FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Redundância de Dados para uma Solução de Voz Sobre IP

Tiago Lúcio Azeredo Lobo de Oliveira Miranda

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Professor Ricardo Santos Morla

23 de junho de 2017

Redundância de Dados Para uma Solução de Voz Sobre IP

Tiago Lúcio Azeredo Lobo de Oliveira Miranda

Mestrado Integrado em Engenharia Informática e Computação

23 de junho de 2017

Resumo

O tema principal deste projeto visa a resolução de determinados tipos de limitação que ocorrem num sistema *multi-site* de processamento de chamadas de voz sobre IP que lida com sistemas de bases de dados relacionais. Tais limitações surgem devido ao mecanismo de replicação de dados utilizado para transportar a informação contida nas bases de dados de um *site* para outro, geograficamente disperso.

A metodologia de trabalho adotada passou pela análise a determinados produtos de gestão de bases de dados, bem conhecidos no mercado, e posterior avaliação de prós e contras da migração dos atuais sistemas. Após esta fase, iniciou-se a construção de uma nova arquitetura de replicação com base num *plugin* aplicado nas bases de dados pré-existentes, e noutras modificações que se mostraram necessárias.

Através deste novo mecanismo, prevê-se que os atuais sistemas em produção possam beneficiar de uma atualização e eliminar completamente as limitações de serviço existentes em determinados casos. Futuramente, outros componentes do produto poderão beneficiar de uma solução semelhante para fazer face a problemas idênticos.

Palavras-chave: comunicações convergentes; voip; base de dados; replicação.

Abstract

The main goal of this project is to solve certain types of limitations that occur on multi-site VoIP call processing systems that work with relational databases. Those limitations are related with the replication method used to carry information from the database located on one site to the database located on a geographically distant opposite site.

The adopted work methodology consisted on a first phase of analysis regarding well known database management systems, followed by an evaluation of the relevance of a technology migration. Next phase involved the creation of a new replication architecture based on a plugin installed on the existing databases, and subsequent changes that appeared to be mandatory.

Using this new mechanism, we can anticipate the benefit that actual production systems may get from this update that completely solves the service limitations that exist in certain scenarios. In the future, other components from the same product could benefit from a similar solution as well, to help face identical problems.

Keywords: convergent communications; voip; database; replication

Agradecimentos

Devo uma palavra de gratidão àqueles que, durante estes cinco meses de trabalho, me ajudaram a atingir o resultado final desta dissertação.

Fica aqui mesmo a minha manifestação de agradecimento à Altice Labs, em especial nas pessoas do Engenheiro José Ribeiro, do Engenheiro Joaquim Azevedo e do Engenheiro Miguel Biscaia, pelo apoio, pela paciência e pelo precioso conhecimento que me transmitiram.

Agradeço também à equipa da InovaRia pela prestabilidade e pela oportunidade que me deu de integrar um projeto como este.

Agradeço-te também, Isabel, pela força, motivação, companhia e pela extraordinária estabilidade que me transmitiste e continuas a transmitir.

Obrigado.

Tiago Lúcio Miranda

"If you believe very strongly in something, stand up and fight for it."

Roy T. Bennett

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

ABC	Advanced Business Communications
SEC	Serviço Empresarial Convergente
VoIP	Voice over Internet Protocol
IP	Internet Protocol
SIP	Session Initiation Protocol
IMS	IP Multimedia Subsystem
PBX	Private Branch Exchange
PT	Portugal Telecom
M2M	Machine to Machine
IoT	Internet of Things

Capítulo 1

Introdução

1.1. Contexto e Enquadramento

Inserimo-nos numa sociedade amplamente conectada, sendo que, em média, por entre os países considerados como possuindo uma economia desenvolvida, observa-se que cerca de 68% da população possui pelo menos um smartphone, e que 87% dessa mesma população utiliza a Internet pelo menos ocasionalmente [2]. Atendendo a estes números, é expectável que, em muitas situações, cada indivíduo possua mais do que um meio de comunicação, resultando em várias formas de o contactar.

Focando o caso dos cenários empresariais, é comum cada indivíduo possuir um telemóvel, um telefone no seu gabinete e ainda, por vezes, um fax, fazendo com que exista uma grande dispersão dos meios de comunicação existentes. Do ponto de vista de uma Empresa, adquirir, gerir e manter estes sistemas torna-se complexo e dispendioso, sendo que muitas vezes o custo destas operações se torna impeditivo do acesso às tecnologias.

Como sendo qualquer sistema em larga escala, existem também necessidades ao nível da resiliência do serviço, sendo que a principal razão para o desenvolvimento do presente trabalho é a melhoria da resistência deste sistema de comunicações unificadas a falhas de certos componentes, sendo que será dada especial ênfase à redundância no sistema de base de dados.

Esta dissertação foi realizada em parceria com a Altice Labs que é uma empresa altamente orientada à inovação na área das telecomunicações digitais avançadas, desenvolvendo soluções que visam a criação de novas tecnologias e serviços orientados não só às empresas, mas também às pessoas.

1.2. A Empresa Altice Labs

Ao longo de décadas foi-se traçando um caminho de inovação no que toca aos serviços de telecomunicações em Portugal. Esse caminho começou por volta dos anos 50 e desde essa época até aos dias de hoje inúmeras soluções foram criadas e colocadas em funcionamento, levando o país a ocupar os primeiros lugares ao nível da qualidade tecnológica das suas redes de telecomunicações.

Durante muitos anos o percurso da então PT Inovação, sediada em Aveiro, trouxe grandes novidades em diversas áreas das comunicações, apostando nos mais variados mercados, orientados não só ao cliente final, mas também ao mundo empresarial, trazendo soluções como comunicação "Máquina a máquina" (M2M), um serviço de comunicações unificadas, o Serviço Empresarial Convergente (SEC), que posteriormente alterou a sua designação para Advanced Business Communications (ABC) derivado à expansão internacional, e soluções de *hardware* inovador adaptado ao funcionamento com os seus serviços.

Em 2015 a compra da PT Portugal pela Altice impulsionou a criação da Empresa Altice Labs que veio substituir a antiga PT Inovação, criando mais uma vez um marco de expansão e inovação e dando continuidade à estratégia de liderança tecnológica em áreas de negócio como as tecnologias *Cloud*, *Smart Living*, Internet das Coisas (IoT), *Big Data*, Serviços Digitais e Redes do Futuro, apostando fortemente na componente de I&D que, segundo a Empresa, "*permite transformar o conhecimento em inovação tecnológica para criar diferenciação e valor no mercado*" [1].

1.3. Objetivos e Contribuições

Com esta dissertação pretende-se encontrar uma abordagem que permita a resolução de determinados problemas relacionados com o armazenamento em bases de dados de determinada informação que decorre do normal funcionamento da aplicação em causa. Estes problemas estão relacionados com o método de transporte de dados a operar na atual solução implementada no produto, causando em última instância indisponibilidades de serviço para o utilizador final.

No decorrer do trabalho foram estudadas diversas soluções que não se adaptaram, até que se optou pela adaptação de uma abordagem de replicação já existente, a replicação lógica, para

que, em conjunto com diversas modificações à aplicação se alcançasse o pretendido, que seria a eliminação de indisponibilidades.

A arquitetura da solução conta com um mecanismo de replicação lógica, assíncrona, no entanto, de interação constante, e com paragem não sinalizada.

Adiante se descreverá em profundidade a solução criada, bem como se detalharão os diversos componentes e configurações.

1.4. Estrutura

Para além da introdução, esta dissertação está dividida em mais seis capítulos.

No capítulo 2 é apresentado o estado da arte e trabalhos relacionados com o assunto, bem como um plano tecnológico geral onde incidiu o estudo desta dissertação.

No capítulo 3 é feita uma descrição do produto em causa e do assunto que motivou a realização da dissertação, e ainda do problema concreto para o qual se procura uma solução.

No capítulo 4 descreve-se o processo de investigação de diversas abordagens que pudessem constituir possíveis soluções.

No capítulo 5 é detalhado e explicado cada ponto da implementação, incluindo todos os pormenores técnicos e funcionais, cujos testes de funcionalidade foram levados a cabo e relatados no capítulo 6 desta dissertação.

Finalmente, no capítulo 7 descrevem-se as conclusões alcançadas por via da realização deste trabalho, deixando em aberto possibilidades de expansão e melhoria do mesmo.

Capítulo 2

Trabalhos Relacionados

Neste capítulo são apresentados os conceitos necessários para uma melhor compreensão do tema apresentado e a forma como estes se relacionam. Consiste na descrição dos vários temas que foram necessários explorar e estudar para o desenvolvimento desta dissertação.

Primeiramente é feita uma descrição do cenário tecnológico que envolve o presente trabalho: uma análise das duas grandes vertentes ao nível das tecnologias existentes no mundo dos sistemas de gestão de bases de dados, NoSQL e Relacional. De seguida são comparadas algumas soluções que se aproximam mais do objeto central de estudo que são os processos de replicação de informação.

2.1. Bases de Dados Relacionais: Garantia de Consistência

Remonta ao fim dos anos 60 o início da caminhada que traçou aquilo que viria a ser uma das tecnologias com mais sucesso nos anos seguintes, no campo do armazenamento de informação. Os primeiros sistemas de gestão de bases de dados, na altura altamente complexos e pouco facilitadores do ponto de vista da interação com a informação que armazenavam, não apresentavam uma clara distinção entre a forma como um programador representava logicamente a informação da forma como esta era fisicamente representada e armazenada; existia a necessidade de lidar com conceitos relacionados com o armazenamento físico, o que tornava a utilização destes sistemas mais um peso por via da avultada quantidade de conceitos adicionais com os quais um programador deveria lidar. De um ponto de vista atual, estes sistemas de gestão de bases de dados apresentavam, segundo E. F. Codd [3], três falhas centrais: os programadores que utilizavam tais sistemas viam-se obrigados a lidar com numerosos conceitos de baixo nível

que pouco se relacionavam com o foco central do seu trabalho, levando a um esforço adicional muitas vezes desnecessário, no âmbito das suas aplicações; o segundo problema está relacionado com a falta de capacidade destes sistemas de processar conjuntos de dados, levando mais uma vez a que o programador seja forçado a escrever código baseado em processos iterativos, desnecessários aos olhos dos dias de hoje; por último, o já mencionado problema da falta de uma linguagem de interação com os dados, isto é, a tecnologia não possuía uma forma bem definida e estruturada de interrogar a informação em si guardada. Estas e outras questões levaram a que os primeiros sistemas de gestão de bases de dados fossem vistos como altamente ineficientes e pouco impulsionadores da produtividade.

A instalação de sistemas deste género mostrava-se altamente morosa, complexa, pelo que na maioria das situações se procurava executar um planeamento tão detalhado quanto possível para que esta fosse feita como que "de uma vez por todas", evitando subsequentes alterações significativas ao nível das aplicações.

Desta forma, a investigação e o aperfeiçoamento destes sistemas avançou com vários objetivos em vista. Também segundo E. F. Codd [3], o mais importante dos quais seria a estruturação e a definição de uma clara divisão entre os aspetos lógicos e físicos de uma base de dados, incluindo áreas como a arquitetura e a manipulação e recolha de dados. Paralelamente, o objetivo de criar um modelo de sistema que fosse estruturalmente simples e inteligível por qualquer programador estava também no horizonte, juntamente com a introdução de um esboço de linguagem de alto nível que permitisse manipular conjuntos de informação de uma única vez com apenas um comando.

Com o avanço dos trabalhos na área das bases de dados relacionais, foi necessário o abandono de alguns conceitos que previamente serviam de base ao armazenamento de informação; a forma como a informação era acedida dependia grandemente de uma lógica posicional, isto é, esta era acedida por intermédio da sua posição, em memória, em disco, etc.
A grande viragem deu-se aquando da criação de um modelo associativo em que um conjunto de dados se associava por intermédio de um nome, uma chave primária ou um atributo. Desta forma, a informação passaria a ser vista como que organizada em tabelas, contrariando a conceção de relacionamento de dados por intermédio de estruturas de dados ligadas [3].

No desenvolvimento do modelo relacional, a integridade dos dados foi tida como um aspeto importante a ter em conta, bem como a possibilidade de manipulação desses mesmos dados por intermédio de operações algébricas como SELECT, JOIN, entre outras. Com o objetivo de aumentar o espetro de aplicabilidade de tal tecnologia, os sistemas de gestão de bases de dados suportariam diferentes sub-linguagens que permitiam a interface entre os dados e diversos tipos de linguagens de programação. Tal abordagem era altamente vantajosa de diversos pontos de vista: do ponto de vista do programador que procura erros na execução de uma aplicação, já que a base de dados passaria a ter uma linguagem própria, permitindo testar separada e isoladamente

comandos a ser incorporados em aplicações; do ponto de vista de todos os envolvidos de alguma forma nas aplicações, sejam programadores, utilizadores finais, analistas, gestores, torna-se vantajoso pela forma inequívoca como se passariam a referir a assuntos relacionados com bases de dados, facilitando assim a comunicação. Por fim, toda a aprendizagem necessária anteriormente para lidar com estruturas armazenadas em memória, de uma forma posicional e não relacional, seria deixada para quem, de facto, necessitava de trabalhar com tais estruturas [3].

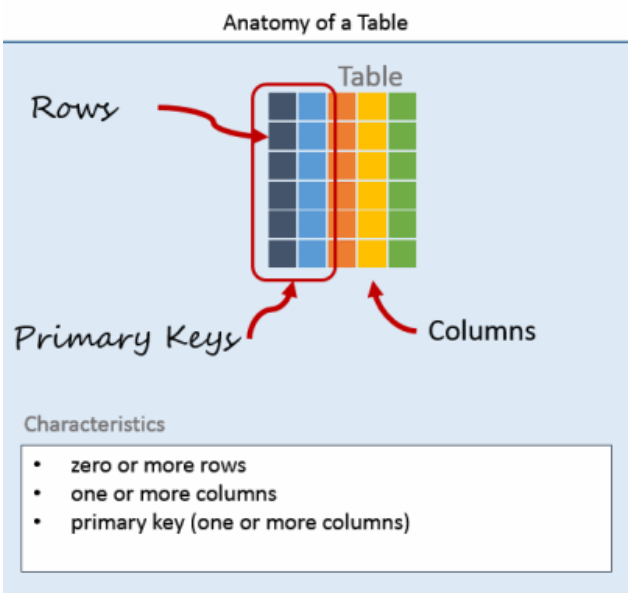

Fig. 2.1 – Exemplo de uma estrutura tabular [4]

Desenvolvidos e aperfeiçoados ao longo do tempo estes sistemas de gestão de bases de dados relacionais, é seguro afirmar que um sistema completo deste tipo providencia as seguintes funcionalidades, segundo E. Codd [3]:

- Armazenamento, recolha e atualização de dados;

- Uma panóplia de tipos de dados acessíveis ao utilizador que permitem a definição dos dados armazenados;

- Suporte a transações que garantem que as alterações a um conjunto de dados apenas se refletem na base de dados caso toda a sequencia de alterações esteja concluída, caso contrário não existirá qualquer alteração;

- Mecanismos de recuperação em caso de falhas diversas;

- Serviços de autenticação e autorização que garantem que a manipulação dos dados é feita de acordo com a especificação de direitos e permissões;

- Suporte aos meios de comunicação de dados típicos;

- Mecanismos de garantia de integridade, que forçam a correta manipulação dos dados segundo as especificações do modelo de dados;

Atendendo a estes atributos, será possível considerar os sistemas de bases de dados relacionais como sendo sistemas altamente consistentes e íntegros do ponto de vista da informação armazenada. Este tipo de valências pode ser extremamente útil e favorável a determinados ambientes.

2.2. NoSQL: uma solução para grandes problemas

As bases de dados NoSQL (Not Only SQL) têm vindo a tornar-se uma sólida solução para os casos de aplicações onde é necessário movimentar grandes quantidades de dados em tempo real, normalmente orientadas à web como é o caso do Facebook, Amazon ou LinkedIn, e em que se torna impraticável a utilização de bases de dados relacionais para armazenamento dessa mesma informação [5].

As limitações das bases de dados relacionais começaram a fazer-se sentir em serviços prestados por Empresas de grande dimensão, pelo que foi necessário iniciar a procura por soluções que se diferenciassem fortemente do paradigma até então. Desta forma começaram rapidamente a surgir os primeiros projetos de bases de dados não relacionais, que prometiam conseguir acomodar as necessidades de armazenamento de informação que surgiam, baseando-se em três premissas: *Big Data, Big Users and Cloud Computing* [6]. *Big Data* é um conceito que se observa nos dias de hoje que indica que lidamos com grandes volumes dos mais variados dados, sejam estes pessoais, de localização, registos de funcionamento de aplicações, conteúdo dos utilizadores, entre outros, e que o acesso e transmissão desses dados se tem tornado cada vez mais fácil e útil nos mais diversos fins. Relacionado com este último, o conceito de *Big Users* dá a entender a grande quantidade de utilizadores com acesso a todo o tipo de serviços e informação através da Internet, podendo um serviço facilmente atingir os milhões de visitas diárias. Não menos relacionado com as duas perspetivas anteriores, o conceito de *Cloud Computing* deixa para trás a realidade de um único utilizador a trabalhar no seu computador pessoal, acedendo a certos serviços numa arquitetura maioritariamente cliente-servidor. Essa realidade evoluiu para milhões

de utilizadores a aceder a serviços distribuídos, disponíveis 24 horas por dia, 365 dias por ano, gerando e consumindo grandes quantidades de informação que necessita de ser tratada e armazenada em tempo real, evolução esta que foi definida por V. G. Cerf como uma avalanche de informação [7].

