MSc
2.º CICLO
FCUP
2013

Sincronização de Bases de Dados Multisite
José Carlos de Moura Carvalho

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

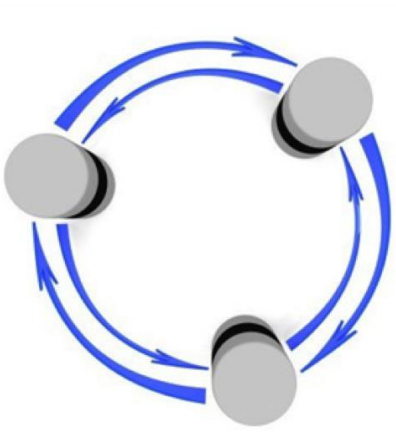
Sincronização de Bases de Dados Multisite

José Carlos de Moura Carvalho

Dissertação de Mestrado apresentada à
Faculdade de Ciências da Universidade do Porto em
Ciência de Computadores

2013

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO



Sincronização de Bases de Dados Multisite

José Carlos de Moura Carvalho
Mestrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2013

Orientador
Francisco José Ferreira de Jesus, Mestre
Faculdade de Engenharia da Universidade do Porto

Coorientador
Sérgio Armindo Lopes Crisóstomo, Doutor
Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Dedico este trabalho às pessoas mais importantes da minha vida.
À minha mãe, ao meu pai, à minha irmã, à minha avó, restante família e amigos

Resumo

A sincronização de bases de dados permite tornar um conjunto de réplicas de uma base de dados mutuamente consistentes. O conjunto das várias réplicas é visto como um sistema de base de dados distribuída.

Este projeto tem como objetivo migrar uma base de dados centralizada para base de dados distribuída. O sistema inicial é um sistema *multi-site* no qual existe apenas uma base de dados armazenada num *site* principal. Pretende-se que os restantes *sites* (*sites* secundários) possuam uma réplica da base de dados central mutuamente consistente com as restantes réplicas.

Dada a complexidade da base de dados, optamos pela estratégia de replicação . Nesta estratégia cada *site* secundário trabalha de forma independente e, periodicamente, cada *site* sincroniza as suas alterações com o *site* principal.

Foi desenvolvido um protótipo para deteção e envio das alterações feitas em cada réplica da base de dados. Este protótipo é baseado no uso de um agente de replicação em cada *site* secundário. Cada agente de replicação, periodicamente, consulta as tabelas da base de dados local para obter os registos modificados. Os registos alterados são "marcados" com uma nova versão baseada num identificador que é gerido pelo *site* principal. O agente de replicação é ainda responsável por pedir ao *site* principal atualizações feitas noutros *sites*. Assim, cada *site* tem acesso aos dados alterados noutras réplicas. No *site* principal foi desenvolvido um controlador de replicação sob a forma de uma aplicação *web*. Este controlador recebe todas as atualizações enviadas pelos agentes de replicação detetando e resolvendo eventuais conflitos entre as atualizações.

O protótipo foi testado por forma a garantir-se que, embora a replicação seja assíncrona, seja possível atingir periodicamente consistência mútua entre as réplicas da base de dados.

Abstract

The goal of database sincronization is to keep a set of replicas from a database mutually consistent. This set of replicas can be seen as a distributed database system.

In this project we wanted to migrate one centralized database into a distributed database. The initial system is a single database multi-site system kept in a main site. It is intended that each of the remaining sites (secondary sites) have a replica of the central database consistent with the other replicas.

Given the complexity of the database, a specific replication strategy is used. In replication, each site works independently and periodically synchronizes the changes with the main site.

A prototype was built for detection and synchronization of changes made in each replica. In each secondary site a replication agent is used, wich queries local database tables to obtain the modified records. The modified records are marked with a new version based on an identifier that is managed by the main site. Moreover, the replication agent is responsible for asking the main site about updates that were made in other sites. Thus, each site has access to changed data that occured on other replicas. In the main site a replication controller was developed in the form of a web application. This controller receives all updates from replication agents detecting and solving possible conflicts between updates.

The prototype was tested to ensure that although replication is asynchronous, mutual consistency of the database replicas can be achieved periodically.

Agradecimentos

Quero começar por agradecer à empresa Tactis pelo excelente acolhimento e pelas condições disponibilizadas para a realização deste projeto. Agradeço de forma especial ao meu orientador Francisco Jesus pelo que aprendi e por todo o apoio a nível profissional e pessoal demonstrado ao longo do estágio. Uma palavra de agradecimento ao diretor técnico, Mário Miguel, pela sua atenção e pelo conhecimento que me transmitiu, e às minhas restantes colegas Márcia Alves e Elsa Lopes.

Pretendo igualmente agradecer ao meu co-orientador Sérgio Crisóstomo, pela sua disponibilidade, paciência e pelas ótimas sugestões ao longo deste projeto.

Agradeço também aos restantes professores da Faculdade de Ciências pelo ambiente de aprendizagem ao longo do meu percurso académico.

Um especial obrigado aos meus pais por terem proporcionado os meios necessários à minha formação e por toda a motivação, apoio e carinho. À minha avó por acreditar sempre em mim e por estar sempre comigo. Não podia deixar de agradecer à minha irmã que ao longo destes anos esteve sempre disponível para me ouvir e para me dar a força necessária.

Agradeço à minha namorada, Jessica Monteiro, pelo carinho e apoio demonstrado ao longo desta etapa da minha vida. Foi com uma sincera e forte amizade que me acompanhaste, amizade essa que tornou possível tudo o que hoje representas para mim.

Finalmente agradeço a todos os meus amigos que, de uma forma ou de outra, estiveram presentes nesta caminhada. Gostava de deixar um agradecimento especial ao Gonçalo Martins, Jorge Gonçalves, Marcelo Teixeira, Daniel Mota, Daniel Moreira, João Rodrigues, Carlos Ferreira, Luís Lobo, Roberto Gouveia, Roberto Caldas, Tânia Lemos e Telma Machado pela amizade e companheirismo.

A todos, um sincero muito obrigado.

Conteúdo

Lista de Tabelas

Lista de Figuras

Acrónimos

2PC Two-Phase Commit Protocol

ACID Atomicidade, Consistência, Isolamento e Durabilidade

API Application Programming Interface

BSD Berkeley Software Distribution

CDDL Common Development and Distribution License

CAP Consistency, Availability, Partition tolerance

CPU Central Processing Unit

DBMS Database Management System

DOM Document Object Model

ECA Event–Condition–Action

EPL Eclipse Public License

EJB Enterprise JavaBeans

GPL GNU General Public License

GUI Graphical User Interface

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

IDE Integrated Development Environment

JDBC Java Database Connectivity

JPA Java Persistence API

JSON JavaScript Object Notation

JAX-WS Java API for XML Web Services

JAX-RS Java API for RESTful Web Services

JAX-RPC Java API for XML-based RPC

JVM Java Virtual Machine

Java EE Java Platform Enterprise Edition

JTA Java Transaction API

LGPL Lesser General Public License

MIME Multipurpose Internet Mail Extensions

PC2PL Primary Copy Two-Phase Locking

PL/SQL Procedural Language/Structured Query Language

RMI Remote Method Invocation

SQL Structured Query Language

SOAP Simple Object Access Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

VPN Virtual Private Network

WAL Write-Ahead Log

XML eXtensible Markup Language

Este relatório de estágio foi escrito ao abrigo do novo Acordo Ortográfico.

Capítulo 1

Introdução

O crescente volume de dados produzidos por sistemas de informação, conjugado com a dispersão geográfica, motiva a migração de bases de dados centralizadas para sistemas de bases de dados distribuídas. A utilização de apenas uma base de dados central constitui um ponto de falha único, onde a indisponibilidade da ligação ao servidor de base de dados impede o normal funcionamento do sistema.

Em sistemas de bases de dados distribuídas, os dados são distribuídos por diferentes servidores de bases de dados que podem estar localizados em diversos pontos geográficos. Cada um destes pontos é denominado por *site*. Entre os vários servidores é criado um mecanismo de comunicação para coordenar as alterações aos dados.

O desenho de uma base de dados distribuída embora seja complexo, diminui a dependência de ligação a um único servidor e permite a criação de várias réplicas de uma base de dados. Entre as várias réplicas é possível criar mecanismos de sincronização para garantir que todas as réplicas possuam os mesmos dados. Assim, é possível otimizar a consulta aos dados, resultando num melhor tempo de resposta.

O projeto descrito neste relatório foi realizado no âmbito da sincronização de bases de dados num cenário *multi-site*. O objetivo é migrar um sistema de base de dados centralizada num sistema a operar sobre uma base de dados distribuída. A base de dados encontrava-se armazenada num *site* (*site* principal) à qual os restantes *sites* (*sites* secundários) se ligam remotamente para acederem aos dados. Quando não existe ligação à base de dados principal, os *sites* secundários não têm acesso aos dados. Utilizando bases de dados distribuídas é possível criar um conjunto de réplicas que, através de um mecanismo de sincronização, contêm os mesmos dados.

1.1 Apresentação da Tactis

A fundação da empresa Tactis data de Maio de 2003, quando antigos sócios e colaboradores da empresa CDFTel decidiram investir e atuar na área de aplicações de gestão para a saúde, tendo optado em particular pela especialidade de medicina dentária.

A empresa CDFTel dedicava-se ao desenvolvimento de aplicações para a gestão de sistemas críticos de telecomunicações, cujas características muito específicas obrigavam a que estas aplicações tivessem por base uma arquitetura robusta e bastante imune a falhas.

A Tactis é uma referência no fornecimento de aplicações de gestão para a área da saúde privada e tem como missão o desenvolvimento e suporte das suas aplicações, bem como a prestação de serviços adequados às necessidades dos seus clientes. Como aplicações de referência, a Tactis é proprietária do *software* Novigest para gestão de clínicas e do *software* Novipem para prescrição eletrónica de medicamentos.

1.2 Contextualização

A empresa comercializa um *software* certificado para gestão de consultórios médicos: o Novigest. O Novigest assenta na tecnologia de um sistema cliente-servidor. Este sistema é constituído por duas componentes: o Noviclient e o Noviserver. O Noviclient é uma aplicação cliente que se liga remotamente ao servidor aplicacional Noviserver para acesso aos dados. O Noviserver comunica com um único servidor de base de dados onde estão armazenados todos os dados do sistema.

Alguns dos clientes da empresa são constituídos por grupos de clínicas. Existem clínicas que, embora pertencendo ao mesmo grupo, se encontram dispersas geograficamente. Cada clínica corresponde a um *site* diferente.

No *site* principal existem aplicações Noviclient, um servidor Noviserver e um servidor de base de dados. Os restantes *sites* são *sites* secundários. Cada *site* secundário tem várias aplicações Noviclient que acedem ao Noviserver através da Internet. A aplicação Noviclient é utilizada por médicos e assistentes (ver Figura 1.1).

A ligação entre *sites* é protegida utilizando uma rede Virtual Private Network (VPN). Cada *site* possui o seu próprio certificado para garantir a privacidade do tráfego entre as aplicações Noviclient e o servidor Noviserver.

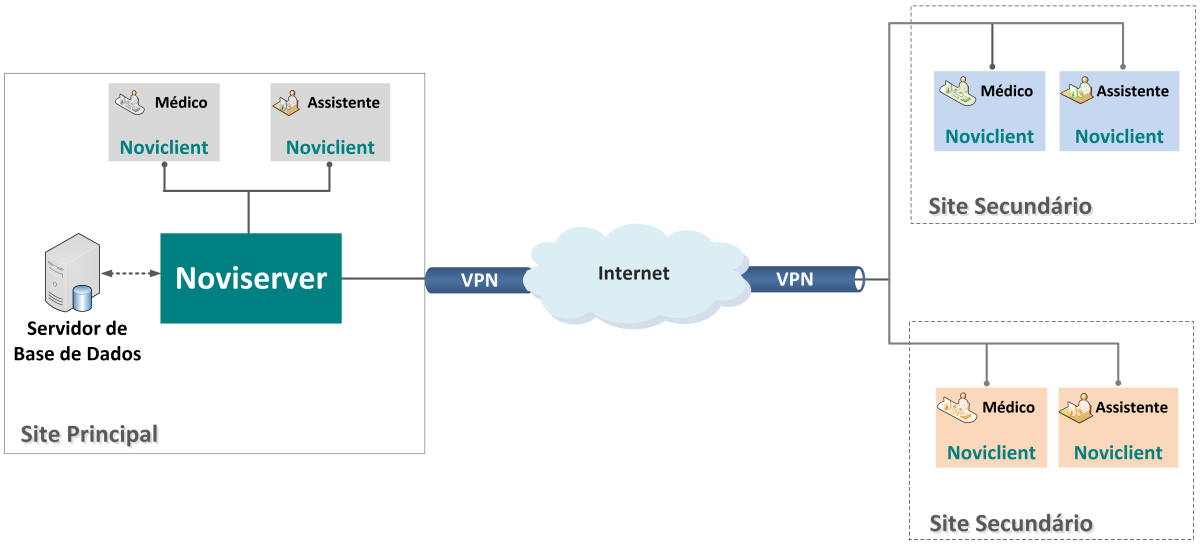


Figura 1.1: Visão geral do sistema Novigest.

1.3 Motivação

No sistema Novigest a base de dados encontra-se centralizada num *site*. Como consequência, os restantes *sites* estão dependentes da ligação à Internet para acesso aos dados. Quando, num *site* secundário, a ligação à Internet está inacessível, as aplicações Noviclient desse *site* ficam sem acesso ao Noviserver e, consequentemente, sem acesso à base de dados. Sendo o Novigest um sistema que depende inteiramente do acesso aos dados, os médicos e assistentes de um *site* que não tenha acesso à base de dados não poderão utilizar a aplicação.

Atualmente, quando em algum *site* não existe ligação remota à base de dados, os assistentes telefonam para o *site* principal e pedem para consultar os dados de um paciente.

Em *sites* com vários médicos a trabalhar em simultâneo, esta abordagem não é viável pois trata-se de um grande número de pacientes por dia e é conveniente consultar todo o histórico de um paciente para que se possa efetuar um tratamento.

Surge então a necessidade de criar um mecanismo capaz de, por um lado, continuar a funcionar em modo *offline* e, por outro, ser capaz de sincronizar os dados alterados com a base de dados central assim que a ligação for reposta.

Com um sistema de sincronização capaz de lidar com o modo de funcionamento *offline*,

uma clínica não tem de estar dependente da ligação à Internet. Todo o processo é transparente para quem utiliza o *software* Novigest, visto que poderá continuar a trabalhar normalmente sem ligação ao servidor de base de dados central.

1.4 Arquitetura do Novigest

O Novigest é um *software* multiplataforma que permite a gestão de consultórios. Este *software* é constituído por duas componentes: o Noviserver e o Noviclient.

O Noviclient é uma aplicação Graphical User Interface (GUI). Esta aplicação disponibiliza um conjunto de funcionalidades como: gerir a agenda de médicos numa clínica, consultar e alterar o histórico clínico de um paciente e adicionar novos pacientes. O Noviclient é utilizado como *interface* gráfica entre os utilizadores e os dados armazenados num servidor de base de dados;

O Noviserver é o servidor aplicacional ao qual as aplicações Noviclient se ligam. Este servidor recebe pedidos do Noviclient para acesso aos dados. Os dados encontram-se armazenados num servidor de base de dados com o qual o Noviserver comunica.

O servidor de base de dados armazena a base de dados utilizada pelo sistema Novigest e responde a pedidos do Noviserver (ver Figura 1.2). No servidor de base de dados, o Database Management System (DBMS)[1] utilizado é o PostgreSQL [1].

O Novigest foi desenvolvido na linguagem Java e utiliza a tecnologia Remote Method Invocation (RMI) para comunicação entre o Noviserver e Noviclient. A utilização desta linguagem deve-se ao facto da sua total interoperabilidade, bastando ter instalada uma versão da Java Virtual Machine (JVM). Além desta última vantagem, o Java é uma linguagem que possui inúmeras Application Programming Interface (API)s e *frameworks* para utilização de diversos recursos.

O servidor Noviserver possui uma *cache* onde guarda temporariamente alguns registos da base de dados. A *cache* do Noviserver reduz o número de pedidos ao servidor de base de dados. Todos os dados guardados em *cache* estão consistentes com a base de dados.

A aplicação Noviclient envia pedidos de leitura e escrita de dados ao Noviserver. Quando o Noviserver recebe um pedido de leitura, verifica se possui em *cache* os registos da base de dados necessários para responder à aplicação Noviclient. Caso possua, devolve a

[1] Um DBMS é um *software* constituído por um conjunto de programas que permitem a um utilizador criar e gerir bases de dados.

resposta sem consultar a base de dados. Caso contrário, (1) consulta a base de dados, (2) obtém os registos, (3) guarda os registos em *cache* e (4) devolve a resposta ao Noviclient.

Se o pedido recebido pelo Noviserver for de escrita, o Noviserver comunica com o servidor de base de dados, guardando na base de dados os registos vindos no pedido. Assim que os registos são guardados na base de dados, o Noviserver atualiza a sua *cache*.

