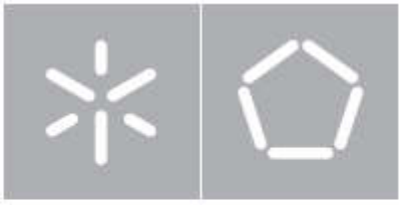
Universidade do Minho
Escola de Engenharia

Hugo André Esteves Gomes

Estudo de Alternativas Open Source Para Soluções IMDG

Julho de 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hugo André Esteves Gomes

Estudo de Alternativas Open Source Para Soluções IMDG

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor Orlando Manuel de Oliveira Belo

Julho de 2016

À minha família.

Agradecimentos

A realização desta dissertação de mestrado contou com importantes apoios e incentivos sem os quais não se teria tornado uma realidade e a os quais estou grato. Agradeço, primeiramente, ao Professor Doutor Orlando Manuel de Oliveira Belo porque, para além de me orientar nesta dissertação, teve toda a sua disponibilidade, interesse e apoio também pela confiança depositada em mim. Ao Professor Doutor José Orlando Pereira, pela disponibilidade, dedicação e paciência demonstrada na resolução de problemas emergentes à implementação das aplicações. À PT Inovação e Sistemas, em especial ao Pedro Salazar, pela cooperação e disponibilidade. À minha família, em especial aos meus pais e irmã, pelo apoio incondicional. À minha namorada, amigos e colegas de laboratório pelo carinho e apoio moral que me demonstraram.

Resumo

Estudo de Alternativas *Open-Source* para Soluções IMDG

Conseguir satisfazer os clientes em mercados altamente competitivos depende diretamente da qualidade e desempenho das aplicações que lhes são direcionadas. Alguns segundos de atraso podem fazer a diferença entre o sucesso e o fracasso de uma empresa. A incapacidade de processar, aceder, analisar e integrar dados rapidamente num dado sistema é bastante problemática para organizações que têm de processar uma grande quantidade e variedade de dados. Os sistemas In Memory Data Grids (IMDG) operam essencialmente com os seus dados em memória, podendo, porém, ser suportados por vários servidores incorporados num sistema distribuído. Estes sistemas são recomendados para aplicações que exijam a manipulação de grandes volumes de dados, uma vez que são facilmente escaláveis e de fácil implementação. Além disso, em termos técnicos, os sistemas IMDG são claramente vantajosos em processos que requeiram rápidas tomadas de decisão, exijam elevados níveis de produtividade e solicitem um atendimento de alta qualidade aos seus sistemas e utilizadores clientes. Neste trabalho de dissertação foram estudas, de forma detalhada, várias alternativas IMDG *open source* existentes na atualidade, tendo como base de trabalho um conjunto de condições funcionais e estruturais definidas por uma empresa de telecomunicações, com o objetivo de viabilizar a utilização de uma solução IMDG *open source* em substituição de uma solução dita comercial. Adicionalmente, idealizou-se um pequeno conjunto de casos de estudo que foram utilizados como base para o processo de criação de duas aplicações práticas reais utilizando duas soluções IMDG *open source* distintas, nomeadamente, o *Hazelcast* e o *Infinispan*. No processo de elaboração destes casos de estudo tomou-se em consideração alguns cenários de aplicação bastante típicos em sistemas de telecomunicações, bem como, nas fases de implementação das aplicações, as funcionalidades mais relevantes que se podem encontrar em sistemas distribuídos deste género, em particular a execução local de dados em ambiente distribuído, a afinidade de dados em casos de particionamento, a capacidade de replicação de *cache* em cenários topológicos com mais de um cluster e, por fim, a integração de *Java Persistence API* (JPA) e *Java Transaction API* (JTA) como mecanismos para controlo e gestão de persistência e das transações distribuídas.

Palavras-chave: Sistemas *In Memory Data Grid, Software Open-Source,* Sistemas de Transações Distribuídas, Avaliação de Soluções IMDG.

Abstract

Study of *Open Source* Alternatives for IMDG Approaches

Being able to satisfy clients in highly competitive markets depends directly on the quality and performance of applications directed to them. Seconds of delay can make the difference between success and failure of a new company. The inability to process, access, analyze and integrate data quickly is more problematic for organizations as they have to process a greater quantity and variety of data. In Memory Data Grids (IMDGs) systems operate with its data in memory, possibly supported by multiple servers embedded in a distributed system. These systems are especially geared to handling large data volumes, featuring a remarkable performance, easily scalable and easy to implement. Furthermore, in technical terms, these systems are clearly advantageous in processes that require quick decision-making, require high levels of productivity and request a high quality service to its customers systems or users. In this dissertation work were studied in detail the existing IMDG *open source* alternatives taking into account a set of functional and structural conditions defined by a telecommunications company, with the aim of enabling *open source* alternatives as paid products substitutes. Additionally was envisioned a small set of case studies as basis for the process of creating two applications with two different IMDGs *open source*, in particular, Hazelcast and Infinispan, demonstrating the versatility of these systems as well as its applicability. In the creation process of the case studies were taken into consideration rather typical application scenarios of telecommunication systems, as well, in the implementation of applications were considered the features relevant to distributed systems of this kind, in particular, local execution of data in a distributed environment, data affinity in partitioning cases, cache replication capacity in topological scenarios with more than one cluster, and integration of Java Persistence API (JPA) and Java Transaction API (JTA) as mechanisms for persistence and distributed transactions management and control.

Keywords: In Memory Data Grid Systems, Open-Source Software, Distributed Transactions Systems, IMDG Solutions Evaluation.

Índice

Introdução 15

1.1 Contextualização 15
1.2 Motivação e Objetivos 16
1.3 Estrutura da Dissertação 18

Sistemas IMDG 21

2.1 Características Base 21
2.2 Principais Funcionalidades 22
2.3 Arquiteturas 27
2.4 Topologias 30

Estudo e Análise de Soluções para *In Memory Data Grid* 33

3.1 Produtos Analisados 33
3.2 Características Relevantes 35
3.3 Os Melhores Candidatos 36
3.3.1 Infinispan 37
3.3.2 Galaxy 38
3.3.3 Ehcache 39
3.3.4 Hazelcast 40
3.3.5 GridGain Data Fabric e Apache Ignite 41
3.4 Comparação Final dos Produtos 45

As Soluções IMDG Hazelcast e Infinispan 49

4.1 Modelo de Dados 50
4.2 Os Casos de Uso 51
4.2.1 Apresentação Geral 51

4.2.2 Os Casos de Uso a Implementar 52
4.3 Implementação das Aplicações 60
4.3.1 A Arquitetura do Sistema *Hazelcast* 60
4.3.2 A Arquitetura do Sistema Infinispan 62
4.3.3 O Ambiente de Desenvolvimento 63
4.3.4 Funcionalidades Chave 66
4.4 Testes de Desempenho 74

Conclusões e Trabalho Futuro 79

Bibliografia 83

Índice de Figuras

Figura 1: Arquitetura base IMDG – figura retirada de Kirby et al. (2009) 28
Figura 2: Arquitetura IMDG – figura retirada de Hazelcast (2015b) 29
Figura 3: Topologia em modo embebido 30
Figura 4: Topologia em modo cliente-servidor 31
Figura 5: Modelo de Dados 49
Figura 6: Referência aos casos de uso selecionados 51
Figura 7: Descrição do caso de uso – Cobrança 53
Figura 8: Diagrama de atividades relativo ao caso de uso Cobrança 54
Figura 9: Descrição do caso de uso – Recarga 55
Figura 10: Diagrama de atividade relativo ao caso de uso Recarga 56
Figura 11: Descrição do caso de uso - Transferência de Saldo 58
Figura 12: Diagrama de atividade relativo ao caso de uso Transferência de Saldo 59
Figura 13: Arquitetura inicial da aplicação utilizando-se Hazelcast 60
Figura 14: Arquitetura atual da aplicação com Hazelcast 61
Figura 15: A arquitetura da aplicação utilizando Infinispan. 63
Figura 16: Definição da Near *Cache* configurada em Hazelcast. 72
Figura 17: Definição da Near *Cache* no cliente Hot Rod. 72
Figura 18: Resultados do Teste 1 – Recarga. 76
Figura 19: Teste 2 – Cobrança 76
Figura 20: Resultados do Teste 3 - Transferência de saldo. 77
Figura 21: Resultados do Teste 4 - Todos os casos de uso. 77

Índice de Tabelas

Tabela 1: IMDGs Estudadas 35
Tabela 2: IMDGs Estudadas 36
Tabela 3: Características das Soluções IMDG Estudadas.....45

Capítulo 1

Introdução

1.1 Contextualização

Usualmente, a ocorrência de situações envolvendo grandes quantidades de dados conduz a dois grandes problemas. O primeiro relaciona-se diretamente com a forma como tirar valor de tão grande melancial de dados em prol do bem-estar de uma organização. Sabe-se que, uma boa análise desses dados pode revelar informação bastante importante para a melhoria dos processos de negócio de uma organização. O segundo problema emerge quando os dados gerados aparecem de forma tão rápida, e em tão grandes quantidades, que as estrutura de suporte ao seu armazenamento e processamento se tornam obsoletas, não sendo mais apropriadas para o suporte dos sistemas operacionais (Prins, 2013).

As infraestruturas informáticas ditas tradicionais têm já grandes dificuldades em gerir volumes de transações muito elevados e lidar com o acesso a dados em tempo real, algo que as novas aplicações hoje exigem. Tendo poucas organizações o orçamento ou o tempo necessários para substituir as aplicações de *backend*, suas bases de dados e infraestruturas, para soluções mais atuais, estas são obrigadas a procurar maneiras alternativas para utilizar o que ainda têm em funcionamento, de forma a poder suportar operacionalmente os novos casos de uso que não tinham sido imaginados aquando os seus sistemas foram projetados e implementados. Com a adoção de uma solução *In-Memory Data Grid* (IMDG), estas organizações poderão ser capazes de acelerar o desempenho das suas aplicações e atingir as expectativas dos clientes com alguma garantia a curto e médio prazo (Kaufman e Kirsch, 2014).

Usualmente, num sistema IMDG os dados estão distribuídos por diversos servidores que asseguram o seu armazenamento em memória, operando em modo ativo. Hoje, no mercado,

encontramos soluções IMDG muito interessantes — *e.g.*, Oracle Coherence, Hazelcast, VMware Gemfire, IBM eXtreme Scale ou JBoss Infinispan —, ocupando já posições de bastante relevo em áreas aplicacionais consideradas de vanguarda. Porém, para que estes sistemas possam utilizar a memória principal como área principal de armazenamento, eles têm de ser concebidos de forma a que sejam fiáveis e que consigam ultrapassar o "limite" da sua própria capacidade de armazenamento. Tal pode ser conseguido através da utilização de uma arquitetura distribuída, com grande facilidade de escalar em termos horizontais e capaz de providenciar mecanismos de replicação efetivos. Apesar das suas excelentes características, os sistemas IMDG levantam inúmeros desafios computacionais. Veja-se, por exemplo, os desafios relacionados com o suporte a transações distribuídas, com o equilíbrio de carga e de dados em ambientes distribuídos ou com as necessidades de pesquisa e indexação de informação persistente.

As IMDG têm vindo a ser adotadas em várias áreas de aplicação, nomeadamente em serviços financeiros, nos quais aumenta a rentabilidade e a competitividade no mercado através do melhoramento de desempenho dos processos de negociação de ações nos mercados financeiros, em revendedores online, oferecendo-lhes uma solução altamente disponível, fácil de manter e escalável, suportando milhões de visitas por mês, ou mesmo na aviação comercial, como sistema de marcação de voos com servidores ativos em vários locais geográficos (Colmer, 2011).

1.2 Motivação e Objetivos

Num mercado tão competitivo como o de hoje, as empresas exigem soluções tecnológicas cada vez mais robustas, seguras e com altos desempenhos. Além disso, para que possam garantir um serviço que respeite os padrões de qualidade requeridos pelos clientes, a garantia da coerência dos dados bem como a rapidez da resposta às suas solicitações são aspetos que não devem ser nunca descurados. As soluções IMDG são, cada vez mais, integradas em aplicações com altos requisitos de qualidade de serviço, devido às capacidades de tolerância a falhas, alta disponibilidade e desempenho que usualmente apresentam. Além disso, em termos técnicos, os sistemas IMDG são claramente vantajosos em processos que requeiram rápidas tomadas de decisão, exijam elevados níveis de produtividade e solicitem um atendimento de alta qualidade aos seus sistemas e utilizadores clientes.

Neste trabalho de dissertação pretendeu-se estudar, de forma detalhada, as várias alternativas IMDG *open source* existentes na atualidade, tendo como base de trabalho um conjunto de condições funcionais e estruturais definidas por uma empresa de telecomunicações, com o objetivo de viabilizar a utilização de uma solução IMDG *open source* em substituição de uma solução dita comercial. Para que isso pudesse acontecer, a empresa de telecomunicação em causa definiu um conjunto de funcionalidades que a aplicação deveria incluir. De referir:

- a execução local dos processos de dados em ambiente distribuído, o que garante um melhor desempenho ao sistema.
- a afinidade de dados em casos de particionamento, que mantém dados relacionados próximos uns dos outros no momento de distribuição dos dados, reduzindo assim o número de "viagens" na rede.
- a capacidade de fazer a replicação da *cache* em cenários com mais de um cluster, algo que é necessário para que não haja perda de dados no caso de ocorrerem situações de falha ou perda de dados;
- a existência de uma *Near Cache* nos clientes que, no caso de ocorrerem muitas operações de leitura, mantém os dados mais próximos do local de processamento e que usualmente traz melhorias no desempenho do sistema;
- a integração de JPA e JTA como mecanismos para controlo e gestão de persistência e das transações distribuídas para tirar partido do ambiente distribuído mantendo sempre os dados persistidos em disco.

As IMDG disponíveis no mercado podem ter custos muito elevados, o que as pode tornar num problema em vez de numa solução. A inclusão de alternativas *open source* – Hazelcast, Infinispan, Galaxy, Apache Ignite, etc. – pode ser considerada a fim de minimizar os custos decorrentes da integração de uma solução IMDG numa dada aplicação do mundo real. Isto justifica a pertinência da realização de um estudo e análise de todas as alternativas *open source* existentes. Tanto quanto nos foi possível descobrir, no mercado atual existem cerca de catorze produtos IMDG pertencentes a diferentes companhias. Deste conjunto de possiblidades, aproximadamente metade são de código livre ou têm uma versão gratuita, o que nos facilitou um pouco o trabalho de estudo e análise de cada uma dessas ferramentas. Para que tal processo pudesse ser levado a cabo com sucesso, e de acordo com os requisitos apresentados pela empresa de telecomunicações envolvida, definimos que devíamos:

- estudar a estrutura e funcionamento típico de uma solução IMDG;
- explorar as soluções IMDG existentes no mercado e fazer o levantamento das suas características segundo a lista de requisitos funcionais apresentados;
- definir um pequeno conjunto de casos de estudo, bem como modelos de dados representativos de aplicações típicas de telecomunicações;
- confirmar as capacidades de duas IMDG, previamente selecionadas, através da implementação de duas aplicações com os casos de estudo, modelo de dados e funcionalidades definidos anteriormente.

De referir, por fim, que neste trabalho não se abordará a avaliação de soluções IMDG implementadas num cluster de várias máquinas, apesar destes sistemas serem projetados para processarem um grande volume de pedidos sobre dados distribuídos por um cluster.

1.3 Estrutura da Dissertação

Para além do presente capítulo, esta dissertação encontra-se estruturada em mais quatro capítulos, cada um deles elaborado de forma a contextualizar o leitor com os diversos assuntos abordados ao longo do trabalho desta dissertação. Esses capítulos são:

- **Capítulo 2** – no qual se faz uma apresentação detalhada dos sistemas IMDG, suas características, funcionalidades e conceitos chave, e ainda a caracterização da sua arquitetura típica bem como as diversas topologias de sistemas nas quais normalmente operam.
- **Capitulo 3** – em que se faz um estudo e a análise das várias soluções IMDG existentes no mercado, realizando-se, inicialmente, uma análise mais genérica de todos os produtos, e aprofundando-se, depois, o estudo abordando as soluções *open source*.
- **Capítulo 4** – neste capítulo faz-se a apresentação e a caracterização de três casos de estudo representativos de operações que ocorrem usualmente num sistema de telecomunicações, descreve-se a implementação das duas aplicações, sua arquitetura, ambiente de desenvolvimento e integração das funcionalidades chave.
- **Capítulo 5** – este é o último capítulo desta dissertação; nele faz-se uma apreciação crítica do trabalho desenvolvido, com referência às vantagens e desvantagens para

cada uma das soluções IMDG utilizadas nas aplicações, e apresenta-se algumas linhas de orientação para trabalho futuro.

Capítulo 2

Sistemas IMDG

2.1 Características Base

A utilização de sistemas IMDG (Red Hat, 2015) (GridGain, 2013) é hoje uma das alternativas tecnológicas mais atrativas para aplicações que têm que manipular grandes volume de dados, com altos índice de variação e de emergência. Usualmente, as soluções baseadas em IMDG têm capacidade para suportar um grande número de eventos de atualização de dados por segundo, apresentando características que as tornam muito escaláveis e capazes de suportar volumes de dados de grande dimensão. Atualmente, já não é difícil encontrar aplicações baseadas em IMDG, utilizando produtos muito conhecidos no domínio – e.g., *Oracle Coherence*, *Hazelcast*, *VMware Gemfire*, *IBM eXtreme Scale* ou *JBoss Infinispan* – e instaladas em áreas de trabalho como as finanças, a banca, os seguros, ou as telecomunicações. Porém, para que uma solução IMDG possa utilizar de forma efetiva a sua memória principal, como área de trabalho prioritária no armazenamento e manipulação das estruturas de dados envolvidas, estas soluções têm que adotar uma estratégia de desenvolvimento e de implementação tal que lhes permita trabalhar para além das suas capacidades de armazenamento. Como referido frequentemente na literatura, isso é conseguido (e garantido) através da utilização de uma arquitetura altamente distribuída, bastante elástica e altamente replicável. Como sabemos, a adoção de um sistema baseado numa solução IMDG traz vantagens significativas quando comparadas com outras alternativas tecnológicas ditas convencionais, nomeadamente:

- o excelente suporte que este tipo de tecnologia providencia na implementação de soluções para sistemas de suporte à tomada de decisão, providenciando uma melhor

qualidade de serviço, uma maior produtividade dos sistemas ou uma maior elasticidade nos serviços e estruturas implementadas;

- a grelha de dados do sistema (*data grid*) é uma estrutura muito elástica, com capacidade para "escalar" facilmente, sendo também capaz de acolher atualizações de uma forma muito simplificada e facilitar o trabalho dos programadores de aplicações ao providenciar uma estrutura de dados do tipo chave/valor ;
- o desempenho muito elevado que este tipo de solução usualmente apresenta, dada a sua capacidade de ser capaz de ler e escrever estruturas de dados de uma forma bastante mais rápida que um sistema que assente em sistemas convencionais de discos.