Dado o facto de esta "avalanche de informação" se situar nos dias de hoje na escala dos *petabytes* de dados, torna-se necessário e de certa forma urgente migrar para soluções que permitam lidar de forma altamente disponível com tal volume de informação, distribuída por diversos servidores, muitas vezes geograficamente dispersos, ainda que possam surgir problemas relativos à consistência dos dados armazenados ao longo de todo o *cluster* da base de dados. É interessante perceber que, com a mudança de cenário descrita acima, surgem permanentemente novos desafios no que toca à arquitetura dos sistemas, e que por vezes não é possível alcançar a funcionalidade perfeita e que consiga responder a toda e qualquer questão advinda desta nova realidade. Por forma a descrever mais detalhadamente esta limitação, o teorema CAP (*Consistency (C), Availability (A), Partition (P)*), de Eric Brewer, mostra como apenas é possível alcançar duas das três características ideais em qualquer sistema distribuído, sendo tarefa do responsável pelo desenho da implementação ponderar quais as características que requerem mais atenção [8].

2.3. O Teorema CAP (Consistency, Availability, Partitioning)

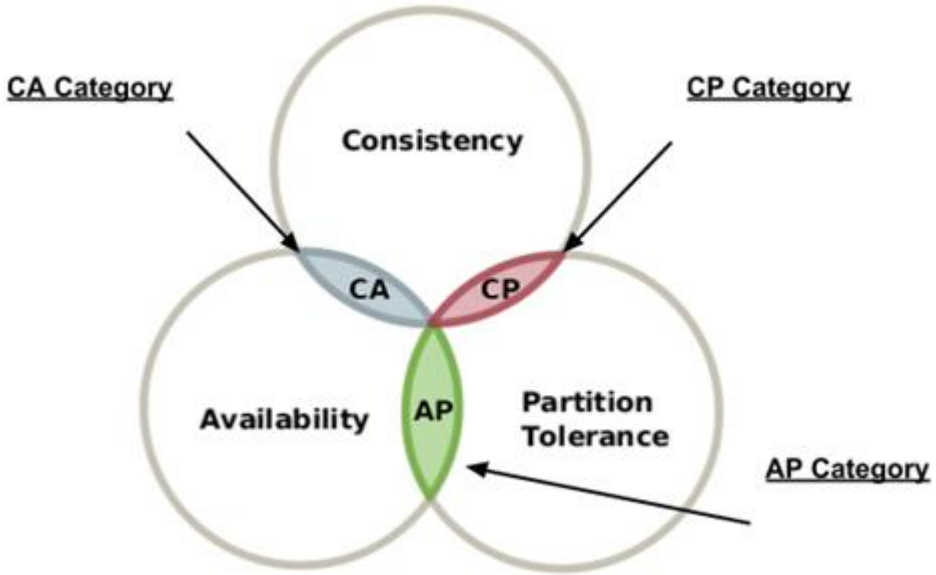

Fig. 2.2 – Representação esquemática do teorema CAP [9]

Este teorema, graficamente representado acima, mostra a interação dos diferentes campos de confiabilidade de um qualquer sistema distribuído:

- **Consistência (C)**: todos os nós devolvem a mesma versão dos dados a um cliente no caso de uma interrogação. Equivalente a ter apenas uma cópia dos dados;

- **Disponibilidade (A)**: em caso de falha de um, ou mais, dos nós que compõe o *cluster* que disponibiliza o serviço, o sistema conseguirá continuar em operação sem que sejam percebidas falhas pelos clientes;

- **Tolerância a Partições (P)**: o sistema é tolerante a partições da rede de comunicação, isto é, falhas na comunicação ou altas latências.

Transportando o teorema para os sistemas de gestão de bases de dados NoSQL, verifica-se que, pelo facto de no centro do esquema não existir espaço de interseção das três circunferências, não é possível criar um sistema que alcance os três níveis de confiabilidade em simultâneo. Por norma, os responsáveis pelo desenho de uma arquitetura distribuída optam por reforçar a disponibilidade (A) em detrimento da consistência (C): é preferível um sistema que está sempre disponível para atender pedidos dos clientes e que, por vezes, poderá devolver uma reposta desatualizada, a um sistema de oferece 100% de consistência dos dados, mas não tolera a falha de qualquer um dos nós, originando indisponibilidade ou perda de informação [9].

2.4. Replicação de Informação

Existem várias razões que levam à necessidade de replicar a informação contida numa base de dados; a mais básica das quais sendo a de obter uma cópia dos dados para, em caso de perda da informação original, a cópia servirá para repor a informação. Este processo é comummente denominado de *backup*.

2.4.1. Classificação dos Métodos de Replicação

Wiesmann et al. [10] consideram que a maioria dos protocolos existentes de replicação se encaixam num método de classificação definido por três parâmetros, como sendo a arquitetura

de replicação, que identifica a forma como as transações poderão ser executadas nos sistemas envolvidos; a forma como as alterações são propagadas pelos restantes participantes, e o protocolo de sinalização da terminação de uma transação.

Através da diversidade de combinações de características que é possível obter por meio deste tipo de classificação, entende-se que consoante o tipo onde se encaixa determinado método de replicação, este irá possuir diferentes requisitos ao nível da infraestrutura de comunicação ou ao nível do sistema de base de dados em si. De uma forma genérica, ao nível da infraestrutura de comunicação, os requisitos prendem-se maioritariamente com o ordenamento das mensagens e a uniformidade e estabilidade da rede; já ao nível da base de dados, o requisito principal é, normalmente, o determinismo das operações, isto é, no final da execução de uma operação incluída numa transação, deve ser sempre possível determinar com certeza a posição da mesma no seu histórico de serialização.

2.4.1.1. Arquitetura dos Processos de Replicação

Partindo agora à definição dos três parâmetros apresentados em [10], o primeiro deles relaciona-se com a arquitetura do sistema de replicação.

Para melhor definir as diferentes arquiteturas, Gray et al. [11] mostram a existência de duas técnicas:

- **Cópia Primária**: cada item existente na base de dados está obrigatoriamente associado a apenas um nó, obrigando a que qualquer atualização a essa informação seja efetuada por intermédio desse nó, denominado de primário, que posteriormente enviará o resultado da operação para os restantes nós. Esta arquitetura apresenta não só a desvantagem de introduzir um ponto único de falha no mecanismo, bem como o efeito "gargalo" ao nível do desempenho.

- ***Update Everywhere***: esta técnica permite que atualizações a um item presente na base de dados sejam feitas em qualquer ponto do sistema, isto é, possibilita a ocorrência de alterações simultâneas aos mesmos dados. Derivado desta propriedade, esta abordagem de replicação possibilita uma simplificada mitigação de falhas, já que não existe a necessidade de eleições para dar continuidade ao processamento da informação. Da mesma forma, e contrariamente ao esquema de cópia primária, esta arquitetura não introduz limitações ao nível do desempenho. No entanto, poderão existir preocupações ao nível do aumento de carga nos sistemas, como se analisará adiante.

2.4.1.2. Interação Entre Nós Replicantes

O segundo parâmetro que se considera relaciona-se com o grau de comunicação entre os diversos servidores de base de dados durante uma transação. Este parâmetro está, portanto, intimamente relacionado com a quantidade de tráfego de rede que é gerado pelos algoritmos de replicação e também com o custo acrescentado ao processamento de uma transação; expressa-se através do número de mensagens necessárias à execução das transações, excluindo eventuais mensagens de terminação.

O grau de interação divide-se nos dois seguintes tipos:

- **Interação Constante**: corresponde às técnicas que envolvem um constante número de mensagens utilizadas para sincronização dos servidores para uma determinada transação, independentemente do número de operações incluídas na transação. Tipicamente, protocolos nesta categoria trocam apenas uma mensagem incluindo um agrupamento de toda a transação e suas operações;

- **Interação Linear**: o número de mensagens transmitidas utilizando este tipo de técnica é proporcional ao número de operações inseridas numa única transação; desta forma, escala ou reduz linearmente não só o número de mensagens, mas também o tráfego de rede envolvido na transmissão das mesmas;

Como se irá aprofundar adiante, o tipo de interação entre os servidores tem um peso significativo no tráfego de rede gerado, peso esse que será agravado pelo incremento de servidores participantes num processo de replicação, podendo, em certos casos extremos trazer complicações caso o sistema não seja corretamente desenhado e dimensionado para contar com a acrescida carga da replicação.

2.4.1.3. Sinalização de Término de Transação

Por último, considera-se que os sistemas se dividem ainda pelo tipo de sinalização que efetuam junto dos restantes nós aquando do término de uma transação, ou, por outras palavras, que garantia ao nível da atomicidade das operações estes sistemas são capazes de providenciar.

Assim, destacam-se dois casos:

- **Terminação Votante**: requer uma ronda adicional de mensagens de comunicação, com o objetivo de coordenar as diferentes réplicas e garantir que o resultado é consistente entre as

mesmas. Só após isto a operação de *commit* será executada. Este protocolo ainda poderá ser tão simples como uma mensagem de confirmação, ou complexo tal como o processo de "*Two-Phase Commitment*" [12];

- **Terminação Não-Votante**: implica que cada nó decida se deve efetuar *commit* à operação ou a deve terminar. Desta forma, é exigido um comportamento determinístico por parte de todos os nós, no entanto, isto apenas afeta as operações serializadas localmente, isto é, operações que não causem nenhum tipo de conflito, poderão ser executadas em ordens e momentos distintos nos diversos nós.

Sumarizando as técnicas de replicação acima descritas, Wiesmann et al. [10] apresentam uma tabela que o faz de forma inequívoca, e que a seguir se apresenta:

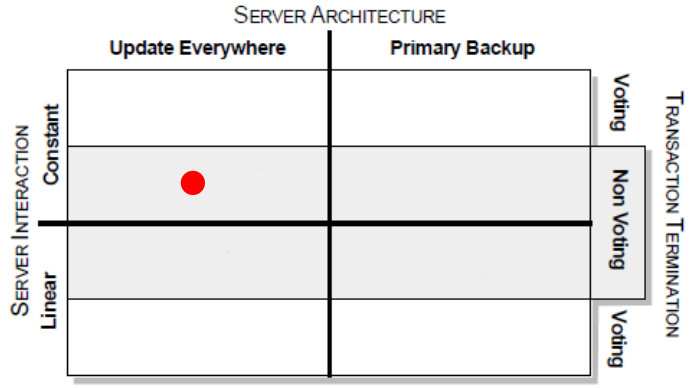

Fig. 2.3 – Tabela de relação dos mecanismos de replicação

Na figura representa-se a tabela que os autores utilizaram para sumarizar e relacionar as diversas combinações possíveis das características que neste documento já se enunciaram. Um sistema que se encaixe numa determinada área, apresentará um conjunto específico de requisitos para a sua operação.

Considera-se que o sistema implementado nesta dissertação se encaixará na área assinalada pelo círculo a vermelho, sendo que atende às características respetivas da mesma.

2.4.2. Análise de Carga em Sistemas com Replicação

Existe ainda uma outra questão levantada por Gray et al. em "*The Dangers of Replication and a Solution*" [11] que evidencia o escalamento acelerado do número de transações por segundo num sistema de base de dados, à medida que se adicionam réplicas.

Gray et al. [11] apresentam um modelo que permite apurar o impacto dos mecanismos de replicação nos sistemas de base de dados. Focando assim este aspeto, é exposto um cenário em que existem *N* nós contendo cada um uma réplica de todos os objetos, cada nó gera *TPS* transações por segundo, e cada transação envolve um número fixo de ações, *A*, cada ação demorando um tempo fixo a executar, *TA*. Desta forma, o tempo de execução de uma transação resultará em $TT = A * TA$. Com base nestas duas observações, o número total de transações a ser executadas em simultâneo num sistema é dado por $TR\_TOT = TPS * A * TA$. Após uma análise cuidada, percebe-se também que à medida que a carga num sistema se eleva, e consequentemente o parâmetro *TR_TOT* se eleva também, o tempo que leva a completar cada uma das ações individuais contidas em cada uma das transações irá aumentar. Observando agora um escalamento de *N* nós neste sistema, *N* vezes mais transações serão originadas por segundo globalmente, dado o facto de existir replicação sempre para *N-1* nós. Observando o comportamento em sistemas que utilizam replicação síncrona, a adição de mais nós ao sistema implica um aumento do tempo de execução de cada transação, já que a replicação está incluída nestas, e, portanto, considera-se que o tamanho de cada transação incrementa no fator de *N* e a taxa de atualização para cada nó incrementa num fator de N^2. Em sistemas cujo mecanismo de replicação não é síncrono, estes valores são idênticos, no entanto aplicam-se a pontos de análise diferentes: neste caso, o número de transações concorrentes a ocorrer nos sistemas é proporcional ao número de nós constituintes, incrementando, portanto, no fator de *N*, à adição de novas réplicas. A taxa de atualização de informação incrementa também num fator de N^2.

Com base nesta análise, percebe-se que existirá, para cada sistema, e dependendo das características de *hardware* de cada máquina, um limiar em que caso se adicione mais nós incluídos na replicação, as máquinas não conseguirão lidar com a carga, levando a uma fila crescente de operações por executar e a uma eventual falha total do sistema de base de dados.

No âmbito desta dissertação, será utilizado um mecanismo de replicação assíncrona dado o facto e não ser crucial lidar com concorrência de operações, pela natureza dos dados que se armazena na tabela em causa.

Capítulo 3

Motivação e Objetivos

O presente trabalho de dissertação surgiu da necessidade de implementar determinadas melhorias ao nível de um produto inovador que se encontra em acelerado desenvolvimento pela Altice Labs no âmbito das telecomunicações de voz sobre IP.

Foi identificada uma falha no produto Advanced Business Communications que em determinadas circunstâncias provocava uma limitação de serviço e consequente indisponibilidade de determinadas funcionalidades para o utilizador final. Sendo esta situação indesejável num produto que se encontra em comercialização, e também já em produção em diversos países do Mundo, faz também parte do processo de inovação a procura de soluções que visem melhorar a sua qualidade global, resolvendo problemas que poderão existir ou adicionando novas funcionalidades.

3.1. O Produto Advanced Business Communications

Integrado num contexto empresarial, e orientado à simplificação da gestão dos serviços, o Advanced Business Communications (ABC) é um produto inovador desenvolvido pela Altice Labs que tem como objetivo fornecer aos clientes uma solução flexível e de simples gestão, com custos moderados e totalmente assente no paradigma dos serviços na *cloud*, capaz de agregar através dos seus múltiplos módulos um grande conjunto de serviços como *hosted* PBX, integração com redes móveis ou fax virtual. A sua gestão baseada num moderno portal *web*, permite um elevado nível de controlo e também um elevado nível de personalização, no entanto de uma forma rápida e pouco complexa.

A arquitetura do produto de elevada modularidade permite uma fácil integração de novas funcionalidades sem comprometer a integridade do serviço previamente existente. Este elevado

nível de flexibilidade permite que os prestadores de serviços sejam capazes de disponibilizar um serviço integrado fixo-móvel de comunicações empresariais que incrementa a produtividade, mantendo, no entanto, um nível de custo reduzido.

3.1.1. ABC Application Server (ABC-AS)

De entre os diversos módulos que compõe e solução global do ABC, destaca-se um por ser como que o "motor" para todos os processos aplicacionais em curso na solução, que é denominado de ABC Application Server, ou ABC-AS.

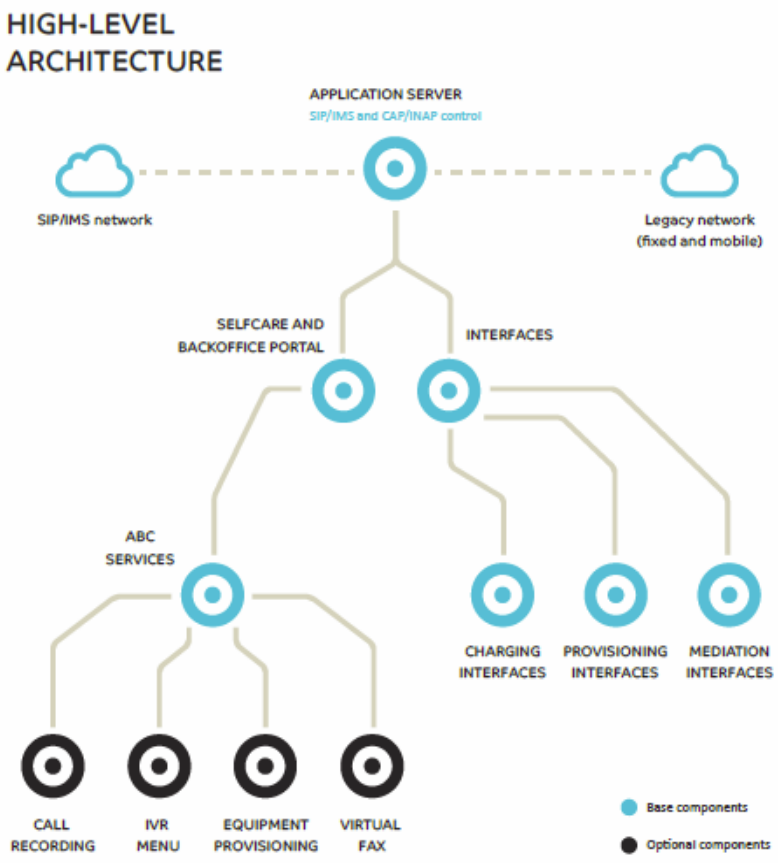

Fig. 3.1 – Arquitetura de alto nível do produto ABC

No ABC-AS estão concentradas as funcionalidades centrais do produto ligadas às comunicações, como o processamento de chamadas e o controlo de utilização dos serviços por parte dos utilizadores, e é feita a integração com serviços externos como por exemplo o serviço de faturação, de aprovisionamento automático de terminais ou o serviço de menu de voz.

