e os novos valores a atualizar), procede à execução das *queries* SQL utilizando os parâmetros recebidos.

7 Cliente

7.1 Funcionalidades

O nosso cliente é munido das funcionalidades principais que um *software* de carrinhos de compra deve possuir. No nosso caso, estas funcionalidades podem ser divididas em subgrupos distintos: gestão de sessão, obtenção de informação e gestão de encomenda.

As funcionalidades de gestão de sessão são os típicos iniciar e cessar sessão. Apesar de simples, estes são necessários pois queremos poder identificar o dono de um carrinho para que seja mais simples para o cliente a gestão de uma encomenda.

Para obtenção de informação, o cliente poderá efetuar uma consulta do catálogo ou uma procura individualizada de um item. No catálogo são listados os itens com apenas a informação essencial e caso o cliente pretenda obter mais detalhes sobre o item, deve efetuar uma procura pelo identificador do item apresentado no catálogo.

As operações de gestão de encomenda tratam-se da adição/remoção de itens do carrinho, da visualização do carrinho e do encomendar/cancelar da encomenda. Com estas operações o cliente pode efetuar todo e qualquer tipo de encomenda que desejar, caso essa seja possível dentro das limitações de *stock*.

7.2 Interface

Para a interação do cliente com o sistema foi desenvolvida uma interface de texto simples. Apesar da sua simplicidade, esta engloba toda a funcionalidade do nosso sistema, incluindo a apresentação das respostas do servidor aos pedidos do cliente.

Na Figura 8, é apresentado o menu principal da nossa interface, onde o cliente deve selecionar a operação que deseja efetuar, através da indicação do número indicativo da operação.

Podemos ver também que, para o utilizador ter acesso a este menu, teve de iniciar sessão com sucesso no ínicio da aplicação.

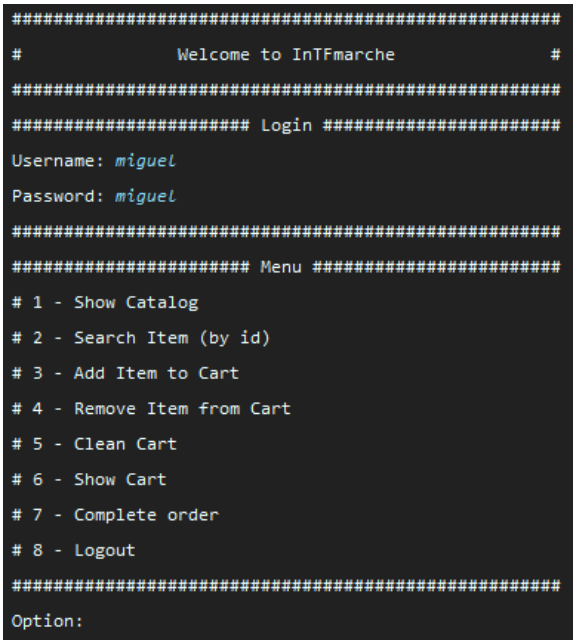

Figura 8: Menu principal

7.3 Comunicação

A comunicação entre o cliente e o sistema é feito, na totalidade, através de *sockets* TCP. Para o efeito, foi criado um *driver* no cliente para garantir que o mesmo possui sempre uma ligação ao sistema, através do *Load Balancer* e dos diversos servidores disponíveis.

O *driver* do cliente, a cada pedido do mesmo, irá verificar se conhece um endereço de um servidor e em caso negativo contacta o *Load Balancer* para o obter. Após a receção do mesmo, o *driver* do cliente redirecionará os pedidos do cliente para esse endereço até que este falhe, sendo nesse momento questionado o *Load Balancer* outra vez.

Deste modo, excetuando o caso em que nenhum *Load Balancer* está a ouvir na *socket* estipulada (o que será complicado devido à redundância introduzida com o uso de vários *Load Balancers*), o cliente terá sempre contacto com o sistema, não percecionando as possíveis falhas no mesmo.