A utilização de sistemas IMDG, em contrapartida com os sistemas baseados em discos, permite aumentar a velocidade dos sistemas computacionais de forma considerável, em aplicações que sejam suportadas por sistemas de gestão de bases de dados. Para além disso, providenciam uma gama completa de capacidades que permitem aos sistemas apresentarem disponibilidade contínua em termos de dados e capacidades muito interessantes em termos de persistência para casos em que seja necessário proceder à recuperação atempada e automatizada de uma qualquer situação que imponha, por qualquer motivo, a interrupção dos serviços do sistema. Podendo ser utilizadas para melhorar o desempenho e escalabilidade de aplicações já existentes ou para criar uma aplicação de grande escala de raiz, as soluções IMDG usam a memória principal para acessos rápidos, dados distribuídos para escalar e trabalhar com outros repositórios de dados ou cópias em nodos remotos para garantir resiliência e persistência.

2.2 Principais Funcionalidades

No essencial, os sistemas IMDG apresentam funcionalidades que os tornam únicos e bastante atrativos, o que pode justificar a sua grande proliferação em domínios de aplicação que envolvam um grande número de utilizadores e a realização de imensas operações num espaço período de tempo. De seguida apresentamos e descrevemos um pouco mais as características base de uma solução IMDG:

- **Armazenamento *key value*** – as *IMDG* são bases de dados *NoSQL* que oferecem uma forma simples e flexível de armazenamento para uma grande variedade de dados sem as limitações de um modelo de dados fixo.

- **Transações ACID distribuídas** – o suporte transacional é assegurado utilizando protocolo *two phase commit* (2PC) para garantir consistência no cluster, este protocolo foi otimizado em alguns casos para minimizar comunicações na rede e concorrência *lock-free*. As IMDG suportam todas as propriedades ACID esperadas, incluindo o suporte para níveis de concorrência otimistas e pessimistas e níveis de isolação *Read-Commiled*, *Repeatable-Read* e *Serializable*. As transações distribuídas abrangem dados em nodos locais e remotos, sendo transações mais leves, por vezes mais convenientes, criadas pelos utilizadores assim como integração com JTA.
- ***Querying*** – É possível encontrar objetos sem ser necessário saber a sua chave, podemos procurar através de correspondência em meta-dados ou por uma procura *full-text*. É também possível executar queries SQL através de mecanismos API ou por interface de leitura JDBC. Quase toda a sintaxe SQL é aceite, incluindo todos os tipos de *joins* e um vasto conjunto de funções SQL. A capacidade de fazer *joins* sobre objetos de classes diferentes em *cache*s diferentes torna as IMDG numa ferramenta potente que, mantendo todos os índices em memória faz com que as queries tenham baixa latência.
- **Persistência de dados** – as IMDG são muitas vezes usadas em conjunto com fontes de dados externas, como bases de dados em disco ou sistemas de ficheiros. Depois de configuradas, carregar ou alterar dados da base de dados são operações que são feitas automaticamente pelo sistema. As bases de dados em disco podem ser inseridas nas transações da IMDG, o que faz com que um *update* na base de dados faça parte da mesma transação que um *update* na IMDG e quando um falha, falha toda a transação.
- **Tolerância a falhas e resiliência de dados** – Se o sistema for construído corretamente, nenhum dado será perdido em caso de falha de um ou de vários nodos. Num cenário com particionamento é possível configurar quantos *backups* forem necessários, na eventualidade de um nodo primário falhar, automaticamente é promovido um dos nodos de *backup* a primário, continuando o acesso aos dados sem interrupções. Além disso, é também possível guardar de várias formas os dados numa camada subjacente de persistência, garantindo assim que os dados não se perdem.
- **Replicação de *datacenters*** – Num cenário que envolva vários *datacenteres* é importante garantir que, se um falhar, existe um outro que continua o seu trabalho e que mantém os seus dados. Quando esta funcionalidade está ativa, a IMDG certifica-se que cada *datacenter* faz um backup num outro *datacenter* definido pelo programador,

mantendo-o atualizado sempre que necessário. Esta replicação tem dois modos, quando ambos os *datacenters* estão ativos e operacionais servindo de *backup* um ao outro e quando só um dos *datacenters* está ativo existindo um outro que servirá de *backup*. A replicação pode ser transacional ou eventualmente-consistente. Na replicação transacional, a transação só vai terminar quando todos os dados de um *datacenter* forem replicados para um outro. Se ocorrer uma falha toda a transação tem de ser revertida nos dois *datacenters*. No caso de ser eventualmente-consistente, a transação termina antes da replicação terminar. Normalmente, um *buffer* é concorrentemente preenchido com os dados de um *datacenter* e é enviado para um outro, quando o *buffer* estiver preenchido ou depois de dado período de tempo especificado. Este método é normalmente mais rápido, mas introduz um atraso entre os *updates* num *datacenter* e a replicação num segundo datacenter, ou seja, as alterações não serão imediatamente refletidas nos dois locais. Se um dos *datacenters* ficar *offline*, o outro irá imediatamente ficar responsável pelo seu trabalho. Caso o *datacenter* volte a ficar *online* receberá os *updates* feitos pelo seu substituto.

Em situações em que os casos de uso exijam a integração com uma camada de persistência de dados é importante definir a forma como ambas as partes deverão interagir. O modo como são feitas as leituras e as escritas para a camada de persistência pode ser dividido pelas seguintes categorias:

- **Read-Through e Write-Through** – Nesta modalidade os dados serão lidos da base de dados se não estiverem disponíveis em *cache* e persistidos sempre que houver um update na *cache*. Estas operações vão fazer parte de transações da *cache* e serão finalizados ou revertidos como um todo.
- **Refresh-Ahead** – Esta categoria faz com que os dados sejam carregados automaticamente na da camada de persistência quando expiram em memória, ou seja, quando termina o seu *time-to-live*. Para prevenir que os dados sejam recarregados cada vez que expiram, o *refresh-ahead* garante que as entradas são automaticamente carregadas para a *cache*, cada vez que o tempo de refrescamento está perto de expirar.
- ***Caching* Write-Behind** – Se cada *update* feito na *cache* envolve o correspondente acesso à camada de persistência, isto pode levar ao aumento do tempo que custa cada *update*. Isto pode agravar-se se a aplicação tiver uma alta taxa de *updates*, o que pode

levar a uma grande pressão sobre a base de dados. Para lidar com estes casos, existe a opção *Write-Behind* que permite executar *updates* à base de dados de forma assíncrona. A ideia é acumular operações feitas em memória e, depois, assincronamente atualizar a base de dados, fazendo os *updates* todos duma vez. Para além das melhorias no desempenho, uma consequência das escritas em *cache* ficarem mais rápidas, este modo também escala melhor, assumindo que é possível atrasar as escritas para a base de dados. Por fim, a IMDG pode continuar a funcionar mesmo que a camada de persistência falhe, continuando simplesmente a acumular os *updates* feitos.

Descritas as principais características de uma IMDG (GridGain, 2013), cabe agora fazer a apresentação de algumas das suas características técnicas gerais (Red Hat, 2015). Estas estão divididas pelos componentes lógicos constituintes das IMDGs e estão, usualmente, organizadas da seguinte maneira:

- **Base** – As características mais importantes e comuns a todas as IMDG.
    - Várias opções de configuração de topologias.
    - Acesso simples aos dados em modo biblioteca, do tipo *map* do Java.
    - Processos de *Eviction e Expiration*, para gestão de objetos em *cache*.
    - Integração com Context Depedency Injection (CDI).
    - API para execução de tarefas assincronamente.
    - Opções de configuração assíncrona de nodos.
    - *Caching* de nível 1.
    - Eventos, notificações e listeners (síncronos e assíncronos).
- **Processamento e Distribuição** – Formas e métodos usados pelas IMDG, desde o nível da rede até ao nível do processamento de dados, tendo em conta aspetos de distribuição da computação e da execução de queries sobre dados.
    - Modelo de programação e framework Map-Reduce, incluindo interfaces de *mapper, collector, reducer e collator.*
    - Distribuição de tarefas para serem executadas em um ou mais nodos, em paralelo.
    - Processamento na grelha e em paralelo.
    - *Hashing* consistente e consciência topológica durante o particionamento de dados.
    - Sistema distribuído com aparência local para o programador.
    - Elasticidade para adicionar ou remover nodos quando necessário.

  - Afinidade de dados via anotações, afinidade por nodos e afinidade por chave.
  - Balanceamento de carga para ter partições altamente disponíveis em múltiplos nodos através de particionamento dinâmico.
  - Não bloqueio aquando duma transação para garantir desempenho e funcionamento contínuo quando é adicionado ou removido um nodo.
  - Descoberta automática de nodos e tolerância a falhas.
  - Protocolo de comunicação entre os clusters de alto desempenho que suporta tanto UDP/IP como TCP/IP.
  - Transferência de mensagens por *Unicast* e *Multicast.*
- **Concorrência** – Mecanismos disponíveis para lidar com situações de concorrência.
  - Integração com transações *Extended Architecture* (XA) através de Java Transactional API (JTA), recuperação de transações e sincronização JTA.
  - Mecanismos de Locking configuráveis, otimista, pessimista, Locking explícito e detecção de *deadlocks*.
  - Níveis de isolação na base de dados.
  - Agrupamento de invocações.
  - Operações com versões.
- **Clientes remotos** – Clientes com capacidade de comunicarem com o cluster.
  - HTTP/REST.
  - Mem*cache*d.
  - Java.
  - C++, C# e .NET .
- **Segurança** – Mecanismos de segurança disponíveis verticais às camadas lógicas, orientados para garantir a segurança desde o nível da rede até às condições de acesso a dados em *cache*.
  - Comunicações seguras entre nodos servidores utilizando algoritmos de criptografia suportados pela *Java Cryptography Architecture* (JCA).
  - Autenticação entre cliente-servidor e nodo-a-nodo através de certificados *SSL Client* (SSL).
  - Autorização e controlo de acesso *role-based* às *cache*s.
- **Armazenamento** – Opções de persistência para uma camada inferior de base de dados disponíveis.

  - *SingleFileCacheStore* para bom desempenho nas leituras e nas escritas onde os índices das chaves são armazenados em memória.
  - *LevelDB cache store* para bom desempenho em grandes quantidades de dados persistidos localmente, em que os índices das chaves não são armazenados em memória.
  - JPA *cache* store para persistir para bases de dados mantendo o esquema de dados.
  - Java Database Connectivity (JDBC) *cache loader/store*.
  - *Read-though*, *write-through* (síncrono) e *write-behind* (assíncrono).
- **Gestão e Monitorização** – Ferramentas de monitorização e gestão.
  - Interface de linha de comandos.
  - Opções de gestão e monitorização via *API Java Management* (JMX).
  - Ferramentas de monitorização com interface web.

2.3 Arquiteturas

Para reforçar um pouco mais o conhecimento sobre IMDG é necessário compreender a sua estrutura e arquitetura base. A estrutura normal de uma grelha de dados de uma IMDG pode ser observada na **Erro! Fonte de referência não encontrada.**, na qual estão apresentados os seus componentes e funções principais (Kirby et al., 2009), bem como as funcionalidades associadas se relacionam entre si numa IMDG (Figura 2).

Tendo como referência a **Erro! Fonte de referência não encontrada.**, de seguida são descritos os componentes que constituem uma grelha de dados típica. De referir:

- **Map –** É uma interface que armazena os pares *key-value* em que não podem haver chaves repetidas. Além disso é considerada uma estrutura associativa porque associa uma chave a um objeto, em que a chave é o meio primário de aceder a dados na grelha de dados. O objeto é normalmente um objeto *Java*.
- **Chave (key) –** É uma instancia de um objeto *Java* que identifica um único valor na *cache*. As chaves não podem mudar e têm que implementar os métodos *equals()* e *hashCode()*.
- **Valor (value) –** É um objeto Java que contém os dados em *cache* e que é identificado pela chave, podendo ser de qualquer tipo.

- **Map set –** É um conjunto de *maps* cujas entradas estão logicamente relacionadas e que partilham o mesmo esquema de particionamento. Os elementos chave destes esquemas são o número de partições e o número de réplicas síncronas e assíncronas.
- **Grelha de dados (Grid) –** É constituída pelo conjunto de *map sets*.
- **Partições –** Uma ação de particionar visa partir blocos de dados em pedaços de menor tamanho, permitindo que a IMDG consiga armazenar mais dados que uma *JVM*. É esta característica que permite a uma IMDG uma escalabilidade linear. O particionamento acontece ao nível dos *map sets* e o número de partições é especificado quando o *map set* é definido. Os dados dos *maps* são divididos pelas N partições, usando o módulo de N no *hashcode* das chaves. A definição do número de partições é importante para a escalabilidade da infraestrutura. Na **Erro! Fonte de referência não encontrada.** podemos ver uma grelha de dados com dois *map sets*, um a vermelho e outro a amarelo, que têm, respectivamente, três e duas partições.
- **Fragmento –** As partições são elementos lógicos, mas os dados de uma partição são fisicamente guardados em fragmentos. Cada partição tem sempre um fragmento primário. Se forem definidas réplicas para um dado *map set*, cada partição terá também esse número de réplicas dos fragmentos que asseguram a disponibilidade da IMDG.

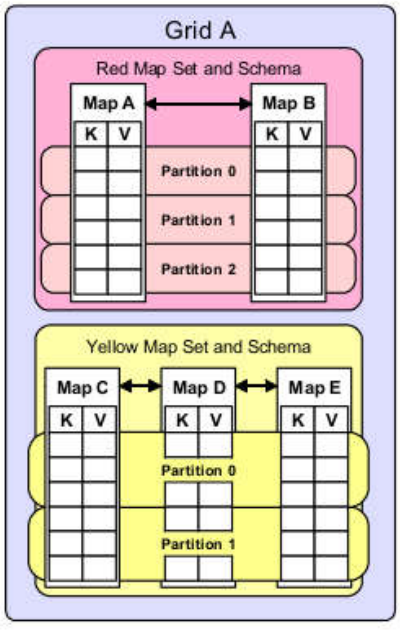


Figura 1: Arquitetura base IMDG – figura retirada de Kirby et al. (2009)

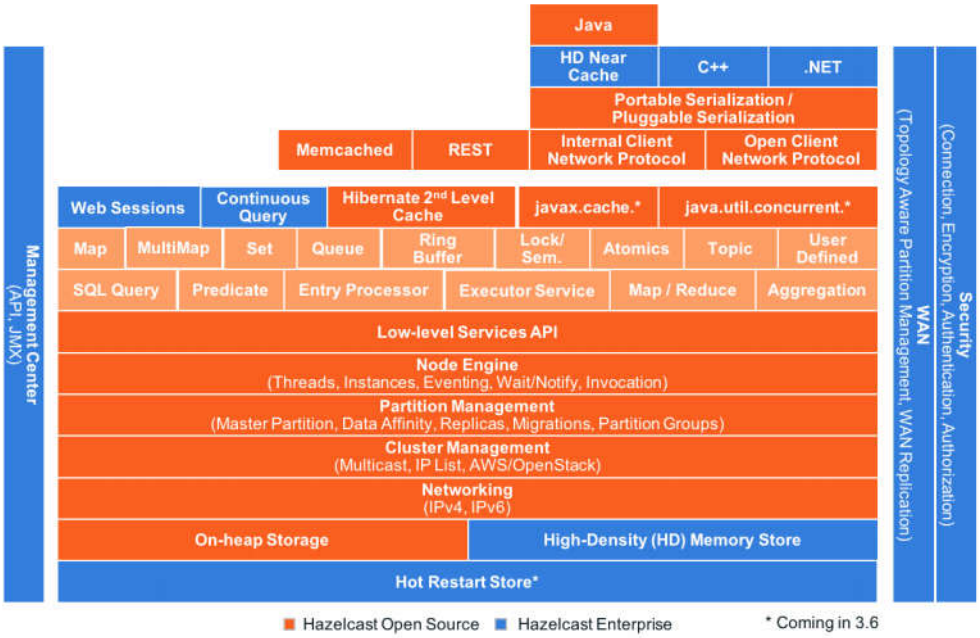


Figura 2: Arquitetura IMDG – figura retirada de Hazelcast (2015b)

A arquitetura funcional apresentada na Figura 2 é próxima à estrutura comum das IMDG. Porém, podem existir algumas variações entre as várias abordagens disponíveis no mercado. Observando a Figura 2, fazendo uma análise de baixo para cima nos diversos componentes apresentados, verificamos que o armazenamento *on-heap* refere-se aos objetos presentes no espaço da *heap* do Java (sujeito a *garbage collector*). Esta *heap* é a memória que o Java pode reservar e usar para fazer a alocação dinâmica de memória, que por omissão é de 128 MB (Hazelcast, 2015b). Acima da camada de dados estão as funcionalidades internas de gestão e de funcionamento, como protocolos de comunicações por rede, de descoberta e de gestão de membros do cluster, opções de particionamento dos dados e a configuração dos nodos do cluster. Toda estas funcionalidades de gestão podem e devem ser configuradas pelo programador, através dum ficheiro XML ou recorrendo a uma linguagem de programação específica. Continuando a subir nos níveis da arquitetura, encontramos as ferramentas que o programador tem à sua disposição para construir a sua aplicação, tais como os mecanismos de computação distribuída, como o *Executor Service*, o *Entry Processor* e alguns serviços definidos pelo utilizador (algumas IMDG disponibilizam implementações próprias deste tipo). Depois aparecem as diversas formas disponíveis para fazer *queries* sobre os dados, neste caso *queries SQL*, *MapReduce*, *Aggregators*, *queries* continuas ou

simplesmente iterando sobre o map distribuído. De seguida, estão integradas as várias estruturas de dados distribuídas disponíveis, no caso do Hazelcast, *Map, MultiMap, Set, List, Queues, Topics, Atomics, Locks, Semaphores* e *RingBuffer*. Por último, encontramos alguns sistemas onde as IMDG podem ser utilizadas como 'fornecedores' de *cache* distribuída – e.g. Hibernate Second Level *Cache*, Tomcat Clustered Web Sessions ou Jetty Clustered Web Sessions – e os clientes para os quais estão disponíveis meios de comunicação com o cluster que, mais uma vez, variam entre as diversas abordagens IMDG.

2.4 Topologias

Normalmente, as IMDG suportam dois modos de operação: o modo embebido (**Erro! Fonte de referência não encontrada.**) e o modo cliente/servidor (**Erro! Fonte de referência não encontrada.**). No primeiro, a JVM que contém o código da aplicação é a mesma que corre a instância da IMDG, juntando-se cluster como seu membro. No segundo modo, as aplicações e os nodos do cluster estão em JVM diferentes, comunicando os nodos das aplicações com o cluster num modelo típico cliente-servidor através de uma API cliente escolhida por quem está a desenvolver a solução.