De uma forma mais detalhada, o ABC-AS está assente numa plataforma Rhino [13] que é um servidor de aplicações Java cuja infraestrutura está altamente orientada a serviços de

telecomunicações em grande escala, implementando por isso diversos mecanismos de resiliência e de alta disponibilidade, bem como um portal que possibilita a gestão das diversas aplicações. A norma implementada pela plataforma Rhino é denominada de JAIN Service Logic Execution Environment (JSLEE), que é uma API de programação Java orientada ao desenvolvimento de aplicações de rede, capaz de atuar como integradora de múltiplos protocolos e serviços de rede na área das telecomunicações. Permite ainda, através do seu desenho de implementação, que o desenvolvimento de software seja robusto pelo facto de as operações cumprirem com o conceito transacional ACID [14].

A implementação da norma JSLEE conta também com a existência de *resource adaptors (RA)*, capazes de adicionar funcionalidades ou acesso a implementações de protocolos bem conhecidos como SIP ou IMS, ao nível da plataforma base Rhino, para que posteriormente uma aplicação possua acesso a tais protocolos ou funcionalidades para sua utilização.

3.2. Pacemaker Cluster Manager

O software Pacemaker é um gestor de recursos orientado a clusters de serviços de praticamente qualquer dimensão, capaz de assumir o compromisso de providenciar alta disponibilidade em certos cenários em que isso é uma mais-valia [15].

De uma forma genérica, um *cluster* Pacemaker é criado por configuração em todos os nós que se pretende incluir no mesmo. A configuração envolve a instalação dos pacotes necessários, a configuração do serviço a controlar e ainda a criação de um IP virtual (VIP), que facilita o acesso por parte das aplicações, já que agrega todas as conexões num único endereço, tornando por isso as eventuais comutações internas entre os vários nós que compõe esse *cluster*, transparentes às aplicações cliente. A instalação e configuração de um *cluster* Pacemaker vai além do âmbito desta dissertação, tendo sido realizada por externos, pelo que não será aqui descrita.

Na figura seguinte ilustra-se a constituição de um *cluster* Pacemaker, composto por dois nós físicos, com dois endereços físicos também, onde estão assentes as bases de dados, um *master* e um *slave*. Existe um VIP que serve o *cluster*, sendo que será responsabilidade do Pacemaker enviar os pedidos que chegam a esse VIP para o nó correto, e em funcionamento em cada instante.

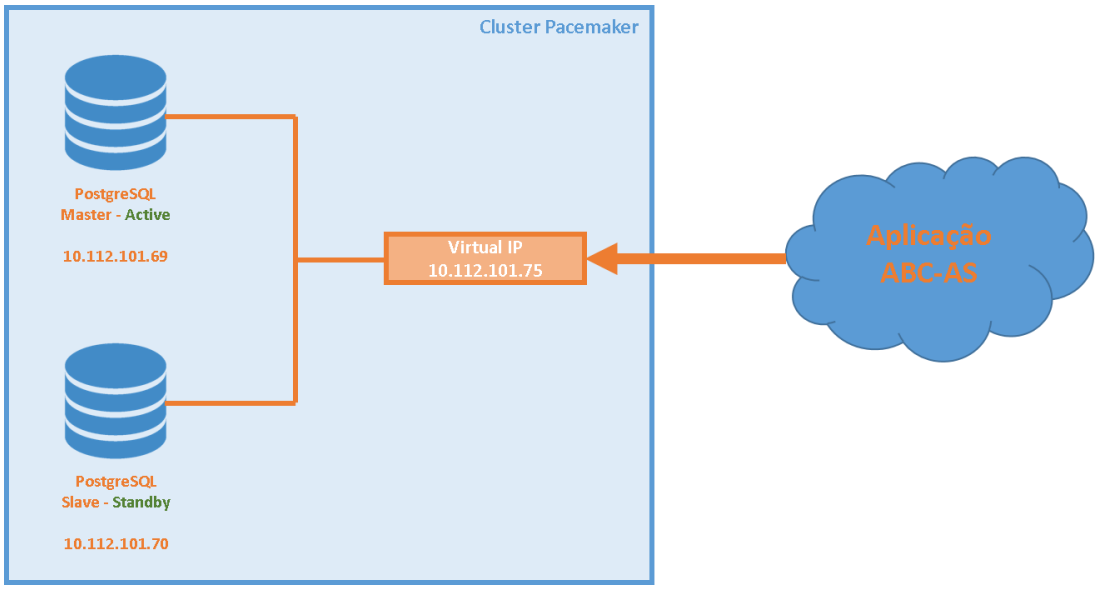

Fig. 3.2 – Representação de um *cluster* Pacemaker

Com esta abordagem, a aplicação nunca toma conhecimento das alterações ocorridas dentro do *cluster*, evitando assim falhas, faltas de serviço, ou configurações manuais. No limite, à ocorrência de uma comutação, poderá haver atrasos de alguns instantes, enquanto o processo de reposição não é terminado. Desta forma alcança-se uma disponibilidade aceitável dos sistemas de base de dados.

3.3. Descrição do Problema

Com a finalidade de suportar as funcionalidades do serviço Advanced Business Communications Application Server (ABC-AS) existem duas bases de dados PostgreSQL que contêm a informação relativa a todos os utilizadores do serviço bem como registos de atividade do mesmo.

A informação contida na primeira base de dados denominada de ‘clientes’ (CLI), é composta por todos os dados que correspondem a informações do cliente, equipamentos associados a este, extensões telefónicas, preferências do serviço, últimas chamadas efetuadas, entre outros. Por outro lado, a base de dados de registo (XDR) contém registos e histórico de todas as ações tomadas pelo cliente para com o serviço, isto é, registo de um terminal, alterações efetuadas no portal de gestão, pedidos de suporte, histórico de utilização de determinadas funcionalidades, etc. Estas bases de dados, ainda que distintas, estão fisicamente colocadas num único servidor e necessitam, portanto, de ser replicadas e de possuir a sua informação copiada quer para outra máquina idêntica no mesmo local, quer para outro local geograficamente

diferente. Assim, e atendendo ao requisito apresentado, o serviço ABC-AS é constituído por 2 *sites* em localidades distintas, sendo que um é designado de principal e o outro de secundário, que assume o serviço aquando da falha do primeiro, possuindo cada um deles duas máquinas físicas que suportam as bases de dados, segundo o esquema abaixo apresentado.

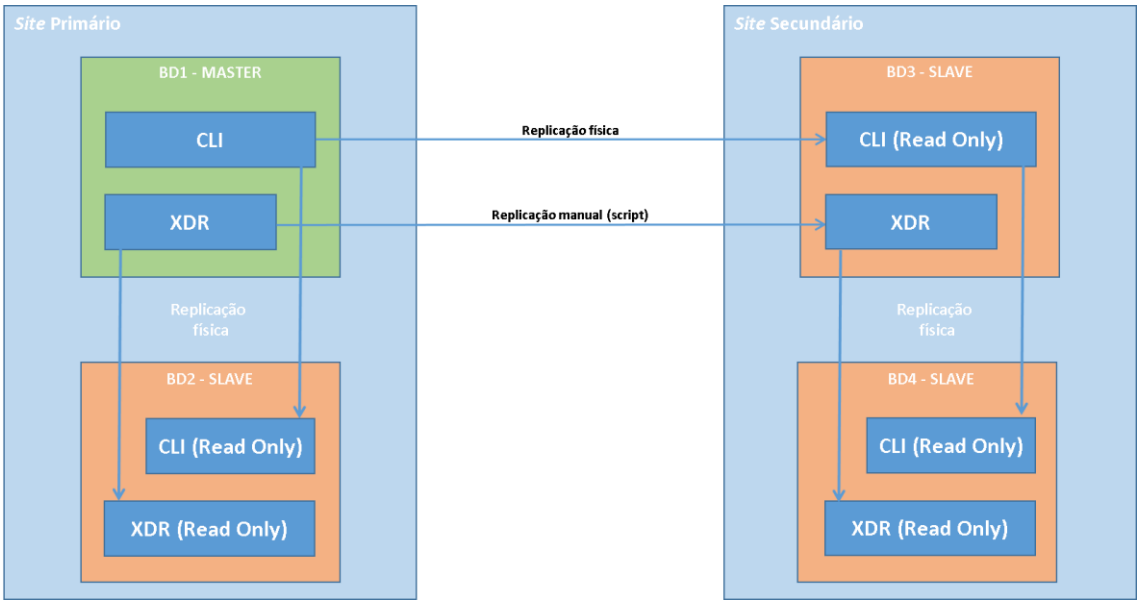

Fig. 3.3 – Esquema atual do fluxo de dados entre bases de dados

Considerando um cenário hipotético e sem condicionantes técnicas, o site secundário deveria assumir o serviço de forma absolutamente transparente para os utilizadores, não criando, desta forma, indisponibilidades nem falhas permanentes que exijam recuperação manual. Não sendo real este cenário por diversas razões, o serviço entra em modo degradado à falha do *site* principal, e, entre outras limitações, as bases de dados que assumem a carga estão em modo de leitura por serem *slaves* das do *site* principal, e por força do tipo de replicação que é física, os slaves são cópias *byte* a *byte* do master. Uma vez estando as bases de dados em modo de leitura, todos os componentes do serviço ABC-AS que necessitem naquele momento de efetuar operações de escrita irão falhar, evidenciando ao cliente falhas no serviço.

De entre as funcionalidades que ficam impactadas pelas razões acima mostradas, serão neste trabalho abordadas e melhoradas duas, por utilizarem nas suas funções uma tabela comum na base de dados, tabela esta que presentemente reside na base de dados de clientes (CLI) e que será transportada para a de registos (XDR). Destacam-se, portanto, os seguintes dois serviços:

- **Call Return:** esta funcionalidade facilita ao utilizador a remarcação automática do número da última chamada recebida, atendida ou não. Não sendo possível registar o número do último chamador na base de dados, que se encontra em

modo de leitura, o número retornado como "último" é, portanto, último antes de o serviço entrar em modo degradado;

- **Last Number Redial:** analogamente ao Call Return, esta funcionalidade permite a remarcação do último número, mas desta vez trata-se do último número marcado e não recebido. Devido às mesmas razões acima apresentadas, e quando o serviço está a operar no *site* secundário, as suas capacidades ver-se-ão reduzidas.

A tabela envolvida nos registos relacionados com as funcionalidades descritas, guarda diversas informações correntes durante o funcionamento do sistema, isto é, os dados inseridos na tabela não são crescentes; a tabela possui apenas tantas linhas quanto o número de terminais telefónicos registados no serviço, sendo responsabilidade deste atualizar a linha correspondente ao terminal que está a efetuar determinada operação, inserindo os dados respetivos nas colunas respeitantes à funcionalidade utilizada, deixando as restantes colunas intactas, já que, como se mostra abaixo, a estrutura da tabela permite guardar informação proveniente de diversas funcionalidades.

```
LASTCALLS (
     TERMINAL_ID                    BIGINT                  NOT NULL,
     USER_ID                        INTEGER                 NOT NULL,
     LAST_DIALED_CALL               VARCHAR(64)             NULL,
     LAST_DIALED_NUM_ID             INTEGER                 NULL,
     LAST_DIALED_DATE               TIMESTAMP               NULL,
     LAST_RECEIVED_CALL             VARCHAR(64)             NULL,
     LAST_RECEIVED_NUM_ID           INTEGER                 NULL,
     LAST_RECEIVED_DATE             TIMESTAMP               NULL,
     LAST_RECEIVED_CALL_ID          CHARACTER VARYING(64)   NULL,
     LAST_RECEIVED_USER_NUM         CHARACTER VARYING(64)   NULL,
     LAST_RECEIVED_USER_NUM_ID      BIGINT                  NULL,
     LAST_RECEIVED_FWD_CALL         CHARACTER VARYING(64)   NULL,
     LAST_RECEIVED_ORIG_CLD_CALL    CHARACTER VARYING(64)   NULL,
     LAST_TERMINATED_CALL_ID        CHARACTER VARYING(64)   NULL,
     LAST_TERMINATED_USER_NUM       CHARACTER VARYING(64)   NULL,
     LAST_TERMINATED_USER_NUM_ID    BIGINT                  NULL,
     LAST_TERMINATED_CALL           CHARACTER VARYING(64)   NULL,
     LAST_TERMINATED_NUM_ID         BIGINT                  NULL,
     LAST_TERMINATED_DATE           TIMESTAMP               NULL,
     LAST_TERMINATED_FWD_CALL       CHARACTER VARYING(64)   NULL,
     LAST_TERMINATED_ORIG_CLD_CALL  CHARACTER VARYING(64)   NULL,
     LAST_TERM_UNKNOWN_PAI_DATE     TIMESTAMP               NULL,
     LAST_TERM_CALL_TRACE_DATE      TIMESTAMP               NULL,
     LAST_RECEIVED_SIP_URI          CHARACTER VARYING(64)   NULL,
     LAST_TERMINATED_SIP_URI        CHARACTER VARYING(64)   NULL,

     CONSTRAINT LASTCALLS_PK PRIMARY KEY (TERMINAL_ID)
)
```

Fig. 3.4 – Modelo de dados da tabela 'lastcalls'

Posteriormente, para cada caso de utilização, a aplicação terá a responsabilidade de recolher da tabela a informação constante da linha correspondente ao terminal e da coluna correspondente à funcionalidade em uso. No final, a informação poderá ser atualizada com os novos dados da última operação, necessitando para isso que a base de dados aceite operações de escrita, caso contrário não será possível guardar a informação mais recente.

De uma forma genérica, o sucesso do presente trabalho estaria garantido caso se conseguisse montar um cenário em que a informação constante dos registos acima descritos fluísse entre os dois *sites* de forma automática, e se reduzisse ao mínimo o tempo de indisponibilidade do serviço do ponto de vista do cliente em caso de falha.

3.4. Objetivos

O objetivo principal deste trabalho é criar um mecanismo de transporte de informação em tempo real entre os dois *sites* que suportam o serviço, para que se eliminem indisponibilidades em caso de falha de um destes. Múltiplos modelos de solução se encontram em aberto, sendo que se considera que o veículo da informação a transportar será uma base de dados, pelo que diversas tecnologias de base de dados serão analisadas para que se venha a conhecer os seus métodos de replicação, e consequentemente se perceba qual a abordagem mais vantajosa face aos requisitos do produto.

Conduzir-se-á um estudo sobre algumas tecnologias de base de dados existentes no mercado, ponderando as vantagens e desvantagens da sua aplicação como potencial solução, e ainda a pertinência de uma mudança de tecnologia no sentido das bases de dados NoSQL.

Capítulo 4

Estudo Prévio

O problema descrito no capítulo anterior prende-se, como já foi explicado, com o facto de os sistemas de gestão de base de dados do *site* secundário se encontrarem a operar como *slaves* de outro nó, ficando por isso em modo só de leitura, não possibilitando, portanto, operações de escrita e limitando o serviço em alguns componentes já descritos.

Assim, o estudo de uma solução que permitisse a utilização dos componentes em pleno, orientou-se maioritariamente à procura de abordagens que facilitassem não só operações de leitura e escrita em qualquer um dos *sites*, bem como a sincronização de informação de forma bidirecional. Durante este processo, múltiplas abordagens para resolução do problema foram apontadas como possíveis soluções, não satisfazendo, no entanto, todos os requisitos, ao nível de robustez técnica, completude da solução ou até eficiência, pelo que foram prematuramente abandonadas.

Inicialmente, foi colocada a hipótese de se iniciar a migração do sistema de base de dados XDR para uma solução baseada em NoSQL, sendo que essa migração teria início precisamente com a tabela de registos que está implicada nos dois serviços abordados neste trabalho (Call Return e Last Number Redial), a tabela 'lastcalls'. Posteriormente, poder-se-ia ponderar a migração gradual das restantes tabelas de registo para a emergente tecnologia NoSQL. Após algum trabalho de pesquisa e estudo mais aprofundado acerca de requisitos técnicos, requisitos de manutenção, esforço de transição e adequação da tecnologia, concluiu-se que a migração iria ser de uma complexidade que não se adequaria ao tempo de realização desta dissertação. Desta forma, prosseguiu-se o estudo tendo em conta que a migração para NoSQL estaria posta de parte num futuro próximo, devendo, portanto, a abordagem utilizar o ambiente tecnológico já existente no produto, designadamente, bases de dados relacionais PostgreSQL. Resumidamente, a solução passará por implementar um esquema de replicação lógica entre *sites* da tabela em questão, para que a informação flua de forma bidirecional, e adequar a aplicação para recolher da base de dados

os dados mais recentes em cada situação, com base nos *timestamps* guardados na tabela para cada campo. Adiante neste documento será explicitada mais profundamente a solução encontrada.

4.1. Análise Tecnológica

Dado que a migração de tecnologia de base de dados foi uma hipótese, foi feita uma análise das características mais importantes de funcionamento a dois produtos que assentam na tecnologia NoSQL com o objetivo de perceber se a solução teria viabilidade de implementação. Adicionalmente, estudou-se a possibilidade de, continuando a utilizar a tecnologia já existente, adicionar determinados procedimentos que conduzissem a uma solução viável.

4.1.1. Base de Dados Não Relacional – NoSQL

De entre os dois produtos abordados, todos apresentaram desvantagens em diferentes áreas específicas, sendo que por essa razão foi decidido abandonar a migração para o paradigma NoSQL num futuro próximo. Apresentam-se de seguida as razões, divididas por tecnologia, que numa abordagem inicial obrigaram à retirada de esforços nesta área:

MongoDB:

Dado um dos requisitos ser a bidirecionalidade do fluxo de dados, a arquitetura master-slave característica deste tipo de bases de dados, bem como a inabilidade dos nós que operam em modo *slave* de aceitar operações de escrita, levou a que esta tecnologia fosse posta de parte. Adicionalmente, foi feito um estudo que abordou outros pontos como sendo a estratégia de replicação, a estratégia de repartição de carga e gestão de acessos e a gestão de falhas. Assim sendo, relativamente à estratégia de replicação, como foi dito, este sistema opera em modo master-slave, com um nó em modo *master* e *n* nós em modo *slave*, o que implica que apenas o *master* aceita operações de escrita. Num segundo momento, todas as operações vão ser lidas pelos seus *slaves* e reproduzidas nos seus *datasets* próprios, o que levanta outra preocupação relacionada com o volume de tráfego gerado entre os dois *sites*: cada operação efetuada no master, vai ser efetuada tantas vezes quanto o número de *slaves* envolvidos, isto é, o tráfego gerado na rede para cada operação será de *n* bytes * *x* slaves, em que *n* representa o tamanho na rede de uma operação e *x* o número de slaves que aplicam a operação. Isto poderá não ser um problema nas

comunicações entre máquinas co-localizadas, mas tornar-se-á se se considerar máquinas geograficamente distantes.