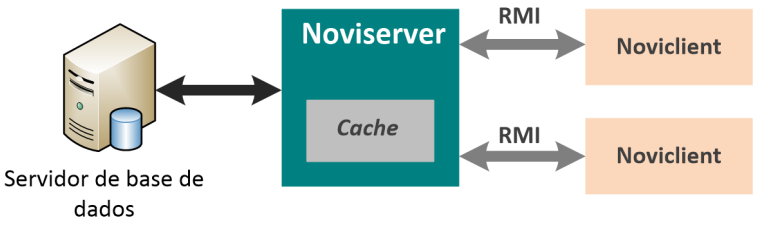


Figura 1.2: Arquitetura do sistema Novigest.

Restrições do Novigest para Implementação de Bases de Dados Distribuídas

O *software* Novigest foi sofrendo várias atualizações à sua estrutura. Estas atualizações resultaram da criação de novas funcionalidades, acrescentando complexidade à sua estrutura. A complexidade do Novigest resulta das seguintes características:

- **Complexidade da base de dados:** a base de dados utilizada pelo Novigest apresenta uma estrutura pouco *standard*. O grande número de relações entre as tabelas da base de dados torna difícil a sua fragmentação.

- **Mecanismo de *cache*:** o Noviserver armazena na sua *cache* alguns registos, o que reduz o número de acessos à base de dados. O mecanismo de *cache* pressupõe que apenas um servidor Noviserver possa alterar os registos na base de dados. Caso os dados sejam alterados na base de dados sem se atualizar de seguida a *cache*, o Noviserver deteta uma inconsistência entre os registos guardados na base de dados e na *cache*, abortando as transações correntes;

- **Transações complexas:** cada transação entre o Noviserver e o servidor de base de dados segue um procedimento específico criado pelo Noviserver. Como exemplo, atualmente é difícil detetar se uma transação tem como objetivo a criação ou alteração de registos;

1.5 Principais Objetivos

O cenário de partida para este projeto é um cenário *multi-site* no qual existem um servidor de base de dados centralizado num *site* principal e vários *sites* secundários ligados remotamente ao servidor de base de dados (através do Noviserver). Pretende-se que qualquer *site* passe a ter acesso aos dados quando perde a ligação à Internet. Para isso, o objetivo principal é migrar a base de dados central para base de dados distribuída sem alterar as aplicações Noviserver e Noviclient.

Dado que o *software* Novigest apresenta características que condicionam a criação de vários fragmentos da base de dados, optou-se por replicar os dados pelos vários *sites*.

Os principais objetivos deste projeto são:

- **Suportar modo *offline*:** permitir que o Noviclient possa funcionar sem acesso à Internet, continuando, no entanto, a ter acesso aos dados e a poder efetuar alterações aos mesmos na base de dados;
- **Suportar sincronização de alterações:** possibilitar a sincronização dos dados alterados durante o período em que o *site* esteve sem acesso à Internet;
- **Minimizar alterações ao sistema Novigest e base de dados:** minimizar alterações às aplicações Noviserver ou Noviclient. Dado que o sistema Novigest é um *software* complexo, não se pretendem criar modificações significativas aos módulos existentes. Além disso, pretende-se minimizar as alterações ao esquema da base de dados atual;
- **Ser configurável:** tornar o sistema de sincronização modular para que permita alterações a curto prazo. O objetivo é poder configurar-se os dados (tabelas e colunas) que se pretendem sincronizar em cada *site* e a periodicidade dessa sincronização;
- **Ser multiplataforma:** garantir que as tecnologias a utilizar no projeto são *standard* e multiplataforma (visto que existem clientes com sistemas operativos Linux, Windows e MacOS);
- **Ser escalável:** o sistema terá de suportar novas bases de dados com o decorrer do tempo, uma vez que o número de bases de dados a utilizar não é uma constante;
- **Utilizar tecnologias *open-source*:** a empresa apresentou preferência por uma solução de *software* não proprietário e *open source*.

1.6 Estrutura do Relatório

Este relatório é constituído por seis capítulos.

No Capítulo 2 é feita uma introdução às bases de dados distribuídas e são identificadas vantagens e desvantagens na sua utilização. Descrevem-se também as estratégias para implementação de bases de dados distribuídas.

O Capítulo 3 apresenta os principais projetos existentes para replicação de bases de dados. Em cada projeto apresentado é feita uma análise das suas principais características, vantagens e desvantagens.

No Capítulo 4 propomos uma solução para migração da base de dados do Novigest para uma base de dados distribuída. Neste capítulo são descritas abordagens para deteção de alterações a registos da base de dados e a arquitetura da solução.

No Capítulo 5 descrevemos a implementação de um protótipo da solução proposta. Apresentamos a arquitetura do sistema, tecnologias utilizadas e implementação.

Por fim, o Capítulo 6 apresenta as principais conclusões deste projeto, apresentando também alguns aspetos a melhorar no sistema proposto.

Capítulo 2

Bases de Dados Distribuídas

Neste capítulo é feita uma introdução a bases de dados distribuídas sendo apresentadas as principais vantagens e desvantagens da sua utilização. Apresentamos várias estratégias para a sua implementação indicando os cenários em que se inserem.

2.1 Introdução a Base de Dados Distribuídas

As bases de dados distribuídas surgiram da junção de duas tecnologias: bases de dados e redes de comunicação. Através de uma rede de computadores é possível distribuir o processamento e armazenamento de dados [26].

Uma base de dados distribuída é um conjunto de bases de dados logicamente relacionadas e distribuídas ao longo de uma rede de computadores. Contrariamente a bases de dados centralizadas, os dados são armazenados em diferentes *sites* [26] [30].

De forma a gerir os dados de um sistema de bases de dados distribuídas são utilizados DBMSs distribuídos. Um DBMS distribuído pode ser homogéneo, quando o *software* utilizado como DBMS nos vários *sites* é igual, ou heterogéneo quando o *software* utilizado como DBMS em cada *site* é diferente [26].

Uma base de dados distribuída deve garantir [25]:

- **Transparência na distribuição:** os utilizadores devem ser capazes de interagir com o sistema como se este se tratasse de um sistema centralizado. O acesso aos dados deve ter a mesma *performance* que num sistema de base de dados centralizado;

- **Transparência nas transações:** cada transação deve manter integridade em todas as bases de dados do sistema.

Transações

Uma transação é uma unidade lógica de instruções na base de dados. Pode ser um programa, parte de um programa ou um simples comando Structured Query Language (SQL). Consiste na execução de uma ou mais instruções sobre a base de dados e é constituída por um conjunto de estados [25]. Estes estados estão representados na Figura 2.1.

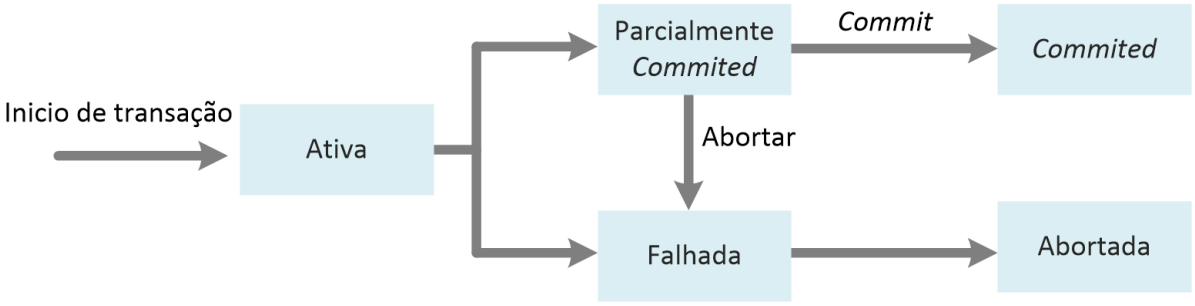


Figura 2.1: Estados de uma transação [25]

Assim que uma transação inicia (início de transação), ela fica no estado de ativa. Após estar ativa, uma transação pode ter dois estados finais possíveis: *commited* ou abortada. No primeiro caso, a transação é executada com sucesso e não pode ser abortada. Isto significa que os dados afetados por ela apenas podem ser novamente alterados caso seja feita uma nova transação que reverta as alterações. Por outro lado, se uma transação falha é abortada. Quando é abortada, as alterações aos dados são revertidas através de um sistema de *rollback*. Quando é utilizado o mecanismo de *rollback*, a transação pode ser re-iniciada mais tarde e, dependendo da causa de falha, pode ser executada com sucesso. Na Figura 2.1, após o estado de ativa, a transação pode falhar (por violar a integridade referencial, por exemplo) e é considerada abortada. Caso não falhe é parcialmente *commited* onde de seguida falha ou faz *commit*.

Propriedades de Transações

Todas as transações devem garantir um conjunto de propriedades, sendo elas: atomici-

dade, consistência, isolamento e durabilidade. São conhecidas por propriedades ACID [25] [31]:

- **Atomicidade:** esta propriedade determina que uma transação deve ser executada na sua integra ou então nada deverá ser executado. Quem garante a atomicidade é o sistema de recuperação do DBMS;
- **Consistência:** uma transação deve ser executada sobre uma base de dados consistente e quando terminar a sua execução, a base de dados deverá continuar consistente. O DBMS e o programador da aplicação que acede à base de dados são responsáveis por garantir a consistência;
- **Isolamento:** as transações são executadas de forma independente umas das outras, ou seja, uma transação não deverá ter acesso aos efeitos parciais de outra. Deverá ser o sistema de controlo de concorrência a garantir esta propriedade;
- **Durabilidade:** o resultado de uma transação que tenha sido executada com sucesso (*commited*), tem de ser permanentemente guardado na base de dados. Esta propriedade é garantida pelo sistema de recuperação.

2.2 Vantagens e Desvantagens de Bases de Dados Distribuídas

Os sistemas de bases de dados distribuídas têm as mesmas vantagens que os sistemas de bases de dados centralizados. No entanto apresentam algumas desvantagens [30][25]:

Como principais vantagens, identificamos:

- **Reflecte a estrutura de uma organização:** várias empresas têm escritórios espalhados por diferentes pontos geográficos. Assim, cada um desses pontos pode deter o seu próprio fragmento da base de dados. Com base nos fragmentos, é possível analisar como se encontra distribuída uma organização;
- **Partilha e autonomia dos dados:** visto que os dados podem ser fragmentados, é possível alguns departamentos de uma empresa, separados geograficamente, terem os seus próprios dados e acederem aos dados de outros departamentos.

- **Disponibilidade:** dependendo da implementação do sistema, um computador não necessita de estar dependente da ligação aos restantes para utilização dos dados;
- **Crescimento modular:** em empresas de grande dimensão é normal a expansão dos dados a armazenar. Com um sistema de base de dados distribuído é relativamente fácil adicionar novas bases de dados ao sistema sem afetar a sua *performance*. Em sistemas centralizados, o crescente volume de dados a armazenar, tem implicações no tempo de resposta do sistema sendo necessária a aquisição de novo *hardware*.

Como desvantagens, podemos destacar:

- **Complexidade:** um sistema de bases de dados distribuído que apresente uma boa *performance* e disponibilidade, tem inevitavelmente uma complexidade maior do que os sistemas de bases de dados centralizados;
- **Custo:** aumentar a complexidade de um sistema de base de dados significa um aumento de custos de manutenção. A ligação entre os vários computadores requer *hardware* adicional;
- **Segurança:** em sistemas centralizados, o acesso aos dados é feito através de um ponto único. Em sistemas distribuídos, existem vários pontos de acesso, o que requer um maior controlo por forma a garantir segurança. Por outro lado, a comunicação entre os vários computadores é um ponto de falha na segurança, se não for devidamente tratada;
- **Desenho complexo:** o desenho de uma solução distribuída é um desafio maior do que o desenho de uma solução centralizada. No desenho de uma solução distribuída é necessário ter em conta a forma como os fragmentos são distribuídos pelos diferentes *sites*.

2.3 Estratégias para Implementação de Bases de Dados Distribuídas

Quando se implementa um sistema de bases de dados distribuído é necessário o levantamento das características que se pretendem implementar no sistema. Com base nessas características é escolhida a estratégia para implementação de bases de dados distribuídas.

As duas estratégias para implementação de bases de dados distribuídas são: fragmentação e replicação. A fragmentação é dirigida a bases de dados onde não seja necessário guardar toda a informação em todos os *sites*. A replicação é utilizada para replicar os dados de forma parcial ou total [29]. Quando se utiliza uma estratégia de replicação é essencial perceber quando deve ser feita a propagação de alterações aos dados e para onde devem ser enviadas.

A propagação das alterações pode ser executada em duas ocasiões específicas: em simultâneo ou após a transação ser guardada na base de dados local. Existem dois métodos que asseguram esta propagação: o método síncrono e o método assíncrono [28].

Outro aspecto a ter em conta é o destino das alterações. Estas alterações podem ser enviadas para um *site* central que possui uma cópia primária dos dados a alterar ou para todos os *sites* em simultâneo. Estes dois métodos são conhecidos como centralizado e distribuído [28].

Nas secções seguintes estão descritas as estratégias de fragmentação e replicação.

2.4 Fragmentação

A fragmentação consiste em dividir uma tabela da base de dados em diferentes subconjuntos da tabela original. Existem três métodos para fragmentação: vertical, horizontal e híbrida [29].

O número de tabelas e colunas guardadas em cada *site* é baseada no tipo de aplicação que irá aceder aos dados e, no tipo de *querys* mais utilizadas. Ao conjunto de dados

guardado em cada *site* é dado o nome de fragmento [29].

2.4.1 Fragmentação Vertical

Na fragmentação vertical as colunas de uma tabela são divididas pelos diferentes *sites*. Este método assume que cada *site* do sistema não necessita de todos os atributos de uma relação guardada.

Uma desvantagem deste método é que diferentes fragmentos não contêm um atributo comum. Após uma tabela ser fragmentada verticalmente, torna-se difícil juntar as suas colunas. Para isso é necessário incluir uma chave única em todos os fragmentos de forma a que, através dessa chave, uma relação seja totalmente restabelecida [26].

A Figura 2.2 mostra uma tabela que armazena informação relativa aos empregados de uma determinada empresa. A empresa contém vários departamentos, localizados em diferentes pontos geográficos. Cada departamento quer apenas ter acesso a algumas colunas (atributos) da tabela. A tabela pode ser fragmentada pelos diferentes departamentos, onde num departamento é guardada informação pessoal dos empregados e noutro apenas o salário. Cada entrada da tabela fragmentada possui um identificador (*id*) como chave primária. Quando se pretende juntar os vários fragmentos de uma tabela, é utilizada a chave primária para possibilitar essa junção [26].

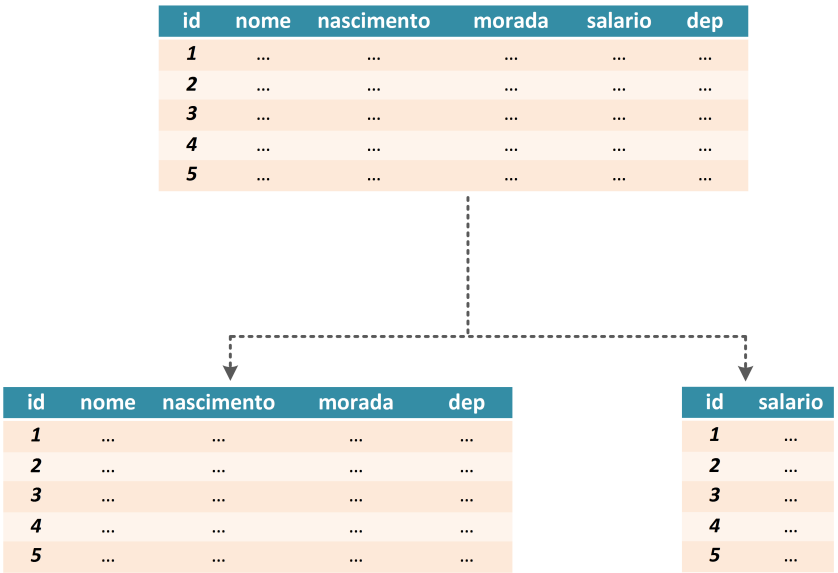


Figura 2.2: Fragmentação vertical [29].

2.4.2 Fragmentação Horizontal

Quando é utilizada a fragmentação horizontal, cada fragmento de uma tabela contém todos os atributos de um registo da tabela original [26].

Consideremos como exemplo uma empresa constituída por diferentes departamentos, onde cada empregado trabalha num departamento. Uma possível base de dados da empresa contém uma tabela com informação acerca de cada funcionário. Essa tabela guarda um atributo que relaciona o funcionário com o departamento onde ele opera. Se cada departamento for um *site*, a tabela dos funcionários pode ser dividida horizontalmente, onde cada fragmento atribuído a um *site* contém os trabalhadores que operam no respetivo departamento [26].