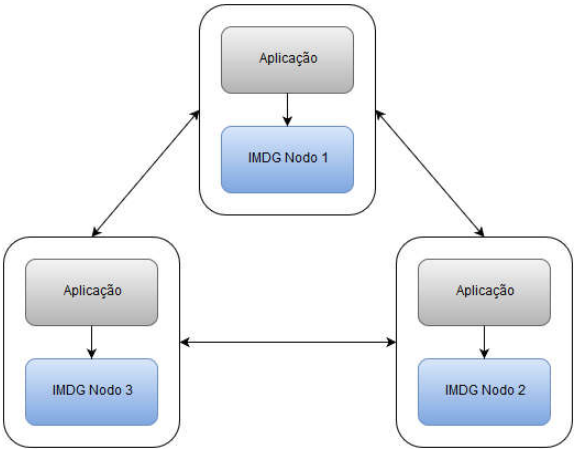


Figura 3: Topologia em modo embebido

A topologia em modo embebido (**Erro! Fonte de referência não encontrada.**) é do tipo *peer-to-peer*, muito simples, na qual a aplicação e as instâncias do cluster estão localizadas na mesma máquina, não sendo necessário fazer *deploy*, gerir ou manter servidores extra. Além disso, tem

muitas vantagens em situações nas quais uma aplicação realize muitas operações locais de leitura e de escrita, bem como em termos de desempenho, tráfego de rede e de consumo de largura de banda, uma vez que os dados estão na mesma JVM.

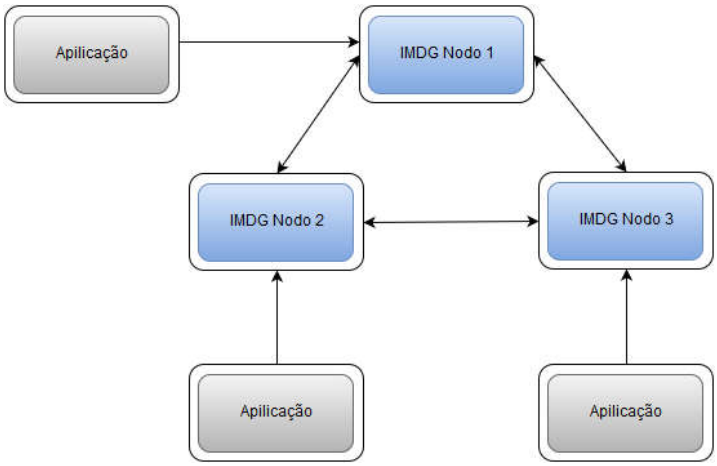


Figura 4: Topologia em modo cliente-servidor

Os sistema em topologia cliente-servidor isolam o código das aplicações dos eventos a nível do cluster, dando assim mais flexibilidade em termos de mecanismos do próprio cluster (Hazelcast, 2016). A adoção deste tipo de topologia traz bastante vantagens, nomeadamente, em termos de:

- **Ciclos de vida diferentes entre nodos servidores e clientes** - é normal estas partes terem ciclos de vida diferentes; no modo embebido estes dois tipos de nodos são, obrigatoriamente, inicializados e encerrados ao mesmo tempo, o que pode aumentar a complexidade operacional.
- **Isolamento de recursos** – assim, os nodos servidores não têm de competir por recursos com a aplicação como CPU, memória ou I/O, o que torna o desempenho dos servidores mais previsível e confiável.
- **Resolução de problemas** – a detecção de problemas e a sua resolução é mais fácil, uma vez que o consumo dos recursos está separada.
- **Partilha da infraestrutura** – quando a IMDG está numa infraestrutura partilhada na qual existem várias aplicações, especialmente em situações em que estas estão sobre o controlo de diferentes grupos de trabalho mas utilizam o mesmo cluster de servidores, é

necessário fazer a sua separação para que todas consigam utilizar a camada de dados em memória.

- **Escalabilidade** – o sistema é mais flexível, já que caso haja a necessidade de acrescentar mais nodos servidores o processo não é complicado. A escalabilidade de clientes e servidores pode ser tratada separadamente como se fossem dois assuntos diferentes.
- **Latência muito baixa** – para que obtenham melhores resultados é possível configurar os clientes para usarem *Near Cache*. Desta forma, garantimos que os dados que estejam a ser constantemente acedidos se mantenham na memória próxima à JVM da aplicação.

Capítulo 3

Estudo e Análise de Soluções para *In Memory Data Grid*

Como foi referido anteriormente, as IMDGs ocupam uma posição importante em soluções aplicacionais de grandes empresas em diversas áreas de negócio. É possível encontrar casos da sua adoção em áreas como finanças, retalho, banca, seguros ou telecomunicações, com soluções IMDG pertencentes a gigantes da tecnologia como Oracle, IBM ou Terracotta. Neste trabalho de dissertação, no total, foram estudadas catorze soluções, das quais, aproximadamente metade foram criadas por empresas dedicadas a este tipo de produtos, enquanto que as restantes disponibilizam versões *open source*. Neste capítulo iremos apresentar e analisar as soluções IMDG mais relevantes atualmente existentes no mercado, analisando-as e comparando-as com base em vários critérios, como, por exemplo, a escalabilidade, a replicação, ou a integração. Assim, fez-se o levantamento dos principais termos técnicos e comerciais que caracterizam tais soluções IMDG e, em seguida, procedeu-se à filtragem das soluções *open source* relevantes, comparando-as com uma lista de funcionalidades previamente definidas, descrevendo-se genericamente o seu comportamento.

3.1 Produtos Analisados

No processo de estudo realizado sobre as IMDGs, estabelecidas atualmente em várias áreas do mercado, foram encontrados catorze sistemas. Cerca de metade pertence a empresas recentes,

que se dedicam exclusivamente a este tipo de produtos. Os produtos encontrados e estudados foram os seguintes:

1. **Oracle Coherence.**
   www.oracle.com/technetwork/middleware/coherence/overview/index.html
2. **Parallel Unverse Galaxy.**
   www.paralleluniverse.co/galaxy/
3. **ScaleOut StateServer.**
   www.scaleoutsoftware.com
4. **Pivotal Gemfire.**
   www.vmware.com/products/vfabric-gemfire/overview
5. **Ehcache.**
   www.ehcache.org
6. **N*cache*.**
   www.alachisoft.com/n*cache*/
7. **Gigaspace XAP.**
   www.gigaspaces.com/xap-in-memory-computing-event-processing/Meet-XAP
8. **TIBCO Active Spaces.**
   www.tibco.com/products/automation/in-memory-computing/in-memory-data-grid/activespaces-enterprise-edition
9. **GridGain InMemory Data Fabric.**
   www.gridgain.com/products/in-memory-data-fabric/
10. **IBM WebSphere eXtreme Scale.**
    www-03.ibm.com/software/products/pt/websphere-extreme-scale
11. **JBoss Infinispan.**
    infinispan.org
12. **Hazelcast.**
    hazelcast.com
13. **Terracotta Enterprise Suite.**
    terracotta.org/products/enterprise-suite
14. **Apache Ignite.**
    ignite.incubator.apache.org

3.2 Características Relevantes

Dado o grande número de produtos existentes no mercado, inicialmente, fez-se um estudo abordando os seus vários aspetos comerciais e algumas das funcionalidades consideradas mais importantes para este tipo de produto. Assim, foi recolhida informação sobre os seus fabricantes, sítios Web e licenciamento, que são, nesta fase, dados bastante relevantes para esta dissertação, uma vez que pretendemos encontrar uma solução IMDG *open source* que seja uma verdadeira alternativa às soluções comerciais mais utilizadas atualmente pelas aplicações neste segmento. Além disso, foram recolhidas e estudadas outras funcionalidades, como sejam os requisitos de instalação ou a capacidade de suportar *cross site replication*, *triggers* ou a integração com *hibernate*, JTA e JPA. Todas estas características são fundamentais para justificar um eventual processo de substituição de qualquer solução IMDG em exploração. Na Tabela 1 e na Tabela 2, podemos ver um primeiro resumo das várias características das catorze soluções IMDG estudadas. As soluções IMDG estão apresentadas nas colunas, enquanto que as suas características nas linhas das tabelas.

Tabela 1: IMDGs Estudadas

	Coherence	**WebSphere eXtreme Scale**	**Ehcache**	**GigaSpace XAP**	**InMemory Data Fabric**	**Infinispan**	**Hazelcast**
Fabricante	Oracle	IBM	Terracotta	GigaSpace Technologies	GridGain	Red Hat	Hazekcast
Site	Coherence	Extreme Scale	Ehcache	GigaSpace XAP	Data Fabric	Infinispan	Hazelcast
Requisitos de instalação	100 MB disco; 1 GB de RAM; Java 7	Java EE ou Java SE	Java 5 ou 6; SLF4J	Java ou .NET	Java 7	Java 7	Java
Licenciamento	Pago	Pago	Pago e *Open source*	Pago	Pago e *Open source*	*Open source*	Pago e *Open source*
Multi Grid	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Integração com Hibernate	Sim	Sim	Sim	Sim	Sim	Sim	Sim

Triggers	Sim	Sim	Sim	Sim	Sim	Sim	Sim
JTA	Sim	Sim	Sim	Sim	Sim	Sim	Sim
JPA	Sim	Sim	Sim	Sim	Não	Sim	Sim

Tabela 2: IMDGs Estudadas

	Galaxy	**StateServer**	**Gemfire**	**N*Cache***	**Active Spaces**	**Terracotta Enterprise Suite**	**Apache Ignite**
Fabricante	Parallel Universe	ScaleOut	Pivotal	Alachisoft	TIBCO	Terracotta	Apache
Site	Galaxy	StateServer	Pivotal Gemfire	N*Cache*	Active Spaces	BigMemoryMax	Ignite
Requisitos de instalação	Java 7	Java EE ou Java SE	Oracle Java SE 7, Update 72 e seguintes	.NET	1GB em disco. 100MB de RAM. Um web browser. Java 6. Compilador C	Java 7; SLF4J	Java
Licenciamento	*Open source*	Pago	Pago	Pago e *Open source*	Pago	Pago	*Open source*
Multi Grid	Não	Sim	Sim	Sim	Sim	Sim	Sim
Integração com Hibernate	Não	Sim	Sim	Sim	Não	Sim	Não
Triggers	Sim	Sim	Sim	Sim	Sim	Sim	Sim
JTA	Não	Não	Sim	Não	Não	Sim	Sim
JPA	Não	Não	Não	Não	Não	Não	Não

3.3 Os Melhores Candidatos

Da análise dos vários elementos apresentados na Tabela 1 e na Tabela 2, podemos concluir que existem cinco sistemas IMDG *open source* com capacidade para satisfazer às funções pretendidas. Esses sistemas são: *JBoss Infinispan*, *Hazelcast, Apache Ignite, Parallel Unverse Galaxy* e Ehcache.

De referir que todas estas ferramentas foram já apresentadas e as suas capacidades descritas detalhadamente na Tabela 3. De seguida, é apresentada uma análise mais detalhada para cada uma das soluções IMDG referidas como melhores candidatos, incluindo uma breve descrição dos seus pontos mais relevantes bem como do seu comportamento.

3.3.1 Infinispan

O Infinispan é um IMDG *open source*, que disponibiliza uma interface *Cache* compatível com JSR-107 (que permite, por sua vez, usar *java.util.Map*) na qual é possível guardar objetos. Este pode correr em modo local ou em modo distribuído num cluster, garantindo resiliência e escalabilidade em modo distribuído caso ocorram falhas de servidores. Algumas das características deste produto são:

- **Escalabilidade** - Como os dados estão igualmente distribuídos não há limite para o tamanho do *cluster*. Porém podem surgir alguns problemas relacionados com as comunicações em grupo que ocorrem quando é necessária a descoberta de novos nodos. Todos os acessos a dados são feitos por comunicação *peer-to-peer,* nos quais os nodos falam diretamente entre si. No caso de ser preciso acrescentar ou retirar nodos, não é necessário que o sistema seja desligado. Pode-se simplesmente adicionar ou remover máquinas sem que para isso seja necessário parar o sistema.
- **Alta disponibilidade** - Num ambiente distribuído, e assumindo que cada item de dados tem pelo menos uma cópia, existem grandes quantidades de dados duplicados que podemos aceder de qualquer parte do cluster. Se um servidor falhar, a IMDG cria cópias novas dos dados perdidos e coloca-as noutro servidor.
- **Distribuição de dados** - O Infinispan utiliza um algoritmo de *hash* consistente para determinar onde as chaves devem ser posicionadas no cluster. A utilização de *hashing* consistente permite fazer uma localização 'barata', rápida e determinística das chaves sem necessidade de meta-dados ou tráfego de rede extra.
- **Persistência de dados** - O produto disponibiliza uma interface *CacheStore* e várias implementações de alto desempenho, incluindo *cache stores* JDBC, *cache stores* baseadas em sistemas de ficheiros, *cache stores* Amazon S3, etc. Esta camada de persistência pode ser usada para *warm starts*, para assegurar que os dados da grelha sobrevivem a um

restart completo da *grid*, ou até para colocar em disco caso o limite da memória seja excedido.

- **Ligação com outras linguagens (PHP, Python, Ruby, C, etc.)** - Este produto oferece suporte tanto para o popular protocolo *memcached* - com clientes em quase todas as linguagens de programação populares -, como para um protocolo otimizado especifico do Infinispan chamado *Hot Rod*. Isto significa que qualquer aplicação que queira tirar proveito desta IMDG, pode-o fazer sem necessidade de grandes alterações.
- **Gestão** - Quando se fala em centenas de servidores, a gestão passa a ser uma necessidade. São disponibilizadas várias formas de fazer a gestão dos membros do cluster, desde o básico JMX até ao *Hawt.io,* passando pelo RHQ ou até criando um *plugin* que se "alimente" dos dados JMX que o Infinispan gera. A ferramenta recomendada é o RHQ, um produto de gestão desenvolvido pela *JBoss* que graças à capacidade de auto aprendizagem do agente RHQ, monitorizar o *Cache Manager* e as instâncias de *cache* torna-se simples.
- **Suporte para grids de computação** - O Infinispan permite executar tarefas do tipo *Runnable* no cluster. Isto faz com que seja possível injetar processamento complexo para o cluster onde os dados estão localizados e recuperar o resultado usando a interface *Future*. Este paradigma, do estilo *map/reduce,* é comum em aplicações nas quais grandes quantidades de dados são necessários para obter pequenos resultados.

3.3.2 Galaxy

O Galaxy diferencia-se de outras soluções IMDG na maneira como distribui os dados pelos nodos do cluster. Este produto, em vez de distribuir os dados pelas chaves com um esquema de *hashing* consistente, move dinamicamente objetos entre os nodos, consoante a necessidade das aplicações envolvidas. Para gerir este método de distribuição de dados, o Galaxy utiliza um protocolo de *cache-coherence* similar ao encontrado nos CPU. Esta característica torna esta solução IMDG bastante apropriada para aplicações com padrões de acesso a dados previsíveis, como por exemplo, aplicações nas quais os acessos a dados se comportam de acordo com alguma métrica de proximidade, ou seja, dados que estão mais "próximos" têm mais probabilidade de serem acedidos do que dados que estão mais "longe". O Galaxy não é uma *key-value store*, embora possa ser usado para a sua construção. O seu propósito é ser usado como uma plataforma para construção de estruturas de dados distribuídas. Além disso, pode usar o ZooKeeper ou JGroups

para gestão do cluster e a BerkeleyDB, ou qualquer base de dados relacional para persistência de dados.

3.3.3 Ehcache

O Ehcache disponibiliza uma *cache* padronizada, *open source*, utilizada para aumentar o desempenho de um sistema, descarregar as bases de dados e simplificar a sua escalabilidade. Este sistema escala desde projetos menores, com um ou mais nodos, até configurações com várias máquinas possuindo *caches* com grandes capacidades. Algumas das características do Ehcache são:

- **Velocidade e peso** - O seu sistema de threads foi desenhado para sistemas de grande porte e muito concorrentes. Os resultados relativos à velocidade de processamento do Ehcache têm-se mantido consistentes entre as suas várias versões. Além disso, também tem mantido a vantagem de ser fácil de utilizar e de requerer uma configuração inicial bastante simples, sendo possível colocá-lo em funcionamento em breves minutos. Outro dos objetivos desta solução IMDG é o de manter as aplicações leves, minimizando as suas dependências. A única dependência para o seu uso mais básico é o SLF4J (*Simple Logging Facade for Java*).
- **Escalabilidade** - O Ehcache foi desenhado para operar com grandes quantidades de dados, existindo exemplos de casos de uso que utilizam espaço de memória e em disco que atingem centenas de *gigabytes* e com centenas de *caches*. Superando problemas relacionados com a relação entre a segurança das *threads* e o desempenho, o E*cache* está agora afinado para cargas concorrentes em grandes servidores multi-CPU. Permite ainda ter vários *CacheManagers* por cada máquina virtual, o que permite à aplicação ter configurações muito diferentes.
- **Flexibilidade** – Esta solução IMDG permite fazer *caching* de objetos sem necessidade de os seriar, não podendo estes ser usados pelo *DisckStore* e na replicação. Se forem usados neste tipo de casos, então são descartados e é emitido um aviso (WARNING). Estas APIs são idênticas exceto pelos métodos de retorno do *Element*. Isto faz com seja fácil começar por fazer *caching* de objetos e mudar para *Serializable* quando for necessário. Podemos definir o tempo de vida (TTL) e o tempo inativo (TTI) como políticas de expiração, por *cache* ou por objecto. Além disso, temos também como políticas e expulsão da *cache* o *Least Recently Used* (LRU), *Less Frequently Used* (LFU) e o *First In First Out* (FIFO).

Configurações como TTL, TTI, capacidade máxima em memória ou em disco, podem ser mudadas enquanto a aplicação está a correr simplesmente alterando o objeto de configuração da *cache*.

- **Extensões** - O Ehcache disponibiliza vários plug-ins que podem ser adicionados e configurados no ficheiro ehcache.xml. Dentro destes, existem *listeners*, descoberta de nodos, replicadores, extensões de *cache*, *cache loaders*, ou *handlers* para exceções.
- **Gestão** – A forma mais simples de que o Ehcache proporciona para gestão é através de Java Management Extensions (JMX), podendo monitorizar e gerir os seguintes *MBeans*: *CacheManager*, *Cache*, *CacheConfiguration* e *CacheStatistics*.
- ***Caching*** – Este IMDG oferece suporte para *caching* distribuído flexível e extensível. E os mecanismos de *caching* que disponibiliza são os seguintes: *caching* distribuído com Terracotta, *Caching* replicado – via RMI, JGroups ou JMS, replicação síncrona, replicação assíncrona, ou cópias – Bootstrapping *cache*s.

Podemos aplicar o E*cache* como uma *cache* convencional ou como uma *cache* de segundo nível para o *Hibernate*. Também podemos integrá-lo com produtos externos como o *ColdFusion*, *Google App Engine* e *Spring*. Integra ainda com Java Transaction API (JTA), XA com *two-phase commit* (2PC). Adicionalmente, esta solução IMDG fornece *cache* por processo, que pode ser replicada por vários nodos. O Ehcache está posicionado no núcleo dos produtos comerciais da Terracotta*,* BigMemory Go e BigMemory Max, que são sistemas de *caching* e armazenamento de dados em memória. De referir que, uma nova versão do Ehcache – o Ehcache 3 – ainda em desenvolvimento, com *milestones* já disponíveis, trabalha nativamente com JSR107 suportando armazenamento *offheap*. O Ehcache é desenvolvido e mantido pela Terracotta, que o suporta ativamente como um projeto *open source* profissional, disponível sob uma licença *Apache 2*.