Como já foi referido, os nós que operam em modo *slave* não aceitam operações de escrita, o que implica que toda a carga respeitante a este tipo de operações será dirigida ao *master*. O contrário sucede com operações de leitura, em que qualquer nó poderá ser contactado para obtenção de informação.

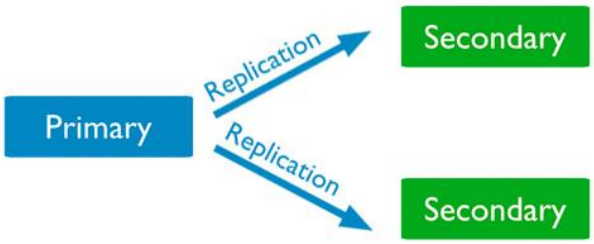

Fig. 4.1 – Esquema de replicação MongoDB

Dado o facto de o *master* possuir um nível crucial de importância no sistema, será necessário que um novo nó seja colocado neste modo após falha do primeiro. O MongoDB utiliza um esquema de deteção de falhas com base em *heartbeat*, em que os nós pertencentes ao *cluster* enviam mutuamente pacotes de *ping*, que deverão retornar uma resposta, no máximo, em dez segundos, e caso não retornem, o nó que não responde será marcado como inacessível. No caso de o nó inacessível ser um *master*, ou no caso de um *master* conseguir apenas contactar uma minoria dos *slaves* do *cluster*, será iniciado um processo de eleição, para que um dos nós restantes seja promovido, como se exemplifica na figura abaixo.

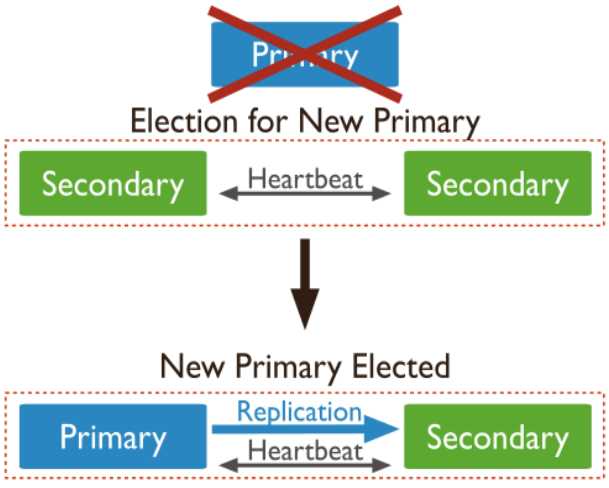

Fig. 4.2 – Ilustração do processo de eleição

Cada nó possui na sua configuração dois parâmetros que definem o seu "peso" nas eleições durante o funcionamento: o parâmetro "*priority*", que representa a preferência de um determinado nó na escolha de um novo *master*, e o parâmetro "*votes*", que representa o número de unidades de voto desse nó, e apenas poderá tomar o valor 0 (zero) ou 1 (um). Segundo a documentação, o parâmetro "votes" não deverá ser alterado, exceto em situações específicas, ou em caso de não se pretender que determinado nó se torne *master* nem solicite eleições, situação em que deverá possuir o valor 0 (zero), bem como o parâmetro "*priority*". Este último poderá tomar qualquer valor inteiro, e será considerado como fator decisivo na eleição.

```
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "arbiterOnly" : false,
   "buildIndexes" : true,
   "hidden" : false,
   "priority" : 0,
   "tags" : {

   },
   "slaveDelay" : NumberLong(0),
   "votes" : 0
}
```

Fig. 4.3 – Exemplo de configuração de nó não votante

O processo de eleição de um novo *master* pode definir-se como um "*best-effort"*, isto é, os nós de prioridade mais baixa deixarão para os de prioridade mais alta o início da operação onde será eleito um *master*. Este é eleito. Caso o eleito não seja o de maior prioridade, novo processo é despoletado pelos nós de maior prioridade. Este processo repete-se até o *master* eleito ser o de maior prioridade.

No caso de o nó em falha ser um *slave*, não haverá lugar a eleição, e o *cluster* continuará em operação normal. O processo de eleição poderá demorar até dois minutos, tendo em conta diversos fatores, e, uma vez que não existe *master* durante esse tempo, o *cluster* não aceitará operações de escrita, limitando, portanto, os serviços.

Finalizado este processo de eleição e existindo já um *master* estável, dar-se-á um outro processo denominado de *rollback*, em que todos os nós irão sincronizar as suas operações em favor da consistência da informação contida no *cluster*.

Apache Cassandra:

Outro produto que mereceu atenção durante o estudo prévio de procura de uma solução para facilitar a geo-redundância do serviço foi o Apache Cassandra, amplamente utilizado no mercado e com potencialidades satisfatórias para alcançar os objetivos pretendidos. No entanto, o processo de migração mostrou-se, mais uma vez, complexo e moroso, tendo também existido aconselhamento interno por parte da Empresa de que esta solução seria de complexa implementação e afinação.

Ainda assim, à semelhança da tecnologia anterior, foi feito um estudo dos principais pontos de importância para a solução: estratégia de replicação, distribuição de carga e gestão de acessos, e tolerância a falhas.

Comparativamente com o produto anterior, o Apache Cassandra apresenta diferenças significativas em várias vertentes, nomeadamente na bidirecionalidade do fluxo de informação, oferecendo uma abordagem verdadeiramente multi-master, e com uma robusta implementação no que toca à resiliência do *cluster*. No entanto, como é clarificado através do teorema CAP explicado em 2.3, um sistema de bases de dados que proporciona alta disponibilidade e forte resistência a falhas, não é um bom candidato ao nível da garantia de consistência da informação. Ainda assim, são disponibilizados pelo Apache Cassandra alguns mecanismos de reforço de consistência.

A arquitetura das bases de dados Cassandra é complexa e a sua configuração exige o perfeito conhecimento das operações efetuadas pelas aplicações, por forma a otimizar o funcionamento do sistema, que foi criado com o objetivo de lidar com grandes cargas de trabalho sem que exista um ponto único de falha. Assim, um *cluster* Cassandra pode ser visto como um anel em que os nós se interligam num esquema *peer-to-peer*. De forma genérica, um sistema possui diversos componentes que interagem na movimentação da informação:

- **Cluster:** representa todo o sistema; contem vários datacenters (um ou mais);

- **Datacenter:** conjunto de nós relacionados que se consideram próximos entre si;

- **Rack:** idêntico a datacenter, no entanto consideram-se ainda com maior proximidade;

- **Nó:** local onde informação é guardada;

- **Mem-Table:** estrutura de dados residente em memória, que alberga as mais recentes operações de escrita num determinado nó;

- **Commit-Log:** registo de todas as operações de escrita de um nó por ordem sequencial; é escrito imediatamente antes da Mem-Table;

- **SSTable:** ficheiro em disco que recebe os dados provenientes de uma Mem-Table, após um determinado espaço de tempo;

- **Bloom Filter:** algoritmo ultrarrápido para determinação da localização da informação; é acedido após toda e qualquer operação e funciona como uma cache;

Relativamente à estratégia de balanceamento de carga, o acesso é feito por intermédio de *drivers* nativos para uma alargada diversidade de linguagens de programação, em que apenas é necessário definir os "pontos de contacto", que são nós pertencentes ao *cluster*. No caso de um "ponto de contacto" não estar disponível, será automaticamente contactado o seguinte.

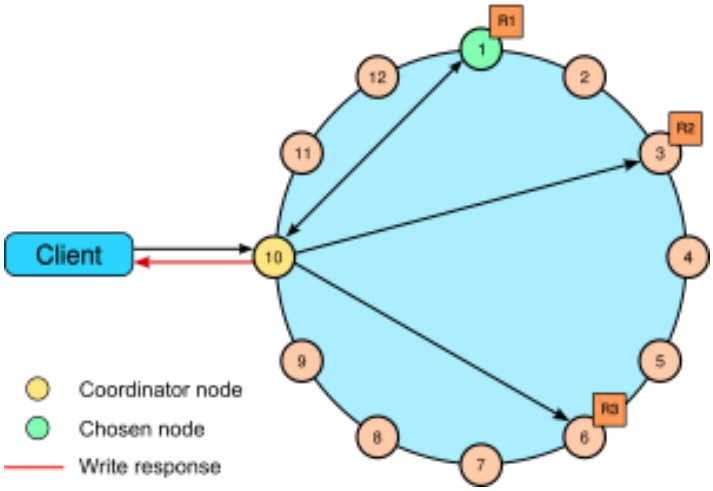

Fig. 4.4 – Esquema de interação numa operação de escrita

Todos os nós constituintes de um cluster aceitam operações de leitura e escrita e têm conhecimento de todos os outros membros desse mesmo cluster, bem como da localização da informação, encarregando-se de replicar ou recolher os dados respeitantes a uma determinada operação que estão a executar, consoante as suas configurações de consistência e disponibilidade. A replicação entre nós é inerente ao funcionamento desta solução, isto é, cada instância que está encarregue de uma determinada operação, seja um SELECT, INSERT, UPDATE ou DELETE, torna-se o coordenador da mesma, isto é, garante a correta execução da operação em, pelo menos, tantos nós quanto os que estão nos parâmetros de configuração relativos aos níveis de consistência das operações. No caso de operações de escrita, na eventualidade de este grau de replicação não poder ser satisfeito, é configurável a ação que o sistema toma, poderá falhar a operação ou guardá-la numa fila no coordenador para mais tarde tentar novamente a operação noutros nós com o objetivo de alcançar a replicação configurada. Já nas operações de leitura, é também configurável o comportamento; ao não ser possível obter um resultado idêntico por parte de *n* nós de uma determinada operação, a reposta ao cliente poderá prosseguir ou falhar.

Um *cluster* Cassandra tolera falhas de uma parte dos nós até um determinado limite, dependendo do fator de replicação definido para as operações específicas, o que significa que a tolerância a falhas de um *cluster* deste tipo depende diretamente da percentagem da informação armazenada em cada nó. Na figura acima representa-se a interação entre um cliente e o cluster, aquando da inserção de dados na BD; assume-se que aquele sistema está configurado para um fator de replicação três e um nível de consistência de 1. Desta forma, a seguinte sequência de operações é desencadeada:

- O cliente contacta o seu "ponto de contacto", o nó 10, e efetua uma operação de inserção;

- O nó denominado de "ponto de contacto" torna-se o coordenador da operação, calculando através de uma função de *hashing* em que nó pertencente ao cluster irá colocar a informação, bem como quais serão as restantes duas réplicas (fator de replicação três). Seleciona o nó 1 como destino, e os nós 3 e 6 como réplicas;

- Uma vez que o nível de consistência é um, o coordenador aguarda a confirmação da correta execução da operação de apenas um dos nós. O nó 1 responde com sucesso;

- A resposta de sucesso é retornada ao Cliente pelo coordenador;

- A operação é terminada.

Um sistema de gestão de bases de dados assente na tecnologia Apache Cassandra mostrou ser um bom candidato a solução pela facilidade em lidar com a distribuição dos sistemas, até mesmo geográfica, e pelo bom controlo da propagação dos dados. No entanto, outros fatores foram tidos em conta, o que levou ao abandono desta solução, nomeadamente, e como já foi referido, a complexidade da migração face ao benefício a curto prazo. Adicionalmente, questões técnicas como a inexistência de *rollback*[1] ou o reduzido tráfego da componente a ser migrada, levaram a que a solução não fosse viável no cenário presente.

4.1.2. Base de Dados Relacional – PostgreSQL

Das duas vertentes de replicação que se observam no mundo das bases de dados relacionais, física e lógica, a última revelou ser um forte candidato a constituir uma solução que alcançasse o objetivo já que apresenta caraterísticas positivas do ponto de vista da facilidade de implementação, da solidez da tecnologia de bases de dados relacionais dentro da Empresa e ainda do ponto de vista da integração com os sistemas pré-existentes.

Foi então utilizado o conceito da replicação lógica do PostgreSQL para se conseguir alcançar o cenário de replicação bidirecional com capacidade de leitura e escrita em ambos os *sites*. Este tipo de replicação é marcado pela flexibilidade do seu modo de funcionamento, e distingue-se da replicação física por essa mesma razão; a simplicidade com que a informação é transmitida entre nós torna-se relevante quando se pretende alcançar certo nível de controlo sobre o fluxo de replicação entre duas bases de dados. Torna-se possível selecionar exatamente o que é copiado, tabela a tabela, sendo que apenas as alterações registadas nos ficheiros de registo de alterações, correspondentes às tabelas selecionadas para replicação, serão transmitidas, contrariamente ao que acontece numa replicação física, conhecida como "*streaming replication*", precisamente pela razão de que toda e qualquer alteração é transmitida, incluindo, por exemplo, as alterações feitas pelos processos de VACUUM, gerando carga e tráfego muitas vezes desnecessários.

A ferramenta que permite adicionar esta funcionalidade ao PostgreSQL é denominada de PGLogical, utilizado na versão 1.2 neste caso, e apresenta-se como um *addon* do sistema de base de dados. É uma tecnologia relativamente recente, pelo que ainda não se encontra integrada numa instalação de raiz da base de dados, estando, no entanto, uma primeira fase dessa integração a ser planeada para a versão seguinte do PostgreSQL, a versão 10.0.

[1] Rollback: operação que permite restaurar a base de dados para um estado anterior, limpo, mesmo após operações falhadas ou dados corrompidos, e que é fundamental para a garantia da integridade dos dados [16].

4.2. Abordagem Selecionada

Tal como já referido anteriormente, a migração para uma tecnologia baseada em NoSQL foi abandonada dado existirem determinados pontos fundamentais em que a tecnologia não respondia aos requisitos do produto. A complexa migração aliada a fatores como falta de determinadas funcionalidades obrigaram à continuidade do vínculo com as bases de dados relacionais.

Após investigação de tecnologias diversas como as que foram explicadas no ponto anterior, foi decidido que a abordagem mais viável para o problema em questão seria, de facto, combinar o atual esquema de replicação física com um outro esquema de replicação lógica entre os dois *sites* para a tabela 'lastcalls'. Na tabela seguinte evidenciam-se algumas das principais características das soluções abordadas, de uma forma comparativa:

	Apache Cassandra	MongoDB	PGLogical
Escrita nos dois *sites*	✓	✗	✓
Gestão de tráfego inter-*site*	✓	✗	✗
Estratégia de recuperação de falhas	✗	✓	✓
Estratégia de replicação de dados	✓	✗	✓

Tabela 4.1 – Comparação das diferentes tecnologias

Analisando a tabela acima, que resume a forma como certas características foram vistas ao abrigo daquilo que seriam os requisitos do produto, percebe-se que, no caso do Apache Cassandra é oferecida a possibilidade de escrita nos dois *sites* bem como controlo do tráfego, já que a informação não é enviada para o *site* oposto tantas vezes quanto o número de réplicas, mas apenas uma, e ainda uma satisfatória replicação de dados através do *cluster*, no entanto, a recuperação de falhas mostrou ser complexa dada a falta do processo de *rollback*, podendo obrigar a que eventuais inconsistências nos dados devam ser resolvidas manualmente. O motor MongoDB claramente não se adaptava à situação desejada, já que as suas características não foram satisfatórias na construção de uma potencial solução; a arquitetura "*master-slave*" em que o *slave*

é só de leitura, ou as exigentes caraterísticas ao nível do esquema de troca de tráfego, levaram ao descarte desta abordagem.

A utilização da replicação lógica em PostgreSQL mostrou ser uma solução que devolvia resultados consistentes na sua operação, no entanto deficitária em alguns pontos como se explicará no capítulo seguinte. A solução passará, portanto, pela utilização do plugin PGLogical, replicando a informação de uma tabela de um *site* para outro, ajustado a um esquema bidirecional, combinado com um *trigger*, uma *view* e modificações em três componentes aplicacionais.

No capítulo seguinte detalhar-se-á a implementação desta solução.

Capítulo 5

Arquitetura da Solução

No presente capítulo detalhar-se-á em profundidade a conceção da solução encontrada, bem como todos os detalhes da sua colocação em funcionamento, adaptada ao cenário real existente no produto ABC da Altice Labs. Recordando o explicitado em 3.3, o sistema existente é composto por dois *sites*, em cada um existindo duas máquinas com bases de dados PostgreSQL instaladas, e em que uma das máquinas é uma cópia física da primeira, correspondendo a um hot-standby. Existe uma ligação dedicada entre os dois *sites* para troca de informação.

5.1. Descrição do Novo Cenário de Replicação

A nova arquitetura projetada para cumprir os requisitos irá coexistir com a arquitetura já existente. Será adicionado a essa arquitetura um plugin denominado PGLogical que irá ficar responsável pelo transporte da informação constante da tabela 'lastcalls', que será repartida em duas, uma por cada *site*, de um *site* para outro.