A Figura 2.3 representa uma tabela dividida em dois fragmentos. Cada fragmento contém toda a informação dos trabalhadores do departamento em causa, e apenas esses. Neste exemplo, existem os departamentos 5 e 9, os fragmentos são divididos pelo número de departamento. A união de todos esses fragmentos, neste caso, será trivial, uma vez que cada registo está sempre guardado de forma única em cada fragmento [26].

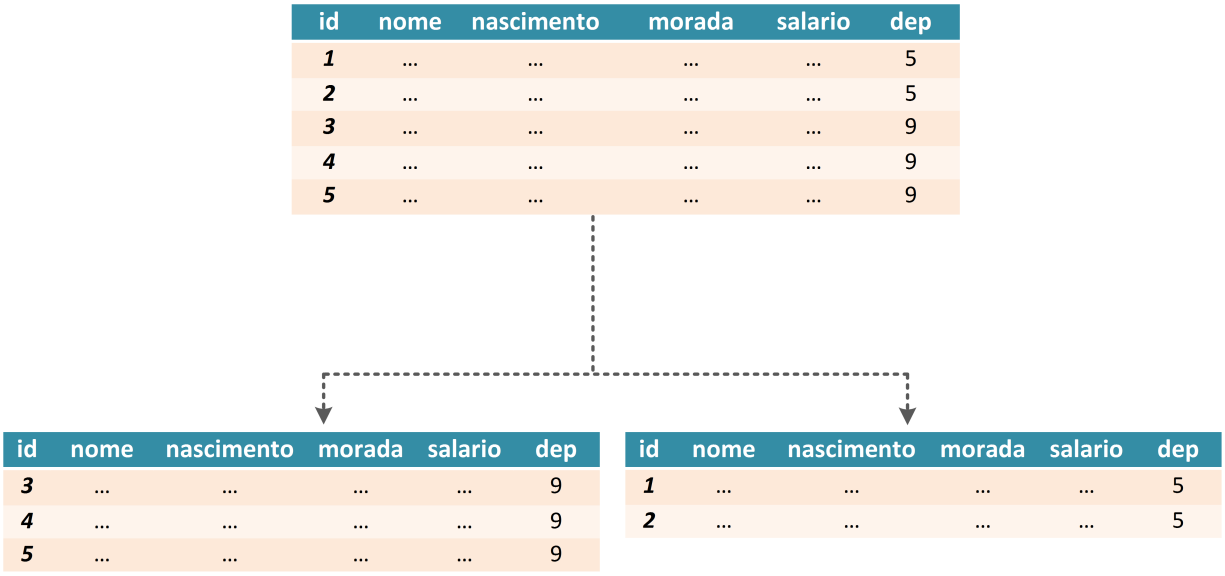


Figura 2.3: Fragmentação horizontal [29].

2.4.3 Fragmentação Híbrida

A fragmentação híbrida consiste numa forma de combinar as vantagens da fragmentação vertical e horizontal. Assume que cada *site* poderá não necessitar de todos os atributos de

um registo, nem de todas os registos de uma tabela [29].

Neste método, podem ser adotadas duas abordagens: fragmentar verticalmente uma tabela e depois fragmentar esse subconjunto de forma horizontal ou começar por fragmentar horizontalmente, obtendo um subconjunto que poderá ser depois fragmentado verticalmente [29].

Este método permite uma forma flexível de fragmentação dos dados. No entanto, a junção dos fragmentos é complexa [29].

Como exemplo deste método, podemos dividir os funcionários de uma empresa que têm um salário acima da média da empresa e por fim, dividi-los por departamentos [29].

2.5 Replicação

O mecanismo de replicação pressupõe que a base de dados seja distribuída, onde cada *site* contém uma cópia parcial ou integral das tabelas [28].

Um dos grandes desafios nesta estratégia é manter consistência mútua entre as diferentes cópias. Consistência mútua é diferente de consistência transacional. A Consistência mútua consiste em que os dados replicados pelas diferentes cópias possuam o mesmo valor. A consistência transacional pretende que o histórico das transações seja serializável [28].

Suponhamos três *sites*: A, B e C. O *site* A guarda uma cópia da entidade X. O *site* B guarda uma cópia das entidades X e Y. O *site* C guarda uma cópia de X, Y e Z. Considere as três transações representadas na Figura 2.4.

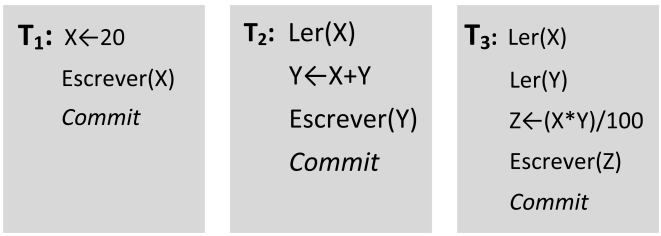


Figura 2.4: Transações T_1, T_2 e T_3.

A transação T_1 é executada nos *sites* onde está guarda uma cópia de X, ou seja, em A, B e C. De igual modo, a transação T_2 é executada nos *sites* B e C, e T_3 apenas é executada em C [28].

Assuma que cada réplica pode ler os dados que guarda localmente, mas tem de enviar

as alterações a todas as outras. Se considerarmos que no *site* B a transação T_2 só é executada depois de T_1. No *site* C é executada primeiro T_2, T_3, e só depois T_1. O histórico das transações executadas não é serializável, porque o conjunto de instruções atómicas correspondentes às transações T_1 e T_2, serão executadas por ordens diferentes nos *sites* B e C [28].

Desta forma não garantimos consistência transacional. No entanto é preservada a consistência mútua entre as várias cópias das entidades X e Y. Por exemplo, o valor de X nos *sites* A, B e C inicialmente é 10, o de Y é 15 e o de Z é 7. No fim, o valor de X nos três *sites* é igual a 20, Y é igual a 35 e Z é igual a 3,5. Ou seja, o valor final das entidades X, Y e Z é o mesmo nos três *sites* [28].

Existem dois tipos de replicação: síncrono e assíncrono, conhecidos como *eager* e *lazy*. Estes métodos diferem no facto das réplicas estarem mutuamente consistentes a cada instante ou de poder haver um período de tempo em que o não estão. O tipo síncrono garante consistência mútua [28].

O tipo de replicação relaciona-se essencialmente com o momento em que é executada a replicação, ou seja, o instante em que as atualizações aos dados são enviadas às restantes bases de dados (réplicas) do sistema [28]

2.5.1 Replicação Síncrona ou *Eager*

No tipo síncrono, todas as réplicas do sistema têm a mesma informação em qualquer instante. Deste modo, qualquer pedido de leitura aos dados feito às diferentes bases de dados retorna o mesmo valor [28].

Atingir tal sincronismo nos dados implica que uma transação sobre uma réplica seja feita em simultâneo nas restantes. Para garantir a total atomicidade dos dados a serem modificados numa transação, é utilizado como *standard*, o protocolo Two-Phase Commit Protocol (2PC)[1] [27]. Existem diferentes implementações do protocolo 2PC, variando de acordo com a estratégia adotada que se define como: centralizada ou distribuída [28].

No caso da replicação síncrona, os modos centralizado e distribuído são idênticos. A única diferença significativa é onde é submetida a transação. No caso do modo centralizado, cada *site* (exceto o próprio *master*) envia a transação para o *master* e espera que ele distribua a alteração pelos restantes *sites*. Por outro lado, o modo distribuído não identifica

[1] O protocolo 2PC permite coordenar de forma distribuída a execução de uma transação, assegurando a atomicidade da transação nas várias bases de dados [27].

qualquer *master*, cada *site* envia a transação aos restantes [27].

Método Centralizado

No método centralizado pode ser utilizado um único *master* para toda a base de dados. Em alternativa, pode definir-se um *master* para cada entidade da base de dados. Cada *master* guarda uma cópia de uma entidade (cópia primária).

- **Um único *master* (*Single Master*):** ter um único *master* para toda a base de dados implica que cada transação seja executada diretamente no *master*. O coordenador de transações do *master* (*Transaction Manager*) é responsável pela execução de cada transação [28]. Cada DBMS integra um coordenador de transações que é responsável por executar o conjunto de instruções de uma transação [24].

  Na Figura 2.5 estão representados dois pedidos: uma operação de escrita sobre uma entidade X e outro de leitura sobre a mesma entidade.

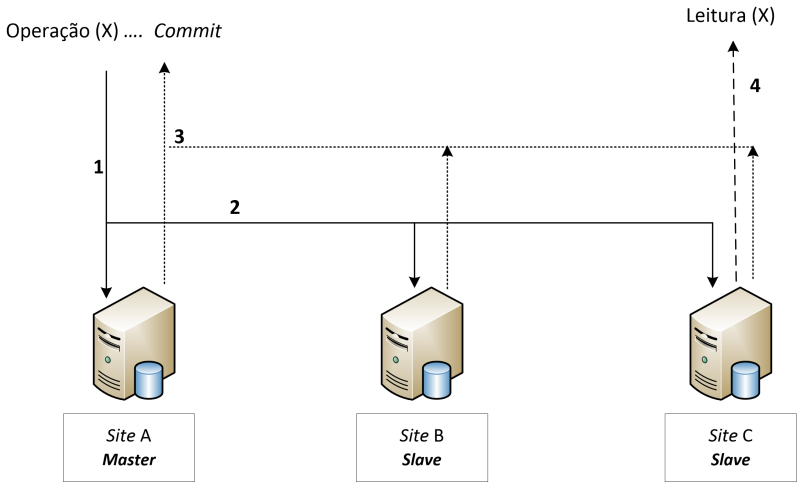


Figura 2.5: Replicação síncrona com um único *master* [28].

Se o pedido consiste numa alteração à entidade X (1), é obtido um *lock*[2] dos dados e efetuada a escrita na base de dados. Simultaneamente, o coordenador envia o mesmo pedido de escrita aos restantes *slaves* do sistema (2). Cada *slave* executa

[2]É um procedimento utilizado para controlar a concorrência no acesso aos dados. Um *lock* pode bloquear o acesso de outra transação aos dados, com o fim de evitar inconsistências na base de dados ou resultados incorretos [24].

um *lock* sobre as tabelas a alterar e faz *commit* da operação (3). O *commit* é feito de forma síncrona [28].

Quando é executado um pedido de leitura, qualquer umas das réplicas *slave* detém os dados necessários para responder (4). Para isso, é executado um *lock* para acesso aos dados e a resposta é enviada para o utilizador que solicitou o pedido [28].

- **Cópia Primária (*Primary Copy*):** a utilização de cópias primárias permite definir um *site master* por cada entidade guardada na base de dados. Cada *master* é responsável pelas alterações feitas a uma entidade. Assim, deixa de existir apenas uma base de dados *master* central e passámos a assumir uma estratégia *multi-master* [28].

  Na Figura 2.6, estão representados os *sites*: A, B, C e D. Onde o *site* A detém a cópia primária da entidade X. O *site* B está configurado como *slave* das entidades X e Y. O *site* C é *slave* da entidade X e cópia primária de Y. Por fim, o *site* D é *slave* da entidade Y.

  Assumindo uma transação que envolve operações sobre as entidades X e Y. O coordenador de transações do *site* onde é originada a transação, envia as operações aos *sites* A e C (1) que são cópias primárias das entidades X e Y. De seguida, as alterações são propagadas às restantes réplicas (2). O *commit* é feito em todos os *sites* em simultâneo (3) [28].

  Este *commit* requer um controlo de concorrência nos diferentes *sites*. Para isso, foi proposto em 1977 o protocolo Primary Copy Two-Phase Locking (PC2PL). O protocolo PC2PL, que é uma extensão do protocolo 2PC, lida com o atraso causado pela necessidade de uma transação esperar pela sua vez para poder ser executada [28].

Método Distribuído

No método distribuído as alterações são originadas em qualquer *site* e distribuídas pelos restantes *sites*. Esta estratégia é também conhecido como *update anywhere*. A diferença em relação ao método centralizado é que, neste caso, não existe o conceito de *master* [28].

A Figura 2.7 mostra quatro *sites*: A, B, C e D. Em cada um deles é guardada uma cópia da base de dados. Considerando duas transações, T_1 e T_2, ambas de escrita sobre a

entidade X. A transação T_1 é enviada ao *site* A (1) e a transação T_2 ao *site* D (1). No método distribuído, T_1, será propagada aos *sites* B, C e D (2), enquanto T_2 será enviada a A, B e C (2) [28].

Estas alterações tornar-se-ão permanentes assim que todos os *sites* fizerem *commit* das transações (3) (na Figura apenas se encontra representado o *commit* de T_1) [28].

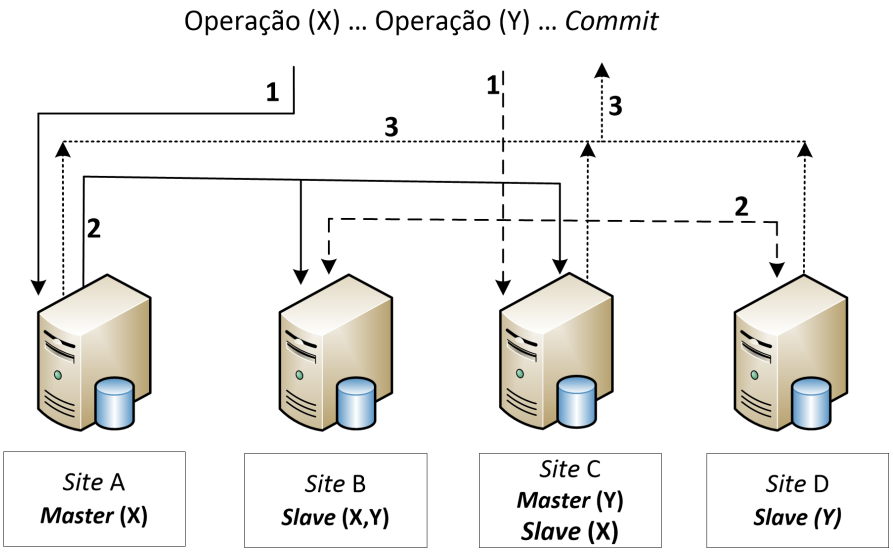


Figura 2.6: Replicação síncrona utilizando cópias primárias [28].

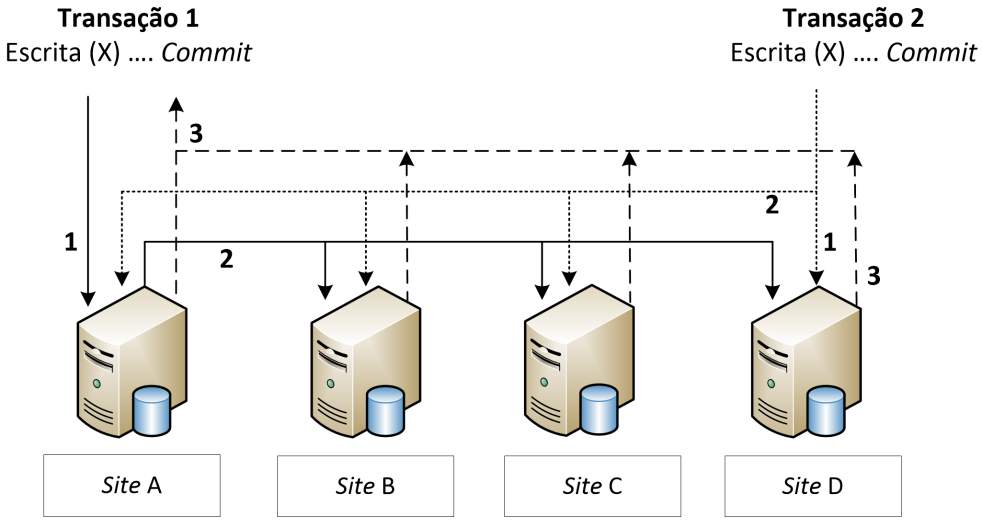


Figura 2.7: Replicação síncrona utilizando o método distribuído [28].

2.5.2 Replicação Assíncrona ou *Lazy*

No tipo assíncrono, uma transação sobre a base de dados é primeiro executada localmente e só posteriormente é propagada aos restantes *sites*. Consequentemente, existe um período de tempo em que os *sites* não possuem os dados consistentes [28].

Tal como no método síncrono, existem os métodos centralizado e distribuído. No entanto, no tipo assíncrono podem ocorrer conflitos [28].

Método Centralizado

No modo centralizado é possível definir um único *master* para toda a base de dados, ou criar um *master* por cada entidade guardada (cópia primária).

- ***Um único Master (Single Master)*:** as transações são aplicadas diretamente ao *site master*. Só após a transação ter feito *commit* na base de dados *master* será enviada aos restantes *sites slave*. Quando um *site slave* recebe um pedido de leitura aos dados, devolve a resposta com base na cópia que possui. Embora essa cópia possa ainda não conter os dados atualizados [28].

  Após o *master* processar localmente as transações, envia-as às réplicas. Nesta arquitetura, o envio das transações pode ser ordenado com base no *timestamp* [3] das transações [28].