3.3.4 Hazelcast

O Hazelcast fornece a alternativa *open source* líder de mercado. Atualmente, consegue chegar a um vasto grupo de indústrias, incluindo telecomunicações, jogos, comércio eletrónico, alta-tecnologia e logística, entre outras. Também, disponibiliza aos programadores uma API familiar e fácil de usar, com funções standard como *Map, Set, List e Queue* da biblioteca *Java.util*, escalável para aplicações que envolvam centenas de nodos de processamento e de armazenamento. Ao

trabalharmos com este produto temos a sensação de estar a programar para uma máquina local, podendo as aplicações serem construídas rapidamente, com um código fácil de ler e de manter. Uma vez em funcionamento, o Hazelcast oferece características operacionais de software empresarial de larga escala e permite criar uma memória principal partilhada, bem como processamento distribuído que pode escalar de forma linear e elasticamente. Com estas características, as aplicações tiram bastante proveito do processamento distribuído. As aplicações em funcionamento são resilientes, sem um ponto de falha único, e têm a capacidade de recuperar de falhas de nodos sem perda de dados e sem quebra de disponibilidade. Utiliza uma topologia *peer-to-peer* que escala linearmente e suporta dois tipos de transações. De referir:

- LOCAL – Ao contrário do que o nome sugere, este tipo de transação é *two phase commit* (2PC). Porém, tem a fragilidade de, durante a fase do *commit*, se um dos membros falha, o sistema poder ficar num estado inconsistente.
- TWO_PHASE – É um pouco mais do que um clássico 2PC. Antes da fase de *commit*, copia o *commit log* para os outros membros. Assim, se um dos membros falha, outro membro pode concluir o *commit*.

Dependendo do tipo de transação definida, é possível influenciar a segurança do sistema quando um qualquer membro falha durante o *commit* da transação. Por omissão, o comportamento é *TWO_PHASE*. Além disso, expõe também uma interface XA entre o gestor de transações globais e o gestor de recursos locais. Adicionalmente, pode ainda ser integrado em J2EE através da utilização do adaptador de recursos do Hazelcast. Depois de configurado, pode participar em transações J2EE standard.

3.3.5 GridGain Data Fabric e Apache Ignite

Este sistema IMDG foi criado de raiz com a noção de escalabilidade horizontal e a capacidade de poder adicionar nodos, quando necessário, em tempo real. É uma ferramenta desenhada para escalar linearmente, em centenas de nodos, com ênfase na localidade de dados e *routing* por afinidade, para reduzir a tráfego de dados na rede. Além disso, suporta modos operacionais com dados locais, replicados e particionados, permitindo realizar queries, com sintaxe SQL, de forma livre sobre estes dados, podendo realizar *joins* distribuídos em SQL. Neste sistema a consistência de dados é garantida enquanto o cluster estiver online. Toda a informação entre os nodos se

mantém consistente em caso de falha de alguns dos nodos ou numa eventual mudança na topologia implementada. Algumas das suas principais características são:

- *Caching* distribuído em memória.
- Escalabilidade elástica.
- Transações distribuídas em memória.
- Grandes ganhos de desempenho.
- Integração com Hibernate.
- Armazenamento *offheap* em camadas.
- Queries SQL ANSI-99 distribuídas com suporte para *Joins*.

Vejamos, agora, com um pouco mais de detalhe, algumas das características dos serviços que este sistema IMDG disponibiliza:

- **Computação** - A computação distribuída é feita em modo paralelo para obter melhorias de desempenho, baixa latência e escalabilidade elástica. O sistema disponibiliza um conjunto de APIs que permitem ao utilizador distribuir computação por vários computadores do cluster. Algumas das suas características são:
    - Execução de encerramento distribuída.
    - Processamento por *MapReduce* e *ForkJoin*.
    - Executor de serviços em cluster.
    - Colocação de computação e de dados.
    - Balanceamento de carga.
    - Tolerância a falhas.
    - *Checkpointing* de estados de trabalho.
    - Agendamento de trabalhos.

- **Serviços na grelha** – O sistema permite o controlo completo sobre os serviços executados no cluster e determina o número de instâncias do serviço que devem ser lançadas em cada nodo do cluster, assegurando sempre a execução correta e tolerância a falhas e garantindo disponibilidade contínua de todos os serviços lançados em caso de falhas de nodos. Algumas das suas principais características são:
    - Disponibilidade contínua.

- Lançar qualquer número de instâncias de serviços no cluster automaticamente.
- Lançar *singletons*, incluindo *cluster-singleton*, *node-singleton* ou *key-affinity-singleton* de forma automática.
- Escolher o nodo de inicio onde os serviços serão lançados especificando isso na configuração.
- Cancelar qualquer serviço previamente lançado.
- Conseguir informação sobre a topologia dos serviços no cluster.
- Criar uma serviço de *proxy* para aceder a serviços remotamente lançados.

- **Streaming** - Permite fazer queries a uma *sliding window* de dados que estão a chegar à aplicação, que conseguem responder a questões sobre eventos acabados de ocorrer. Esta funcionalidade tem as seguintes características base:
  - Queries programáveis.
  - Fluxo de eventos personalizado.
  - Garantia de processamento *At-Least-Once*.
  - *Sliding Windows*.
  - Indexação de dados.
  - *Streamer* queries distribuídas.
  - Co-localização na IMDG.

- **Clustering avançado** – Neste tipo de serviço, o sistema disponibiliza uma das mais sofisticadas tecnologias de *clustering* em *Java Virtual Machine* (JVM). Os nodos do cluster descobrem-se automaticamente, o que ajuda a escalar o cluster quando necessário sem ter que se reiniciar todo o cluster. Os programadores podem beneficiar do suporte dado para *clouds* híbridas, que permite estabelecer a ligação entre *clouds* privadas e públicas, dando o melhor dos dois mundos. Isto permite ter funções como:
  - Gestão dinâmica da topologia.
  - Clouds públicas e privadas.
  - Zero Deployment.
  - Estado partilhado por nodo.
  - Métricas e monitorização do cluster em tempo real.

  - Descoberta automática em LAN, WAN e AWS (Amazon Web Services).
  - Permite clusters e grupos virtuais.

- **Mensagens distribuídas** – O sistema IMDG também disponibiliza funções de trocas de mensagens ao nível do cluster via os modelos de comunicação *publish-subscribe* e ponto-a-ponto direto. As características desta funcionalidade são:
  - Suporte do modelo *publish-subscribe* baseado em tópicos.
  - Suporta comunicação ponto-a-ponto direta.
  - Camada de transporte de comunicações *pluggable*.
  - Permite definir a ordenação de mensagens.
  - *Listener* de mensagens consciente do cluster.

- **Eventos distribuídos** - Nesta funcionalidade, o sistema permite que as aplicações recebam notificações, quando uma série de eventos ocorre num ambiente distribuído. Os programadores podem aplicar esta funcionalidade para serem notificados sobre execuções remotas de tarefas ou de alterações de dados em *cache* no cluster. As notificações de eventos podem ser agrupadas e enviadas de uma só vez ou em intervalos de tempo, para reduzir o tráfego de rede. Isto permite ações como:
  - Subscrever *listeners* locais e remotos.
  - Ativar e desativar qualquer evento.
  - Criar filtros para eventos locais e remotos.
  - Notificações de eventos agrupados.

- **Estruturas de dados distribuídas** – Estas estruturas funcionam como a maioria das estruturas de dados da *framework java.util.concurrent* que são usadas de maneira distribuída. Para além de permitir um armazenamento *key-value* standard, também fornece implementações de uma *blocking queue* distribuída e de um *Set* distribuído. Em suma, fornece implementações para as seguintes primitivas:
  - *Map* concorrente.
  - *Queues* e *Sets* distribuídos.
  - *AtomicLong.*

- *AtomicSequence.*
- *AtomicReference.*
- *CountDownLatch.*
- *ExecutorService*.

- **Sistema de ficheiros distribuídos** - O Ignite disponibiliza uma interface para os seus dados em memória chamada *Ignite File System* (IGFS). Esta interface tem funcionalidades similares às do *Hadoop Distributed File System* (HDFS), mas isto em memória. Os dados de cada ficheiro são partidos em blocos de dados separados e armazenados em *cache*. Podemos aceder a cada ficheiro a partir da API de *streaming Java* e, para cada parte do ficheiro, calcular a afinidade e processar o conteúdo do ficheiro no nodo correspondente, evitando assim tráfego desnecessário na rede. Algumas das suas características mais específicas são:
  - *In-Memory File System*.
  - Listar diretorias.
  - Conseguir informação para um caminho único.
  - Criar/Mover/Apagar ficheiros ou diretorias.
  - Escrever/Ler *streams* de dados de/para ficheiros.

3.4 Comparação Final dos Produtos

Ainda no âmbito da descrição dos produtos escolhidos é importante fazer a sua comparação com as funcionalidades descritas como objetivos funcionais propostos por uma empresa de telecomunicações e que considerámos importantes para o processo de seleção de soluções IMDG alternativas. A Tabela 3 é o resultado dessa comparação e complementa a descrição de cada uma das soluções IMDG selecionadas como alternativa realizada nas seções anteriores.

Tabela 3: Características das Soluções IMDG Estudadas

Funcionalidade	Hazelcast	Infinispan	Galaxy	Ehcache	Apache Ignite
Multigrid	Sim	Sim	?	Sim	Sim
Invocation	Sim	Sim	Sim	Sim	Sim

services					
Entry processors	Sim	Sim	Sim	Sim	Sim
***Cache* replicada**	Sim	Sim	Sim	Sim	Sim
***Cache* particionada**	Sim	Sim	Sim	Sim	Sim
Near *Cache*	Sim	Sim	?	?	Sim
Afinidade de dados	Sim	Key affinity e Grouping API	Sim	?	Key affinity
Map listeners	Sim	Sim	Sim	Sim	Sim
Triggers	Sim	Sim	Sim	Sim	Sim
JPA	Integração com Hibernate	Integração com Hibernate	?	Integração com Hibernate	Integração com Hibernate
Persistência	Memória volátil; BDs relacionais	Memória volátil; Berkeley DB; Cassandra; Sistema de ficheiros; ExFat; LevelDB; MongoDB; Ficheiros Memorymaped; BDs relacionais;	BerkeleyDB; Bases de dados SQL	Memória volátil; Sistema de ficheiros;	Sistema de ficheiros; ExFat; LevelDB; RAM; BDs relacionais; SSD;
Transações	Utiliza uma interface de transações própria., Locking Distribuído; Locking otimista; Locking pessimista;	Pode usar JTA; Commitment ordering; Locking Distribuído; Locking otimista; Locking pessimista; Modelo lock livre;	Utiliza um protocolo similar ao usado pelos CPUs para coordenar *cache*s L1.	Pode usar JTA; Optimistic Locking;	Pode usar JTA; API calls; JSON; Java API; MapReduce; Mem*cache*d-protocol; SQL; REST;
Alta disponibilidade	Sim	Sim	Sim	Sim	Sim
Clientes	.NET Framework; C#; C++; Java (qualquer linguagem de	Java; Ruby; Python; C#; C++; Scala;	Java	Java (qualquer linguagem de scripting JVM)	Scala; C++; Java (qualquer linguagem de scripting JVM);

	scripting JVM);				C#; .NET Framework

Considerando as catorze ferramentas analisadas, podemos concluir que todas são semelhantes em termos de funcionalidades base. Porém, divergem um pouco quanto ao suporte, ferramentas complementares de monitorização e de gestão e na documentação que disponibilizam. Além disso, podemos também concluir que os produtos pagos das maiores empresas estão normalmente incluídos em pacotes com outros produtos proprietários com o objetivo de maximizar lucros, mas, porém, acarretando custos adicionais para os clientes. Complementarmente, podemos ver pelos resultados alcançados que existem ferramentas *open source* muito completas e com capacidade para desempenhar as mesmas funções que as que os produtos comerciais oferecem, para além de oferecerem também um bom nível de suporte e confiança aos clientes.

A análise realizada permitiu-nos obter os elementos necessários para que pudéssemos suportar a nossa opção por uma solução IMDG *open source* alternativa. Com a ajuda da informação contida na Tabela 3, decidimos optar pelos sistemas IMDG Hazelcast e Infinispan, uma vez que são produtos que, funcionalmente, são muito completos, e que disponibilizam um excelente suporte, não só ao nível da comunidade *open source* como também ao nível comercial no mercado de produtos IMDG.

Capítulo 4

As Soluções IMDG Hazelcast e Infinispan

De maneira a validar o Hazelcast e o Infinispan como alternativas tecnológicas a outras IMDGs pagas, projetou-se e desenvolveu-se duas aplicações, cada uma com uma IMDG diferente, com o objetivo de as testar relativamente aos requisitos funcionais estabelecidos (Secção 1.2). Assim, no presente capítulo ir-se-á descrever o processo de concepção das duas aplicações referidas. Inicialmente, descrever-se-á o modelo de dados comum às aplicações, assim como os casos de uso que foram implementados, apresentando depois o ambiente de desenvolvimento e, por último, descrever detalhadamente cada uma das funcionalidades e como foram integradas em cada aplicação.

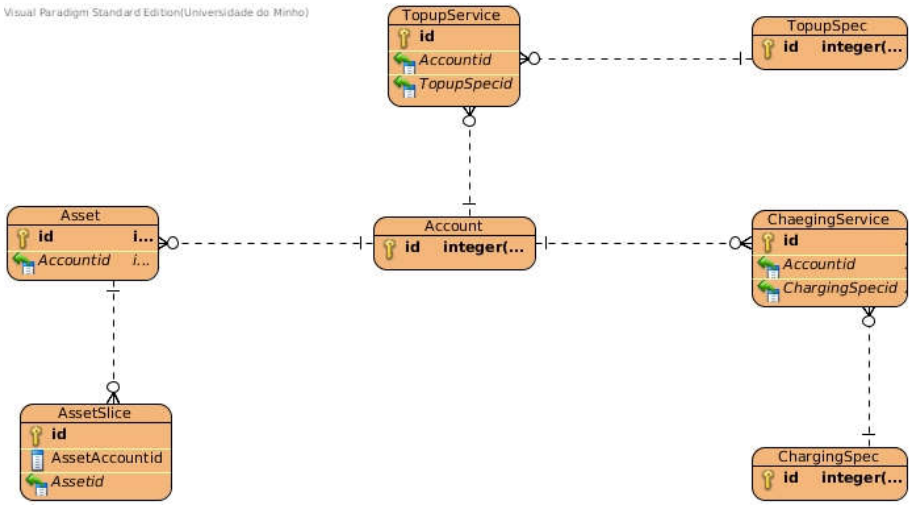


Figura 5: Modelo de Dados

4.1 Modelo de Dados

O modelo de dados estabelecido para suporte ao nosso estudo de comparação de soluções IMDG teve como objetivo estabelecer uma base sólida para acolhimento da informação das várias aplicações desenvolvidas. Nesse modelo (Figura 5), cada uma das entidades envolvidas está representada por uma tabela da base de dados relacional, representando os relacionamentos as formas como cada uma das tabelas se relaciona com outras tabelas no modelo.

No modelo definido (Figura 5) podemos encontrar sete tabelas, *Account, Asset, AssetSlices, ChargingServices, ChargingSpecs, TopUpServices* e *TopUpSpecs*, cuja descrição se apresenta de seguida:

- **Account** – esta é a tabela da entidade mais importante do sistema, pois agrega toda a informação relativa à subscrição de serviços de um determinado cliente. Todas as outras tabelas estão ligadas a esta, tendo o seu *id* como chave estrangeira. Esta tabela também contém os *AssetSlice* de onde o cliente deve consumir saldo.
- **ChargingSpec** – esta tabela contém informação sobre os serviços disponíveis. Especificação das regras de valorização e cobrança dos serviços. Aqui está guardada a informação do valor a debitar por cada serviço.
- **ChargingService** – esta tabela contém as instanciações de serviços para contas, representando as subscrições de serviços de cobrança por parte de um cliente.
- **Asset** – esta é a tabela da entidade cobrável, que é constituída por várias parcelas chamadas *slices*. Estas *slices,* que se encontram limitadas por um intervalo de tempo, têm um dado valor associado, que corresponde ao saldo do cliente para cada um desses períodos de tempo. Este valor de saldo só pode ser utilizado se o instante de utilização do serviço se encontrar dentro do seu intervalo de definição. Associada com esta tabela, está a tabela *AssetSlices* que acolhe uma lista de parcelas do salto correspondente, ordenada por data de término, para a qual a data da *Slice* é válida.
- **AssetSlice** – tal como referido anteriormente para a tabela "Asset", esta tabela acolhe as parcela de saldo, que são limitadas por um período de validade, tal como a própria estrutura da tabela o revela.
- **TopupSpec** – esta tabela acolhe a especificação das regras de criação e distribuição de recargas nos vários *Assets*. Além disso, contém outra informação, como a validade das *slices* a criar e o seu valor máximo.

- **TopupService** – as instanciações de serviços para contas são guardadas nesta tabela, que representam as subscrições de serviços de recarga por parte de um cliente.

4.2 Os Casos de Uso

4.2.1 Apresentação Geral

Nesta fase de desenvolvimento da dissertação estipulou-se um pequeno conjunto de caso de estudo com vista à realização do teste e do desenvolvimento de uma solução para um sistema assente numa plataforma computacional IMDG. Os casos de uso selecionados foram definidos a partir de algumas das situações mais vulgares que usualmente consideram processos de clientes envolvidos num sistema de taxação de telecomunicações convencional, com objetivo de constituírem elementos de trabalho para teste da aplicação IMDG referida.

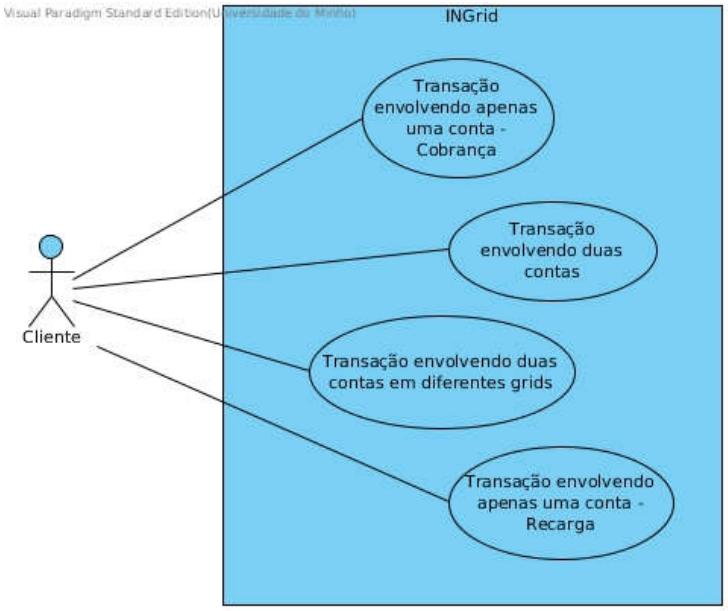


Figura 6: Referência aos casos de uso selecionados

No processo de seleção dos casos de uso esteve sempre presente a assunção de que estes deveriam permitir a realização de testes especialmente orientados para a análise dos diferentes aspetos operacionais vulgarmente relacionados com a implementação de uma solução IMDG, em particular os relacionados com a utilização comum de um sistema de telecomunicações, como

sejam a garantia da atomicidade de transações, a concorrência de processos e, por fim, o desempenho global do próprio sistema. Nesse leque de casos de estudo foram incluídos alguns processos de trabalhos que envolviam a realização de um conjunto de tarefas bastante típicas no domínio dos sistemas de telecomunicações. Os casos selecionados (Figura 6) envolvem, na prática, respectivamente:

1. Uma transação envolvendo apenas uma conta de um cliente - na lógica de negócio, esta transação pode ser vista como uma simples operação de débito na qual o acesso aos dados e consequentes alterações são realizadas sobre uma mesma conta.
2. Uma transação envolvendo apenas uma conta de um cliente - na ótica de negócio esta operação corresponderia a um carregamento de dinheiro feito pelo cliente para gerar saldo, que provocará a criação de *asset* e *asset slices* por parte de um cliente, permitindo assim que os serviços que envolvem cobrança sejam possíveis.
3. Uma transação envolvendo duas contas de clientes - uma operação que pode ser vista como uma operação de transferência de dinheiro de uma conta para outra, que implica a consulta da informação relacionada com as contas em causa e que, no final, obriga à realização de operações de alteração sobre ambas as contas.