As alterações a introduzir ao nível dos sistemas de gestão de bases de dados incidirão maioritariamente nas instâncias responsáveis pela base de dados XDR (registos), em que serão criadas duas novas tabelas ('lastcalls_a' e 'lastcalls_b') que corresponderão à informação contida em cada *site*, para além da instalação do plugin e criação das subscrições, ação que permite que uma instância de base de dados subscreva os dados e todas as alterações que ocorrem numa, ou num conjunto, de tabelas de uma instância remota e as reproduza na sua tabela local que se pretende que seja sempre um espelho da tabela remota. Na figura seguinte, recorre-se ao esquema utilizado em 3.3 para mostrar onde se irão inserir as alterações:

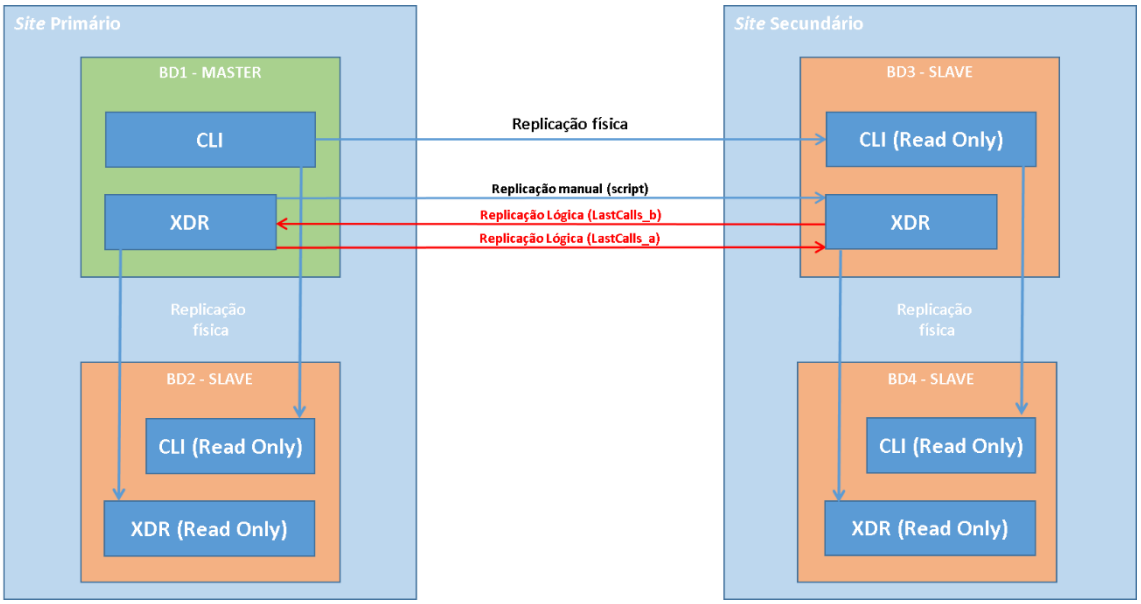

Fig. 5.1 – Esquema da nova arquitetura de replicação

Como se observa na figura, está assinalado a vermelho o novo fluxo de dados que é criado pelo plugin PGLogical e que se refere ao transporte de informação das tabelas, ou seja, a tabela lastcalls_b pertencente ao *site* secundário é copiada para a tabela lastcalls_b no *site* primário, e a tabela lastcalls_a pertencente ao *site* primário é copiada para a tabela lastcalls_a no *site* secundário. Com este esquema, é possível ter a informação atualizada quase em tempo real nas duas direções, permitindo por isso aos utilizadores registados num qualquer dos dois *sites* aceder às informações das suas últimas chamadas no outro *site*. Esta abordagem é vantajosa na eventualidade de uma falha no serviço que provoque a comutação de tráfego do *site* primário para o secundário, permitindo ao cliente a continuidade da utilização dos dois serviços enunciados em 3.3.

Após estas alterações, a tabela inicial 'lastcalls' não figurará mais na tabela de clientes (CLI).

5.2. PGLogical v1.2

PGLogical [17] é uma extensão desenhada com o objetivo de dotar o sistema de gestão de bases de dados PostgreSQL com a funcionalidade de replicação lógica, mantida pela empresa 2ndQuadrant, e que, como já referido anteriormente possibilita uma filtragem fina da informação que se pretende replicar. Desta forma, este plugin permite alcançar uma vasta diversidade de objetivos, bem como integrar-se num alargado espectro de arquiteturas que podem ir desde a

simples cópia de informação de A para B até à agregação de informação proveniente de diversas fontes, filtrada por diversas cláusulas. Os seus casos de utilização incluem, entre outros, a migração seletiva de informação, a agregação de dados e a distribuição de dados por diversas outras instâncias.

A sua grande flexibilidade ao nível da configuração permite que, de facto, se adapte a uma variedade considerável de cenários e providencie diversos benefícios como sendo a replicação síncrona ou assíncrona, resolução de conflitos configurável baseada em *timestamps*, replicação de sequências, filtragem da informação replicada e replicação entre diferentes versões PostgreSQL. Existiu um esforço por parte da equipa de criação da extensão uma vez que foram tidos em conta alguns parâmetros com vista à melhoria da eficiência da utilização de recursos, como sendo a não utilização de *triggers* internos para desencadeamento de ações ou a não repetição de comandos SQL para cópia de informação. Em detrimento destas ações, é utilizado o mecanismo de *WAL*[2] *decoding*, em que é feita a interpretação do registo de operações do servidor principal e a posterior reprodução dessas alterações no *dataset* local.

O método de configuração desta ferramenta baseia-se, antes de mais, na instalação da extensão no sistema e da sua instanciação na base de dados onde a replicação irá trabalhar. Seguidamente, através de comandos específicos do *plugin*, executados já ao nível de uma determinada base de dados, a criação de uma entidade que representa um nó deverá ser feita quer no nó emissor de informação, quer no nó recetor. Posteriormente, passa-se à definição exata do que deverá ser replicado, utilizando para isso *replication sets*, que no fundo são conjuntos de tabelas em que o sistema deverá trabalhar sobre. Criados estes *replication sets*, adicionam-se as tabelas pretendidas aos mesmos. Todas as entidades descritas poderão ser criadas mais do que uma vez, contendo diversos conjuntos distintos de tabelas, o que torna o processo de replicação verdadeiramente flexível. Finalmente, é necessário informar o nó recetor de que a informação no emissor está pronta a ser enviada, sendo para isso necessário criar no recetor uma subscrição do *replication set* criado. A partir deste momento, o nó emissor cria um *replication slot* e autoriza a ligação do recetor, que irá procurar o ponto mais recente do ficheiro WAL do primeiro, iniciando a partir desse ponto o trabalho de replicação.

Adiante em 5.3.2.1 explicar-se-á a configuração deste *plugin* contextualizada no cenário real do produto.

[2] WAL (Write Ahead Log): conjunto de ficheiros que descreve detalhadamente todas as alterações ao conteúdo da base de dados; o motor da base de dados garante que o registo de todas as transações concluídas é escrito neste ficheiro e guardado no armazenamento permanente antes de alterar os ficheiros de dados onde as tabelas, os índices e restantes informações residem. Este procedimento resulta em escritas no disco significativamente menores e permite ter uma garantia da possibilidade de recuperação transação a transação [18].

5.3. Requisitos de Implementação

Para ser possível a integração desta nova abordagem no sistema existente, foi necessário proceder a algumas alterações quer ao nível das instalações PostgreSQL, quer ao nível da aplicação ABC-AS. O ambiente em que o sistema está assente é RedHat Enterprise Linux Server (RHEL) 6.8.

5.3.1. Alterações ao Modelo de Dados

As alterações que foi necessário levar a cabo no modelo de dados de ambas as bases de dados (CLI e XDR) estavam relacionadas com o transporte da tabela 'lastcalls' de uma base de dados para outra. Assim, esta tabela deixou de pertencer à BD de clientes e passou a figurar na BD de registos sob a forma de lastcalls_a e lastcalls_b. A constituição destas duas tabelas é idêntica à inicial, tal como mostrado na figura 3.4.

5.3.2. Configurações PostgreSQL

A primeira grande alteração que foi necessário levar a cabo para que este cenário fosse viável foi a migração da versão do PostgreSQL de 9.5 para 9.6, por força dos requisitos de compatibilidade do plugin que foi utilizado.

Determinadas alterações foram também efetuadas ao nível das configurações internas do sistema de base de dados para que fosse possível instanciar um processo de replicação lógica. Para este fim, é necessário que seja alterado o tipo de registo que a BD efetua no WAL, caso contrário o plugin utilizado não será capaz de seguir detalhadamente todas as alterações feitas na base de dados. Foi ainda necessária a criação de um *trigger* e uma *view*, sendo que o *trigger* ficou responsável por despoletar a remoção de registos na tabela local aquando da remoção na tabela remota e a *view* foi útil para abstrair, de certa forma, do nível aplicacional os nomes das novas tabelas. De seguida aprofundar-se-ão cada uma das alterações introduzidas no sistema PostgreSQL, aplicadas igualmente em cada uma das duas instâncias envolvidas na replicação lógica.

5.3.2.1. Instalação e configuração PGLogical v1.2

Após migração de versão da base de dados, procedimento que foi efetuado internamente pela Empresa, foi necessário proceder à instalação do plugin PGLogical. Esta instalação foi

facilmente levada a cabo por intermédio do repositório *yum* do RHEL que dispunha dos pacotes necessários para a instalação.

Concluída a instalação dos pacotes necessários, o próximo passo será estabelecer uma ligação à base de dados através do aplicativo psql, que é uma ferramenta destinada a ser utilizada numa consola e que permite executar código SQL sobre uma base de dados. Será utilizada para levar a cabo as configurações necessárias. A sequência das configurações será idêntica à explicitada acima, e assume-se que o pacote está já instalado no sistema operativo e as alterações necessárias no ficheiro *postgresql.conf* estão feitas.

Assim no *site* primário (A):

a) Estabelece-se uma ligação à BD de registos denominada 'sec_sdr' com o psql, utilizando o comando "**`psql –U postgres –d sec_sdr`**";

b) Instancia-se a extensão 'pglogical' na BD através do comando "**`CREATE EXTENSION pglogical;`**"

c) Cria-se um nó PGLogical denominado 'BD1', à escuta no IP 10.112.101.69 e no porto 5432, associado à base de dados 'sec_sdr', utilizando o comando "**`SELECT pglogical.create_node(node_name := 'BD1', dsn := 'host=10.112.101.69 port=5432 dbname=sec_sdr');`**":

```
[root@secabcas-dev-evl-bd1 ~]# psql -U postgres -d sec_sdr
psql (9.6.2)
Type "help" for help.

sec_sdr=# SELECT pglogical.create_node(node_name := 'BD1', dsn := 'host=10.112.101.69 port=5432 dbname=sec_sdr');
 create_node
-------------
   527255768
(1 row)

sec_sdr=#
```

Fig. 5.2 – Criação de nó PGLogical

d) Por uma questão de organização, apagam-se os *replication sets* instalados por defeito com o comando "**`SELECT pglogical.drop_replication_set('<nome do replication set>');`**":

e) Cria-se um novo *replication set*, a ser utilizado na solução proposta, denominado "lastcalls_a_rep" e que indica através dos quatro parâmetros a *true* que todas as operações devem ser replicadas (INSERT, UPDATE, DELETE, TRUNCATE),

com o comado "**`SELECT pglogical.create_replication_set('lastcalls_a_rep', true, true, true, true);`**":

```
[root@secabcas-dev-evl-bd1 ~]# psql -U postgres -d sec_sdr
psql (9.6.2)
Type "help" for help.

sec_sdr=# SELECT pglogical.create_replication_set('lastcalls_a_rep',  true,  true, true,  true);
 create_replication_set
------------------------
             2914777707
(1 row)

sec_sdr=#
```

Fig. 5.3 – Criação de *replication set* adequado

f) Em último lugar, nesta primeira fase de configuração, adiciona-se a tabela 'lastcalls_a' ao *replication set* criado na alínea anterior com o comando "**`SELECT pglogical.replication_set_add_table('lastcalls_a_rep', 'sec_sdr.lastcalls_a', false);`**", em que o parâmetro *false* indica que não deve haver sincronização de dados entre todos os subscritores, já que só existirá apenas um:

```
[root@secabcas-dev-evl-bd1 ~]# psql -U postgres -d sec_sdr
psql (9.6.2)
Type "help" for help.

sec_sdr=# SELECT pglogical.replication_set_add_table('lastcalls_a_rep', 'sec_sdr.lastcalls_a', false);
 replication_set_add_table
---------------------------
 t
(1 row)

sec_sdr=#
```

Fig. 5.4 – Adição da tabela 'lastcalls_a' ao *replication set*

Após a execução dos passos descritos, esta base de dados encontra-se preparada para fornecer a informação constante da tabela 'lastcalls_a' a um subscritor, que posteriormente será a base de dados do *site* B. Prossegue-se a configuração do PGLogical na próxima instância PostgreSQL, desta vez, no *site* B. Não serão explicitados os passos a), b) e d) já que são idênticos nos dois locais, no entanto, necessitam de ser executados na mesma ordem:

a) Ver acima;

b) Ver acima;

c) Cria-se um nó PGLogical denominado ‘BD2’, à escuta no IP 10.112.101.71 e no porto 5432, associado à base de dados ‘sec_sdr’, utilizando o comando “**SELECT pglogical.create_node(node_name := 'BD2', dsn := 'host= 10.112.101.71 port=5432 dbname=sec_sdr');**”:

d) Ver acima;

e) Cria-se um novo replication set, a ser utilizado na solução proposta, denominado “lastcalls_b_rep” e que indica através dos quatro parâmetros a true que todas as operações devem ser replicadas (INSERT, UPDATE, DELETE, TRUNCATE), com o comado “**SELECT pglogical.create_replication_set('lastcalls_b_rep', true, true, true, true);**”:

f) Por último, adiciona-se a tabela ‘lastcalls_b’ ao replication set criado na alínea anterior com o comando “**SELECT pglogical.replication_set_ add_table(' lastcalls_b_rep', 'sec_sdr.lastcalls_b', false);**”:

Portanto, a segunda base de dados está agora configurada também para enviar as suas informações da tabela ‘lastcalls_b’ aos respetivos subscritores.

Finalmente, resta apenas que as bases de dados façam as subscrições mútuas para que a informação comece a fluir e a replicação inicie. Este passo foi propositadamente deixado para o final já que, aquando da criação de uma subscrição, esta ir-se-á ligar ao *replication set* do servidor remoto, que deverá estar previamente criado e configurado com as tabelas corretas; de outra forma, a subscrição irá falhar. Como tal, possuindo nesta fase duas instâncias PostgreSQL configuradas para providenciar replicação lógica, e prontas a aceitar subscrições, procede-se à criação das mesmas da seguinte forma:

a) Nó ‘BD1’ pertencente ao *site* A, subscreve as alterações à tabela ‘lastcalls_b’ presente em ‘BD2’, no site B, com o comando “**SELECT pglogical.create_subscription('lastcalls_b_rep_subs', 'host =10.112.101.71 port=5432 dbname=sec_sdr', ARRAY[' lastcalls_b_rep'], false, true);**”, em que os parâmetros, por ordem, indicam o nome da subscrição, a *string* de ligação, o *replication slot* remoto ao qual a subscrição se vai ligar, negação da replicação da estrutura da BD, ativação da sincronização dos dados:

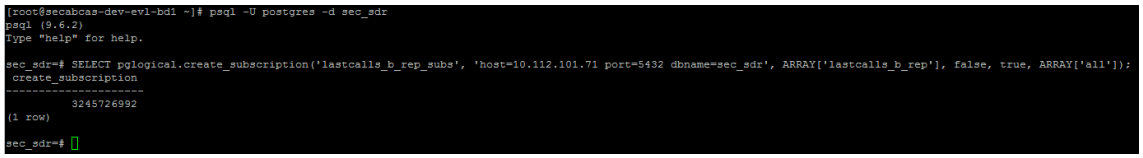

Fig. 5.5 – Criação de subscrição no nó BD1

b) Nó 'BD2' pertencente ao site B, subscreve as alterações à tabela 'lastcalls_a' presente em 'BD1', no site A, com o comando "**`SELECT pglogical.create_subscription('lastcalls_a_rep_subs', 'host=10.112.101.69 port=5432 dbname=sec_sdr', ARRAY['lastcalls_a_rep'], false, true);`**":

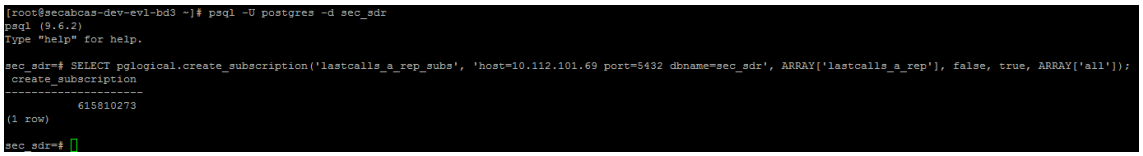

Fig. 5.6 – Criação de subscrição no nó BD2

O plugin PGLogical encontra-se, neste momento, completamente configurado e o cenário de replicação totalmente funcional. Adiante no capítulo seguinte, efetuar-se-ão testes de funcionalidade e verificação de funcionamento deste esquema de replicação.

5.3.2.2. Alterações 'postgresql.conf'

O ficheiro *postgresql.conf* é responsável por reunir num só local todas as configurações operacionais do sistema PostgreSQL, e é lido e carregado de cada vez que a instância é iniciada, pelo que determinados parâmetros presentes neste ficheiro foram alterados para estar de acordo com as necessidades da arquitetura.