  A Figura 2.8 representa três *sites*: o *site* A em modo *master* e os *sites* B e C em modo *slave*. A transação T_1 consiste numa alteração sobre a entidade X e T_2 é um pedido de leitura sobre a mesma entidade. Assim que T_1 chega ao *master* (1), este executa-a e faz *commit*. De seguida, é enviada às restantes réplicas *slave* (3). Por fim, é feito um pedido de leitura no *site* C sobre a entidade X (4) [28].

  A transação T_2 é executada depois de T_1 ter sido propagada à réplica no *site* C. No entanto, o sucedido nem sempre acontece, uma vez que poderia ter-se verificado a ordem inversa [28].

  A Figura 2.9 mostra uma representação temporal onde T_2 é executada antes de T_1 ser propagada ao *site* C. Assuma que, inicialmente, o valor de um registo na entidade X é igual a 100 nos dois *sites*. A transação T_1 consiste em alterar o valor desse registo para 0. Após a execução de T_1 no *site* A, existe um intervalo de tempo *delta* em que

[3] Definido como uma representação do tempo corrente quando um evento ocorreu, uma marca temporal guardada pelo computador [21]

o valor do registo diverge, sendo em A igual a 0 e em B igual a 100. A diferença dos valores nos dois *sites* viola o princípio de consistência mútua [28].

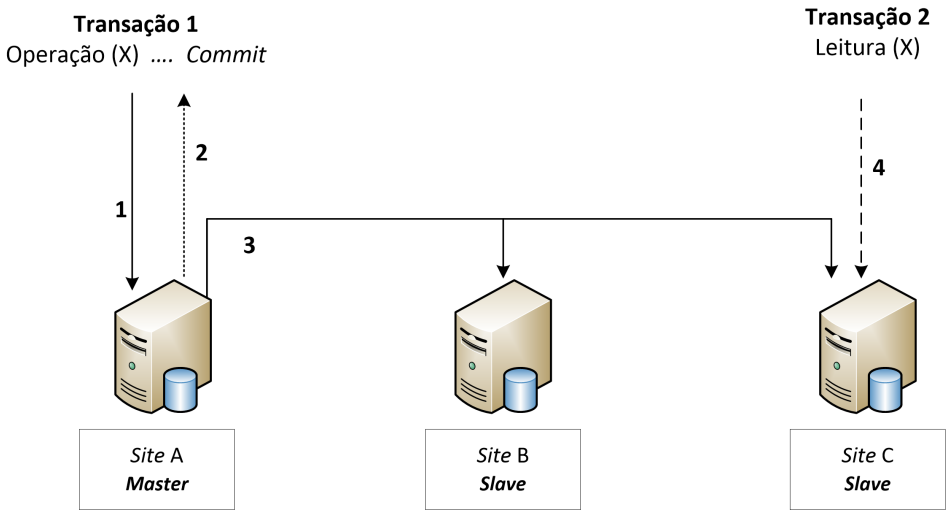


Figura 2.8: Replicação assíncrona com um único *master* [28].

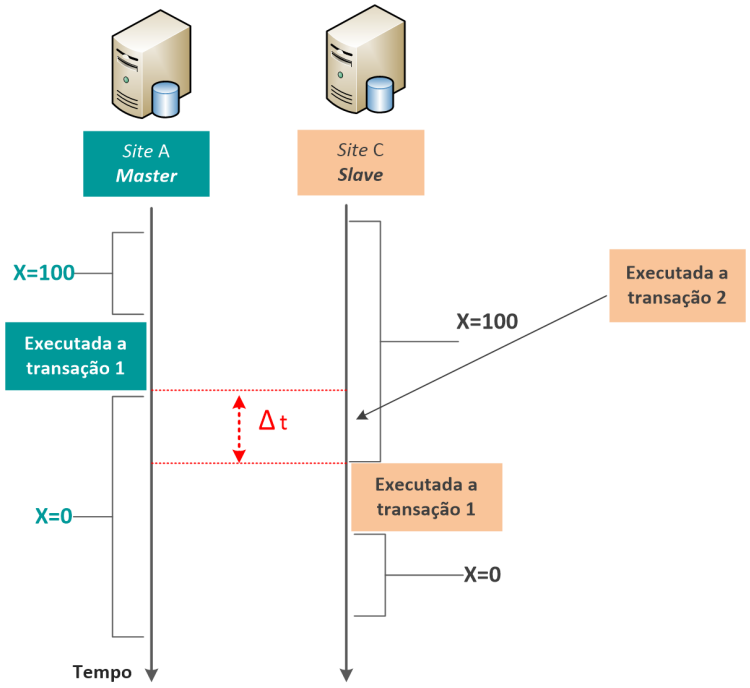


Figura 2.9: Consistência mútua violada em replicação assíncrona.

- ***Cópia Primária***: neste método existe uma réplica responsável por uma ou mais entidades da base de dados. Qualquer alteração feita a essas entidades, deverá ser efetuada primeiro na sua cópia primária [27]

Método Distribuído

O método distribuído é o mais complexo, pois qualquer *site* executa localmente atualizações nos dados e só depois as propaga às restantes réplicas. Na Figura 2.10 temos um exemplo deste método e dos conflitos que origina [27].

Suponha duas réplicas: R_A e R_B, guardadas nos *sites* A e B respetivamente. A réplica R_A executa uma transação T_1, e R_B uma transação T_2. As duas transações alteram o valor de um registo na tabela X que inicialmente é 0 em ambas as réplicas [27].

As transações são executadas localmente (1,2) sendo feito o seu *commit* (3,4). O registo da entidade X, no *site* A, fica com o valor de 1 e no *site* B com o valor 2.

Assim que T_1 é enviada a R_B (5) e T_2 é enviada a R_A (8), é detetado um conflito. Este conflito deve-se ao facto de, quando as réplicas recebem as transações sobre X, verificarem o seu valor antigo e novo, onde o valor antigo difere do que guardam localmente (7,8) [27].

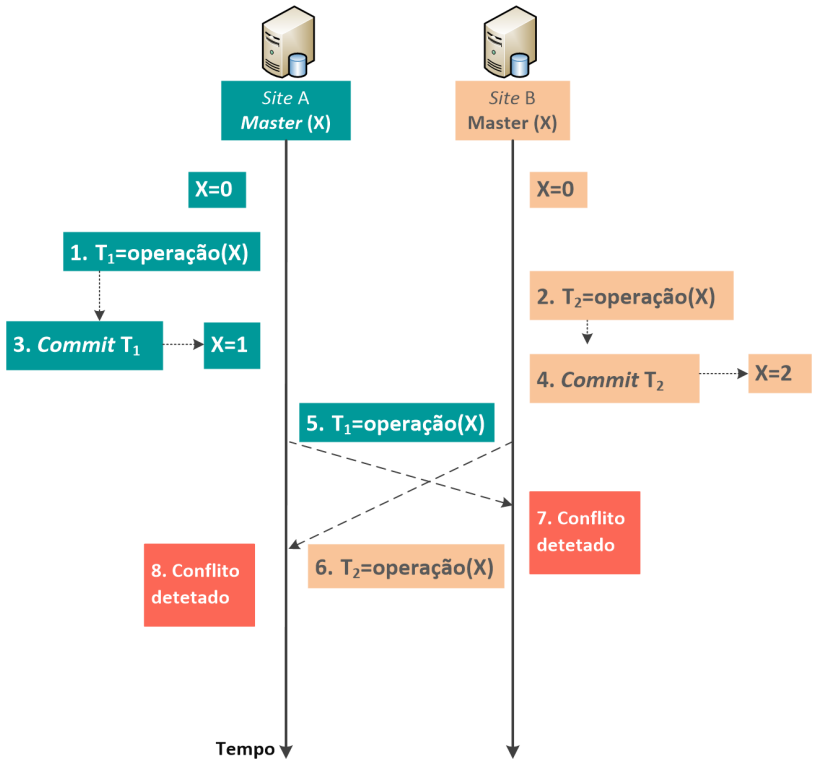


Figura 2.10: Replicação assíncrona utilizando o método distribuído [27].

2.5.3 Comparação entre Métodos Síncrono e Assíncrono

Na Tabela 2.1 encontram-se sintetizadas as principais características dos métodos síncrono e assíncrono.

O método síncrono assegura o princípio de consistência mútua, uma vez que todas as réplicas da base de dados fazem *commit* da mesma transação em simultâneo. Por outro lado, o método assíncrono viola esse princípio porque existe um atraso na propagação das transações às réplicas.

Dado que o tipo assíncrono permite um atraso na propagação de transações, suporta o modo *offline*. O protocolo 2PC utilizado no tipo síncrono aborta uma transação caso uma das réplicas esteja *offline*. Este protocolo implica que haja um atraso no tempo de resposta por parte do sistema, no entanto impede a ocorrência de conflitos. No tipo assíncrono o tempo de resposta é menor, mas podem ocorrer conflitos.

Tabela 2.1: Tabela de comparação dos tipos síncrono e assíncrono.

	Tipo	
Característica	Síncrono	Assíncrono
Consistência mútua	✓	–
Suporta modo *offline*	–	✓
Ocorrência de conflitos	–	✓
Atraso na propagação dos dados	–	✓
Consistência transacional	✓	–
Tempo de resposta	Maior	Menor

2.5.4 Comparação entre Métodos Centralizado e Distribuído

Os métodos centralizado e distribuído têm diferentes propriedades. O método centralizado refere-se não só a sistemas onde existe um *master* para toda a base de dados, mas também ao uso de cópias primárias por cada entidade. Criar uma cópia primária de uma entidade é equivalente a centralizar todas as operações sobre essa entidade.

Existem algumas vantagens e desvantagens nos métodos centralizado e distribuído [27]:

- **Perda de consistência:** utilizar o método distribuído pode causar problemas de consistência. A falta de consistência é causada pelos conflitos. Esses conflitos tornam-se difíceis de resolver, uma vez que nenhuma réplica detém um controlo total do sistema, contrariamente ao que acontece no método centralizado;

- **Perda da transparência na replicação:** no método centralizado, cada *site* deve ter informação acerca do *master* da entidade a alterar. Antes de uma transação ser enviada a um *site*, é necessário saber o tipo de transação (leitura ou escrita) antes de iniciar a sua execução. Algumas *interfaces* de ligação a bases de dados como Java Database Connectivity (JDBC) já permitem declarar o tipo de transação. No método distribuído, um *site* apenas precisa de conhecer outro que seja uma réplica do seu;

- **Efeito *bottleneck*:** no método centralizado todas as transações de atualização têm de passar pelo *site master*. Quando existem muitas transações em simultâneo, o *master* pode ficar sobrecarregado de pedidos. Esta sobrecarga irá revelar-se num atraso diretamente proporcional com o número de transações com que o *site* tem de lidar. No método distribuído, a carga de transações é distribuída pelas várias réplicas;

- **Um único ponto de falha:** no método distribuído quando o *site* principal falha, existem réplicas que impedem a falha total do sistema. No método centralizado, quando o *master* falha, todo o sistema é interrompido;

- **Ocorrência de *deadlock*:** em sistemas que utilizem o método distribuído com recurso ao *lock*, é possível que ocorram *deadlocks*[4] distribuídos.

2.5.5 Deteção de Conflitos

Por definição, um conflito ocorre quando duas transações concorrentes tentam alterar o mesmo registo na base de dados [29][24]. Num cenário de replicação assíncrona, um conflito pode também ser originado por uma transação que cause inconsistência entre as várias réplicas. A frequência com que um conflito ocorre aumenta quando é utilizado o método distribuído. No método centralizado é possível gerir os conflitos de forma a enviar para os restantes *sites* apenas as transações que não originam conflitos. Em cenários descentralizados, não existe uma entidade responsável por essa função, visto que todos os *sites* enviam informação para os restantes [27].

[4] Um *deadlock* consiste num "impasse" que resulta de uma ou mais transações estarem à espera que um *lock* seja libertado por outra que o retem [24].

Existem três tipos de conflitos principais [27]:

- **Conflitos de chave primária:** ocorrem quando duas transações tentam inserir dados com a mesma chave primária;
- **Conflitos de atualização:** quando duas transações concorrentes tentam atualizar os mesmos dados;
- **Conflitos de eliminação:** ocorrem quando uma transação apaga dados que outra transação visa atualizar ou apagar.

Podemos distinguir dois tipos de deteção de conflitos segundo os métodos centralizado e distribuído [27]:

- **Deteção de conflitos de forma centralizada:** neste método, a deteção de conflitos torna-se mais simples, uma vez que é possível a utilização de um identificador de versões nos dados. Este identificador pode ser controlado no *site master* do sistema de replicação. O processo decorre da seguinte forma:
  1. O *site master* atualiza os seus próprios dados gerando um identificador da versão desses dados;
  2. Quando uma réplica aplica essa alteração recebida do *site master*, mantém um histórico da versão;
  3. Assim que uma réplica envia uma alteração para o *site master*, associa-lhe o valor da versão corrente;
  4. O *site master* compara a versão recebida da réplica com a versão que tem atualmente guardada;
  5. Se a versão for a mesma, nenhuma outra réplica alterou, ainda, esses dados e portanto nenhum conflito é detetado. Caso os valores comparados sejam diferentes, então alguma réplica alterou já os dados no *site master* e é declarado como um conflito.
- **Deteção de conflitos de forma distribuída:** no método distribuído, é pouco eficiente manter um histórico das versões dos dados porque cada réplica tem a sua própria versão. Uma possível abordagem é utilizar vectores de versões.

  Na Figura 2.11 está representado um sistema que implementa o uso de vetores de versões. Cada vector é guardado junto aos dados e contém uma entrada para cada

réplica de uma entidade na base de dados. Assim, sempre que uma réplica altera os seus dados, incrementa a versão na posição correspondente [27].

O sistema representado na Figura 2.11 é constituído por três *sites*: A, B e C. Cada *site* armazena uma réplica da base de dados: R_A, R_B e R_C, respetivamente. Cada réplica guarda uma cópia de uma entidade X. Onde:

1. Inicialmente cada réplica possui o seu próprio vector da entidade X (1): V_A=(0,0,0), V_B=(0,0,0) e V_C=(0,0,0);
2. É executada uma transação T_1 no *site* A (2);
3. Após o *commit* (3) de T_1, V_A é atualizado para (1,0,0) (4);
4. O mesmo sucede no *site* C quando é executada uma transação T_2, resultando em V_C=(0,0,1);
5. Assim que R_A recebe T_2 (5), compara o seu vector (V_A) com V_C, e conclui que V_C[1]<V_A[1] (7). Ou seja, quando T_2 ocorreu, T_1 ainda não tinha sido executada em R_C, originando um conflito. O mesmo se verifica quando T_1 é enviado a R_C;
6. Se T_1 for enviada à réplica R_B (6) antes de T_2: V_A [1]=V_B[1]+1, V_A[2]=V_B[2] e V_A[3]=V_B[3], portanto nenhum conflito é detetado (8).

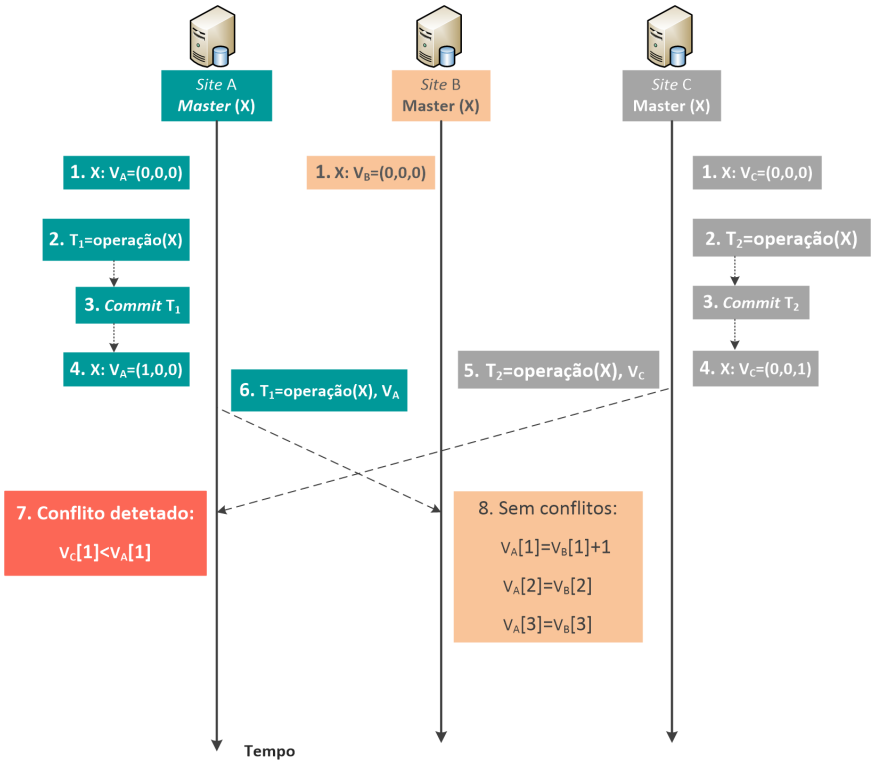


Figura 2.11: Deteção de conflitos utilizando vectores de versões [27].