Todos estes casos de uso serão utilizados no processo de avaliação tecnológica que pretendemos realizar, tendo em conta as funcionalidades e características técnicas necessárias para demonstrar a aplicação de um sistema IMDG no mercado das telecomunicações.

4.2.2 Os Casos de Uso a Implementar

Neste trabalho de dissertação foram implementados os três casos de estudo descritos anteriormente:

1. a realização de transações envolvendo apenas uma conta de um cliente.
2. a cobrança de um serviço e recarga de saldo.
3. a realização de uma transação envolvendo duas contas de clientes.

Estes casos de estudo serão descritos pormenorizadamente nas secções seguintes.

Transação Envolvendo uma Conta – Cobrança

Para este caso de uso, assumiu-se que um cliente usufrui de um serviço, que é subscrito por si, e, como tal, o valor do custo desse serviço ser-lhe-á debitado na sua conta, com a devida atualização do seu saldo. O motivo que conduziu à seleção deste caso de uso prendeu-se, simplesmente, com o facto de ele nos permitir fazer, de uma maneira muito simples, a demonstração das funcionalidades mais básicas, e ao mesmo tempo mais importantes, de uma solução IMDG. A descrição desse caso de estudo está apresentada na Figura 7.

Super Use Case	Transação envolvendo apenas uma conta - Cobrança		
Author	hugo		
Date	Oct, 2015 2:20:41 PM		
Brief Description	Quando um cliente utiliza um serviço que tem subscrito e é necessário debitar o valor dessa operação do seu saldo		
Preconditions	O cliente tem conta, o serviço subscrito e saldo suficiente		
Post-conditions	O valor do serviço é debitado e o cliente é notificado		
Interação na Grid		**Actor Input**	**System Response**
	1	Pedido é enviado para processamento na Grid	
	2		O pedido é avaliado e encaminhado para processamento um nó da Grid
	3		Uma lógica funcional é executada nesse nó da Grid no contexto de uma transacção
	4		Se tudo correr bem, no final da transacção as actualizações deverão ser reflectidas na Grid e persistidas de forma consistente na base de dados. Em caso de erro, a transacção deverá ser abortada
	5		Deverá ser devolvida uma resposta para o cliente com o resultado da operação
	6	Recebe notificação do novo saldo	
Execução no nó da Grid (passo 3)		**Actor Input**	**System Response**
	1		Acesso e leitura a um conjunto de informação da conta na Grid
	2		Atualização dum conjunto de informações na conta
	3		Deverá ser gerado um registo de actividade com toda a informação relevante à operação.

Figura 7: Descrição do caso de uso – Cobrança

O processo base deste caso de estudo tem um funcionamento bastante simples. É uma das atividades mais básicas que se realiza num sistema de taxação de serviços de telecomunicações. Para suportar uma dada transação de atualização, que afete exclusivamente uma única conta de cliente, como é sabido, é necessário receber, como parâmetros de entrada (*input*), a identificação

do cliente e o tipo de serviço que este utilizou. Neste processo pressupõe-se que o cliente existe, que tem o serviço em questão subscrito e que tem saldo suficiente para cobrir o seu custo. Mesmo nestas circunstâncias, estas condições são verificadas durante a execução do processo. Num passo seguinte, é necessário aceder às especificações do serviço e verificar a forma como este deve ser taxado. Em suma determinar o seu custo. De seguida, verifica-se se o cliente tem dinheiro para pagar o serviço utilizado. Caso isso se verifique é possível obter a informação das *asset slices* que podem ser utilizadas. Por fim, procede-se à atualização dos diversos saldos envolvidos: o saldo geral no *asset* e o saldo da *asset slice*. Caso o custo seja superior ao valor da *asset slice*, a diferença é consumida na próxima transação a ser realizada.

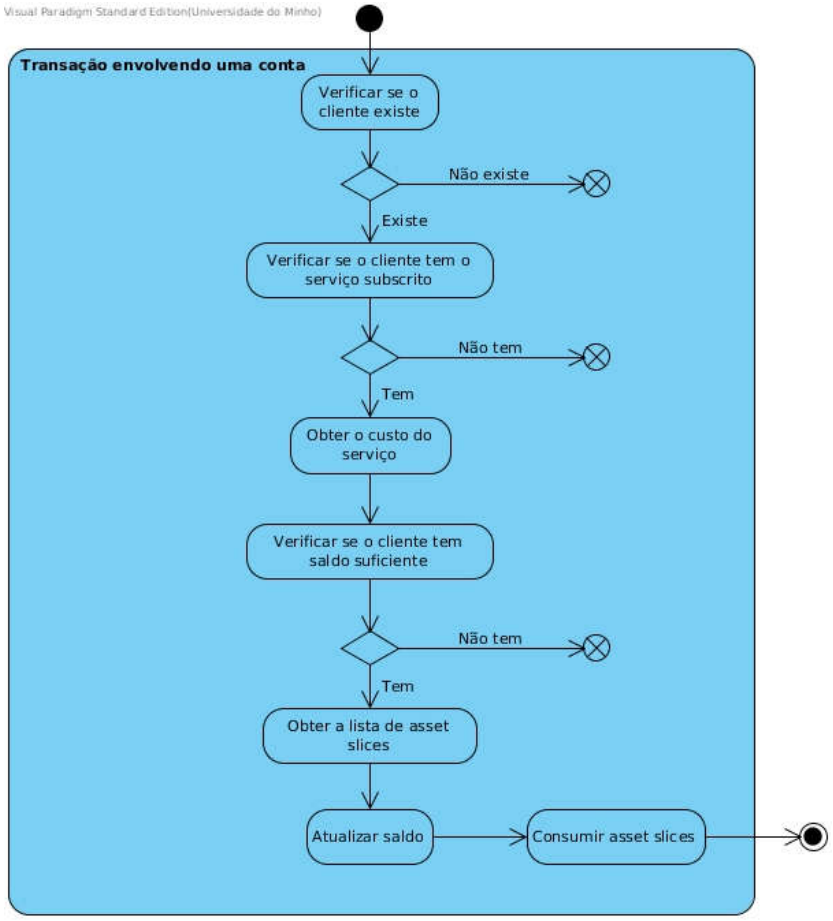


Figura 8: Diagrama de atividades relativo ao caso de uso Cobrança

Todo este processo está sinteticamente exposto no diagrama de atividades apresentado na Figura 8. O procedimento descrito neste diagrama reflete, basicamente, a lógica que é executada num dos nodos do cluster, no contexto da transação que estiver subjacente ao processo. Como já foi referido, a informação de entrada deste processo é constituída pela identificação do cliente e o tipo do serviço prestado. Como consequência, espera-se que as operações de atualização geradas durante a execução da transação sejam adequadamente refletidas nos dados em memória e refletidas na base de dados.

Super Use Case	Transação envolvendo apenas uma conta - Recarga		
Author	hugo		
Date	Jan 9, 2016 4:58:31 PM		
Brief Description	O cliente efetua um carregamento para criar Asset Slices que possam ser consumidas quando utilizar um serviço de cobrança		
Preconditions	O cliente tem conta e tem um serviço Topup subscrito		
Post-conditions	O valor que carrega fica disponível para consumo		
Interação na Grid		**Actor Input**	**System Response**
	1	Pedido é enviado para processamento na Grid	
	2		O pedido é avaliado e encaminhado para processamento um nó da Grid
	3		Uma lógica funcional é executada nesse nó da Grid no contexto de uma transacção
	4		Se tudo correr bem, no final da transacção as actualizações deverão ser reflectidas na Grid e persistidas de forma consistente na base de dados. Em caso de erro, a transacção deverá ser abortada
	5		Deverá ser devolvida uma resposta para o cliente com o resultado da operação
	6	Recebe notificação do novo saldo	
Execução no nó da Grid (passo 3)		**Actor Input**	**System Response**
	1		Acesso e leitura a um conjunto de informação da conta na Grid
	2		Atualização do saldo do cliente e criação dos Asset Slices
	3		Deverá ser gerado um registo de actividade com toda a informação relevante à operação

Figura 9: Descrição do caso de uso – Recarga

De um ponto de vista mais técnico, pode-se dizer que, no processo relativo a este caso de uso é enviado um pedido para processamento na grelha que, depois de avaliado, é encaminhado para processamento num dos seus nodos. Nesse nodo, e dentro do contexto da transação em questão, são executados os acessos e as leituras da informação necessários, e atualizados os dados que tiverem sido alterados. Se tudo correr de acordo com o esperado, no final da transação todas as

atualizações deverão estar refletidas na grelha de dados e mantidas de forma persistente e consistente na base de dados do sistema.

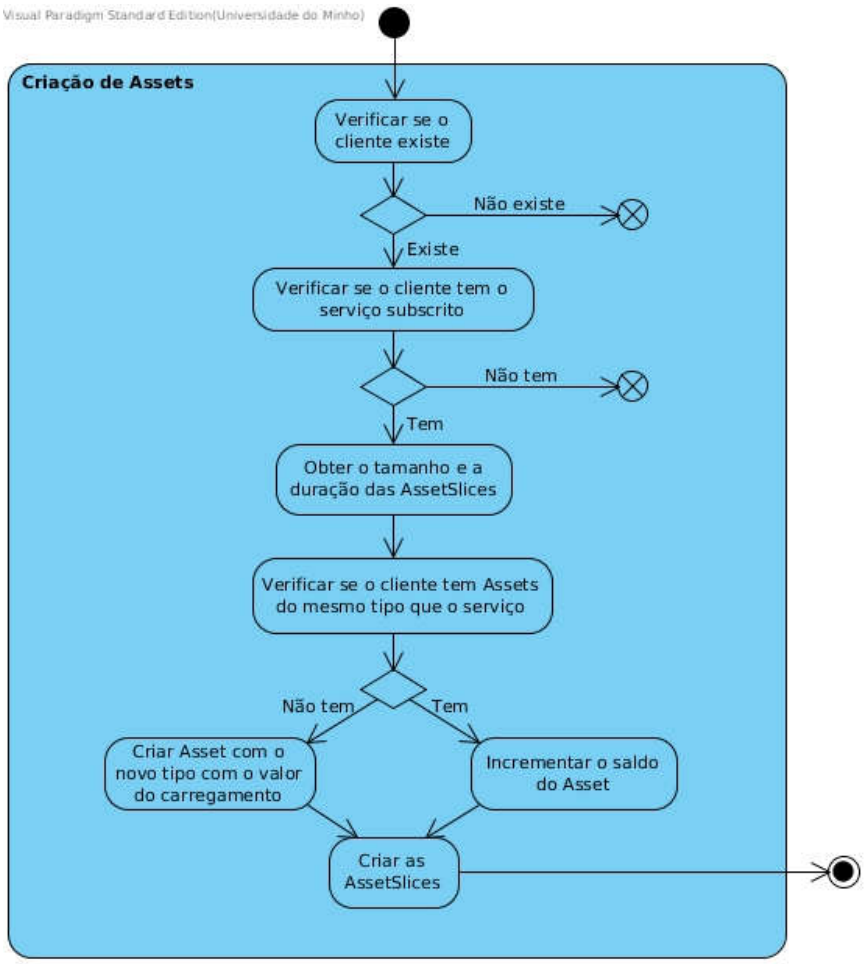


Figura 10: Diagrama de atividade relativo ao caso de uso Recarga

Transação Envolvendo uma Conta – Operação de Recarga

O caso de uso de uma operação de recarga representa um carregamento feito por determinado cliente, que gerando um saldo, permite o usufruto de serviços que envolvam cobranças ou transferências. Este caso assemelha-se muito ao caso de uso de cobrança, uma vez que também acede, altera e gera dados referentes à conta dum só cliente, dados estes que, idealmente,

estarão localizados no mesmo nodo da *grid*. Não obstante, distingue-se da cobrança na medida em que, incorpora a mais-valia da criação e persistência de novos objetos, nomeadamente, *Assets* e *Asset Slices*. A descrição deste processo pode ser observada na Figura 9.

O processo por trás deste caso de uso é um dos processos base de qualquer sistema de telecomunicações. É com este processo que o cliente paga os serviços que futuramente irá utilizar. Para poder executar este caso de uso são necessários, como parâmetros de entrada, a identificação do cliente, o tipo do serviço – e.g., pré-pago e pós-pago – e, por fim, o valor do carregamento efetuado.

De forma semelhante ao caso de uso anterior, aqui é pressuposto que o cliente exista e que tenha o serviço em questão subscrito, condições estas que deverão sempre ser verificadas durante a execução do processo. O passo seguinte é a obtenção dos dados necessários para a criação das *Asset Slices*, ou seja, o tamanho que as fatias (*asset slices*) terão e a duração de cada uma delas, sendo que o valor da duração é incrementado caso haja mais que uma fatia. Em seguida, verificamos se o cliente já tem *Assets* do mesmo tipo do carregamento, ou seja, se já tinha realizado carregamentos deste tipo anteriormente ou se é a primeira vez. Caso seja a primeira vez, temos de criar um *Asset* desse mesmo tipo, com um saldo total igual ao valor do carregamento. Caso não seja, o valor do carregamento é adicionado ao valor do saldo existente na *Asset*. Por fim criamos as *Asset Slices*, nunca maiores que o tamanho nem com duração superior que as especificações do serviço. A validade das fatias é acumulativa. Se existirem fatias válidas, acrescenta-se a duração à data da última fatia, caso não existam fatias válidas, acrescenta-se a duração ao dia de hoje e assim sucessivamente. As fatias são guardadas por ordem ascendente da data de término e são também consumidas nessa ordem. Na Figura 10 podemos ver o diagrama de atividades relativo a este processo. Do ponto de vista técnico, a interação com cluster e com os seus nodos é similar à do primeiro caso de uso.

Transação envolvendo duas contas - Transferência de saldo

Uma transação entre duas contas pode ser vista como uma transferência de saldo, na qual, do lado do primeiro cliente, é debitado o valor da transferência e, do lado do segundo, é creditada essa mesma quantia. Este caso de uso difere dos dois anteriores porque envolve dois clientes, o que, num cenário distribuído, com *cache* distribuída, pode significar que os dados dos clientes

estão localizados em dois nodos da grelha diferentes, ou seja, a própria transação poder ser distribuída. Na Figura 11 podemos observar a descrição deste caso de uso.

Super Use Case	Transação envolvendo duas duas contas na mesma Grid		
Author	Hugo		
Date	May 6, 2015 6:54:21 PM		
Brief Description	Quando um cliente faz uma transferência de saldo para outro cliente. O valor é debitado numa das contas e creditado na outra.		
Preconditions	O cliente que transfere saldo tem conta, o serviço subscrito e saldo suficiente		
Post-conditions	O soma dos dois saldos envolvidos mantem-se igual e os clientes são notificados		
Interação na Grid		**Actor Input**	**System Response**
	1	Pedido é enviado para processamento na Grid	
	2		O pedido é avaliado e encaminhado para processamento um nó da Grid
	3		Uma lógica funcional é executada nesse nó da Grid no contexto de uma transacção
	4		Se tudo correr bem, no final da transacção as actualizações deverão ser reflectidas na Grid e persistidas de forma consistente na base de dados. Em caso de erro, a transacção deverá ser abortada
	5		Deverá ser devolvida uma resposta para o cliente com o resultado da operação
	6	Recebe notificação do novo saldo	
Execução no nó da Grid (passo 3)		**Actor Input**	**System Response**
	1		Acesso e leitura de um conjunto de informação da conta na Grid
	2		Acesso e leitura de um conjunto de informação de uma segunda conta na Grid Local
	3		Actualização de um conjunto de informação de ambas as contas
	4		Deverá ser gerado um registo de actividade com toda a informação relevante à operação.

Figura 11: Descrição do caso de uso - Transferência de Saldo

Para que seja possível fazer uma transferência são necessários, como parâmetros de entrada, a identificação das duas contas envolvidas na transferência, o tipo do serviço e a quantia a transferir. Neste caso, o tipo do serviço é necessário porque o cliente que quer transferir saldo tem de ter o serviço subscrito. Na Figura 12 pode-se observar uma representação do processo em questão. Como é possível depreender através da observação do diagrama de atividades apresentado na Figura 12, é possível dividir o processo em duas partes. A parte relativa ao primeiro cliente e a parte relativa ao segundo cliente. Na parte do primeiro cliente temos um processo semelhante ao caso de uso de cobrança, em que a única diferença que se identifica é a de não ser necessário aceder às especificações do serviço para obter o seu custo. Na parte do segundo cliente o

processo é igual ao do caso de uso de recarga, mas com a quantia igual à debitada ao primeiro cliente. Esta parte só é executada se a primeira parte ocorrer sem erros. As verificações de conta, subscrições de serviços e de saldo são feitas durante a execução do processo, tal como nos casos anteriores. Do ponto de vista técnico, a interação com o cluster e com os seus nodos é similar às dos casos de uso anteriores, sendo que neste processo estejam envolvidos dois nodos da grelha e não apenas um só. Isto faz com que este processo implique uma transação distribuída, envolvendo duas transações locais, vistas como um bloco indivisível, que é executado completamente ou então simplesmente falha. No final, os dados mantêm-se coerentes e a soma dos saldos dos dois clientes é a mesma que no início do processo.