Os parâmetros alterados foram os que a seguir se explicam:

- **`shared_preload_libraries:`** este parâmetro encontrava-se vazio e passou a ter o valor 'pglogical'; especifica bibliotecas que deverão ser carregadas aquando da inicialização do servidor de base de dados;

- **`max_worker_processes:`** possibilita a definição do número de processos que está autorizado a correr na retaguarda, isto é, define o número de processos em simultâneo que poderão estar ativamente a satisfazer interrogações; o parâmetro possuía o valor por defeito de oito e foi incrementado para dez;

- **`wal_level:`** este parâmetro é responsável por definir a quantidade de informação que é colocada nos ficheiros de WAL, e poderá assumir os valores *'minimal'*, *'replica'* e *'logical'*; para o presente cenário, foi retirado o valor por defeito *'minimal'*, que regista exclusivamente a informação necessária para recuperação de uma falha, e foi colocado o valor *'logical'* que possibilita a descodificação lógica das operações;

- **`max_wal_senders:`** define o número de ligações que o servidor permite provenientes de outros servidores encarregues de replicar a informação do primeiro; para colocar este parâmetro com um valor superior a zero (causando a ativação da replicação), é necessário que o parâmetro anterior esteja definido como *'replica'* ou *'logical'*; o valor definido de um foi incrementado para dois;

- **`max_replication_slots:`** permite a definição do número de *slots* de replicação que o servidor suporta; cada slot de replicação tem a responsabilidade de seguir o trabalho de replicação dos *slaves* e de registar a localização de cada slave na leitura do registo WAL, para que este registo, que é rotativamente apagado, mantenha, no mínimo, toda a informação até ao ponto onde se encontra o *slave* mais atrasado; foi incrementado de um para dois;

- **`track_commit_timestamp:`** define se o motor de base de dados deve registar os *timestamps* de todas as operações de COMMIT, que poderão ser úteis na resolução de conflitos de replicação; foi retirado o valor por defeito FALSE e colocado TRUE;

Após efetuadas as alterações descritas é necessário proceder à reinicialização do serviço da base de dados através do comando '**`service postgresql restart`**'.

5.3.2.3. Alterações 'pg_hba.conf'

Este ficheiro, *pg_hba.conf*, lido e carregado também no arranque do PostgreSQL, é responsável por indicar quais são as máquinas autorizadas a ligar-se ao servidor local e com que nomes de utilizador, quais são as bases de dados a que possuem acesso, quais os segmentos IP que estão autorizados e ainda o método de autenticação.

Com base no descrito, foi, portanto, necessário autorizar mutuamente os servidores que iriam executar a replicação lógica, acrescentando para isso uma nova linha por cada servidor a autorizar, incluindo o próprio.

5.3.3. Trigger DELETE_ON_OPPOSITE

Estando o mecanismo de replicação em funcionamento, a informação flui do *site* A para o *site* B e vice-versa. Do mesmo modo, como foi introduzido no início do capítulo 5, a aplicação irá recolher a informação mais recente da base de dados, respeitante a cada terminal, esteja esta informação presente na tabela 'lastcalls_a', 'lastcalls_b' ou em ambas, sendo que neste último caso serão comparados os *timestamps* referentes às colunas consultadas em ambas as tabelas.

Na eventualidade de ocorrer eliminação de um terminal, isto é, o terminal deixará de utilizar o sistema, é um requisito que a sua informação relativa às últimas chamadas seja removida da base de dados. Uma vez que a aplicação tem a capacidade de ir buscar informação às duas tabelas, não será suficiente apagar a linha correspondente ao terminal na tabela do *site* local, já que poderá continuar a figurar informação na outra tabela.

Foi, portanto, necessário não só automatizar o processo de remoção na tabela oposta, mas também garantir que a informação é apagada. Para isso foi criado um *trigger* que remove a linha em causa na tabela contrária à tabela onde ocorreu o DELETE original:

```
CREATE OR REPLACE FUNCTION sec_sdr.delete_on_opposite() RETURNS TRIGGER AS $lastcalls$
BEGIN
DELETE FROM sec_sdr.lastcalls_a WHERE sec_sdr.lastcalls_a.terminal_id = OLD.terminal_id;
RETURN OLD;
END;
$lastcalls$ LANGUAGE plpgsql;

CREATE TRIGGER on_delete
AFTER DELETE ON sec_sdr.lastcalls_b
FOR EACH ROW
EXECUTE PROCEDURE sec_sdr.delete_on_opposite();

ALTER TABLE sec_sdr.lastcalls_b ENABLE REPLICA TRIGGER on_delete;
```

Fig. 5.7 – Código SQL respeitante à criação do *trigger* DELETE_ON_OPPOSITE no *site* A

```
CREATE OR REPLACE FUNCTION sec_sdr.delete_on_opposite() RETURNS TRIGGER AS $lastcalls$
BEGIN
DELETE FROM sec_sdr.lastcalls_b WHERE sec_sdr.lastcalls_b.terminal_id = OLD.terminal_id;
RETURN OLD;
END;
$lastcalls$ LANGUAGE plpgsql;

CREATE TRIGGER on_delete
AFTER DELETE ON sec_sdr.lastcalls_a
FOR EACH ROW
EXECUTE PROCEDURE sec_sdr.delete_on_opposite();

ALTER TABLE sec_sdr.lastcalls_a ENABLE REPLICA TRIGGER on_delete;
```

Fig. 5.8 – Código SQL respeitante à criação do *trigger* DELETE_ON_OPPOSITE no *site* B

Assim, e assumindo que a operação DELETE é propagada pelo PGLogical para o *site* oposto, quando esta operação é recebida neste último, o *trigger* irá disparar e apagar na restante tabela a informação relativa ao terminal. Na figura seguinte exemplifica-se o procedimento:

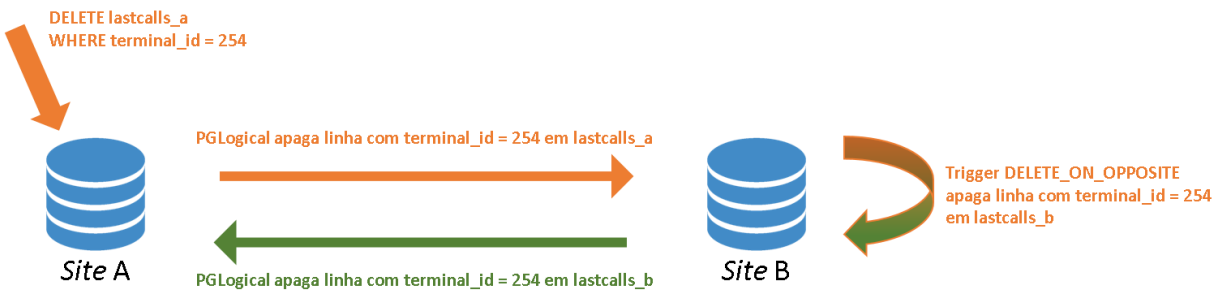

Fig. 5.9 – Esquematização da interação PGLogical/*trigger*

Desta forma se garante a completa eliminação dos dados referentes a um determinado terminal, cumprindo desta forma o requisito de eliminação de dados na situação de remoção de um terminal.

5.3.4. View LASTCALLS

Foi necessária a inclusão de uma *view* SQL na arquitetura dado que originalmente a tabela possuía o nome 'lastcalls'. A *view* possui o mesmo nome da tabela antiga (lastcalls) e permitiu alcançar um nível de abstração necessário para operações de escrita do ponto de vista da

aplicação. Não seria prático existir a necessidade de configurar a aplicação especificamente para o caso de estar a correr no *site* A ou no *site* B.

Como tal, e uma vez que a criação da *view* na base de dado faz parte da configuração inicial do sistema, não se torna necessária qualquer alteração nas operações de escrita da aplicação, nem qualquer outra configuração aplicacional dependente do *site*.

As views foram criadas da seguinte forma:

```
CREATE VIEW sec_sdr.lastcalls AS
    SELECT *
    FROM sec_sdr.lastcalls_a;
GRANT ALL ON TABLE sec_sdr.lastcalls TO sec_sdr;
```

Fig. 5.10 - Código SQL respeitante à criação da *view* LASTCALLS no *site* A

```
CREATE VIEW sec_sdr.lastcalls AS
    SELECT *
    FROM sec_sdr.lastcalls_b;
GRANT ALL ON TABLE sec_sdr.lastcalls TO sec_sdr;
```

Fig. 5.11 - Código SQL respeitante à criação da *view* LASTCALLS no *site* B

5.3.5. Ajuste de Permissões

Um ajuste de permissões provou-se necessário com o objetivo de proteger a integridade do processo de replicação. Ainda que as operações de escrita da aplicação sejam dirigidas à *view* 'lastcalls', e esta por sua vez, estando corretamente configurada, as encaminha para a tabela respetiva dependente do site, optou-se por impedir escritas na tabela correspondente ao *site* oposto; a aplicação residente no *site* A não poderá escrever na tabela 'lastcalls_b', e vice-versa. Caso isto acontecesse, a informação escrita não iria ser replicada, e na eventualidade de uma escrita correta na tabela 'lastcalls_b' efetuada no *site* B, com um 'terminal_id' igual ao escrito incorretamente no *site* A, a replicação devolveria conflito de PRIMARY KEY, o que não é desejável.

Assim, procedeu-se ao ajuste de permissões da seguinte forma:

```
ALTER TABLE sec_sdr.lastcalls_a OWNER TO sec_sdr;
ALTER TABLE sec_sdr.lastcalls_b OWNER TO sec_sdr;
GRANT ALL ON TABLE sec_sdr.lastcalls_a TO sec_sdr;
REVOKE ALL ON TABLE sec_sdr.lastcalls_b FROM sec_sdr;
GRANT SELECT ON TABLE sec_sdr.lastcalls_b TO sec_sdr;
```

Fig. 5.12 - Código SQL respeitante ao ajuste de permissões no *site* A

```
ALTER TABLE sec_sdr.lastcalls_b OWNER TO sec_sdr;
ALTER TABLE sec_sdr.lastcalls_a OWNER TO sec_sdr;
GRANT ALL ON TABLE sec_sdr.lastcalls_b TO sec_sdr;
REVOKE ALL ON TABLE sec_sdr.lastcalls_a FROM sec_sdr;
GRANT SELECT ON TABLE sec_sdr.lastcalls_a TO sec_sdr;
```

Fig. 5.13 - Código SQL respeitante ao ajuste de permissões no *site* B

5.4. Alterações ao ABC-AS

Sendo o ABC Application Server um software construído na linguagem Java, foi necessário adaptar o código para que pudesse lidar corretamente com a existência de duas tabelas 'lastcalls'. Antes de avançar para a descrição das alterações, importa detalhar o processo de interação com a base de dados realizado pela aplicação aquando da utilização de um dos dois serviços abordados; de uma forma genérica, o processo é quase idêntico, na medida em que a sequência e tipo das operações é semelhante, no entanto, existem diferenças ao nível e quais as colunas da tabela 'lastcalls' estão envolvidas em cada operação. A seguir explicam-se as etapas de uma forma genérica, transversais aos dois serviços, antes da implementação de qualquer solução:

- Utilizador marca no terminal um código de serviço, neste caso, o *52, correspondente ao serviço "*Last Number Redial*" e o *69, correspondente ao serviço "*Call Return*";

- O aplicativo, após executar determinadas validações, como validade do código de serviço, permissão de o utilizador recorrer ao serviço associado, entre outras, utiliza a tabela 'lastcalls' e recolhe os dados necessários para o serviço (as colunas recolhidas dependem do código de serviço a ser utilizado no momento), bem como a data de atualização dos mesmos, sob a forma de um *timestamp*;

- É estabelecida a chamada para o número pretendido, previamente obtido a partir da base de dados;

- São atualizadas na BD, para o terminal em causa, as colunas referentes ao último número marcado, refletindo o destinatário, data e hora da chamada acabada de realizar.

Considerando agora o novo cenário a ser implementado, em que de uma tabela passam a existir duas, cada uma contendo informação gerada num dos *sites* onde se aloja o ABC, poderá ocorrer a situação em que, para o mesmo terminal, existe informação nas duas tabelas. Desta forma, a aplicação necessita de ser capaz de recolher dados de ambas as tabelas e decidir quais os que deverão ser utilizados, com base no *timestamp* mais recente, estejam estes dados localizados na tabela A ou na tabela B.

Às etapas acima mencionadas foi acrescentada uma nova, que visa comparar os *timestamps* de atualização da informação contidos nas duas novas tabelas, 'lastcalls_a' e 'lastcalls_b'.

5.4.1. Interações com a BD por Serviço

Cada código de serviço marcado desencadeia um conjunto de operações a nível aplicacional, que por sua vez procederão de forma diferente junto da base de dados. Relativamente aos dois códigos abordados, explica-se de seguida, para cada um deles, o novo procedimento:

5.4.1.1. *52 – Last Number Redial

O serviço "*Last Number Redial*" permite ao utilizador remarcar o último número por si marcado.

Quando o utilizador marca a sequência *52, ativando, portanto, a funcionalidade respetiva, a aplicação procede à interrogação da base de dados para obter os seguintes campos das tabelas 'lastcalls_a' e 'lastcalls_b', utilizando para isso duas operações SELECT:

- **terminal_id**: identificador do terminal chamador;

- **user_id:** identificador do utilizador ao qual o terminal chamador está associado;

- **last_dialed_call:** último número marcado;

- **last_dialed_num_id:** identificador do número correspondente ao último marcado;

- **last_dialed_date:** *timestamp* correspondente à última alteração dos dados nas colunas acima;

De seguida, são feitas verificações da informação recebida:

- ‘lastcalls_a’ e ‘lastcalls_b’ retornam *false* ou as colunas necessárias estão a NULL em ambas as tabelas: não existe informação relativa ao ‘terminal_id’ em causa, logo o serviço não poderá ser utilizado. A aplicação irá devolver ao cliente uma mensagem SIP 480 “*Service Unavailable*”;

- Apenas uma das interrogações retorna *false*: indica falta de informação para o ‘terminal_id’ em causa numa das tabelas, pelo que a informação da outra é imediatamente retornada, caso as colunas necessárias não contenham NULL;

- Ambas as tabelas retornam uma linha que corresponde ao ‘terminal_id’, no entanto uma das quais contém nas colunas necessárias NULL: a informação da outra tabela é retornada;

- Ambas as tabelas retornam uma linha para aquele ‘terminal_id’ e contém informação válida: será retornada a informação correspondente à tabela na qual o campo ‘last_dialed_date’ é mais recente.

Após a realização da lógica descrita, a chamada prosseguirá para o seu destino, excetuando-se nas situações compreendidas no primeiro ponto.

Ao prosseguir com a chamada, as colunas ‘last_dialed_call’, ‘last_dialed_num_id’ e ‘last_dialed_date’ irão ser atualizadas na tabela local de cada *site*, refletindo as informações mais recentes.

5.4.1.2. *69 – Call Return

O serviço “*Call Return*” permite a remarcação do último número de onde o utilizador recebeu uma chamada, que não atendeu.

Ativada a funcionalidade pelo utilizador, utilizando a sequência *69, a base de dados será interrogada nos mesmos moldes anteriormente explicados, mas desta vez incidindo sobre as seguintes colunas:

- **terminal_id**: identificador do terminal chamador;

- **user_id:** identificador do utilizador ao qual o terminal chamador está associado;

- **last_received_call:** último número recebido, não atendido;

- **last_received_num_id:** identificador do número correspondente ao último recebido e não atendido;

- **last_received_date:** *timestamp* correspondente à última alteração dos dados nas colunas acima;

As validações efetuadas de seguida são idênticas às efetuadas no ponto anterior para o serviço "*Last Number Redial*". Após obtenção da informação mais recente, a chamada prosseguirá para o seu destino, sendo que, novamente, os campos 'last_dialed_call', 'last_dialed_num_id' e 'last_dialed_date' serão atualizados na tabela local de cada *site*, refletindo as informações mais recentes.

5.5. Integração com Procedimentos de *Disaster Recovery*

Como seria expectável em sistemas com determinado nível de complexidade, existe redundância ao nível não só do *hardware*, mas também dos serviços. Já foi também mostrado que a arquitetura existente ao nível das bases de dados contempla um determinado número de réplicas de informação e de *hardware*. Tais réplicas mostram-se de utilidade inquestionável em caso de falha de algum dos componentes, visto que rapidamente outro nó assumirá o trabalho, permitindo uma continuidade na prestação do serviço. A transição de uma máquina para outra, no caso do PostgreSQL, não é automática, principalmente se se considerar que, existindo duas instâncias de cada BD em cada *site*, uma atua como *master* e outra como *hot standby* ou *slave*; torna-se necessário, em caso de falha da instância principal, seja por falha de *hardware*, ou falha de um serviço, promover o nó secundário a primário, controlando de perto um eventual re-arranque do nó principal num estado mais atrasado da informação. Assim, a automatização deste processo deverá ser feita através de um aplicativo externo, que neste caso, é o Pacemaker.

5.5.1. Processo de Recuperação com Pacemaker

Quando é detetada uma falha não recuperável automaticamente pelo aplicativo Pacemaker, um processo mais profundo de recuperação de falhas é iniciado, contemplando uma panóplia de ações para restabelecimento do serviço, entre as quais a mudança de instância da base de dados do antigo *master* para o *slave*, promovendo este a novo *master*. Assumindo o novo *master* as suas funções, é necessário restabelecer as subscrições para que a replicação lógica fique

de novo ativa, sendo que para isso ambas as tabelas replicadas necessitam obrigatoriamente de ser truncadas para não haver conflitos de PRIMARY KEY. O Pacemaker possui a funcionalidade de execução de *scripts* em situações de recuperação, pelo que o procedimento a seguir explicado foi integrado em *scripts* de recuperação pré-existentes.

O procedimento para restabelecer a replicação lógica é composto pelos seguintes passos:

a) Estabelece-se uma ligação à base de dados local utilizando psql, e um *user* com permissões de escrita nas duas tabelas 'lastcalls';
b) Executa-se o TRUNCATE da tabela 'lastcalls' que recebe informação remota ('lastcalls_b' para o *site* A e 'lastcalls_a' para o *site* B);
c) Garante-se que não existe nenhuma subscrição da tabela 'lastcalls_a' ou 'lastcalls_b' no PGLogical, forçando a remoção da mesma com **`pglogical.drop_subscription()`**;
d) Cria-se a nova subscrição com **`pglogical.create_subscription()`**, de forma idêntica ao mostrado em 5.3.2.1;
e) A ligação psql é terminada;

Um procedimento idêntico é efetuado de seguida, a partir do nó local, mas desta vez executando operações na instância PostgreSQL do *site* remoto. A partir daqui a informação será integralmente sincronizada de novo e a replicação estará em funcionamento.