Esta técnica é eficaz, mas tem problemas de escalabilidade. Se o sistema tiver muitas réplicas, os vectores tendem a ter uma grande dimensão.

A abordagem mais eficaz é enviar sempre a representação da entidade *X* antes e depois da alteração. Um conflito é detetado quando a representação de *X* antes da alteração não é igual ao valor de *X* guardado localmente.

2.5.6 Resolução de Conflitos

Quando ocorre um conflito, a sua resolução resulta em: manter os dados inalterados ou alterar os dados. É necessário decidir o valor final a assumir no conflito e para tal, existem algumas heurísticas de suporte à decisão. O seu objetivo é tornar o sistema consistente, de forma a que todas as réplicas contenham o mesmo conjunto de dados.

As Principais heurísticas são [27]:

- **Ignorar a alteração:** uma possível solução em caso de conflito é ignorar a atualização. Esta opção só é viável caso se tenha um mecanismo de deteção e resolução de conflitos centralizado. Caso seja distribuído, algumas réplicas podem não decidir ignorar a atualização, o que leva a inconsistências. Em ambos os casos podemos ter perda de dados;

- **Sobreposição:** esta opção é o oposto da anterior. Neste caso é sempre executada a atualização, sobrepondo o valor novo ao antigo.

- **Prioridade por *site*:** é atribuída uma prioridade a cada *site* do sistema. Em caso de conflito opta-se pela atualização gerada pelo *site* com maior prioridade. Esta heurística origina perda de dados, na medida em que atualizações (em conflito) geradas por *sites* com menor prioridade são ignoradas;

- **Prioridade por valor:** neste caso a prioridade é atribuída de acordo com os valores que uma atualização pode tomar. Dadas duas transações em conflito, é escolhida a que tem um valor com maior prioridade;

- **Prioridade por *timestamp*:** dadas duas transações em conflito, é escolhida a que foi originada em primeiro ou segundo lugar. É utilizada uma marca temporal para identificar qual das transações foi criada primeiro e segue-se um critério onde se escolhe a transação que foi originada primeiro ou a mais recente;

- **Decisão pelo máximo ou mínimo:** este método é utilizado apenas para tipos de dados específicos, onde é possível uma comparação direta dos valores em conflito. O objetivo é escolher o maior ou o menor valor em conflito.

2.6 Resumo

Neste capítulo fizemos uma breve introdução a bases de dados distribuídas definindo alguns conceitos importantes. Verificamos que uma base de dados distribuída apresenta algumas vantagens como: partilha e autonomia dos dados, disponibilidade, modularidade e *performance*. No entanto tem desvantagens como: complexidade na arquitetura e no desenho da base de dados, requisitos de segurança e custo a nível de *hardware*.

Existem duas estratégias para implementação de bases de dados distribuídas: fragmentação e replicação. A fragmentação permite dividir a bases de dados original em vários fragmentos. Cada fragmento contém um conjunto de tabelas. Os fragmentos são construídos recorrendo a três diferentes métodos de fragmentação: vertical, horizontal e híbrido. Cada *site* guarda um fragmento e é responsável pelas alterações feitas aos registos armazenados nesse fragmento.

A replicação permite ter dados replicados por vários *sites*. A Figura 2.12 sintetiza os métodos utilizados para replicação de bases de dados. A utilização de um mecanismo de replicação implica responder a duas questões:

1. **Quando replicar a informação:** os dados podem ser replicados antes ou depois de serem guardados localmente utilizando o tipo síncrono ou assíncrono, respetivamente. No tipo síncrono, o *commit* das transações é feito em simultâneo em todos os *sites*.
2. **Para onde enviar os dados:** os dados podem ser enviados para um único *site*, utilizando o método centralizado. No método centralizado é possível definir dois métodos:

   - ***Single master:*** através do método *single master* é definido um único *master* para todas as entidades da base de dados;
   - ***Primary copy:*** no método *primary copy*, cada entidade tem um servidor *master*;

   O método distribuído permite o uso do modo *update anywhere*. Neste método cada réplica envia as atualizações aos restantes *sites* do sistema.

A replicação assíncrona gera conflitos quando uma transação é propagada entre diferentes réplicas. Um conflito compromete a consistência mútua. É possível a criação de um mecanismo para deteção de conflitos e utilizar heurísticas para a sua resolução.

Figura 2.12: Resumo dos tipos de replicação.

Capítulo 3

Sistemas para Replicação de Bases de Dados

No capítulo anterior descrevemos os princípios de bases de dados distribuídas, tendo sido analisados métodos para sincronização de bases de dados.

Neste capítulo é feita uma análise aos principais projetos que implementam mecanismos de sincronização de bases de dados PostgreSQL. Na Secção 3.1 é descrito o sistema de replicação disponibilizado pelo PostgreSQL. A Secção 3.2 apresenta outros projetos de sincronização de bases de dados. É feita uma descrição destes projetos, analisando as suas principais caraterísticas.

3.1 Mecanismos de Replicação do PostgreSQL

O mecanismo de replicação do PostgreSQL permite que dois servidores de bases de dados trabalhem em conjunto. Quando um deles falha, o outro continua a garantir disponibilidade dos dados. Contudo, não implementa um mecanismo capaz de lidar com a comunicação de dois servidores que estejam em modo de leitura e escrita em simultâneo [16].

A versão 9.2 disponibiliza um mecanismo de replicação baseado no envio de ficheiro de *log* [17]. Este mecanismo permite a sincronização dos dados, onde apenas um servidor é capaz de alterar os dados. A este tipo de servidor dá-se o nome de servidor *master* ou *read/write*. Os servidores que podem apenas responder a pedidos de leitura, enquanto

recebem os dados replicados, têm o nome de servidor *slave* ou *read only* [16].

Envio do Ficheiro de *log*

No envio do ficheiro de *log* é definido um servidor *master* e um conjunto de servidores em modo *slave*. Cada servidor *slave* é atualizado de acordo com registos que resultam de alterações feitas aos dados guardados no servidor *master*.

Os registos são previamente guardados no mecanismo Write-Ahead Log (WAL) do *master*. Este mecanismo de WAL consiste numa solução *standard* baseada no *logging* de transações, onde são registadas as alterações a serem feitas sobre dados. Antes das alterações serem escritas em disco, elas são registadas num ficheiro *log*. O servidor mantém o ficheiro *log* para que não seja necessário estar sempre a efetuar escritas em disco e para que se possa recuperar a base de dados caso ela fique inconsistente [23].

Se o servidor principal falhar, como qualquer um dos *slaves* contém uma cópia integral dos dados, poderá passar ao modo *master*. A replicação pode ser feita de forma síncrona ou assíncrona. Esta opção é utilizada para todo o sistema não sendo possível utilizar os dois tipos de sincronização em simultâneo para diferentes partes do mesmo sistema.

O servidor *master* envia os dados em modo *batch* (*file-based log shipping*) ou em modo *streaming*:

- **Modo *batch*:** o servidor *slave* recebe periodicamente os ficheiros de *log* gerados pelo *master*. Os ficheiros que contêm os registos (*WAL records*) são transportados do *master* para o *slave* através da rede.

  É utilizado o modo assíncrono, o que significa que os ficheiros de WAL são enviados após o *commit* das transações no *master*. Este pequeno intervalo de tempo entre o momento em que as transações são executadas no *master* e o momento em que são enviados os ficheiros para o *slave*, constitui um ponto de falha. Caso o *master* falhe definitivamente, as transações que estavam à espera de serem transportadas são perdidas.

  Este intervalo é parametrizável. No entanto, se o intervalo for demasiado pequeno, estamos constantemente a enviar ficheiros na rede.

- **Modo *streaming*:** permite que um servidor *slave* esteja a receber atualizações continuamente. Neste método, o *slave* recebe os registos modificados à medida

que eles vão sendo alterados (não tendo que esperar pelos ficheiros WAL). Usa um mecanismo assíncrono. Existe um pequeno atraso entre o momento da execução de uma transação no *master* e a propagação das alterações nos servidores *slave*. Este atraso é muito menor do que no modo baseado nos ficheiros de *log*.

As duas implementações suportam métodos de autenticação. A autenticação permite que um *slave* se autentique na ligação ao respetivo *master*.

Um servidor em modo *slave* deverá ser configurado para operar num dos seguintes modos:

- ***Warm standby*:** o servidor *slave* é apenas usado como um servidor de *backup*, ou seja, apenas é utilizado em caso de falha do servidor principal [13];

- ***Hot standby*:** o servidor *slave* pode responder a pedidos de leitura com base na cópia que vai mantendo do servidor principal [8].

A topologia da ligação entre os servidores poderá ser em estrela ou em cascata. Na topologia em estrela, vários *slaves* ligam-se diretamente ao *master* (ver Figura 3.1). Na topologia em cascata um *slave* liga-se ao *master* e os restantes servidores *slave* ao primeiro *slave* (Figura 3.2). No modo cascata, reduz-se o número de ligações ao master diminuindo o congestionamento de tráfego de rede entre diferentes *sites*.

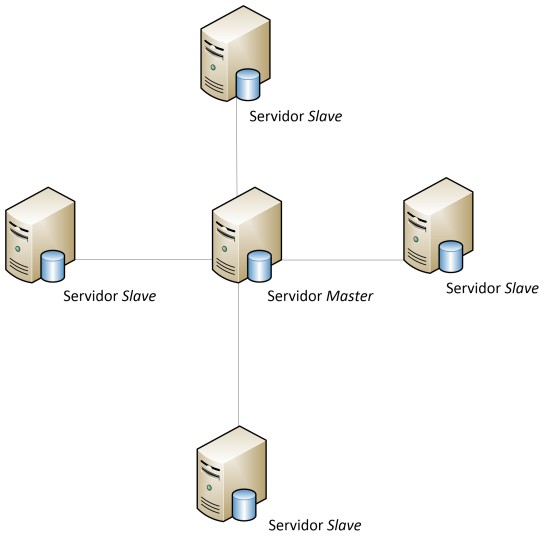


Figura 3.1: Topologia em estrela.

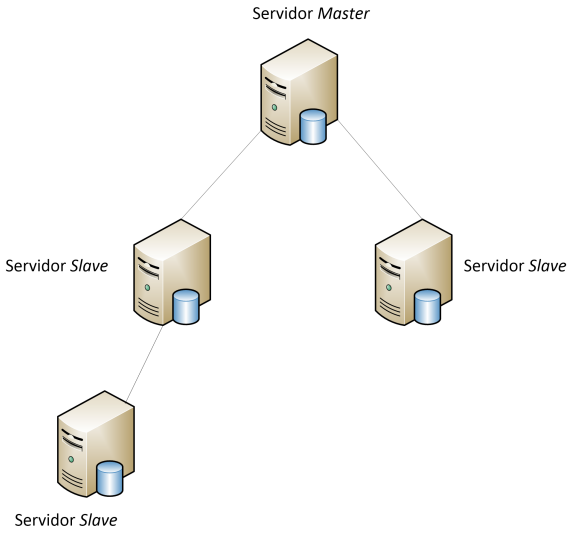


Figura 3.2: Topologia em cascata.

3.2 Projetos de Replicação de Bases de Dados

A crescente necessidade de distribuir os dados por diferentes servidores de bases de dados, levou a que surgissem novas soluções de replicação. Como exemplo dessas soluções temos os projetos PgCluster[1], Pgpool-II[2], Slony[3], Bucardo[4] e Symmetric DS[5].

3.2.1 PgCluster

O PgCluster é um sistema de replicação que utiliza replicação síncrona *multi-master*. Permite a comunicação e sincronização entre diferentes servidores configurados em modo *master* [14].

Este mecanismo é composto por três tipos de servidores: um distribuidor de carga dos pedidos (*load balancer*), um *cluster* de bases de dados (onde estão guardadas as bases de dados) e um servidor de replicação. O projeto implementa um sistema que permite a total disponibilidade em caso de falha de uma base de dados [14].

Quando uma das bases de dados falha, o distribuidor e o servidor de replicação continuam

[1] PgCluster: http://pgcluster.projects.pgfoundry.org/
[2] Pgpool-II: http://www.pgpool.net/mediawiki/index.php/Main_Page
[3] Slony: http://slony.info/
[4] Bucardo: http://bucardo.org/
[5] Symmetric DS: http://www.symmetricds.org/

a funcionar de forma a que o sistema não tenha de parar. Para isto, uma segunda base de dados, réplica da que falhou, assume a sua "posição" de *master* de imediato no sistema. Quando o *cluster* é reposto, não há paragem das funcionalidades do sistema, esse *cluster* será sincronizado com o *cluster* que esteve a receber os pedidos [14].

3.2.2 Pgpool-II

O funcionamento do projeto Pgpool-II é baseado num *middleware replication*. O *middleware replication* utiliza uma "camada" intermédia para deteção de transações. Neste caso em concreto, funciona entre o servidor PostgreSQL e os clientes. É expansível até cerca de 128 bases de dados no sistema. Tem algumas características interessantes como [15] :

- ***Pool* de Ligações:** o Pgpool-II guarda as ligações feitas ao servidor PostgreSQL permitindo reutilizá-las quando uma nova ligação com o mesmo nome de utilizador, base de dados e versão do protocolo contacta o servidor. Desta forma diminui e melhora o tempo de resposta do sistema;
- **Replicação:** integra um mecanismo de replicação entre diferentes servidores PostgreSQL, de forma a criar backups dos dados em diferentes discos. É utilizado o método síncrono;
- ***Load Balancer*:** como este sistema implementa um mecanismo de replicação, um pedido de consulta aos dados feito nos diferentes servidores retorna a mesma resposta. Assim, é possível distribuir os pedidos evitando sobrecarga num servidor;
- **Paralelização de *querys*:** permite dividir um pedido em várias *sub-querys* e executar cada uma delas em diferentes servidores, diminuindo o tempo de execução de um pedido.

3.2.3 Slony

O projeto Slony é um sistema de replicação baseado na arquitetura *master-slave* que utiliza o método de deteção de alterações baseado em *triggers*. Permite o modo assíncrono e suporta replicação em cascata. Como desvantagem, apresenta falta de escalabilidade e requer muitos pontos de monitorização. Uma vez que utiliza *triggers* para detetar atualizações feitas aos dados é possível configurar quais as tabelas que se pretendem replicar [18].

3.2.4 Bucardo

O sistema Bucardo suporta as arquiteturas *master-slave* e *multi-master*. Utiliza o modo assíncrono e *triggers* para a deteção de alterações [3]. A replicação é feita através de um processo *daemon* escrito na linguagem de programação Perl. O processo notifica as restantes bases de dados do sistema acerca das alterações feitas no *master*. Utiliza uma base de dados própria para guardar informação de configuração. Essa informação inclui uma lista de todas as bases de dados que fazem parte da replicação e as respetivas tabelas a replicar [3].

Após a configuração, os *triggers* detetam quais as linhas que são alteradas nas tabelas previamente configuradas. O processo de replicação no modo *multi-master* segue o seguinte procedimento [3]:

1. Uma alteração que seja detetada é guardada numa tabela própria do sistema de replicação;
2. Um processo *daemon* deteta que ocorreu uma alteração;
3. O *daemon* notifica o controlador do Bucardo acerca dessa alteração e volta a ficar à escuta de novas alterações;
4. O controlador cria um processo filho para lidar com a replicação ou envia um sinal a um já existente;
5. O processo filho inicia uma nova transação e desativa os *triggers* das tabelas em questão (para evitar ciclos entre os servidores);
6. Cria uma lista das alterações feitas na última replicação e compara-as com as atuais de forma a perceber o que deve fazer;
7. Se detetar um conflito, consulta as configurações de resolução de conflitos por defeito ou o administrador;
8. Volta a ativar os *triggers* e executa as transações;
9. Se a transação falhar, corre um gestor de exceções;
10. Por fim, o "filho" notifica o controlador que o processo está terminado.

O modo *multi-master* apenas suporta dois servidores *master* e tem um mecanismo de deteção e gestão de conflitos. Permite distribuir os pedidos pelos vários servidores *slave* [3].

3.2.5 Symmetric DS

O Symmetric DS é uma aplicação Java de *software* livre para sincronização de bases de dados em diferentes ambientes [20].

Suporta sincronização assíncrona bi-direcional, ou seja tem suporte para o modo *multi-master*. Foi desenvolvido para lidar com questões de escalabilidade e é tolerante a falhas na ligação [20].

Cada instância do Symmetric DS é vista como um nó. O conjunto dos vários nós constitui uma rede. Cada nó acede a uma base de dados a sincronizar. O Symmetric DS cria nas bases de dados a sincronizar algumas tabelas específicas para controlo da sincronização. Nessas tabelas são configurados os *triggers* a serem utilizados para detetar as alterações aos dados. As alterações aos dados são armazenadas em tabelas auxiliares para serem propagadas às restantes instâncias [20].

Os nós que estão ligados à rede sincronizam os dados utilizando os métodos *Pull* e *Push*. A Figura 3.3 mostra esses métodos. O método *Pull* implica que dados dois nós A e B, o nó B aguarde um pedido do nó A (1) e após esse pedido, o nó B envia as suas alterações (2). Assim que A recebe os dados, envia uma confirmação (3). No método *Push*, dados os nós A e B, o nó B irá enviar os seus dados a A sem que este lhe tenha feito qualquer pedido [20].