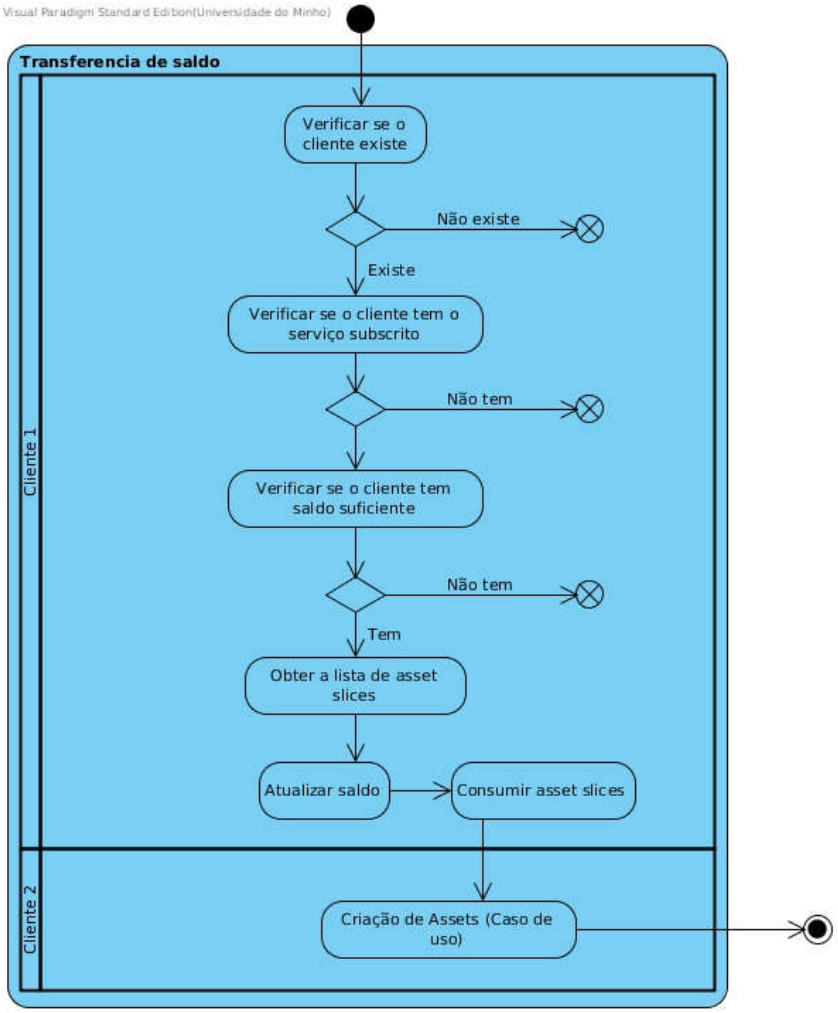


Figura 12: Diagrama de atividade relativo ao caso de uso Transferência de Saldo

4.3 Implementação das Aplicações

Para melhor compreensão dos sistemas implementados, devemos explicar a sua estrutura base e o ambiente de desenvolvimento das aplicações. Assim, de seguida, iremos descrever as arquiteturas da aplicação desenvolvida com Hazelcast e da aplicação desenvolvida com Infinispan. De referir que, a arquitetura da primeira destas aplicações sofreu mais alterações devido a ter sido implementada em primeiro lugar e por se ter incluído novas funcionalidades ao longo do processo de estudo e análise desta dissertação.

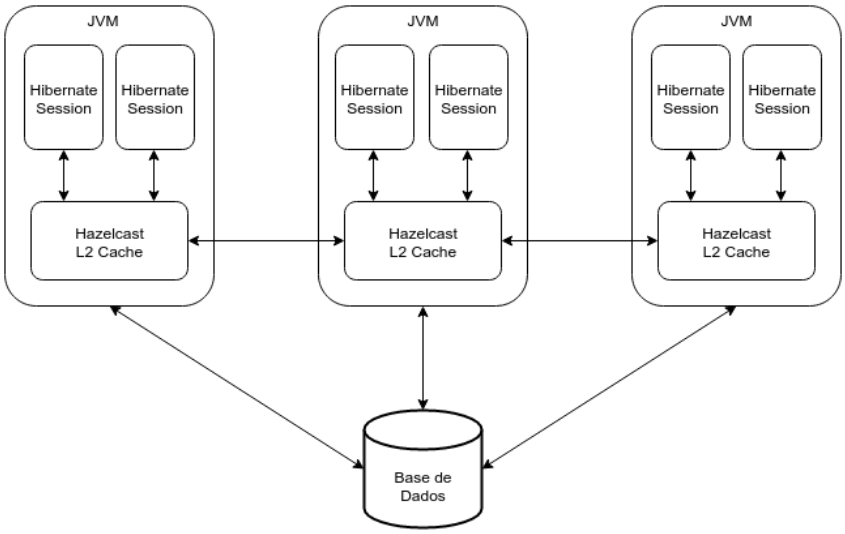


Figura 13: Arquitetura inicial da aplicação utilizando-se Hazelcast

4.3.1 A Arquitetura do Sistema *Hazelcast*

Tal como acontece na maioria dos sistemas deste tipo, o sistema protótipo foi organizado num sistema em três camadas, nomeadamente: persistência, dados distribuídos em memória, e camada aplicacional. Inicialmente, a aplicação utilizava a mesma classe na qual estavam tanto os seus métodos específicos como os métodos responsáveis pela inicialização da instância do cluster Hazelcast. Do ponto de vista estrutural, podemos considerar esta estrutura semelhante à apresentada anteriormente na **Erro! Fonte de referência não encontrada.**, na qual podemos ver uma camada de persistência separada, em que o seu servidor esperava os pedidos que teria

que satisfazer. Criaram-se, também, diversos servidores Hazelcast, nos quais cada JVM corre numa instância Hazelcast com sessões *Hibernate*. Neste cenário os casos de uso são executados através de uma interface de linha de comandos. Tal como se pode ver na **Erro! Fonte de referência não encontrada.**, todas as componentes Hazelcast ligam-se entre si, o que torna possível a disponibilização de uma camada de dados distribuída em memória - uma *Cache* L2 -, revelando uma arquitetura embebida típica (Hazelcast, 2016a).

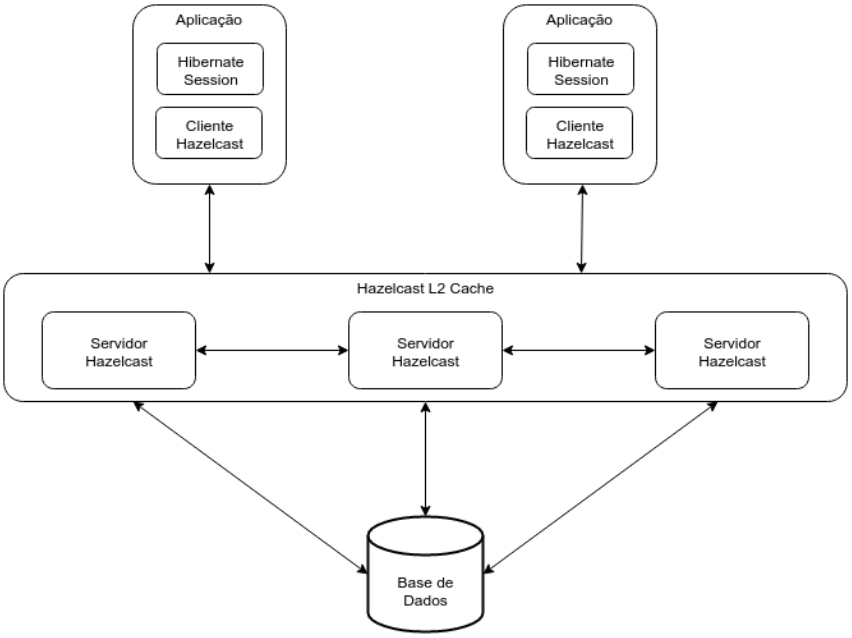


Figura 14: Arquitetura atual da aplicação com Hazelcast

Numa fase posterior do desenvolvimento do sistema, a parte aplicacional foi separada do cluster de servidores Hazelcast. Ao fazermos isso, a execução dos casos de uso passou a ser feita como um teste Java. Para isso foi criada uma instância cliente Hazelcast, em vez de um servidor, que no final da execução do processo é desligada, mantendo-se, porém, o cluster ativo. Ao optarmos por esta solução, a camada de dados do sistema, que está em memória, pode ser vista como uma camada horizontal acessível a todos os clientes (Figura 14).

A arquitetura alcançada é uma arquitetura típica cliente-servidor (Hazelcast, 2016a), na qual a interação entre a camada aplicacional e a *cache* L2 distribuída é realizada através da interface *EntityManager*. No entanto, na versão atual da aplicação, devido à inclusão do serviço de execução

distribuída (*IExecutorService*), na camada aplicacional teve de ser criada uma instância servidor Hazelcast em vez de uma instância cliente. Isto deve-se ao facto dos clientes Hazelcast não terem conhecimento sobre a localização dos dados no cluster. Porém, se o pretendido era fazer o direcionamento dos pedidos de execução para o dono das chaves manipuladas isso não poderá ser realizado. Assim, mantivemos a arquitetura tal como a Figura 14 apresenta, mas com um servidor Hazelcast em vez de um cliente Hazelcast na camada aplicacional. A camada aplicacional recebeu a implementação dos diversos casos de uso, nomeadamente: um processo de cobrança de um serviço de telecomunicações subscrito por um dado cliente, um processo de geração de *AssetSlices* como carregamento de conta, e a transferência de saldo entre clientes.

4.3.2 A Arquitetura do Sistema Infinispan

De forma semelhante às outras IMDG, o Infinispan também disponibiliza dois modos de funcionamento: *Peer-to-Peer* (P2P) e Cliente-Servidor. Segundo a documentação do Infinispan, o modo cliente-servidor tem desvantagens sobre o modo P2P. O modo P2P é mais simples, uma vez que os nodos são iguais do ponto de vista funcional, facilitando a sua gestão e o seu *deployment*. É, também, provável que pedidos em cliente-servidor sejam mais lentos, devido ao custo de serialização e utilização da rede em chamadas remotas. Normalmente, neste tipo de implementações aconselha-se a ter clientes externos mais "leves" ligados a uma aplicação do lado dos servidores que acede ao Infinispan em modo P2P, em vez de uma aplicação mais "pesada" do lado do cliente conectada ao Infinispan em modo cliente-servidor. Esta diferença de desempenho é mais acentuada se usarmos uma *cache* Infinispan replicada. Em modo distribuído, a diferença é menor, uma vez que não há garantias que os dados estejam disponíveis localmente (Surtani et al., 2015a). Porém, mais importante do que isto, aquando da decisão da arquitetura da aplicação, foi conseguir implementar as funcionalidades que tinham sido previamente definidas, em particular, a submissão das tarefas aos nodos donos dos dados (execução local) e a definição de Near *Cache*s. Estas duas funcionalidades motivaram a integração do protocolo *Hot Rod* numa topologia cliente-servidor na aplicação. O modulo servidor *Hot Rod* é uma implementação do protocolo binário *Hot Rod* suportado pelo Infinispan, que permite que os seus clientes façam balanceamento dinâmico de carga, tolerem falhas de servidores e direcionem os pedidos aos nodos onde estão localizados os dados (*smart routing*). Os clientes Hot Rod conseguem assim detectar dinamicamente mudanças na topologia do cluster, atualizando a sua *view* do cluster sempre que sai algum nodo

ou novos nodos se juntam. Além disso, quando as *cache*s são distribuídas, os clientes têm conhecimento da localização de todas as chaves, conseguindo direcionar os seus pedidos de forma inteligente. Isto é conseguido através do envio de informação dos servidores Hot Rod para os seus clientes. Quando os clientes são aplicações Java, este protocolo é altamente recomendado (Surtani et al., 2015a). A arquitetura final da aplicação desenvolvida utilizando Infinispan pode ser observada na Figura 15.

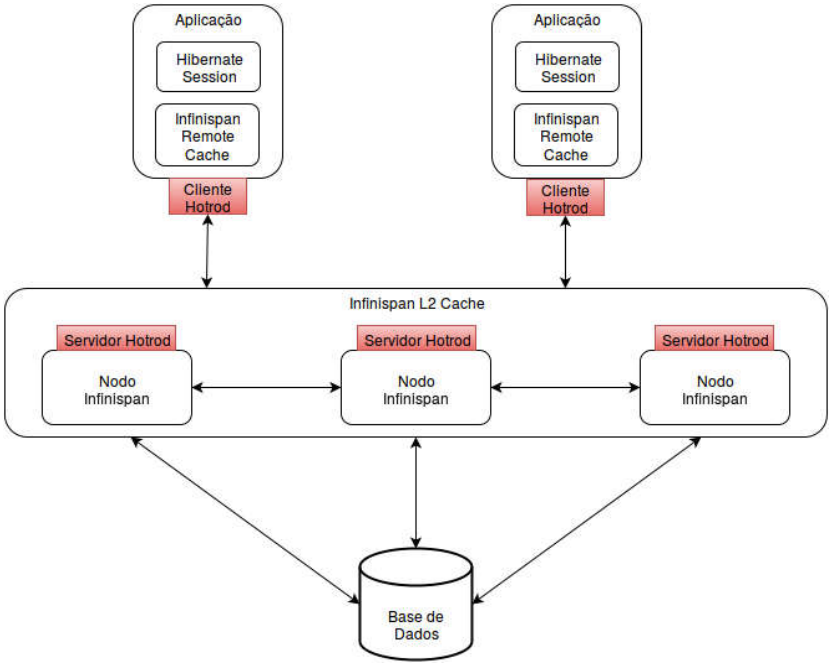


Figura 15: A arquitetura da aplicação utilizando Infinispan.

4.3.3 O Ambiente de Desenvolvimento

O Ambiente de Desenvolvimento Integrado (IDE) utilizado para o desenvolvimento das aplicações envolvidas neste trabalho de avaliação de soluções IMDG foi o *IntelliJ IDEA*[1]. Quanto às aplicações Hazelcast e Infinispan estas são bastante semelhantes, divergindo apenas em algumas das classes e dependências que utilizam para algumas funcionalidades que não são comuns às duas aplicações e nos ficheiros de configuração que variam de acordo com o fabricante da solução IMDG. Não

[1] https://www.jetbrains.com/idea/

obstante, vamos dividir esta secção tendo em conta as características e as funcionalidades das duas aplicações.

A Solução Hazelcast

No processo de desenvolvimento da solução baseada em Hazelcast foram concebidos e implementados quatro *package*s distintos. De referir:

- **Database** – este *package* possui duas classes distintas, que trabalham diretamente sobre a base de dados; uma dessas classes é utilizada para fazer a criação da própria base de dados enquanto que a outra classe é utilizada para fazer o povoamento das entidades que são necessárias para a execução dos diversos casos de uso.
- **Entities** – no qual estão definidas as estruturas das entidades, as suas ligações, sempre bidirecionais conforme o definido no modelo de dados, e alguns métodos comuns a todos as classes, como por exemplo os métodos *setters* e *getters*. Aqui, está também definida uma classe por cada tabela existente na base de dados, num total de sete classes. Todas estas classes têm as respectivas anotações *Java Persistence API* (JPA) e *Hibernate* de forma a que seja possível fazer um mapeamento objeto-relacional de forma correta.
- **Instances** – é um *package* que é constituído por uma única classe que, quando executada, cria um nodo servidor Hazelcast.
- **UseCases** - neste *package* implementámos os casos de uso já enunciados, dando origem a três classes distintas, nomeadamente, *SimpleChargingService, TopupService* e *TransactionBetweenAccounts*. Estas três classes incorporam os métodos necessários para executar a lógica funcional dos casos de uso. Ainda neste *package*, existem mais duas classes que nasceram da necessidade da inclusão de chaves compostas para garantir a afinidade de dados no cluster. Uma dessas classes - *MyKey* - representa a constituição da chave composta e implementa o método *getPartitionKey*, enquanto que a outra - *MyPartitioningStrategy* - implementa o método que define a estratégia de particionamento dos dados pelo cluster.

Para além destes quatro *package*s, existe ainda a pasta de *Resources* e o módulo de *testes*. A diretoria - *Resources* - é o local por omissão de todos os ficheiros de configuração da aplicação. No caso do Hazelcast esta pasta contém dois ficheiros XML:

- *persistence.xml* que define a unidade de persistência (*persistence-unit*) e as propriedades JPA, que são sempre utilizadas aquando da criação de uma *Entity Manager Factory*;
- *hazelcast.xml*, ficheiro de configuração do Hazelcast, em que é possível definir os protocolos de comunicação e os métodos de descoberta de membros do cluster, configurar o comportamento da *cache*, incluindo *near cachês,* e, por fim, definir as comunicações com outros clusters - *cross site replication* .

Quanto ao módulo de testes, neste existe apenas uma classe que executa os três casos de uso como três testes unitários. Os testes são executados pela seguinte ordem: uma recarga, uma série de cobranças e uma transferência. Antes da realização dos testes é criado uma instância servidor Hazelcast que, caso exista, comunica com o cluster e é desligada no final dos testes. Por último, temos que referir a existência de um ficheiro POM. Neste ficheiro é identificado o projeto e todas as dependências externas de bibliotecas – e.g., *hazelcast, hazelcast-hibernate, hibernate-entitymanager, hibernate-jpa, derby e derby-client*. Estas bibliotecas disponibilizam os meios necessários para o funcionamento da aplicação.

A Solução Infinispan

A aplicação que foi desenvolvida com o Infinispan é constituída pelos mesmos quatro *package*s. Os *package*s Database e Entities, que estão diretamente relacionados com o modelo de dados definido anteriormente, são formados pelas mesmas classes, com algumas pequenas diferenças. Os restantes *package*s desta aplicação são:

- **Instances** – que é formado pelas classes *EmbeddedInstance* e *MyJndiServer*. A primeira possui método *main* que quando executada cria um nodo servidor Infinispan que implementa o serviço *Hot Rod Server* e regista o recurso XA. A segunda classe - *MyJndiServer* - é uma implementação própria dum servidor *Java Naming and Directory Interface* (JNDI) que vai servir para registar os gestores de *cache*, o recurso XA e o gestor de transações.
- **UseCases** – que é mais simples que na aplicação com Hazelcast, contém simplesmente três classes que são as implementações dos três casos de uso existentes, cobrança, recarga e transferência de saldo. As três classes são respectivamente *SimpleChargingService, TopupService e TransactionBetweenAccounts*.

Para além destes *package*s, e tal como na aplicação anterior, existe ainda a pasta de *Resources* e o módulo de *testes*. Na pasta *resources* estão localizados os vários ficheiros de configuração, que são sete: *persistence.xml, infinispan.xml, jgroups.xml, jgroups-relay2.xml, relay2.xml, jndi.properties* e *jta.properties* . Os ficheiros *persistence.xml* e o *infinispan.xml* têm funções idênticas às do *persistence.xml* e *hazelcast.xml* da aplicação anterior. Os três ficheiros seguintes - *jgroups.xml, jgroups-relay2.xml* e *relay2.xml* - são relativos às comunicações, dois com definições *JGroups,* para as comunicações dentro do cluster e para as comunicações entre clusters, e ainda o *relay.xml,* no qual são especificados os clusters existentes. Os dois últimos ficheiros - *jndi.properties* e *jta.properties* - são utilizados pela framework para definir as propriedades do *Java Naming Server* e do *Atomikos* como implementação de transações globais. Quanto ao **módulo de teste, este** tem a mesma estrutura que o da aplicação Hazelcast. Porém, neste módulo existe uma única classe que executa os três casos de uso em três testes unitários. Os testes são executados pela seguinte ordem: uma recarga, uma série de cobranças e uma transferência. Antes da execução dos testes é criado um cliente *Hot Rod* como gestor de *cache* remota, definida a nossa base de dados como recurso XA, e gerada uma implementação de um *UserTransaction.* Todos são registados no servidor JNDI. As dependências da aplicação Infinispan estão também definidas no ficheiro POM do projeto.