É possível encontrar uma compilação dos comandos acima descritos nos anexos C e D, correspondentes às ações a efetuar no *site* A, e nos anexos E e F, correspondentes às ações a efetuar no *site* B.

Capítulo 6

Testes à Implementação

No decorrer do presente capítulo, efetuam-se testes de diversas naturezas à solução implementada, com o objetivo de garantir uma fiabilidade mínima que permita a sua integração no produto existente.

Numa primeira fase, são conduzidos alguns testes de funcionalidade cujo objetivo será testar se a solução apresenta um comportamento ideal nas situações que se enquadram naquilo que se pretende que seja o seu modo de funcionamento normal, não devolvendo erros ou outro tipo de ocorrências indesejadas.

Os testes de robustez incluem duas vertentes: a introdução de instabilidade na rede de comunicação e a introdução de conflitos nos dados contidos nas tabelas replicadas. No final, é expectável que as tabelas se encontrem absolutamente idênticas em ambos os locais e a replicação mantenha o seu funcionamento normal.

Finalmente, é feito um teste de performance ao *plugin* PGLogical, por forma a avaliar a capacidade de este software lidar com cargas significativas.

O cenário de testes inclui dois servidores de base de dados, com replicação lógica ativa, um servidor a correr o aplicativo ABC-AS e ainda três computadores com *softphones* instalados, nomeadamente o software Zoiper, na versão 3.15.40006 32 *bit*.

6.1. Testes de Funcionalidade

A primeira bateria de testes efetuada sobre a implementação, uma vez estando esta concluída, focou-se em testar a funcionalidade da solução como um todo, garantindo que os requisitos mínimos estivessem cumpridos, e que os sistemas apresentassem comportamentos

expectáveis nos diversos casos de utilização. Estes testes foram divididos em duas partes, sendo que a primeira parte é dirigida à base de dados, em que se experimentam todas as funcionalidades relativas ao cenário de replicação lógica, relativas à utilização da *view* e se comprova a funcionalidade do *trigger*; a segunda parte é dirigida à aplicação, onde se verifica a recolha de dados da base e dados, a decisão baseada nos *timestamps* guardados e a correta devolução da informação.

6.1.1. Avaliação de Funcionalidade da Base de Dados

Como foi já explicado, numa primeira fase os testes de funcionalidade orientaram-se à base de dados. Como tal, para efetuar estes testes foi necessário alterar manualmente as configurações do *datasource* da plataforma Rhino, alterando o IP relativo à base de dados de registos para que os pedidos fossem consistentemente enviados para apenas uma das instâncias e se conseguisse simular escritas provenientes de ambos os *sites*.

Procurou-se que os testes efetuados para comprovar o funcionamento dos mecanismos ao nível da base de dados fossem o mais abrangentes possível, de modo a garantir que todas as situações que pudessem vir a ocorrer em ambiente de produção estivessem cobertas e obtivessem resultados positivos de funcionamento. Desta forma, as situações experimentadas nestes testes foram:

- Inserção de novos dados; permite validar o funcionamento da *view* e replicação da nova linha inserida;

- Atualização de dados existentes; valida novamente o funcionamento da *view* e replicação da informação atualizada;

- Remoção de uma linha; permite testar o funcionamento da *view*, da replicação e ainda do *trigger* de remoção remota.

Para validar os pontos que acima de descrevem, foram utilizados comandos simples SQL para interagir com a informação constante da base de dados, avaliando simultaneamente o seu comportamento.

Assim, para testar a inserção de novos dados, foi feito um INSERT na base de dados respeitante a cada *site*, utilizando um ‘terminal_id’ e um ‘user_id’ aleatórios e diferentes para cada um, e contendo os restantes campos a NULL. Verificou-se que, após inserção na *view* ‘lastcalls’ com um comando semelhante a “**`INSERT INTO sec_sdr.lastcalls VALUES`**

`(...)`", no *site* A a informação era corretamente colocada em 'lastcalls_a' e, no *site* B, corretamente colocada em 'lastcalls_b'. De seguida, com o objetivo de comprovar a replicação, efetuou-se um SELECT à tabela oposta ao *site* para verificar a existência da nova linha: no *site* A inquiriu-se a tabela 'lastcalls_b' e no *site* B a tabela 'lastcalls_a' e comprovou-se a correta replicação das novas linhas.

Foi, de seguida, efetuada uma operação de UPDATE às duas linhas anteriormente inseridas por intermédio da *view*; comprovou-se novamente o correto funcionamento da mesma, bem como da replicação da atualização aos dados.

Finalmente, foi necessário testar o mecanismo de eliminação de linhas; esperava-se que, ao apagar uma linha num dos *sites*, esta deixasse de figurar no conjunto de dados total. Para melhor verificar o funcionamento do mecanismo, inseriu-se uma linha em cada *site* contendo o campo 'terminal_id' idêntico, pelo que desta forma a mesma informação estivesse presente nas quatro tabelas. De seguida, apagou-se a linha inserida, utilizando um comando DELETE, no *site* B. Verificou-se que, tal como se esperava, o mecanismo operou corretamente e a informação foi totalmente apagada.

No final das experimentações acima descritas, concluiu-se que os mecanismos de movimentação de informação ao nível da base de dados estavam em pleno funcionamento, pelo que se podia avançar para um novo âmbito de testes.

6.1.2. Avaliação de Funcionalidade da Aplicação

Do ponto de vista aplicacional, e assumindo que as funcionalidades ao nível da base de dados estão em pleno, pretende-se validar a recolha e tratamento da informação proveniente da mesma e a sua correta utilização no serviço.

Assim, ao nível da aplicação foram validadas as situações em que:

- Existe informação relativa a um terminal na tabela **local** e não na remota;

- Existe informação relativa a um terminal na tabela **remota** e não na local;

- Existe informação relativa a um terminal em **ambas** as tabelas e a tabela **local** contém informação mais recente;

- Existe informação relativa a um terminal em **ambas** as tabelas e a tabela **remota** contém informação mais recente.

Para validação dos cenários acima elencados, utilizou-se um procedimento o mais próximo do real possível, o que significa que as funcionalidades foram testadas por intermédio da realização de chamadas reais utilizando *softphones*. Dado que a lógica interna da aplicação é semelhante para os dois serviços, "*Call Return*" e "*Last Number Redial*", apenas o último foi alvo de experimentações. Desta forma, foram validados os quatro fluxos seguintes:

a) Fluxo 1 – Simula registo e utilização de informação local

O primeiro fluxo submetido a teste tem como objetivo o teste do cenário em que a informação a ser utilizada está na tabela correspondente ao *site* local. Para efetuar o teste, efetua-se uma chamada do utilizador A para o utilizador B. Seguidamente, e após a chamada ter sido terminada, o utilizador A marca o código de serviço *52, esperando que a aplicação encaminhe a sua chamada novamente para B. A aplicação recolhe a informação correspondente da base de dados e estabelece corretamente a chamada para B, comprovando o correto funcionamento da funcionalidade;

b) Fluxo 2 – Simula utilização de informação remota

O segundo cenário tem por base a utilização da informação guardada no fluxo anterior, no entanto, é utilizada do ponto de vista do *site* oposto. Para isso, foi necessário alterar a base de dados com a qual a aplicação interage, para simular desta forma a mudança de operação do *site* primário para o *site* secundário. Após a alteração, o utilizador A marca o código de serviço *52, com o objetivo de contactar novamente B. A aplicação verificará a existência da informação na tabela correspondente ao *site* primário, estabelecendo de seguida a chamada para B.

c) Fluxo 3 – Simula utilização de informação local, obrigando à comparação de *timestamps*

O fluxo 3 pretende simular a existência de informação relativa a um utilizador nas duas tabelas, fazendo com que haja lugar a uma comparação de *timestamps* relativos à atualização dessa mesma informação. Asism, o utilizador A efetua uma chamada para C, sendo que, após o seu término, a aplicação procede ao seu registo na base de dados. De seguida, o utilizador A marca o código *52 que leva a que haja uma

comparação da informação contida em ‘lastcalls_a’ e ‘lastcalls_b’. A chamada é então estabelecida para C, já que foi quem recebeu por último uma chamada de A.

d) Fluxo 4 – Simula a utilização de informação remota, obrigando à comparação de *timestamps*:

Por último, o fluxo 4 testa a utilização de informação remota com comparação de *timestamps.* Após a validação do fluxo 3, repõe-se a ligação da aplicação à base de dados original, simulando nova mudança de *site*. Para efetuar o teste, bastará o utilizador marcar novamente *52. Uma vez que a última chamada por si efetuada foi no *site* oposto e teve como destino o utilizador C, a aplicação estabelecerá novamente uma chamada para esse mesmo utilizador.

No final dos testes acima, que todos retornaram sucesso, considerou-se que, ao nível da aplicação, a funcionalidade era plena.

6.2. Testes de Robustez

Garantido o funcionamento da arquitetura implementada, por intermédio dos testes acima descritos, foi necessário avaliar a robustez da mesma por intermédio de alguns testes que introduziram certo tipo de instabilidades ou situações de carga, verificando no final a consistência dos dados presentes nas tabelas em causa. Estes ensaios basearam-se em duas vertentes de falha, provenientes de duas diferentes origens.

Os primeiros testes de robustez pretendiam simular distúrbios na rede de comunicação entre os dois *sites*, prejudicando desta forma a comunicação entre os dois servidores de base de dados. É inegável à partida a existência de instabilidade numa ligação de dados a grande distância, pelo que neste teste é fundamental obter a garantia de bom funcionamento do sistema de replicação. Assim, e por forma a simular condições marcadamente adversas na rede, foi feita uma simulação de carga que durou vinte minutos e que contava com uma rede de comunicação com as características que se observam no seguinte quadro:

Latência	**5000ms ± 1000ms**
Perda	**25 %**

Corrupção	**25%**
Duplicação	**25%**
Fora de Ordem	**25%**

Tabela 6.1 – Características da rede de comunicação

Tais condições na rede foram introduzidas utilizando para isso o pacote 'tc' do Linux, um pacote que permite a aplicação de medidas de controlo de tráfego às interfaces de rede de uma máquina, sob a forma do comando "**`tc qdisc add dev bond0 root netem delay 5000ms 1000ms 25% loss 25% 25% duplicate 25% currupt 25% reorder 25% 50%`**". A título demonstrativo, executou-se um comando de *ping* dirigido ao servidor onde estas condições foram colocadas, que mostrou com clareza a adversidade do ambiente de rede, introduzida pelo filtro mencionado acima:

```
PING blade4.ct.ptin.corppt.com (10.112.76.35) 56(84) bytes of data.
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=1 ttl=64 time=1042 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=2 ttl=64 time=101 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=3 ttl=64 time=522 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=4 ttl=64 time=1010 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=5 ttl=64 time=1178 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=6 ttl=64 time=729 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=6 ttl=64 time=773 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=7 ttl=64 time=1336 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=8 ttl=64 time=781 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=9 ttl=64 time=687 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=9 ttl=64 time=1442 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=10 ttl=64 time=1613 ms
64 bytes from blade4.ct.ptin.corppt.com (10.112.76.35): icmp_seq=11 ttl=64 time=1001 ms
```

Fig. 6.1 – Demonstração de resultado do comando *ping*

Para geração de carga utilizou-se um gerador de transações incluído já na distribuição do PostgreSQL, o 'pgbench', que permitiu gerar transações correspondentes a operações de INSERT na tabela 'lastcalls_b', com a cadência definida de uma inserção por segundo. No final da execução, verificou-se a consistência do processo de replicação por intermédio da contagem das linhas existentes quer na tabela e origem, quer na tabela de destino, o que mostrou que a replicação funcionou de forma correta, caso contrário o número de linhas seria diferente. Com isto, conclui-

se que ao nível da resiliência em cenários de forte instabilidade de rede, a ferramenta PGLogical apresenta uma robustez bastante aceitável.

Ainda no campo dos testes à robustez da solução, uma segunda vertente incidiu sobre a queda de serviços, provocando a indisponibilidade dos nós de replicação. Neste segundo teste, provocou-se a terminação abrupta de uma instância de cada vez, o que invalidou a replicação durante alguns minutos, continuando, no entanto, as operações na base de dados oposta. No final, ao arrancar novamente a instância previamente parada, o sistema deveria sincronizar as tabelas, obrigando a que o resultado fosse consistente. Este teste pretende simular a indisponibilidade de um dos lados da replicação, seja por falha de um serviço ou por falha total temporária na rede, e a recuperação após tal acontecimento. O procedimento iniciou-se por garantir que a replicação estava em funcionamento e a informação fluía nas duas direções. Após isto, procedeu-se ao término do processo referente ao PostgreSQL do *site* secundário, interrompendo assim o funcionamento da base de dados, e consequentemente a replicação. A base de dados encontrava-se, no momento da paragem, contendo 1706 linhas. O teste decorreu por novos vinte minutos, efetuando uma operação de inserção por segundo na tabela "lastcalls_b". No final do teste, a base de dados do *site* primário encontrava-se da seguinte forma:

```
Blade3:
postgres=# select count(*) from lastcalls_b;
count
-------
  2907
```

Fig. 6.2 – Resultado após inserções no *site* primário

O próximo passo seria iniciar a base de dados do *site* secundário. Assim se procedeu, e, após um pequeno período de espera de cinco segundos que permitiu a sincronização, interrogou-se a base de dados solicitando uma contagem de linhas da tabela "lastcalls_b":

```
Blade4:
postgres=# select count(*) from lastcalls_b;
count
-------
  2907
```

Fig. 6.3 – Resultado após reinicialização da base de dados no *site* secundário

O resultado foi, como se mostra, favorável. O *plugin* resistiu à quebra da ligação de sincronização, conseguindo repor a consistência nos dados imediatamente após o restabelecimento do serviço.

6.3. Testes de *Performance*

Os testes de *performance* efetuados à solução incidiram sobretudo em perceber de que forma a carga ao nível da base de dados poderia influenciar a integridade do sistema. Assim, utilizando a ferramenta 'pgbench', uma ferramenta do PostgreSQL que permite efetuar testes de carga a uma determinada base de dados, procedeu-se ao teste de esforço sobre a arquitetura final criada para o projeto, incluindo a replicação de informação.

O cenário de teste envolveu a invocação da ferramenta com os seguintes parâmetros:

Concorrência (*threads*)	**8**
Ligações à BD	**8**
Tempo de Execução	**20s**
Intervalo de *Log*	**2s**
Tipo de Interrogação	**Preparada**

Tabela 6.2 – Características do teste de carga

O teste foi arrancado com as características acima mencionadas utilizando o comando "`pgbench -v -c8 -T 20 -P 2 -M prepared -j 8 -f abc_bench.sql abcdb`" e utilizando o ficheiro 'abc_bench.sql' como *template* para as operações a realizar.

No final da execução do teste a consola possuía indicação dos resultados como a seguir se mostra:

```
starting vacuum... end.
starting vacuum pgbench_accounts... end.
progress: 2.0 s, 13924.0 tps, lat 0.569 ms stddev 0.123
progress: 4.0 s, 14059.4 tps, lat 0.566 ms stddev 0.099
progress: 6.0 s, 14083.6 tps, lat 0.565 ms stddev 0.100
progress: 8.0 s, 14608.0 tps, lat 0.545 ms stddev 0.107
progress: 10.0 s, 14226.5 tps, lat 0.559 ms stddev 0.094
progress: 12.0 s, 14057.0 tps, lat 0.566 ms stddev 0.102
progress: 14.0 s, 14285.5 tps, lat 0.557 ms stddev 0.098
progress: 16.0 s, 14352.4 tps, lat 0.554 ms stddev 0.118
progress: 18.0 s, 14113.0 tps, lat 0.564 ms stddev 0.125
progress: 20.0 s, 14324.4 tps, lat 0.555 ms stddev 0.094
transaction type: abc_bench.sql
scaling factor: 1
query mode: prepared
number of clients: 8
number of threads: 8
duration: 20 s

number of transactions actually processed: 284076
latency average = 0.560 ms latency stddev = 0.107 ms
tps = 14203.376739 (including connections establishing)
tps = 14207.793011 (excluding connections establishing)
```

Fig. 6.4 – Resultados do teste de *performance*

Como seria de esperar, o número de transações por segundo (TPS) observado está intimamente relacionado com as características físicas do servidor. Neste caso específico, a máquina em questão permitiu a execução de uma média de 14203 TPS. Pelo facto de se estar a executar um teste de carga, este valor corresponde ao nível máximo de esforço por parte de um dos componentes da máquina, que no presente ensaio se observou ser o disco rígido. Por esta razão, existem operações ao nível do sistema operativo que irão ser colocadas numa fila de execução e que, consequentemente, ganharão algum atraso. Com base nisto, e comparando a complexidade de uma operação comum na base de dados (INSERT, UPDATE, ...) face à complexidade de uma operação de replicação, entende-se que esta última terá o mesmo peso da operação comum, acrescido ao trabalho de descodificação lógica do *Write Ahead Log* (WAL), e que, portanto, exigirá ligeiramente mais tempo de processamento, facto que poderá ser agravado pelo excesso de carga no sistema.

Verificou-se, portanto, que no decorrer do teste, a operação de replicação se ia atrasando sucessivamente face às operações a ser executadas em tempo real, e que por essa razão, o número

de operações não exportadas do WAL aumentava também, fazendo crescer o tamanho desde último de forma linear.