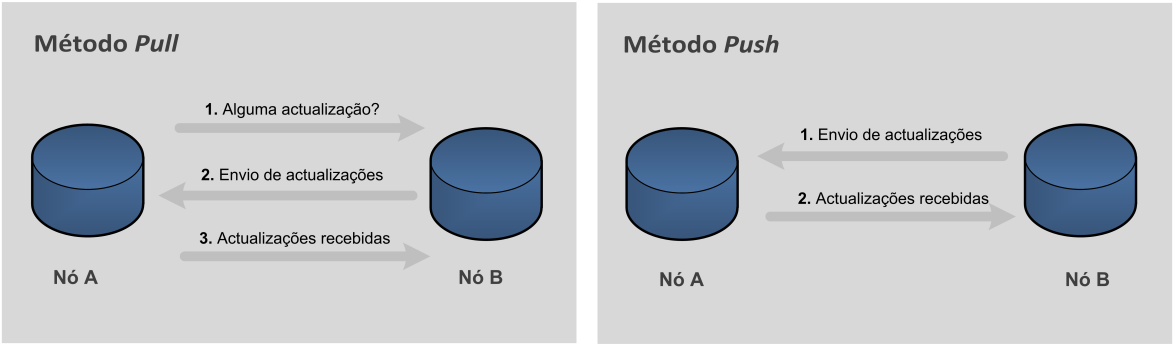


Figura 3.3: Métodos Pull e Push [20].

3.3 Resumo

Neste capítulo descrevemos os principais sistemas de replicação para bases de dados PostgreSQL. A Tabela 3.1 sumariza as características dos projetos PgCluster, Pgpool-II,

Slony, Bucardo e Symmetric DS.

O método de envio do Ficheiro de *log* disponibilizado pelo PostgreSQL, embora suporte replicação assíncrona, não suporta vários servidores de bases de dados em modo *master*.

Os projetos PgCluster e Pgpool-II não satisfazem um requisito essencial da solução pretendida: não permitem replicação assíncrona.

O sistema Bucardo apresenta duas desvantagens: não é multiplataforma (exige uma máquina Unix onde se possa executar o *daemon*) e não é escalável (apenas suporta duas bases de dados no modo *master*). Por estes motivos, o sistema Bucardo não reúne as características necessárias para os objetivos propostos neste projeto.

O Slony apresenta características interessantes como ser assíncrono e independente do sistema operativo. Contudo, apresenta falta de escalabilidade e não suporta mais do que um servidor *master*.

A solução mais interessante dos projetos analisados é o Symmetric DS, pois suporta o modo *multi-master* e o método de replicação assíncrona. É independente da plataforma e é um sistema estável. Foi testado num ambiente com duas bases de dados em modo *master*. Neste sistema, a sincronização é feita de forma totalmente configurável e os *triggers* detetam todas as alterações efetuadas aos dados.

Porém, encontramos algumas limitações:

1. **O mecanismo de resolução de conflitos:** permite selecionar heurísticas como: optar pelos dados mais "recentes" (baseado numa marca temporal), sobrepor as alterações ou ignorar os dados em conflito. No entanto, não permite definir outras heurísticas.
2. **Configuração dos dados a replicar:** permite configurar as tabelas a replicar, no entanto não permite selecionar as colunas a replicar.
3. **Integração com Novigest:** integrar esta solução no Novigest iria originar conflitos. O Novigest tem um mecanismo de *cache* próprio. Quando o Symmetric DS atualizasse os dados na base de dados, o conteúdo da *cache* do Novigest fica inconsistente.

Após a análise feita neste capítulo, concluímos que nenhum dos projetos apresentados reúne os requisitos necessários para ser utilizado como solução. Por isso, propomos e implementamos uma solução baseada nos requisitos do projeto.

Tabela 3.1: Tabela de comparação dos projetos relacionados.

	Projecto				
Característica	PgCluster	Pgpool-II	Slony	Bucardo	Symmetric DS
Versão	1.3	3.2	2.1.3	4.5	3.3
Licença	Berkeley Software Distribution (BSD)	BSD	BSD	BSD	Lesser General Public License (LGPL)
Tipo Sincronização	*Master-Master*	*Statement-Based Middleware*	*Master-Slave*	*Master-Master*[a], *Master-Slave*	*Master-Master*
Método de replicação	Síncrona	Síncrona	Assíncrona	Assíncrona	Assíncrona
Plataforma	FreeBSD, Linux,Solaris	FreeBSD, Linux,Solaris	FreeBSD, Linux,Solaris	Unix[b]	Independente
DBMS Suportados	PostgreSQL	PostgreSQL	PostgreSQL	PostgreSQL	Multi-plataforma[c]
Estado	Estável	Release recente	Estável	Estável	Estável

[a] Apenas dois *masters*.
[b] Apenas o daemon Perl tem de correr em ambiente UNIX. As bases de dados a sincronizar podem estar numa máquina com sistema operativo Windows.
[c] Oracle, IBM DB2, MS SQL Server, MySQL, Informix, Interbase, PostgreSQL, Firebird, Apache Derby, HSQLDB, H2 e Greenplum.

Capítulo 4

Solução Proposta

No Capítulo 3 identificámos e descrevemos diferentes soluções existentes para replicação de bases de dados. Após a sua análise concluímos que nenhuma dessas soluções reúne as características adequadas aos objetivos deste projeto.

Neste capítulo apresentamos uma solução para deteção e replicação de alterações em bases de dados, baseada em agentes de replicação. Após uma visão geral da solução, é descrita a sua arquitetura e é feita uma análise das suas características.

4.1 Descrição do sistema

A solução proposta consiste em ter um *site* principal e vários *sites* secundários. O *site* principal armazena um servidor de base de dados e um controlador de replicação. Os *sites* secundários contêm um servidor de base de dados, um servidor Noviserver, um agente de replicação e várias aplicações Noviclient (ver Figura 4.1).

As bases de dados armazenadas nos *sites* secundários são réplicas da base de dados original do sistema Novigest. Deste modo, cada *site* secundário utiliza a sua própria base de dados na qual o servidor Noviserver tem permissões de leitura e de escrita.

Os agentes de replicação são responsáveis pela sincronização dos dados entre as várias réplicas. Para isso, comunicam com o controlador de replicação localizado no *site* principal.

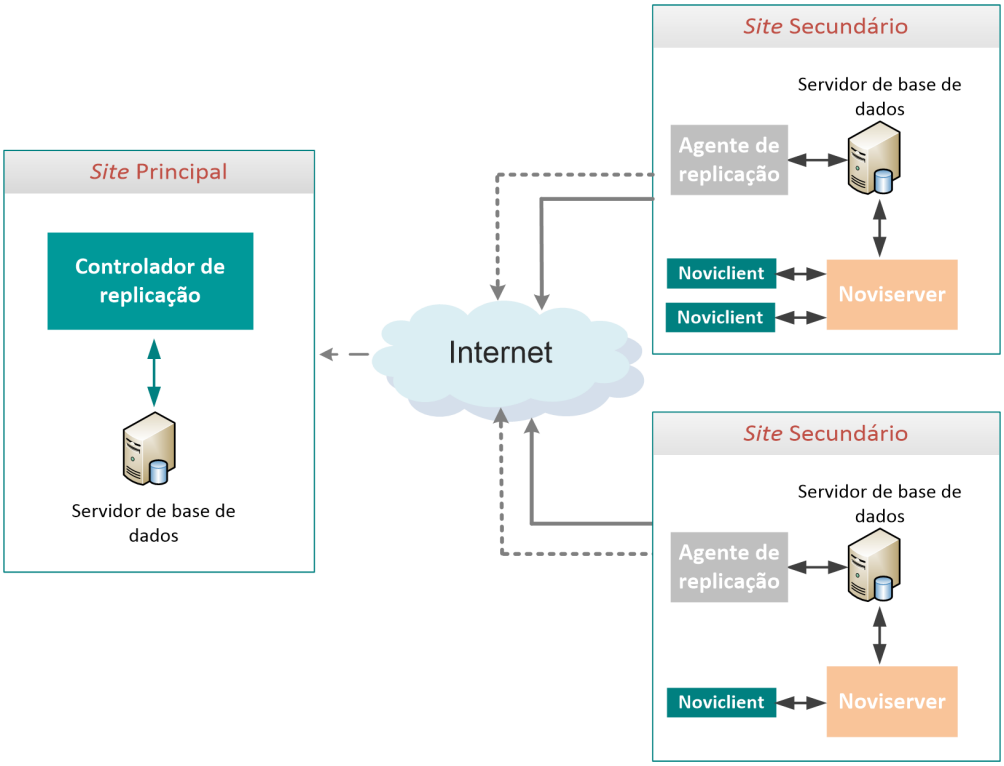


Figura 4.1: Sistema baseado em agentes de replicação.

Descrição dos *Sites* Secundários

A arquitetura dos *sites* secundários é composta pelos seguintes elementos:

- **Servidores de bases de dados secundárias**: os servidores de base de dados dos *sites* secundários são responsáveis pelo armazenamento dos dados gerados e alterados pelo servidor Noviserver local;
- **Agentes de Replicação**: os agentes de replicação são executados periodicamente. Cada uma destas entidades tem como funções: a deteção de alterações feitas na base de dados do *site* onde está a ser executado; a comunicação com o *site* principal; e a atualização da base de dados do seu *site*;
- **Servidores Noviserver e aplicações Noviclient**: cada *site* secundário contém um servidor Noviserver ao qual se ligam as várias aplicações Noviclient desse *site*.

Os agentes de replicação dos *sites* secundários efetuam a comunicação com o *site* principal. A comunicação com o *site* principal consiste no envio e pedido de atualizações:

- **Envio de atualizações:** no envio de atualizações, o agente de replicação de cada *site* secundário envia ao *site* principal as atualizações feitas aos dados;
- **Pedido de atualizações:** no pedido de atualizações, o agente de replicação requisita ao *site* principal atualizações feitas por outros *sites*.

Descrição do *Site* Principal

O *site* principal é constituído por um agente de replicação e um servidor de base de dados, onde está armazenada a base de dados principal.

A base de dados principal armazena todos os dados do sistema Novigest e informação para controlo de replicação. O controlador de replicação é responsável pelo processamento das atualizações recebidas no *site* principal. O processamento de atualizações é composto pelas seguintes fases:

1. **Gestão de conflitos:** as atualizações são passadas a um gestor de conflitos. Este gestor é responsável por verificar se existem duas atualizações sobre um mesmo registo da base de dados. Caso existam, será feita a resolução do conflito;
2. ***Log* de conflitos:** quando é detetado um conflito, é registado no *log* de conflitos. Neste *log* são guardados os dados em conflito e o resultado da resolução do conflito;
3. **Atualização da base de dados central:** uma vez resolvidos todos os conflitos, os dados são armazenados na base de dados central;
4. **Disponibilização de atualizações:** após a atualização da base de dados central, as atualizações são disponibilizadas aos *sites* secundários. Para isso, as atualizações são armazenadas numa fila de saída até que um *site* secundário envie um pedido de requisição de atualizações.

4.2 Requisitos da Solução

A utilização de um agente de replicação apresenta os seguintes requisitos:

- **Configurar dados a replicar**: uma vez que não se pretende que todos os dados sejam replicados, é necessário parametrizar o que deve ser replicado. Com base na

parametrização dos dados a replicar, o agente de replicação apenas deteta alterações aos dados pretendidos;

- **Detetar alterações**: um ponto crucial do sistema é o agente de replicação detetar que dados sofreram atualizações. Dado que o processo de replicação é executado periodicamente, é necessário definir um mecanismo para apenas detetar o que foi alterado desde a última sincronização com o *site* principal;
- **Armazenar as alterações**: existe um intervalo de tempo em que o agente de replicação está inativo. Durante este intervalo de tempo os dados alterados têm de ser armazenados;
- **Evitar ciclos na replicação**: a sincronização é bi-direcional, ou seja, um *site* secundário envia atualizações ao *site* principal e pede novas atualizações. O *site* principal só deve responder com as atualizações feitas por outros *sites*;
- **Detetar e resolver conflitos**: no *site* principal, é necessária a deteção de conflitos e a sua resolução. A resolução deverá ser feita de forma automática;
- **Manter a integridade referencial**: após as alterações serem enviadas para o *site* principal, deve-se manter a integridade referencial dos dados replicados. Tal só é possível se as chaves primárias dos dados se mantiverem.

Nas sub-secções seguintes são descritos dois métodos para deteção de atualizações à base de dados: utilização de *triggers* e utilização de versão dos dados. Para cada um destes métodos é feita uma descrição das suas principais vantagens e desvantagens.

4.3 Deteção de Alterações Baseada em *Triggers*

A deteção de alterações baseada no uso de *triggers* utiliza *triggers* SQL para detetar alterações aos dados. Cada tabela a replicar deverá ter um conjunto de *triggers* configurados.

4.3.1 Arquitetura dos *Triggers*

Os *triggers* são configurados para cada tabela e são desencadeados quando o servidor Noviserver efetua pedidos de alteração à base de dados através de instruções SQL (como

insert, *update*, *delete*). Cada pedido de alteração a registos de uma tabela, resulta num evento que desperta os *triggers* nela configurados.

A Figura 4.2 representa a arquitetura de deteção de alterações baseada em *triggers*.

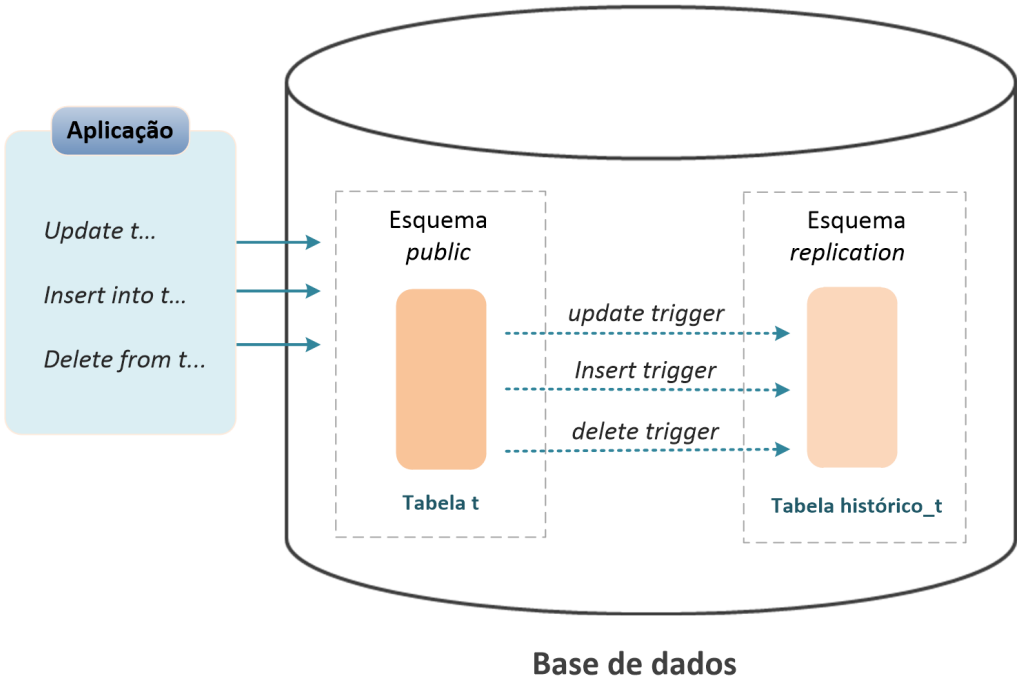


Figura 4.2: Arquitetura de deteção de alterações baseada em *triggers*.

Na base dados de cada *site* secundário são utilizados dois esquemas[1]: *public* e *replication*. O esquema *public* guarda os dados da aplicação Noviclient. O esquema *replication* guarda tabelas onde são armazenadas as alterações feitas aos dados. Os dados armazenados correspondem ao histórico das atualizações.

Os *triggers* são responsáveis por guardar no esquema *replication* as alterações que detetam. Em cada alteração são guardados: o evento que despertou os *triggers*, a hora, e a imagem dos dados antes e depois da atualização. Através dos *correlation names* *NEW* e *OLD*, os *triggers* guardam uma imagem dos dados antes e após uma alteração [19]. A variável *NEW* representa os dados alterados e a variável *OLD* representa os dados armazenados antes da alteração.

Cada tabela que se pretenda replicar tem os seus *triggers* e uma tabela correspondente no esquema *replication*. As tabelas do esquema *replication* guardam o histórico de alterações

[1] Um esquema define o modelo de dados, faz a descrição da base de dados e é definido quando se desenha a sua estrutura, nomeadamente as suas tabelas e relações entre elas [26].

que foram sendo feitas aos dados.