4.3.4 Funcionalidades Chave

Durante a implementação das aplicações manteve-se sempre presente as funcionalidades consideradas essenciais para a concretização do projeto de implementação dos casos de estudo. Estas funcionalidades foram definidas no início da dissertação e podem ser dividas em cinco grupos: execuções distribuídas, afinidade de dados, persistência e transações, *multigrid* e *near cache*. De seguida explicaremos cada uma delas e a forma como foram integradas nas aplicações desenvolvidas.

Execuções Distribuídas

Usualmente, numa execução distribuída um problema é dividido em várias tarefas, sendo cada uma destas resolvida por um ou mais computadores. Alguns frameworks, como a *Executor* do Java, permitem que algumas tarefas normalmente mais pesadas sejam executadas de forma assíncrona - e.g., *queries* a base de dados, cálculos complexos ou renderização de imagens.

Porém, tais frameworks por vezes são desenhados para serem executados numa só JVM, o que os torna inadequados para ambientes de sistemas distribuídos, nos quais uma tarefa pode ser submetida numa JVM, mas com o objetivo de ser executada numa outra qualquer. Os criadores do Hazelcast e do Infinispan aplicaram os princípios da *framework Executor* para sistemas distribuídos, mas acrescentaram a possibilidade das tarefas submetidas serem executadas poderem ser executadas num cluster de servidores (Surtani et al., 2015b). No caso do Hazelcast a interface disponibilizada (*IExecutorService*) é mais flexível, oferecendo várias opções do local no qual se vai executar o código dentro do cluster (Hazelcast, 2016b). Isso, permite-nos executar as tarefas num membro:

- especificado pelo utilizador;
- dono duma chave (key) à escolha;
- escolhido pelo *Hazelcast*;
- num subconjunto de membros do *cluster*.

O Infinispan, através da sua framework de execução distribuída (*DistributedExecutorService*), permite distribuir a execução por todos os membros do cluster de uma forma única. Isto faz com que esta framework não seja a mais adequada, quando o objetivo principal é a execução local de dados. Ou seja, visto que os dados estão guardados em pares chave-valor, queremos que a execução de dados seja feita no servidor dono das chaves envolvidas. Isto consegue-se de formas diferentes no Hazelcast e no Infinispan.

Na aplicação implementada com Hazelcast utilizámos o método descrito acima, que consiste na utilização do serviço criado para ambientes distribuídos semelhante à *Executor framework* existente no *Java*, o *IExecutorService*. Para conseguir esta funcionalidade foram feitas alterações nas partes das aplicações que tratam da submissão das tarefas (classe de teste) e na parte das próprias tarefas, que neste caso são os casos de uso. Para que as tarefas possam ser submetidas, estas têm que implementar a interface *Runnable* ou *Callable,* e para que possam ser transmitidas entre nodos, estas têm também de implementar a interface *Serializable*. Para concluir as alterações da parte das tarefas, acrescentaram-se ainda alguns outros métodos inerentes à implementação destas mesmas interfaces.

Para submeter as tarefas é necessário criar o *IExecutorService* através do nossa instância Hazelcast, que depois nos vai permitir submeter a tarefa no dono da chave

(*executorService.submitToKeyOwner*), que em conjunto com a interface *Future* do *Java,* para execução assíncrona de tarefas, permite recuperar os resultados da execução.

Com o Infinispan optou-se por não utilizar o *DistributedExecutorService*. A execução local dos dados foi obtida a partir da utilização do protocolo *Hot Rod*. Este protocolo cliente-servidor, sobre TCP, é usado pelo Infinispan, tendo sido criado para superar algumas falhas que outros protocolos cliente-servidor, como o *Memcached,* apresentam. Uma das vantagens do protocolo *Hot Rod* é a sua capacidade de recuperação em caso de mudanças de topologia que possam ocorrer num cluster de servidores Infinispan. Isto só é possível, porque o *Hot Rod* informa regularmente os clientes sobre a topologia do cluster. Para além disso, os seus clientes direcionam de forma inteligente os seus pedidos em clusters com *cache* particionada ou distribuída. Tal significa que, os clientes *Hot Rod* conseguem determinar em qual partição as chaves estão localizadas, comunicando diretamente com o servidor dono da chave. Isto só é possível graças aos servidores Infinispan enviarem informação sobre a topologia do cluster para os clientes e estes usarem o mesmo *consistent hash* que os servidores. Para podermos utilizar o protocolo *Hot Rod* na aplicação, tivemos que realizar as seguintes operações:

1. Acrescentar as dependências Infinispan para o servidor e para o cliente *Hot Rod*.
2. Do lado dos nodos servidor, criar o serviço *Hot Rod Server,* passando-lhe o endereço e porta na qual recebe pedidos.
3. Na parte do cliente, criar um gestor de *cache* remota, no qual é configurado um servidor *Hot Rod* como *host*. Este gestor de *cache* remota é o cliente *Hot Rod*.

Afinidade de Dados

A afinidade de dados de um sistema é a característica que descreve a forma como assegurar que um dado grupo de entradas *cache* relacionadas sejam guardadas na mesma partição de *cache*. Isto permite assegurar que todos os dados importantes são geridos no mesmo nodo de *cache* primária sem comprometer a tolerância a falhas (Oracle, 2015a). Se os dados relacionados estiverem no mesmo nodo, as operações podem ser executadas sem que seja necessário aceder a tantos nodos do cluster, tornando assim a execução de tarefas mais eficiente, o que reduz também o consumo de tráfego de rede. Isto é possível através da utilização da mesma chave de particionamento para os dados relacionados. Normalmente, quando uma entrada é guardada, utiliza-se o código *hash* da chave, mapeando-se esse código para o segmento de *hash* o que faz

com que a entrada seja guardada no nodo dono desse segmento. Posteriormente, é utilizado um algoritmo para localizar a chave, não sendo permitido por isso que os nodos nos quais as chaves estão guardadas sejam definidos manualmente. Este esquema permite que cada nodo saiba onde está o nodo que é dono de uma certa chave, sem que para isso tenha que distribuir a informação do local dos dados (Surtani et al., 2015c). Daqui, podemos concluir que a afinidade de dados tem que ser sempre definida relativamente às chaves das entradas, ou seja, a informação sobre a associação entre dados deve estar sempre presente na classe da chave, bem como a lógica de associação deve ser também aplicada à classe da chave. As soluções IMDG são as responsáveis pelo particionamento, distribuição e replicação de dados sobre o cluster e disponibilizam mecanismos para garantir a afinidade de dados.

Os mecanismos que o Hazelcast oferece em termo de afinidade de dados permitem relacionar chaves de *maps* diferentes, através da criação de uma classe que implementa a interface *PartitionAware*. Esta classe tem que ter um método que devolva a chave de partição (*getPartitionKey()*) que vai decidir a localização dos dados no cluster. Na aplicação com Hazelcast não é possível aplicar diretamente este procedimento, mas pode-se, contudo, seguir a mesma lógica, isto é, os objetos são geridos pelo *Hibernate* que os envolve num envelope próprio, incluindo a chave que está dentro da *CacheKey*, o que faz com que não haja acesso direto às chaves dos *maps*. O que foi feito nesta situação foi criar uma classe com a definição da estratégia de afinidade, que tivesse em atenção a *CacheKey*. Desta forma, as nossas entidades passaram a ter uma chave composta, em que parte dela é a identificação do objeto e a restante parte é referente ao local onde o objeto deve ser guardado - esta chave composta é representada por uma classe (*MyKey*). Por fim, criamos uma classe na qual é definida a estratégia de particionamento em função da chave composta, acrescentando-a de seguida às definições do *Hibernate* a propriedade que indica que a estratégia de particionamento a ser usada deve ser a nossa.

O Infinispan disponibiliza dois mecanismos para garantir uma boa afinidade de dados, o *Key Affinity Service* e a *Grouping API*. O primeiro, permite que o programador decida que entradas colocar num certo nodo do cluster, enquanto que o segundo permite co-localizar um grupo de entradas num certo nodo, porém não podendo escolher qual o nodo em questão. Para que pudéssemos ter as vantagens pretendidas na aplicação utilizámos o *Grouping API*. Ao aplicarmos este mecanismo, o Infinispan vai ignorar o *hash* da chave no momento da decisão do nodo que quer colocar a entrada e, em vez disso, vai usar o *hash* do grupo. Ao se utilizar a *Grouping API* não

vai tornar a aplicação mais lenta, uma vez que o Infinispan continuará a usar nas suas estruturas internas o *hash* da chave.

Quanto aos grupos, estes podem ser intrínsecos à entrada (gerado pela classe da chave) ou extrínsecos (gerado por uma função externa). Na aplicação criada os grupos foram definidos intrinsecamente. Para isso, foi necessário ativar primeiro essa funcionalidade, acrescentando a *tag groups* ao ficheiro de configuração do Infinispan (*infinispan.xml*). De seguida, acrescentou-se às entidades que pretendíamos agrupar a anotação *@Group,* seguida do *getter* que devolve a chave pela qual pretendemos que as entidades sejam agrupadas.

Multigrid

A referência *Multigrid* utiliza-se quando o nosso cenário aplicacional envolve mais do que um cluster de servidores, em locais geográficos distintos. Nestes casos, as operações disponibilizadas por cada IMDG envolvem operações de replicação entre clusters, o que permite que estes sirvam de backup entre si (Surtani et al., 2015d). A replicação pode ser vista como tendo dois modos diferentes, dependendo se for realizada num sentido ou nos dois sentidos entre dois clusters. Estes modos são designados por ativo-ativo e ativo-passivo. O primeiro é maioritariamente utilizado para cenários de falha completa onde o objetivo é replicar o cluster ativo para um ou mais clusters passivos que servem de *backup*. No segundo modo todos os clusters são ativos e replicam-se entre si. Este é normalmente utilizado para ligar diferentes clientes a diferentes clusters, com o objetivo de aproximar os clientes dos servidores, utilizando o caminho mais curto (Hazelcast, 2015a).

A replicação entre diferentes clusters é conseguida no Hazelcast e no Infinispan através de uma definição incorporada nos seus ficheiros de configuração. No caso do Hazelcast é necessário declarar no ficheiro de configuração *hazelcast.xml*, com a *tag 'wan-replication'*, os clusters entre os quais existe comunicações, ou seja, os clusters para os quais os dados vão ser replicados. Se os clusters remotos tiverem uma configuração semelhante, então o modo de operação é ativo-ativo. Neste modo, a mesma entrada pode ser alterada em vários clusters simultaneamente. Para estes casos, deve-se definir uma política de convergência (*merge policy*) para resolver estes conflitos. Como a versão Hazelcast que utilizámos é a versão gratuita, só temos acesso a uma implementação de replicação, nomeadamente a *com.hazelcast.wan.impl.WanNoDelayReplication,* que, na prática, representa uma replicação síncrona entre clusters. Porém, na versão *Enterprise* do

produto é disponibilizada uma implementação de replicação assíncrona ou em *batch*, em que os dados são replicados após um certo número de alterações de dados ter sido realizado ou após um certo período de tempo - ambos os parâmetros referidos são definidos pelo utilizador. Por fim, a configuração de replicação entre clusters pode ser feita por cada *cache-map*,acrescentando uma pequena *tag* com nome a cada *map* existente.

No caso do Infinispan, são necessários três ficheiros de configuração de comunicações: o primeiro, que anteriormente já foi necessário, é o ficheiro que define as comunicações dentro do próprio cluster; o segundo, é o ficheiro no qual estão declarados os clusters existentes e qual o ficheiro a utilizar para comunicar entre clusters; por último, o ficheiro que define as comunicações entre os clusters. Todas as comunicações utilizam o *jGroups* como protocolo de comunicação. Normalmente as comunicações intra-cluster são feitas por UDP enquanto que as inter-cluster são feitas sobre TCP. Para além disto, é ainda necessário especificar no ficheiro de configuração *infinispan.xml* quais os clusters que vão servir de backup, se a replicação é assíncrona ou síncrona, bem como os parâmetros de *timeout* e *failure-policy*. Aquilo que define se a relação entre os clusters é do tipo ativo-ativo ou ativo-passivo são definições semelhantes dos clusters que estamos a definir.

Near *Cache*

Uma *near cache* é uma *cache* local que está mais próxima do processamento, cujo objetivo é garantir um acesso rápido em operações de leitura dos dados que são mais frequentemente usados (MFU) e mais recentemente usados (MRU). Se as ações sobre a *cache* forem principalmente de leitura, então deve-se considerar a criação de uma *near cache* para a *cache*. Desta forma, consegue-se obter maior ganhos no desempenho dos processos de leitura e um menor consumo de tráfego de rede (Oracle, 2015b).

O Hazelcast permite definir *near caches* por *map,* através do seu ficheiro de configuração *hazelcast.xml*. Desta forma é possível configurar o tamanho máximo da *cache*, o tempo de vida de cada entrada, o tempo máximo de inatividade, a *eviction-policy*, as entradas em *cache* que devem ser despejadas caso sejam alteradas, e as entradas locais que devem ficar em *cache*. Esta funcionalidade é recomendável para aplicações que realizam muitas operações de leitura (Hazelcast, 2016c). A configuração da *Near Cache* do sistema protótipo que foi desenvolvido pode ser observada na Figura 16.

```
<near-cache>
      <max-size>0</max-size>
      <time-to-live-seconds>0</time-to-live-seconds>
      <max-idle-seconds>0</max-idle-seconds>
      <eviction-policy>LFU</eviction-policy>
      <invalidate-on-change>true</invalidate-on-change>
      <cache-local-entries>false</cache-local-entries>
</near-cache>
```

Figura 16: Definição da Near *Cache* configurada em Hazelcast.

Para o Infinispan definir uma *near cache* tem-se que adicionar a *cache* opcional de nível 1 ao cliente *Hot Rod,* com o propósito de manter os dados recentemente acedidos mais próximos do utilizador. Esta *cache* L1 é, essencialmente, uma *cache* local do nosso cliente, que é atualizada sempre que uma entrada remota é acedida via operações *get* ou *getVersioned*. Por omissão, os clientes *Hot Rod* não têm a *near cache* ativa. A consistência na *near cache* é conseguida com a ajuda de eventos remotos, os quais enviam notificações quando as entradas são modificadas ou removidas. Da parte do cliente, a *near cache* pode ser configurada como *Lazy* ou *Eager*. Estes dois modos, assim como o número de entradas máximo por *near cache*, são definidos pelo utilizador ao iniciar o cliente *Hot Rod*. No código apresentado na Figura 17 podemos ver como a *near cache* foi configurada para o cliente *Hot Rod* do Infinispan.

```
ConfigurationBuilder config = new ConfigurationBuilder();
        config.addServer().host("127.0.0.1").port(11322);
        config.nearCache().mode(NearCacheMode.LAZY).maxEntries(100);
        manager = new RemoteCacheManager(config.build());
```

Figura 17: Definição da Near *Cache* no cliente Hot Rod.

Persistência e Transações

Uma transação é um conjunto de operações que são tratadas como um único bloco indivisível. Estas seguem quatro propriedades fundamentais, nomeadamente (Medeiros, 2015): atomicidade, propriedade que garante que as operações que são executadas dentro de uma transação devem

ser executadas por inteiro, ou seja, todas as operações devem ser executadas, ou então nenhuma delas o será; consistência, propriedade que garante que uma transação deve respeitar as relações existentes entre os dados envolvidos na transação; isolamento, que garante que as transações que estão a ser executadas em paralelo fiquem isoladas uma da outra, de forma a que o trabalho realizado por uma não seja afetado pela execução da outra e vice-versa; durabilidade, que garante que os dados gravados não são perdidos, mesmo em caso de reinício ou falha no sistema. Estas propriedades são uma parte fundamental no processo de persistência dos dados.

A *Java Persistence API* (JPA) disponibiliza dois mecanismos para suporte a transações: *Java EE* ou com uma implementação *Java Transaction API* (JTA) *open source* - e.g., *JBossTS, Atomikos TransactionsEssentials e Bitronix JTA*. Quando usado com *Java SE* ou num modo não gerido com *Java EE*, o JPA disponibiliza ainda o sua própria implementação de *EntityTransaction* (*Resource Local Transactions*). As transações com JPA são sempre realizadas ao nível dos objetos, o que significa que todas as alterações feitas aos objetos persistidos fazem parte duma transação (Wikibooks, 2016).

No âmbito desta dissertação, pretendeu-se que existissem várias transações a ocorrer em paralelo, uma vez que estamos a utilizar um ambiente distribuído, sendo a camada de persistência comum às várias máquinas envolvidas. Para que isso seja possível necessitamos de ter um gestor de transações distribuídas. Além disso, as transações têm de ser do tipo JTA. A JTA especifica uma interface *Java* standard entre um gestor de transações e as diversas partes envolvidas num sistema de transações distribuídas, nomeadamente o gestor de recursos, o servidor aplicacional e as aplicações transacionais (Oracle, 2015c). Uma transação distribuída é uma transação que acede e altera dados em dois ou mais recursos computacionais ligados em rede e que, por isso, deve ser coordenada entre estes.

Uma transação XA descreve uma interface entre o gestor de transações globais e o gestor de recursos e permite que vários recursos - e.g., bases de dados, servidores aplicacionais, *cache*s transacionais, etc. – possam ser acedidos no contexto de uma mesma transação, isto é, preservando as propriedades ACID entre aplicações envolvidas. Uma transação XA utiliza o protocolo *two-phase commit* (2PC) para garantir que todos os recursos fazem a mesma ação, *commit* ou *rollback* (Hazelcast, 2016d).

Para as aplicações desenvolvidas, com Hazelcast e com Infinispan, foi integrado o *Atomikos* como gestor de transações *open source* conjuntamente com *Hibernate* como implementação de JPA.

O Hazelcast disponibiliza transações XA a partir da implementação da interface *HazelcastXAResource* que, por sua vez, é obtida a partir duma *HazelcastInstance*. Esta interface permite-nos declarar a *cache* de segundo nível como recurso XA. Contudo, como se pretendia que este recurso fosse uma base de dados relacional, tivemos que declarar a base de dados como um *AtomikosDataSourceBean* e depois associá-lo ao servidor de nomes (JNDI). O procedimento que seguimos para incluir esta funcionalidade na aplicação foi o seguinte:

1) Acrescentar a dependência do *Atomikos* que gere as transações quando integrado com *Hibernate*.
2) Alterar o conteúdo do ficheiro *persistence.xml* de forma a alterar o tipo de transações do *persistence unit* para JTA, acrescentar o nome da *data source* declarada no servidor de nomes JNDI, e incluir a propriedade relativa ao *lookup* do gestor de transações.
3) Declarar a *data source* no código de teste e de inicialização de instâncias, antes de fazer qualquer operação sobre os dados.
4) Substituir as transações locais, feitas através da interface *EntityManager,* por transações de utilizador, feitas através da implementação *Atomikos* do *UserTransaction.*

No Infinispan o procedimento é semelhante ao do Hazelcast. Nessa solução IMDG é necessário acrescentar as dependências, alterar o tipo de transação e acrescentar as propriedades relativas ao *lookup* do gestor de transações e à plataforma JTA a utilizar no ficheiro *persistence.xml*. Quanto à parte relativa à definição da base de dados como recurso XA, esta é feita com programação na classe de teste, que com ajuda do servidor de nomes JNDI, se liga aos restantes participantes da aplicação. O restante processo para que as transações sejam do tipo JTA é semelhante aos passos três e quatro do processo descrito acima. Em ambos os casos, é a interface *UserTransaction* que permite que o programador controle os limites das transações através de programação.