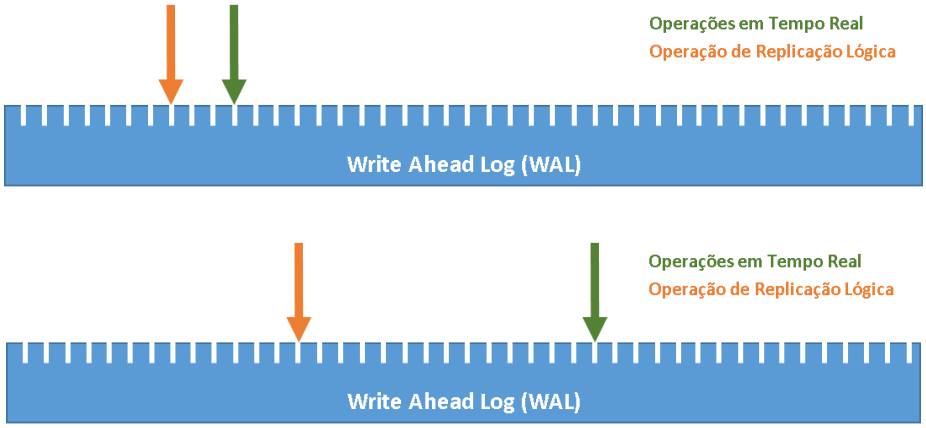

Fig. 6.5 – Representação do avanço do processamento do WAL

Adicionalmente, após término do teste, verificou-se por intermédio do pacote pmap, que a utilização da memória por parte dos *workers* de replicação se comportava igualmente de forma crescente, como mostra o excerto da consola abaixo:

```
-bash-4.1$ pmap -x 49355 | grep zero
00007f0786104000 45098400  264268  264268 rw-s-  zero (deleted)
-bash-4.1$ pmap -x 49355 | grep zero
00007f0786104000 45098400  264480  264480 rw-s-  zero (deleted)
-bash-4.1$ pmap -x 49355 | grep zero
00007f0786104000 45098400  264680  264680 rw-s-  zero (deleted)
-bash-4.1$ pmap -x 49355 | grep zero
00007f0786104000 45098400  299664  299664 rw-s-  zero (deleted)
-bash-4.1$ pmap -x 49355 | grep zero
00007f0786104000 45098400  300432  300432 rw-s-  zero (deleted)
```

Fig. 6.6 – Observação da crescente utilização de memória

Atendendo a todos estes factos, a conclusão que o presente ensaio permitiu retirar foi de que, dependendo das características físicas da máquina em questão, existe um limiar de carga em que as operações de replicação começam a ganhar atraso de forma linearmente crescente relativamente às operações em tempo real, como evidencia a figura acima. Este comportamento leva, caso a condição de sobrecarga seja mantida por tempo suficiente, a um esgotamento inevitável de recursos computacionais e a uma falha completa do sistema de base de dados, num qualquer ponto do tempo.

Capítulo 7

Conclusões e Trabalho Futuro

Acentuando-se cada vez mais a utilização de meios de comunicação eletrónicos, em específico aqueles que recorrem à tecnologia de voz sobre IP, torna-se importante que as empresas de telecomunicações prestem serviços com taxas de falha e indisponibilidades extremamente reduzidas.

Esta dissertação teve como objetivo a resolução de determinadas lacunas no produto em comercialização pela Altice Labs, o ABC – Advanced Business Communications, que levavam a que, sob determinadas circunstâncias, o serviço apresentasse limitações ao cliente. Com o intuito de resolver tais limitações, procurou-se criar uma arquitetura de transporte de informação ao nível das bases de dados que permitisse que essa mesma informação estivesse disponível em tempo real nos dois *sites* que servem de base ao serviço, para que caso exista uma falha num deles, o outro seja capaz de assumir o serviço sem que o cliente final se aperceba que tal falha aconteceu.

Em primeiro lugar foi realizado um estudo relativo à viabilidade de migração do sistema de base de dados para uma solução assente na tecnologia NoSQL, o que, por diversos motivos, se mostrou impraticável. Seguidamente, tomou-se a decisão de manter a tecnologia atual, PostgreSQL, adicionando, no entanto, uma camada adicional de replicação lógica entre os dois *sites*.

Com base na análise de trabalhos relacionados com o tema central da presente dissertação, a arquitetura desenhada, capaz de dar solução ao problema em questão, enquadra-se, segundo a classificação em três parâmetros de Wiesmann et al. [10], num modelo "*Update Everywhere"*, de interação constante e sem sinalização de término de transação. Esta qualificação baseia-se na avaliação das funcionalidades que a solução implementada disponibiliza; do ponto de vista aplicacional, torna-se agora possível a atualização da informação em qualquer um dos dois *sites*, que por via da replicação passará a estar presente nos dois, sendo que, no final, esta se apresentará

como “um todo”. A interação constante é dada pelo facto de cada transação se transmitir para o *site* oposto por si só, independentemente do número de operações em si contidas.

Com esta alteração à arquitetura, o serviço passa a beneficiar de um nível mais elevado de resiliência, sendo capaz de resistir à falha de um dos *sites* sem que o cliente final se aperceba do sucedido.

7.1. Trabalho Futuro

Futuramente, está planeada a implementação da presente solução em sistemas do ABC que utilizem como sistema de gestão de base de dados o software Oracle DB, em vez de PostgreSQL. A arquitetura de alto nível da solução será idêntica, no entanto, existirão pequenas modificações ao nível da configuração da base de dados, já que o plugin PGLogical não é compatível com Oracle DB.

Adicionalmente, outros componentes do produto ABC que atualmente utilizam PostgreSQL como sistema de gestão de base de dados e necessitam de redundância geográfica, poderão beneficiar da solução apresentada nesta dissertação.

Referências

[1] Altice Labs. 2017. "Altice Labs | Sobre Nós." accessed 5 Jun. http:// www.alticelabs.com/pt/sobre.html.

[2] Poushter, Jacob. 2016. "Smartphone ownership and Internet usage continues to climb in emerging economies." *Pew Research Center*.

[3] Codd, Edgar F. 1982. "Relational database: a practical foundation for productivity." Communications of the ACM 25 (2):109-117.

[4] Easy Computer Academy, LLC. 2017. "What Is a Database Table? - Essential SQL." accessed 14 Jun. https://www.essentialsql.com/what-is-a-database-table/.

[5] Padhy, Rabi Prasad, Manas Ranjan Patra, and Suresh Chandra Satapathy. 2011. "RDBMS to NoSQL: reviewing some next-generation non-relational database's." International Journal of Advanced Engineering Science and Technologies 11 (1):15-30.

[6] Băzăr, Cristina, and Cosmin Sebastian Iosif. 2014. "The transition from rdbms to nosql. a comparative analysis of three popular non-relational solutions: Cassandra, mongodb and couchbase." Database Syst J 5 (2):49-59.

[7] Cerf, Vinton G. 2007. "An information avalanche." Computer 40 (1).

[8] Shim, Simon SY. 2012. "Guest editor's introduction: The cap theorem's growing impact." Computer 45 (2):21-22.

[9] Brewer, Eric. 2012. "CAP twelve years later: How the" rules" have changed." Computer 45 (2):23-29.

[10] Wiesmann, Matthias, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. 2000. "Database replication techniques: A three parameter classification." Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on.

[11] Gray, Jim, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. "The dangers of replication and a solution." ACM SIGMOD Record 25 (2):173-182.

[12] Bernstein, Philip A, Vassos Hadzilacos, and Nathan Goodman. 1987. CONCURRENCY CONTROL AND RECOVERY IN DATABASE SYSTEMS: Addison− Wesley.

[13] Metaswitch. 2017. "Rhino Telephone Application Server (TAS) for Mobile." accessed 7 Jun. http://www.metaswitch.com/rhino-mobile-telephony-application-server-tas.

[14] Metaswitch. 2017. "1.4 About JAIN SLEE - Rhino v2.2 Documentation - OpenCloud Developer's Portal." accessed 7 Jun. https://developer.opencloud.com/devportal/display/RD2v2/1.4+About+JAIN+SLEE.

[15] ClusterLabs. 2017. "Pacemaker - ClusterLabs." accessed 16 Mai. http://wiki.clusterlabs.org/wiki/Pacemaker.

[16] Elmasri, Ramez, and Shamkant B Navathe. 2015. Fundamentals of database systems: Pearson.

[17] 2nd Quadrant Ltd. 2017. "pglogical Docs | 2ndQuadrant." accessed 13 Abr. https://www.2ndquadrant.com/en/resources/pglogical/pglogical-docs/.

[18] The PostgreSQL Global Development Group. 2017. "PostgreSQL: Documentation: 9.6: Write-Ahead Logging (WAL)." accessed 19 Mai. https://www.postgresql.org/docs/current/static/wal-intro.html.

Anexos

Anexo A

Script de Configuração Total *site* A

config_raiz_a.sql

```
\set node_name 'BD1'
\set local_dsn 'host=10.112.101.69 port=5432 dbname=sec_sdr'
\set remote_dsn 'host=10.112.101.71 port=5432 dbname=sec_sdr'
\set schema_name 'sec_sdr'


/* ************************************************************** */
/* **********  Trigger para deletes em aprovisionamento  *********** */


CREATE OR REPLACE FUNCTION :schema_name.delete_on_opposite()
RETURNS TRIGGER AS $lastcalls$
   BEGIN
      DELETE FROM :schema_name.lastcalls_a
      WHERE :schema_name.lastcalls_a.terminal_id = OLD.terminal_id;
      RETURN OLD;
   END;
$lastcalls$ LANGUAGE plpgsql;

CREATE TRIGGER on_delete
AFTER DELETE ON :schema_name.lastcalls_b
FOR EACH ROW
EXECUTE PROCEDURE :schema_name.delete_on_opposite();

ALTER TABLE :schema_name.lastcalls_b ENABLE REPLICA TRIGGER on_delete;
```

```
/* ************************************************************** */
/* ***  Ajuste de permissoes para acesso por parte da aplicacao  *** */


ALTER TABLE :schema_name.lastcalls_a OWNER TO sec_sdr;
ALTER TABLE :schema_name.lastcalls_b OWNER TO sec_sdr;
GRANT ALL ON TABLE :schema_name.lastcalls_a TO sec_sdr;
REVOKE ALL ON TABLE :schema_name.lastcalls_b FROM sec_sdr;
GRANT SELECT ON TABLE :schema_name.lastcalls_b TO sec_sdr;


/* ************************************************************** */
/* **********************  Criacao de view  ********************** */


CREATE VIEW :schema_name.lastcalls AS
    SELECT *
    FROM :schema_name.lastcalls_a;
GRANT ALL ON TABLE :schema_name.lastcalls TO sec_sdr;


/* ************************************************************** */
/* ******************  Configuracao do PGLogical  ***************** */


DROP EXTENSION IF EXISTS pglogical CASCADE;
CREATE EXTENSION pglogical;
SELECT pglogical.create_node(
    node_name := :'node_name',
    dsn := :'local_dsn'
);
SELECT pglogical.drop_replication_set('default');
SELECT pglogical.drop_replication_set('default_insert_only');
SELECT pglogical.drop_replication_set('ddl_sql');
SELECT pglogical.create_replication_set(
       'lastcalls_a_rep', TRUE, TRUE, TRUE, TRUE);
SELECT pglogical.replication_set_add_table(
       'lastcalls_a_rep', :'schema_name' || '.lastcalls_a', FALSE);
SET pglogical.conflict_resolution = 'last_update_wins';
```

```
/*
SELECT pglogical.create_subscription(
       'lastcalls_b_rep_subs', :'remote_dsn', ARRAY['lastcalls_b_rep'],
       false, true, ARRAY['all']);
*/
```

Anexo B

Script de Configuração Total *site* B

config_raiz_b.sql

```
\set node_name 'BD3'
\set local_dsn 'host=10.112.101.71 port=5432 dbname=sec_sdr'
\set remote_dsn 'host=10.112.101.69 port=5432 dbname=sec_sdr'
\set schema_name 'sec_sdr'


/* ************************************************************** */
/* **********  Trigger para deletes em aprovisionamento  *********** */


CREATE OR REPLACE FUNCTION :schema_name.delete_on_opposite()
RETURNS TRIGGER AS $lastcalls$
   BEGIN
      DELETE FROM :schema_name.lastcalls_b
      WHERE :schema_name.lastcalls_b.terminal_id = OLD.terminal_id;
      RETURN OLD;
   END;
$lastcalls$ LANGUAGE plpgsql;

CREATE TRIGGER on_delete
AFTER DELETE ON :schema_name.lastcalls_a
FOR EACH ROW
EXECUTE PROCEDURE :schema_name.delete_on_opposite();

ALTER TABLE :schema_name.lastcalls_a ENABLE REPLICA TRIGGER on_delete;
```

```
/* ************************************************************** */
/* ***  Ajuste de permissoes para acesso por parte da aplicacao  *** */


ALTER TABLE :schema_name.lastcalls_a OWNER TO sec_sdr;
ALTER TABLE :schema_name.lastcalls_b OWNER TO sec_sdr;
GRANT ALL ON TABLE :schema_name.lastcalls_b TO sec_sdr;
REVOKE ALL ON TABLE :schema_name.lastcalls_a FROM sec_sdr;
GRANT SELECT ON TABLE :schema_name.lastcalls_a TO sec_sdr;


/* ************************************************************** */
/* ************************  Criacao de view  ********************** */


CREATE VIEW :schema_name.lastcalls AS
    SELECT *
    FROM :schema_name.lastcalls_b;
GRANT ALL ON TABLE :schema_name.lastcalls TO sec_sdr;


/* ************************************************************** */
/* ********************  Configuracao do PGLogical  ***************** */


DROP EXTENSION IF EXISTS pglogical CASCADE;
CREATE EXTENSION pglogical;
SELECT pglogical.create_node(
    node_name := :'node_name',
    dsn := :'local_dsn'
);
SELECT pglogical.drop_replication_set('default');
SELECT pglogical.drop_replication_set('default_insert_only');
SELECT pglogical.drop_replication_set('ddl_sql');
SELECT pglogical.create_replication_set(
       'lastcalls_b_rep', TRUE, TRUE, TRUE, TRUE);
SELECT pglogical.replication_set_add_table(
       'lastcalls_b_rep', :'schema_name' || '.lastcalls_b', FALSE);
SET pglogical.conflict_resolution = 'last_update_wins';
```

```
/*
SELECT pglogical.create_subscription(
       'lastcalls_a_rep_subs', :'remote_dsn', ARRAY['lastcalls_a_rep'],
       false, true, ARRAY['all']);
*/
```

Anexo C

Script de Recuperação *site* A

script_dr_a.sql

```
#!/bin/sh

LOCAL_HOST=10.112.101.69
REMOTE_HOST=10.112.101.70
DB_NAME='sec'

psql -h $LOCAL_HOST -U postgres -d $DB_NAME -f ./config_dr_a.sql;
psql_local_exit_status=$?

if [ $psql_local_exit_status != 0 ]; then
       echo "ERROR: PSQL failed to execute the local script!" 1>&2
       exit $psql_local_exit_status
fi

echo "PSQL executed the local script successfully."

psql -h $REMOTE_HOST -U postgres -d $DB_NAME -f ./config_dr_b.sql;
psql_remote_exit_status=$?

if [ $psql_remote_exit_status != 0 ]; then
       echo "ERROR: PSQL failed to execute the remote script!" 1>&2
       exit $psql_remote_exit_status
fi

echo "PSQL executed the remote script successfully."

exit 0
```

Anexo D

Script SQL para Recuperação *site* A

config_dr_a.sql

```
\set remote_dsn 'host=10.112.101.71 port=5432 dbname=sec'
\set schema_name 'sec_sdr'

/* ************************************************************** */
/*  Limpeza da tabela de info 'remota' para evitar conflitos de PK   */

TRUNCATE :schema_name.lastcalls_b;


/* ************************************************************** */
/* *****************  Configuracao do PGLogical  ***************** */

SELECT pglogical.drop_subscription('lastcalls_b_rep_subs');
SELECT pglogical.create_subscription(
       'lastcalls_b_rep_subs', :'remote_dsn', ARRAY['lastcalls_b_rep'],
       FALSE, TRUE, ARRAY['all']);
```

Anexo E

Script de Recuperação *site* B

script_dr_b.sql

```
#!/bin/sh

LOCAL_HOST=10.112.101.70
REMOTE_HOST=10.112.101.69
DB_NAME='sec'

psql -h $LOCAL_HOST -U postgres -d $DB_NAME -f ./config_dr_b.sql;
psql_local_exit_status=$?

if [ $psql_local_exit_status != 0 ]; then
       echo "ERROR: PSQL failed to execute the local script!" 1>&2
       exit $psql_local_exit_status
fi

echo "PSQL executed the local script successfully."

psql -h $REMOTE_HOST -U postgres -d $DB_NAME -f ./config_dr_a.sql;
psql_remote_exit_status=$?

if [ $psql_remote_exit_status != 0 ]; then
       echo "ERROR: PSQL failed to execute the remote script!" 1>&2
       exit $psql_remote_exit_status
fi

echo "PSQL executed the remote script successfully."

exit 0
```

Anexo F

Script SQL para Recuperação *site* B

config_dr_b.sql

```
\set remote_dsn 'host=10.112.101.69 port=5432 dbname=sec'
\set schema_name 'sec_sdr'

/* ************************************************************** */
/*   Limpeza da tabela de info 'remota' para evitar conflitos de PK   */

TRUNCATE :schema_name.lastcalls_a;


/* ************************************************************** */
/* *****************  Configuracao do PGLogical  ****************** */

SELECT pglogical.drop_subscription('lastcalls_a_rep_subs');
SELECT pglogical.create_subscription(
       'lastcalls_a_rep_subs', :'remote_dsn', ARRAY['lastcalls_a_rep'],
       FALSE, TRUE, ARRAY['all']);
```