Na figura 4.3 apresentamos um exemplo dos *triggers* configurados. O *trigger* foi configurado numa tabela onde são guardados dados pessoais de clientes. Inicialmente, é criado um *trigger* com o nome *replicate_dadospessoais* (1). Este *trigger* é executado após um evento de *insert*, *update* ou *delete* (2). Está configurado na tabela dados pessoais (3) e para cada linha alterada (4) invoca o procedimento *function_replicate_dadospessoais()* (5).

```
1 CREATE TRIGGER replicate_dadospessoais
2  AFTER INSERT OR UPDATE OR DELETE
3  ON dadospessoais
4  FOR EACH ROW
5 EXECUTE PROCEDURE function_replicate_dadospessoais();
```

Figura 4.3: Exemplo de *Trigger*.

No Apêndice A apresentamos o procedimento *function_replicate_dadospessoais()*, escrito na linguagem Procedural Language/Structured Query Language (PL/SQL). O procedimento *function_replicate_dadospessoais()* guarda as alterações (detetadas pelo *trigger*) no esquema *replication*.

4.3.2 Avaliação da Deteção de Alterações Baseada em *Triggers*

As principais vantagens da solução de deteção de alterações baseada em *triggers* são:

- **Configuração de dados a replicar**: os *triggers* são configurados apenas nas tabelas a replicar. Deste modo é possível configurar que dados se pretendem replicar;
- **Deteção automática de alterações**: um *trigger* é desencadeado assim que ocorre um evento, detetando quais as alterações efetuadas;
- **Armazenamento do histórico de alterações**: o histórico de alterações é guardado num esquema independente do utilizado pelo Noviserver. As alterações ficam armazenadas no esquema *replication* até que o agente de replicação as envie ao *site* principal;
- **Resolução de conflitos**: através da imagem dos dados antes e após as alterações, é mais fácil resolver eventuais conflitos.

As principais desvantagens são:

- **Ocorrência de ciclos na replicação**: um *site* secundário tem de atualizar a base de dados quando recebe atualizações do *site* principal. Caso os *triggers* estejam ativos, serão detetadas essas atualizações e guardadas para posterior envio ao *site* principal. Por outro lado, caso se desativem os *triggers*, os dados que estão a ser modificados pelo Noviserver não são detetados. Para evitar ciclos, seria necessário bloquear temporariamente o acesso do servidor Noviserver à base de dados, o que é obviamente indesejável;

- **Pouca modularidade**: a base de dados utilizada pelo sistema Novigest está em constante evolução. Por esta razão é essencial garantir modularidade na configuração dos *triggers*. Os procedimentos invocados pelos *triggers* são difíceis de manter. Isto é, alterar um procedimento implica alterar o seu código e as tabelas do esquema *replication*;

- **Criação de novo esquema na base de dados**: é necessário utilizar um novo esquema na base de dados (esquema *replication* representado na Figura 4.2) para guardar o histórico de alterações de cada tabela que se pretenda replicar. O número de tabelas do novo esquema aumenta proporcionalmente com o número de tabelas a replicar;

- **Erros originados pelos *triggers***: caso um *trigger*, por algum motivo, desperte um erro na invocação duma função PL/SQL, o servidor Noviserver recebe um erro da base de dados. Como consequência, todas as aplicações Noviclient ligadas ao servidor Noviserver encerram de imediato.

4.4 Deteção de Alterações Baseada na Versão dos Dados

A deteção de alterações baseada na versão dos dados consiste na utilização de um identificador atribuído a cada registo onde é guardada a sua versão. Os identificadores analisados são: marca temporal e número de série de replicação. Com base na versão dos registos, o agente de replicação executa periodicamente *querys* à base de dados para detetar os registos alterados.

4.4.1 Utilização de Marca Temporal

A utilização de uma marca temporal para identificar a versão de um registo permite identificar quando é que o registo foi alterado. Os registos criados ou alterados pelo servidor Noviserver contêm um campo no qual é guardado o instante da última alteração. Contudo, esta solução apresenta uma desvantagem: o uso de marcas temporais num ambiente *multi-site* não é fiável. Isto deve-se ao facto dos relógios utilizados pelas máquinas que executam o servidor Noviserver não se encontrarem sincronizados. Logo, utilizar uma marca temporal como versão de um registo não é um identificador que se possa utilizar para serializar as alterações em diferentes réplicas.

4.4.2 Utilização de Número de Série

Outra forma de identificar a versão de um registo seria a utilização de um identificador gerido pelo *site* principal. Dado que os *sites* secundários comunicam periodicamente com o *site* principal, este gere o identificador como sendo um número de série de sincronização.

Quando um *site* secundário comunica com o principal, este responde com um novo número de série. Este número de série é guardado na base de dados. Sempre que o Noviserver altera um registo, o número de série é armazenado no registo para identificar a sua versão.

A Figura 4.4 representa a estrutura de um registo guardado numa tabela da base de dados. Em cada registo, para além dos dados da aplicação (como o *id*, *nome*, *morada*, *telefone* e *notas*), é guardada uma versão. Esta versão permite ao agente de replicação detetar quais os dados alterados desde a última comunicação com o *site* principal.

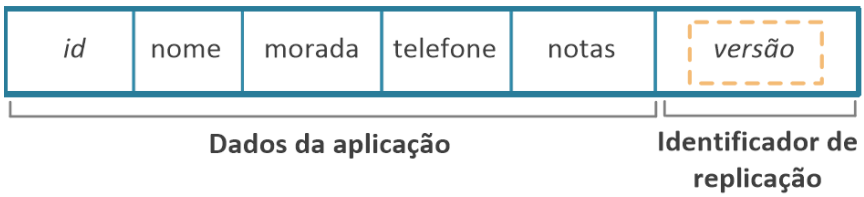


Figura 4.4: Estrutura de registo com versão.

O agente de replicação utiliza um ficheiro de configuração onde se definem todas as tabelas a replicar e os campos que se pretendem replicar em cada tabela. A Figura 4.5 apresenta um exemplo de um ficheiro de configuração.

O ficheiro é composto por um conjunto de elementos *tabela_a_replicar* que indica quais as tabelas que devem de ser replicadas (1) (neste exemplo é replicada apenas a tabela *dadospessoais*). Para cada tabela a replicar, são declaradas quais as colunas que devem ser replicados (2). Para isso, é utilizado o elemento *campo* que é constituído por vários atributos como: o nome da coluna, o tipo de dados, e referência a outras tabelas.

Para deteção de alterações o agente de replicação:

1. Lê o ficheiro de configuração onde constam as tabelas e campos a replicar;
2. Verifica o último número de série sincronizado;
3. Executa uma *query* sobre cada tabela a replicar, obtendo os registos que contêm uma versão superior ao último número de série sincronizado. Em cada registo obtido, são selecionados os campos a sincronizar.

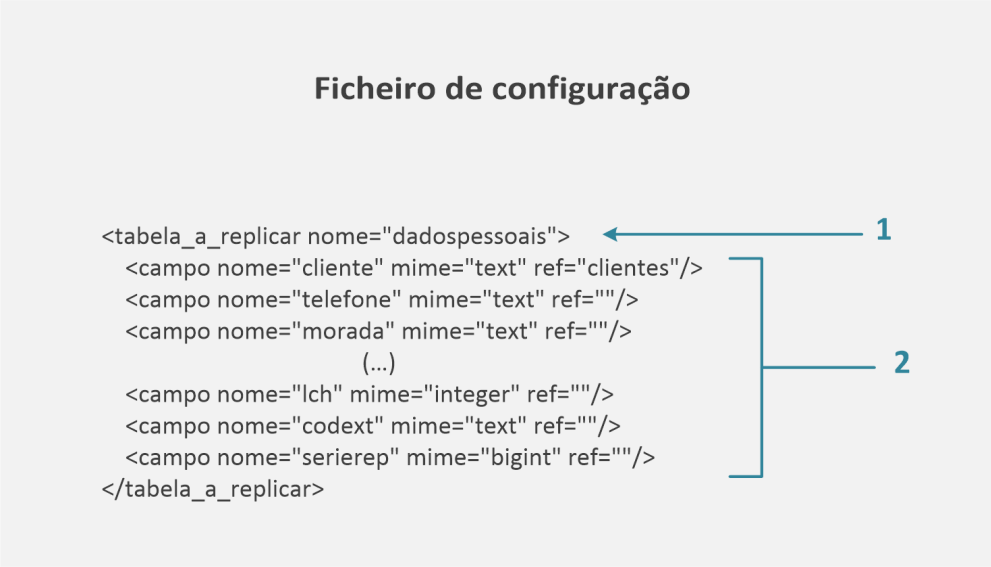


Figura 4.5: Ficheiro de configuração.

4.4.3 Avaliação da Deteção de Alterações Baseada na Versão dos Dados

A utilização de deteção de alterações baseada na versão dos dados satisfaz os seguintes requisitos:

- **Configuração de dados a replicar**: a utilização de um ficheiro de configuração permite descrever que dados se pretendem replicar. Neste ficheiro são identificadas as tabelas e campos a replicar;
- **Deteção de alterações**: através da utilização de um identificador, é possível detetar os dados que foram alterados desde a última sincronização;
- **Armazenamento das alterações**: o agente de replicação deteta as alterações feitas aos dados consultando as tabelas da base de dados;
- **Ausência de ciclos na replicação**: os dados alterados pelo servidor Noviserver contêm uma versão baseada num número de série armazenado na base de dados (previamente recebido do *site* principal). O *site* principal guarda o histórico de números de série que fornece aos *sites* secundários;
- **Resolução de conflitos**: a deteção e resolução de conflitos é feita no *site* principal durante o processamento das atualizações;
- **Manutenção de integridade referencial**: no ficheiro de configuração são incluídas as referências externas de cada campo. O agente de replicação acede à tabela referenciada e obtém o código externo do registo guardado na tabela que é referenciada.

4.5 Discussão

A solução descrita neste capítulo é baseada na utilização de agentes de replicação para sincronização de bases de dados *multisite*. Cada agente de replicação deteta periodicamente as alterações numa base de dados. Os dados presentes nas alterações são replicados de forma assíncrona para os restantes *sites* através de um controlador de replicação no *site* principal. Dado que nem todos os dados são replicados, trata-se de uma replicação parcial.

A utilização de um agente de replicação torna o sistema modular, pois permite a configuração da informação que se pretende replicar: tabelas e colunas. Nesta abordagem são analisadas duas estratégias para deteção das alterações feitas aos dados (ver Tabela 4.1): utilização de *triggers* e utilização de uma versão nos registos.

O uso de *triggers* permite uma deteção automática das atualizações e a configuração das tabelas a replicar. Os *triggers* possuem a vantagem de guardarem uma versão dos dados antes e após a atualização. No entanto, uma alteração nos dados a replicar ou no esquema

da base de dados implica alterar a configuração dos *triggers*. Além disso, pressupõem o uso de um novo esquema na base de dados. No novo esquema, por cada tabela a replicar terá que existir uma tabela onde são armazenadas as alterações. Caso se pretendam adicionar novas tabelas ao sistema de replicação, é necessária a criação de novos *triggers* e novas tabelas para armazenamento das alterações.

A utilização de um identificador consiste em adicionar uma versão aos registos de cada tabela. Através da versão, o agente de replicação consulta as tabelas da base de dados e deteta os dados alterados desde a última sincronização. A versão é baseada num número de série gerido pelo *site* principal que o disponibiliza aos *sites* secundários. Neste método, contrariamente à utilização de *triggers*, não é necessário criar um novo esquema na base de dados.

A estratégia de um agente de replicação baseado num identificador para deteção de alterações é a que apresenta melhores características. Esta abordagem não requer demasiadas alterações à base de dados atual e permite uma configuração modular.

O método utilizado para envio das atualizações é o método centralizado. Ou seja, existe um *site* (*site* principal) para o qual todos os restantes *sites* propagam as alterações aos dados. O *site* principal contém um servidor de base de dados onde está guardada uma cópia de todas as entidades guardadas nas bases de dados dos *sites* secundários. Só após ser feito o *commit* de todas as alterações (que não gerem conflito) na base de dados principal, é que as alterações são disponibilizadas aos restantes *sites*.

A deteção de alterações não é baseada em transações. As alterações são detetadas através de uma consulta a cada tabela da base de dados. Por este motivo, perde-se consistência transacional.

Contrariamente à deteção de alterações baseada em transações, a deteção de alterações baseada em consultas às tabelas da base de dados para obter os dados com uma dada versão, não permite ao agente de replicação detetar a eliminação de dados. No entanto, o Novigest atualmente não elimina dados. Em vez disso, o Novigest inclui um campo junto dos dados que os identifica como inválidos.

As bases de dados dos *sites* secundários não se encontram mutuamente consistentes a cada instante. Isto deve-se ao uso do tipo de replicação assíncrono. No entanto, periodicamente, a consistência mútua é reposta através da utilização de um controlador de replicação. A consistência mútua é atingida quando entre os instantes em que o agente de replicação envia e pede atualizações ao controlador de replicação, não são feitas atualizações aos dados nas várias réplicas.

Nesta solução, cada *site* secundário possui um servidor de base de dados local. Neste servidor está armazenada a base de dados utilizada pelo *software* Novigest. Deste modo, quando não existe ligação à Internet os dados continuam disponíveis.

Quando um dos *sites* secundários fica *offline*, todo o sistema continua a funcionar. Esta solução garante tolerância a falhas de comunicação. Caso seja o *site* principal a falhar, a replicação dos dados permanece suspensa até que o *site* principal seja reposto. No entanto, cada *site* secundário continua com acesso aos dados guardados localmente.

Tabela 4.1: Comparação de estratégias para deteção de alterações.

Característica	**Abordagem**	
	Deteção Baseada em *Triggers*	Deteção Baseada na Versão dos Dados
Configurável por tabela	✓	✓
Ocorrência de conflitos	✓	✓
Deteção de alterações	*Triggers*	Versão
Armazenamento de alterações	Novo esquema	Desnecessário

Capítulo 5

Implementação

No Capítulo 4 apresentámos uma solução baseada em agentes de replicação constituída por agentes de replicação e por um controlador de replicação que gere as alterações aos dados comunicadas pelos agentes.

Neste capítulo descrevemos o desenvolvimento de um protótipo dessa solução. Inicialmente é dada uma visão geral do sistema. De seguida, é descrito em pormenor o desenvolvimento do agente de replicação (nos *sites* secundários) e do controlador de replicação (no *site* principal). Por fim, é testado o protocolo de comunicação entre os agentes e o controlador de replicação.

5.1 Visão Geral do Sistema

O protótipo implementado permite a sincronização das bases de dados de múltiplos *sites* secundários. Para isso, cada *site* secundário utilizada um agente de replicação, um servidor de base de dados e um servidor Noviserver. O Noviserver tem acesso de leitura e escrita à base de dados do seu *site*. No *site* principal é utilizado um controlador de replicação e um servidor de base de dados (ver Figura 5.1).

Os agentes de replicação detetam alterações feitas na base de dados através da consulta de tabelas. A consulta de cada tabela consiste em procurar os registos com uma versão superior à última sincronizada com o *site* principal.

Os agentes de replicação comunicam com o *site* principal e com Noviserver. O Noviserver contém um módulo, desenvolvido neste projeto, para o processamento de documentos

estruturados recebidos do agente de replicação. Assim, quando o *site* principal envia um conjunto de atualizações, o agente de replicação comunica-as ao Noviserver que atualiza os dados da sua *cache* e a base de dados.

A base de dados do *site* principal armazena os dados dos *sites* secundários e informação acerca do controlo de replicação. O controlador de replicação é responsável por:

- Processar as atualizações vindas dos *sites* secundários;
- Detetar/resolver possíveis conflitos;
- Guardar as alterações na base de dados principal;
- Disponibilizar as atualizações aos *sites* secundários.

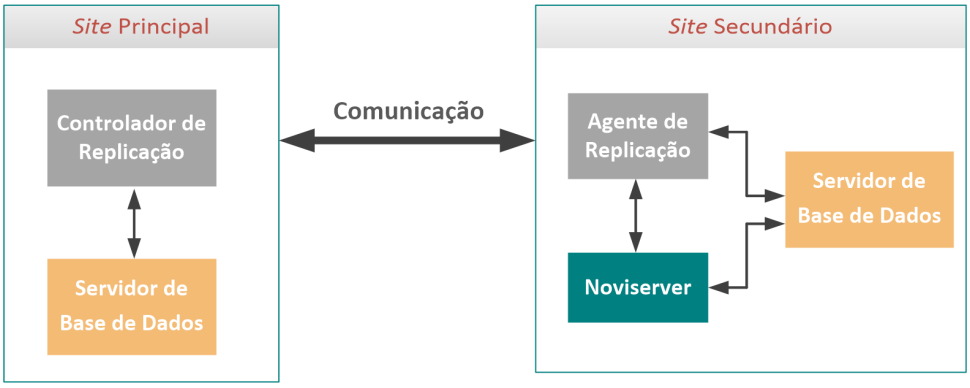


Figura 5.1: Visão geral do sistema implementado.

5.2 *Sites* Secundários

Seguidamente descrevemos os agentes de replicação e as alterações introduzidas nas bases de dados de cada *site* secundário.