4.4 Testes de Desempenho

Com o objetivo de complementar a comparação entre as aplicações implementadas com diferentes IMDG foi realizado um conjunto de testes para a avaliação do seu desempenho. Este estudo integrou quatro testes realizados em ambiente local, numa única máquina *Linux,* baseado em *Debian*. Três dos testes avaliaram a execução dos casos de uso que foram definidos, enquanto que

o quarto teste envolveu a junção dos três anteriores e respetiva avaliação. As especificações da máquina e do *software* que foram utilizados são as seguintes:

Hardware:

- Processador: Intel Core i5-2430M CPU @ 2.40GHz.
- Memória: 3.8 GB.
- Sistema operativo: Ubuntu 14.04 LTS 32 bits.
- Disco rigido: SATA 2.6, 3.0 Gb/s, 5400 RPM.

Software:

- Java: Versão 7 update 95.
- Base de dados: Derby DB 10.11.1.1.

Software Hazelcast:

- Hazelcast versão 3.5.3.
- *Hibernate* versão 3.6.9.Final.
- *Atomikos* versão 3.9.3.

Software Infinispan:

- Infinispan versão 7.2.1.Final.
- *Hibernate* versão 4.0.10.Final.
- *Atomikos* versão 4.0.0M4.

Resultados

Os gráficos apresentados de seguida revelam o desempenho das aplicações nos quatro testes realizados. O desempenho foi medido em nanossegundos e os valores utilizados para os gráficos foram obtidos após a execução de cem vezes de cada caso de uso, tirando os cinco melhores e os cinco piores tempos. Por fim, fizemos a média dos restantes valores obtidos. O primeiro teste realizado (Figura 18) envolveu a execução do caso de uso de recarga. Na lógica de negócio, este representa um carregamento no valor de trinta euros por parte de um cliente que gerará saldo para mais tarde poder usufruir de serviços. Do ponto de vista técnico, durante a execução foi criada uma transação que envolveu dados de cinco tabelas da base de dados e resultou na criação e persistência de um *Asset* e três *Asset Slices,* na primeira execução, e na criação de três *Asset Slices* nas restantes execuções.

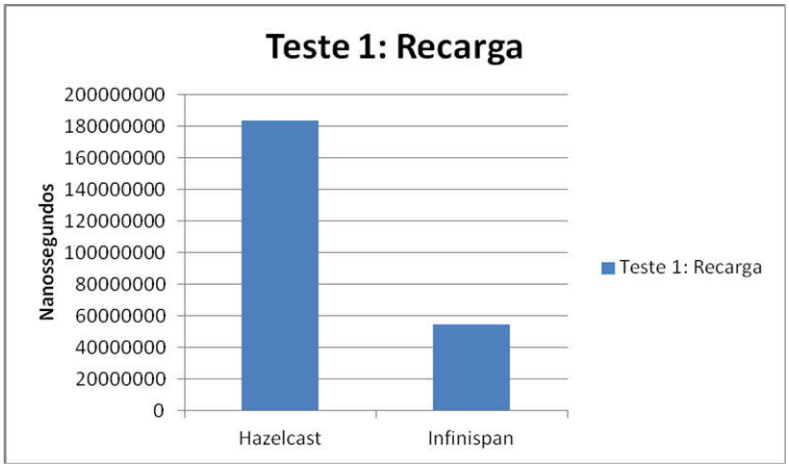


Figura 18: Resultados do Teste 1 – Recarga.

O segundo teste realizado envolveu a taxação da utilização de um serviço por parte de um cliente. Neste caso, definimos uma cobrança de cinquenta cêntimos pela utilização dum serviço de SMS. Do ponto de vista técnico, é o caso de uso mais simples, se ocorrer sem problemas, consulta informação de cinco tabelas da base de dados, atualizando informação em duas dessas tabelas - a única informação atualizada é o saldo do cliente.

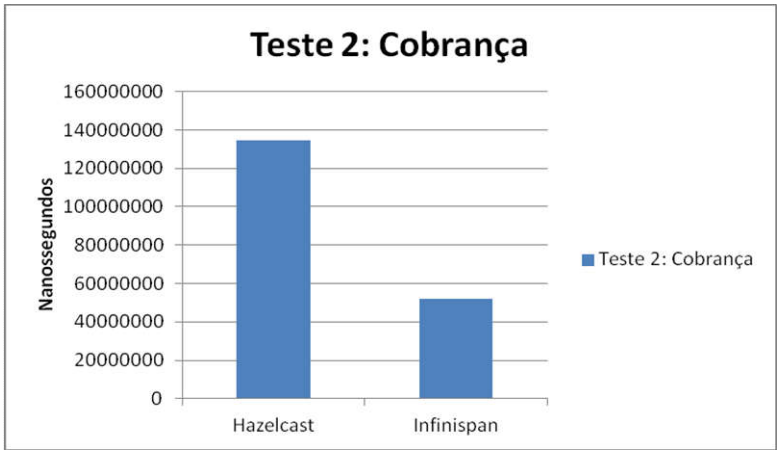


Figura 19: Teste 2 – Cobrança

O terceiro teste foi realizado com o caso de uso da transferência de saldo entre clientes. Este teste pode ser visto como a combinação dos dois testes anteriores. Aqui foram debitados ao primeiro

cliente cinco euros que, de seguida, serão creditados ao segundo cliente. Tecnicamente, neste teste foi acedida informação de todas as tabelas da base de dados, terminando com a criação e persistência do Asset e Asset Slices, que foram gerados no segundo cliente.

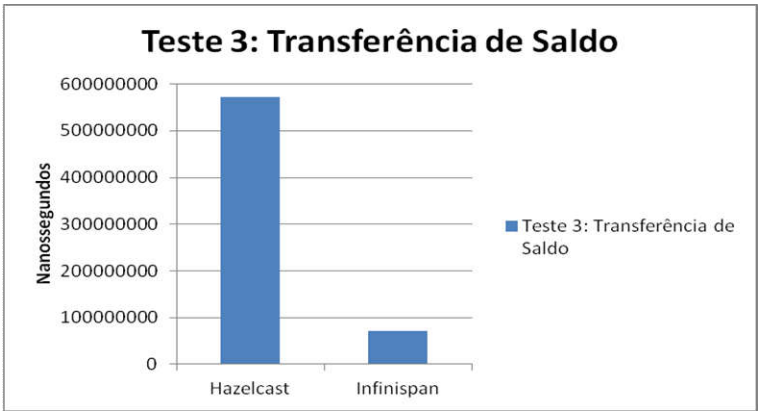


Figura 20: Resultados do Teste 3 - Transferência de saldo.

Por último, como já referido, o teste 4 foi constituído pelos três testes anteriores, que foram executados em série. Este teste começou com um carregamento no valor de trinta euros, realizando depois uma série de trinta cobranças no valor de cinquenta cêntimos cada. Por fim, terminou, fazendo uma transferência de saldo entre os clientes envolvidos. Este foi o teste mais exigente em termos computacionais.

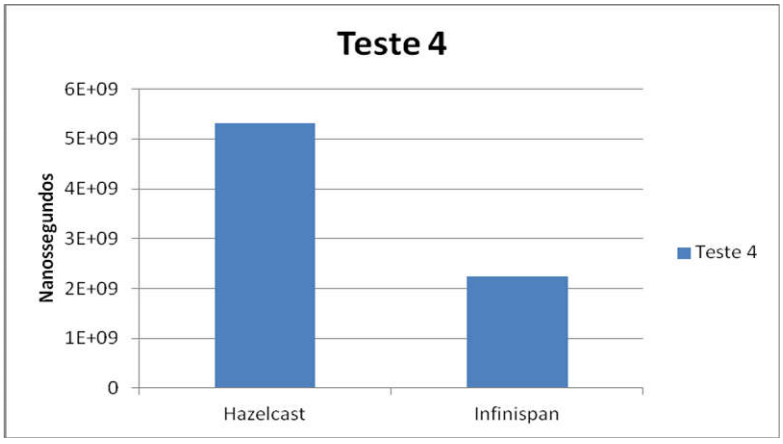


Figura 21: Resultados do Teste 4 - Todos os casos de uso.

As aplicações, assim como as funcionalidades que integram, foram idealizadas para apresentarem bons tempos de resposta quando instaladas num ambiente distribuído. Provavelmente, a falta de tal instalação afetou negativamente os tempos de execução em ambiente local. Funcionalidades como transações globais, execução de tarefas em *background* (no cluster) ou criação de *near cache*, assim como procedimentos internos de gestão e comunicação em cluster, não foram exploradas por estes testes. Mesmo assim, estas funcionalidades ao estarem implementadas influenciam o desempenho da solução desenvolvida em ambiente local.

Capítulo 5

Conclusões e Trabalho Futuro

Alta disponibilidade e coerência de dados são características fundamentais para qualquer plataforma aplicacional dirigida a clientes. Com plataformas a serem sobrecarregadas por pedidos e com um crescimento rápido, o desempenho e a escalabilidade revelam-se também de grande importância, o rápido processamento de dados e pedidos por parte de aplicações de organizações e empresas é essencial para o seu sucesso. Os sistemas IMDG ganharam espaço no mercado devido aos seus elevados níveis de produtividade, satisfazendo as referidas necessidades. Com a vantagem de ser possível manter as infra-estruturas de dados e substituir só a parte aplicacional, estes sistemas têm um menor custo de integração que outras soluções concorrentes – e.g., in memory databases -, porém, os sistemas IMDG pertencentes aos gigantes desta tecnologia podem ter associados custos de licenciamento e manutenção elevados. Este problema dá relevância ao estudo dos sistemas IMDG *open source* para o mercado.

Neste sentido, o trabalho de dissertação foi conduzido de forma a desenvolver duas aplicações materializando assim vários casos de uso que simulam operações bastante frequentes de um sistema de telecomunicações convencional, utilizando como base de dados em memória distribuída as IMDG *Hazelcast* e *Infinispan*. Durante o processo de estudo e análise das soluções IMDG, foram descritos os casos de uso criados, assim como o processo de implementação das aplicações que acolheram cada um desses casos de uso.

Analisando comparativamente o processo de desenvolvimento das duas aplicações, verificaram-se desafios inerentes aos dois sistemas IMDG. O Infinispan peca por ter uma documentação pobre e pouco suporte online pela comunidade de utilizadores, o que torna difícil o desenvolvimento de aplicações sem ter contacto direto com o suporte remunerado oferecido pela *JBoss*. Tendo

também documentação menos completa para casos de integração com outras *frameworks*, como *Hibernate* ou *Atomikos*, exige um conhecimento mais aprofundado por parte de quem desenvolve as aplicações. Requere também conhecimento sobre *JGroups*, utilizado para comunicações entre membros do cluster e entre clusters, sendo necessário criar ficheiros de configuração para o efeito.

No entanto, quando em funcionamento, verificamos que o Infinispan, em ambiente local, é mais rápido e leve que o Hazelcast no processo de inicialização das instâncias, nos processos internos e na execução dos casos de uso. Bem como, a integração de algumas das funcionalidades chave, como a afinidade de dados e execução local dos mesmos em ambiente distribuído, é realizada com menos complexidade. Isto deve-se, respectivamente, à funcionalidade de *Grouping API* disponibilizada pelo Infinispan através de anotações nas entidades, e à integração com o protocolo cliente-servidor *Hot Rod* que, entre outras vantagens, dirige os pedidos aos donos dos dados e nos permite definir uma *Near Cache* para os clientes. Estas duas funcionalidades no Infinispan foram detalhadas nesta dissertação.

No que diz respeito ao Hazelcast, a sua maior desvantagem nesta aplicação foi a necessidade de utilização de chaves compostas, que vieram acrescentar complexidade e trabalho. Para a integração de chaves compostas com JPA decidimos utilizar uma *@IdClass*, que estabelece uma classe identificadora das entidades, sendo esta aproximação ao problema desaconselhada na documentação do Hibernate (Bernard, 2010).

Ao não ser suportada a função *@GeneratedValues* dentro de uma *IdClass*, fomos obrigados a abdicar da geração automática dos valores do identificador único de cada entidade aquando da criação das *Assets* e *Asset Slices*. As entidades que tinham relações *@ManyToOne* ficaram também mais complexas uma vez que as chaves estrangeiras fazem referência a duas colunas em vez de uma, neste sentido foi necessária a utilização das anotações *@JoinColumns* afim de solucionar o problema.

Contudo, o Hazelcast apresenta uma documentação bastante completa, incluindo material para integração com frameworks externas, assim como uma comunidade de utilizadores ativa com fóruns de suporte e resposta rápida em caso de necessidade, exigindo uma menor intervenção e configuração por parte do utilizador, o que facilita o processo de desenvolvimento de aplicações.

Em relação aos testes de desempenho, como já foi referido, foram realizados em ambiente local numa só máquina. Estes servem apenas como mais uma referência de comparação e não como *benchmarking* definitivo, pois não tiram proveito das funcionalidades implementadas pensadas para bons desempenhos em ambiente distribuído. Ainda assim, processos como mapeamento objeto-relacional, persistência para disco ou gestão de memória, são contabilizados nestes testes. Nestes, podemos concluir que o Infinispan obteve melhores desempenhos.

De forma a dar continuidade a este trabalho e a aproximá-lo mais da realidade, explorando a potencialidade destes sistemas, numa próxima iteração deveríamos desenvolver as soluções IMDG para execução num ambiente distribuído sobre um cluster de servidores. Nessa altura, teriam que testados vários aspectos da solução implementada, como a coerência de dados, tolerância a falhas, co-localização de dados, distribuição de computação entre outras características inerentes a sistemas distribuídos. Posteriormente, deveríamos realizar alguns processos de *benchmarking* e testes de desempenho standard de forma a poder comparar os dois sistemas.

Não obstante as várias diferenças entre os sistemas IMDG, ambas as aplicações integram JPA e JTA e concretizam as expectativas funcionais requeridas. Para além da escalabilidade, alta disponibilidade, fiabilidade, tolerância a falhas e consistência dos dados em memória, as aplicações estão preparadas para oferecer bons desempenhos e persistência para disco com transações distribuídas. Desta forma podemos concluir que as IMDGs *open source* conseguem ser funcionalmente adequadas às exigências do mercado assim como os produtos concorrentes que exigem grandes custos.

Bibliografia

Bernard, E., 2010. Hibernate Annotations [WWW Document]. URL https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/#entitymapping-identifier (accessed 1.15.16).

Oracle, 2015a. Coherence Developer's Guide - Data Affinity [WWW Document]. URL https://docs.oracle.com/cd/E18686_01/coh.37/e18677/api_dataaffinity.htm#COHDG131 (accessed 4.15.16).

Oracle, 2015b. Coherence Getting Started Guide - Near *Cache* [WWW Document]. URL https://docs.oracle.com/cd/E24290_01/coh.371/e22840/near*cache*.htm#COHGS227 (accessed 4.15.16).

Colmer, P., 2011. In Memory Data Grid Technologies - High Scalability - [WWW Document]. URL http://highscalability.com/blog/2011/12/21/in-memory-data-grid-technologies.html (accessed 8.30.15).

GridGain, 2013. In-Memory Data Grid (WHITE PAPER). GridGain Systems.

Hazelcast, 2016a. Hazelcast Deployment and Operations Guide (Guide). Hazelcast Inc.

Hazelcast, 2016b. Hazelcast Documentation - Distributed Computing [WWW Document]. URL http://docs.hazelcast.org/docs/3.5/manual/html/execution.html (accessed 4.14.16).

Hazelcast, 2016c. Hazelcast Documentation - Near *Cache* [WWW Document]. URL http://docs.hazelcast.org/docs/3.5/manual/html/map-near*cache*.html (accessed 4.15.16).

Hazelcast, 2015.Hazelcast Documentation - WAN Replication [WWW Document], 2015. URL http://docs.hazelcast.org/docs/3.5/manual/html/wan.html (accessed 4.15.16).

Hazelcast, 2016d. Hazelcast Documentation - XA Transactions [WWW Document]. URL http://docs.hazelcast.org/docs/3.5/manual/html/xatransactions.html (accessed 4.16.16).

Hazelcast, 2015b. Hazelcast – The Operational In-Memory Computing Platform [WWW Document]. URL https://hazelcast.com/products/ (accessed 10.10.15).

Oracle, 2015c. Java Transaction API (JTA) [WWW Document]. URL http://www.oracle.com/technetwork/java/javaee/jta/index.html (accessed 4.15.16).

Kaufman, M., Kirsch, D., 2014. Improving Application Scalability with In-Memory Data Grids (WHITE PAPER). HURWITZ.

Kirby, T., Matthew, J., Stone, G., 2009. IBM WebSphere eXtreme Scale V7: Solutions Architecture (WHITE PAPER). IBM Redpaper publication.

Medeiros, H., 2015. Java Transaction API (JTA) na Plataforma Java EE 7 [WWW Document]. URL http://www.devmedia.com.br/java-transaction-api-jta-na-plataforma-java-ee-7/31820 (accessed 4.15.16).

Prins, R., 2013. In-Memory Data Grids - DZone Big Data [WWW Document]. dzone.com. URL https://dzone.com/articles/memory-data-grids (accessed 8.24.15).

Red Hat, 2015. IN-MEMORY DATA GRIDS - The perfect solution for big data and application performance (WHITE PAPER). redhat.

Surtani, M., Markus, M., Zamarreño, G., Muir, P., 2015a. Infinispan User Guide - Server Modules [WWW Document]. URL http://infinispan.org/docs/7.0.x/user_guide/user_guide.html#_server_modules (accessed 4.14.16).

Surtani, M., Markus, M., Zamarreño, G., Muir, P., 2015b. Infinispan User Guide - Distributed Executor [WWW Document]. URL http://infinispan.org/docs/7.0.x/user_guide/user_guide.html#DistributedExecutor (accessed 4.14.16).

Surtani, M., Markus, M., Zamarreño, G., Muir, P., 2015c. Infinispan User Guide - Data Affinity [WWW Document]. URL http://infinispan.org/docs/7.2.x/user_guide/user_guide.html#_the_grouping_api (accessed 4.14.16).

Surtani, M., Markus, M., Zamarreño, G., Muir, P., 2015d. Infinispan User Guide - Cross Site Replication [WWW Document]. URL http://infinispan.org/docs/7.2.x/user_guide/user_guide.html#CrossSiteReplication (accessed 4.14.16).

Wikibooks, 2016. Java Persistence/Transactions - Wikibooks, open books for an open world [WWW Document]. URL https://en.wikibooks.org/wiki/Java_Persistence/Transactions#JTA_Transactions (accessed 4.15.16).