8 Benchmarking

De modo a avaliarmos o desempenho do sistema, montamos um sistema simples de *benchmarking*. Para tal, bastou-nos adaptar os Driver e o Stub do cliente para não utilizar *singletons*, uma vez que teremos várias *threads* a executar e necessitam de usar uma instância diferente das classes supramencionadas.

Assim, decidimos correr um programa com um número de *threads* a representar o número de clientes simultâneos (que definimos como uma lista de números exponencialmente crescente, de 1 a 1024), que se irão *loggar* como um dos utilizadores existentes de forma aleatória e, durante trinta segundos, fazer pedidos de forma aleatória (ou seja, tanto pode ir buscar os items disponíveis no catálogo como encomendar os items presentes no seu carrinho) aos servidores. No final destas execuções registamos o tempo médio de resposta e o número de operações médio por segundo (*throughput*), relativos ao número de clientes em simultâneo.

Este tipo de clientes não simula uma interação real com o sistema, até porque não se simulam tempos de espera entre pedidos, nem se tem um fio de execução de pedidos, por isso quando se diz que o teste tem 1024 clientes, na verdade são 1024 Threads a fazerem pedidos sequencialmente, o que na verdade representa um número de clientes bem maior que 1024.

Tendo isto em atenção, decidimos fazer uma análise do desempenho com clientes simultâneos entre 1 e 1024 e utilizando 1, 2, 3, 4 e 5 servidores. Com os dados obtidos, pudemos gerar os gráficos que se encontram de seguida:

Inicialmente, testamos com apenas uma Thread a fazer operações da Base de Dados (execução sequencial).

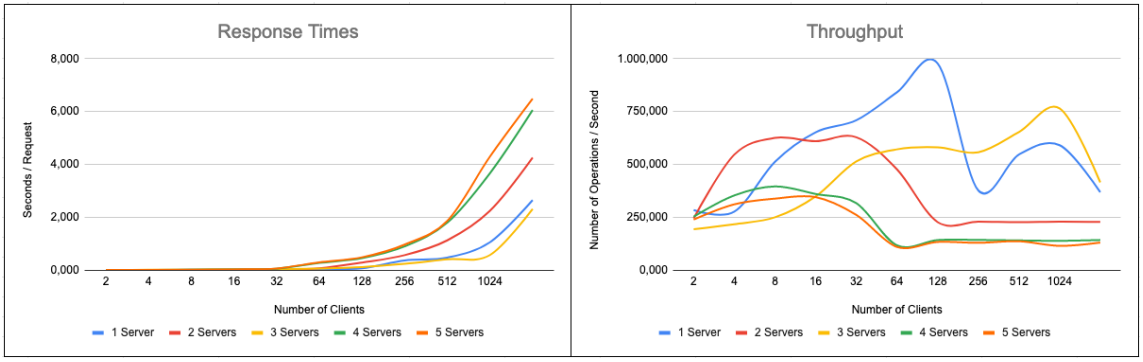

Figura 9: Benchmarking com operações de escrita sequenciais na Base de Dados

Observando o gráfico, podemos concluir que à medida que mais servidores são adicionados mais operações o sistema pode receber, e como executa os pedidos sequencialmente na Base de Dados, os tempos de resposta e respetivos débitos tendem a piorar com o aumento de clientes, tal como esperado.

Após este teste, aumentamos o número de Threads a realizarem operações da Base de Dados para **10**, onde se obteve resultados bem melhores que os anteriores.

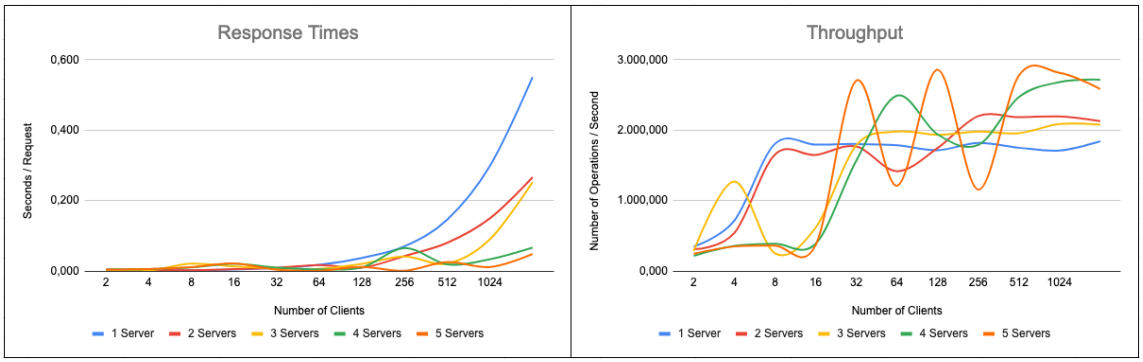

Figura 10: Benchmarking com operações de escrita concorrentes na Base de Dados

Ainda que os gráficos não pareçam apresentar uma tendência, os resultados obtidos deta vez são bem diferentes dos do teste anterior, pois com mais servidores e com mais clientes, observam-se respetivamente, tempos de resposta mais pequenos, e um débito maior.

Desta forma, acentuou-se a necessidade da existência de operações concorrentes no sistema para melhorar a sua escalabilidade.

Para uma configuração com **5 servidores**, e **512 clientes** (melhor débito nos testes anteriores), testamos a falha e recuperação de 1 a 4 servidores. Obtivemos resultados anormais qua não seguem uma tendência o que é no mínimo expectável uma vez que pomos o sistema perante uma situação caótica. Estes resultados devem-se também ao facto de quando um servidor falhar, os clientes a ele ligados procurarem outro servidor ativo, e por exemplo no caso, de falharem 4 servidores, todos os clientes vão se conectar a um servidor o que resulta em toda a carga direcionada a um servidor. Ainda que no fim estejam 5 servidores funcionáveis e recuperados, apenas um estaria com todos os clientes.

Estes resultados são, por isso, pouco fidedignos e irreais, pelo que não merecem um gráfico ilustrativo.

9 Requisitos e Valorizações

Assim, os ***requisitos*** requeridos pelos professores foram cumpridos:

- Par cliente/servidor da interface descrita, replicado para tolerância a faltas usando qualquer um dos protocolos estudados nas aulas : justificado nas secções **2.3**, **4.5** e **5**.
- Armazenamento persistente do estado dos servidores na base de dados HSQLDB : justificado na secção **6**.
- Transferência de estado para permitir a reposição em funcionamento de servidores sem interrupção do serviço : justificado nas secções **5.2**, **5.3** e **5.4**.
- Interface do utilizador mínima para teste do serviço : justificado na secção **7.2**.
- Existência um tempo limite TMAX para a concretização da encomenda: justificado na secção **5.4**.

Relativamente às ***valorizações*** enunciadas, cumprimos as seguintes:

- Separar claramente o código entre middleware genérico de replicação e aplicação : justificado na secção **2.4**
- Permitir o tratamento de varias operações concorrentemente : justificado na secção **5.1**
- Minimizar as encomendas canceladas como consequência de faltas ou do funcionamento do mecanismo de replicação : justificado na secção **5.1**
- Tirar partido do sistema de bases de dados para atualizar os estado dos servidores nas diversas situações em que isso for necessário, nomeadamente, para diminuir o volume de informação copiada : justificado nas secções **5.2** e **5.3**
- Avaliação de desempenho : justificado na secção **8**

10 Conclusões

Olhando em retrospetiva para o trabalho realizado, achamos que fizemos um trabalho completo e que cumpre todos os requisitos referidos no enunciado e praticamente todas as valorizações, exceto a respetiva às partições do *Spread*.

O sistema que montamos é tolerante a faltas, permitindo quer a falha das ligações quer a falha de servidores, mantendo sempre a consistência do sistema.

Como trabalho futuro, uma vez que temos já métodos de obter as modificações realizadas nas bases de dados, poderíamos dar o salto da replicação passiva das BDs para uma execução conservadora, necessitando apenas de um método eficaz de comparação destas modificações e assim identificar-mos as zonas de conflito.

Este projeto permitiu-nos colocar em prática os conceitos abordados na Unidade Curricular Tolerância a Faltas e colocar lado a lado as várias opções de implementação, levando-nos assim a estudar cada uma delas e a selecionar aquela que o grupo considerou a mais adequada.