5.2.1 Agente de Replicação

O agente de replicação é uma aplicação para execução nos *sites* secundários. Esta aplicação foi desenvolvida na linguagem Java e contém um módulo para comunicação com o controlador de replicação e para comunicação com o servidor Noviserver.

5.2.1.1 Arquitetura do Agente de Replicação

A Figura 5.2 ilustra a arquitetura do agente de replicação. Este agente utiliza as APIs: JDBC API, Document Object Model (DOM) API e Jersey API, para acesso à base de dados, processamento de documentos eXtensible Markup Language (XML) e configuração do módulo de comunicação, respetivamente.

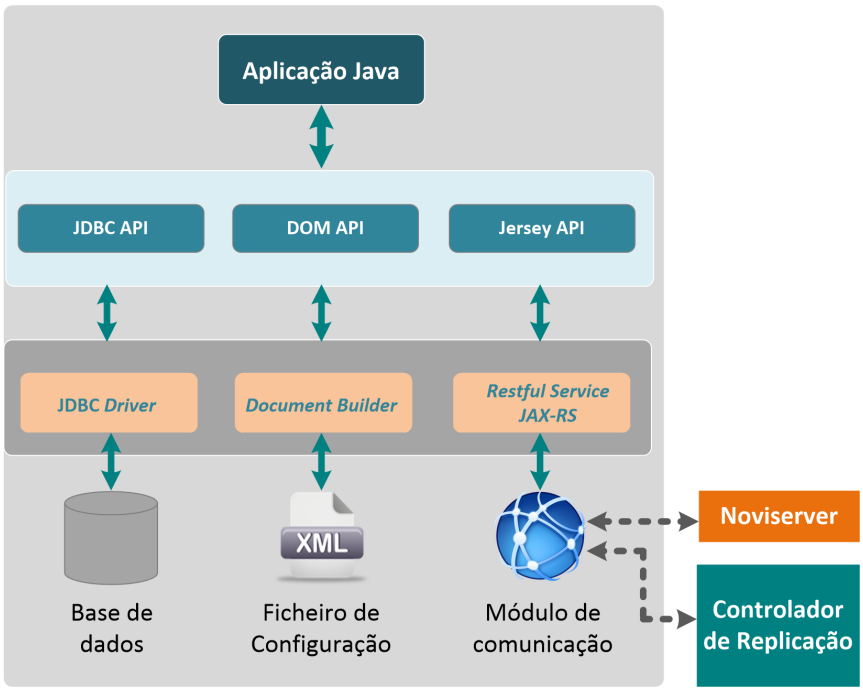


Figura 5.2: Arquitetura do agente de replicação.

O XML é um formato flexível de texto para transferência de dados entre diferentes ambientes (e.g. ambientes *web*). Foi concebido com a preocupação de transportar dados e não a sua apresentação [10][6].

Neste projeto, o XML é utilizado para: criar o ficheiro de configurações em cada *site* secundário, troca de atualizações entre os *sites* secundários e principal e envio de atualizações do agente de replicação ao Noviserver.

A API Jersey é uma implementação *standard open-source* de criação de *web services* RESTful [11]. Esta API utiliza duas licenças, uma GNU General Public License (GPL) e a outra Common Development and Distribution License (CDDL) [11].

A API DOM permite processar documentos XML formados segundo um *standard* próprio. Esta API possibilita navegar na estrutura do documento, adicionar, alterar e remover ele-

mentos [22].

No módulo de comunicação, as mensagens trocadas entre o *site* principal e secundário utilizam os formatos XML e JavaScript Object Notation (JSON). No envio de atualizações do agente de replicação para o servidor Noviserver é utilizado o XML. O JSON é um formato de transferência de dados bastante "leve" do ponto de vista da largura de banda necessária. É um formato de texto independente de qualquer linguagem em particular, o que permite que possa ser lido e escrito por um humano [9].

5.2.1.2 Descrição do Agente de Replicação

O agente de replicação executa várias tarefas de forma periódica. Estas encontram-se representadas no fluxograma da Figura 5.3.

Quando o agente inicia, pede ao *site* principal um novo número de série e guarda-o na base de dados. Sempre que o Noviserver altera registos na base de dados, associa-lhe a versão dos dados, utilizando o número de série guardado.

De seguida, o agente de replicação lê o ficheiro de configurações, fazendo o seu *parsing*. Com base nas tabelas e campos a replicar lidos do ficheiro, o agente faz um *query* às tabelas pedindo os registos com uma versão maior ou igual à do último número de série sincronizado. Para cada registo no resultado desse *query*, é criada uma nova entrada num ficheiro XML único.

Após a consulta de todas as tabelas, se o ficheiro resultante não contiver atualizações, não será enviado. Caso existam atualizações o ficheiro é enviado ao *site* principal. Em qualquer cenário (envio ou falha) será guardado o número de série que se sincronizou (ou que se tentou sincronizar).

De seguida, o agente requisita atualizações ao *site* principal. Caso não existam atualizações, o agente termina a sua execução. Se o *site* principal devolver atualizações, a resposta conterá um ficheiro com atualizações e uma *flag* indicando se existem atualizações pendentes.

Quando o agente recebe atualizações provenientes do controlador de replicação, envia-as para o Noviserver. Assim que o módulo de processamento de documentos estruturados do Noviserver recebe o ficheiro, processa-o, atualizando a base de dados e atualizando a sua memória *cache*. Por fim, envia uma resposta ao agente de replicação. Esta resposta, em caso de sucesso, será encaminhada para o *site* principal para que o controlador de

replicação marque o conjunto de atualizações como enviado.

O agente de replicação termina esta iteração de sincronização se o *site* principal não enviar mais atualizações.

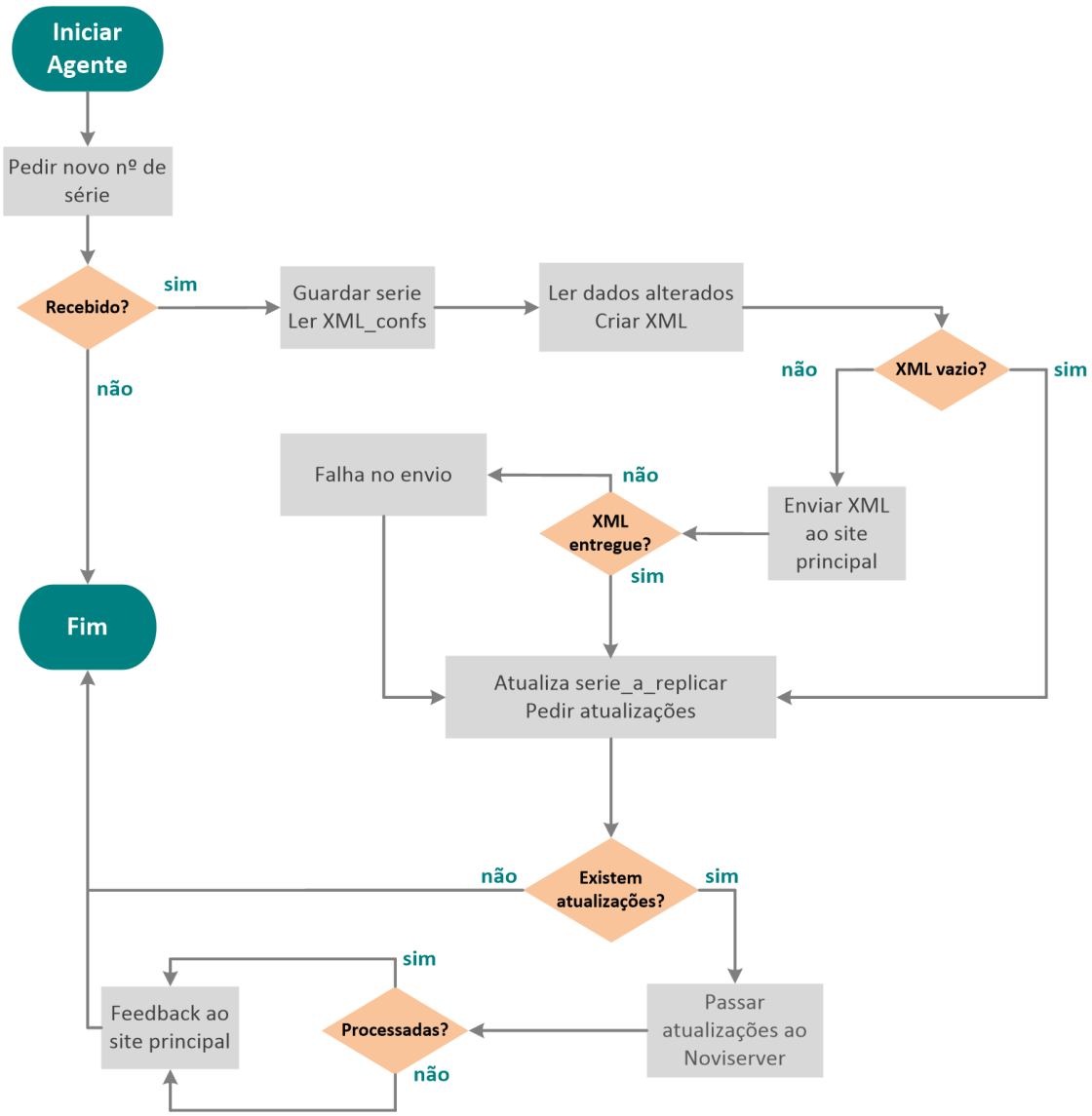


Figura 5.3: Fluxograma do agente de replicação.

5.2.1.3 Gestão do Número de Série

O número de série é utilizado para distinguir versões dos dados, permitindo ao agente de replicação identificar as atualizações que deve sincronizar com o *site* principal. Em cada *site* secundário são guardados dois números de série: o atual e o último sincronizado. O

número de série atual é sempre maior ou igual ao último sincronizado.

O número de série atual é atualizado sempre que é executada uma sincronização com o *site* principal. Para isso, o agente de replicação solicita um novo número e guarda-o localmente. Este número é utilizado pelo Noviserver sempre que altera algum registo na base de dados (associando-lhe o número de série).

O último número de série sincronizado é utilizado pelo agente de replicação para poder identificar os últimos dados enviados ao *site* principal. O agente assume que todos os registos na base de dados com um número de série maior do que o último sincronizado terão de ser enviados ao *site* principal. Quando o agente de replicação envia atualizações ao *site* principal, atualiza o último número de série sincronizado.

A figura 5.4 representa um exemplo da utilização do número de série.

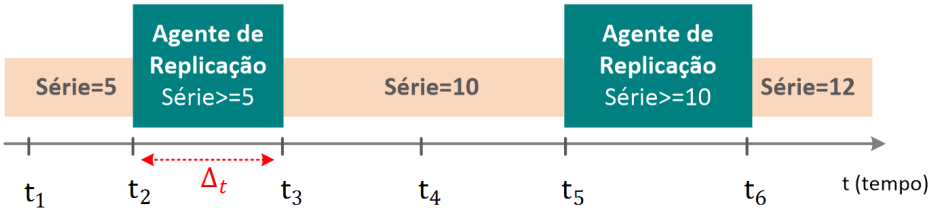


Figura 5.4: Gestão do número de série.

Suponhamos que no intervalo de tempo entre o instante t_1 e t_2 o número de série atual é cinco. A partir de t_2 o agente de replicação é executado durante um intervalo de tempo Δt. Nesse intervalo o agente de replicação (ver fluxograma da Figura 5.3):

1. Pede um novo número de série ao *site* principal;
2. Guarda-o na base de dados;
3. Efetua uma *query* sobre as tabelas a replicar, utilizando a condição *serie>=5*.

Supondo que o novo número de série recebido foi dez, o número de série a utilizar pelo Noviserver até ao instante t_5 é dez. No instante t_5, o agente de replicação é executado novamente repetindo-se o processo.

A utilização de dois números de série nos *sites* secundários impede a ocorrência dos seguintes problemas:

1. **Falha na deteção de atualizações:** o sistema Novigest está continuamente a alterar dados. Logo, o Noviserver terá que passar a utilizar um novo número de série antes do agente de replicação iniciar a *query* sobre os dados a replicar. Enquanto o agente de replicação não obtiver um novo número de série, não executa o *query* sobre os dados a replicar.
2. **Falha no envio de atualizações:** se ocorrer um erro no envio de atualizações ao *site* principal, na próxima execução o agente de replicação terá que saber qual o último número de série sincronizado com sucesso.

Desta forma, caso ocorra algum erro durante a execução do agente de replicação, o último número de série sincronizado identifica os últimos registos enviados com sucesso ao *site* principal.

5.2.1.4 Módulo de Comunicação

A comunicação com o *site* principal (controlador de replicação) é feita através do módulo de comunicação (cliente *web service*) do agente de replicação.

Os pedidos enviados ao controlador de replicação utilizam os seguintes métodos:

1. **Método *postUpdates*:** envio de um ficheiro XML contendo os registos alterados na base de dados, incluindo o identificador do *site*;
2. **Método *getSerie*:** pedido de um novo número de série ao *site* principal, enviando o identificador do *site*;
3. **Método *getUpdates*:** pedido de atualizações ao *site* principal, através do envio de identificador do *site*;
4. **Método *updateCheckout*:** envio de *feedback* sobre o sucesso/insucesso de um conjunto de atualizações. É incluído no pedido o identificador do conjunto de atualizações.

5.2.2 Bases de Dados Secundárias

Nas bases de dados secundárias (réplicas), foi necessário adicionar algumas colunas às tabelas existentes, bem como tabelas para guardar informação de replicação. Estabeleceu-

se que cada *site* secundário teria que possuir um identificador único guardado na base de dados local.

A cada tabela a replicar foram adicionados os seguintes campos:

1. **Campo *lch*:** O campo *lch* representa o último *site* que alterou o registo. Sempre que algum *site* altera um registo numa tabela, modifica este campo colocando nele o seu identificador. Este campo é utilizado no *site* principal para resolução de conflitos;

2. **Campo *serierep*:** O campo *serierep* guarda a versão do registo. Este número de série é alterado pelo Noviserver sempre que altera um registo na base de dados. Deste modo o agente de replicação consegue distinguir os registos que foram alterados desde a última sincronização;

3. **Campo *idext*:** O campo *idext* representa a chave primária de um registo num contexto externo ao *site* secundário. Uma vez que os dados podem ser alterados por qualquer servidor de base de dados, cada registo contém duas chaves primárias: chave local e chave global. A chave primária local é utilizada localmente para manter a integridade referencial. A chave primária global é composta pela chave local e por um identificador do *site* e permite saber, posteriormente, no *site* principal onde foi criado o registo pela primeira vez e relacioná-lo com o registo local.

Para se guardar informação acerca do processo de sincronização, foram criadas as tabelas *location* e *seriereplicacao*.

A tabela *location* contém dois campos: *site_id* e *server_address*. No campo *site_id* é guardado o identificador do *site*. No campo *server_address* é guardado o endereço IP do *site* principal.

A tabela *seriereplicacao* é utilizada para guardar os números de série. Contém os campos: *serie* e *serie_a_replicar*. O campo *serie* guarda o último número de série recebido do servidor. O Noviserver utiliza o valor deste campo sempre que altera um registo da base de dados. No campo *serie_a_replicar* é guardado o último número de série sincronizado. O agente de replicação guarda neste campo o último número de série que sincronizou com o *site* principal.

5.3 *Site* Principal

Nesta secção descrevemos o desenvolvimento do controlador de replicação no *site* principal. Descrevemos os pedidos recebidos pelo controlador de replicação, o processamento de atualizações, o mecanismo de resolução de conflitos e a estrutura da base de dados utilizada no *site* principal.

5.3.1 Controlador de Replicação

O controlador de replicação é uma aplicação *web* desenvolvida no servidor aplicacional Glassfish (a configuração do servidor Glassfish pode ser consultada no Apêndice B).

O Glassfish é um servidor *open source*, com uma boa performance e escalável. Atualmente tem uma comunidade dedicada ao seu suporte e desenvolvimento [7].

A aplicação *web* é baseada na tecnologia Java Platform Enterprise Edition (Java EE) que permite utilizar vários módulos do servidor Glassfish para processamento de atualizações, recepção de pedidos vindos dos *sites* secundários e interação com a base de dados do *site* principal.

5.3.1.1 Arquitetura do Controlador de Replicação

A Figura 5.5 representa a arquitetura do *site* principal.

O servidor Glassfish permite utilizar vários módulos (contentores). Neste projeto foram utilizados os módulos Enterprise JavaBeans (EJB) e *web*. O módulo EJB é utilizado no processamento de atualizações e o módulo *web* é utilizado na comunicação com os *sites* secundários.

O acesso do controlador de replicação à base de dados é feito pela Java Persistence API (JPA). A *framework* de persistência de dados utilizada é o EclipseLink.

O EclipseLink é uma *framework open-source* sobre uma licença Eclipse Public License (EPL). Pode ser utilizada em *software* proprietário [2]. Existem outras *frameworks* JPA, como a Hibernate. No entanto, o EclipseLink apresenta melhores características tais como performance e escalabilidade na interação com o PostgreSQL [12].

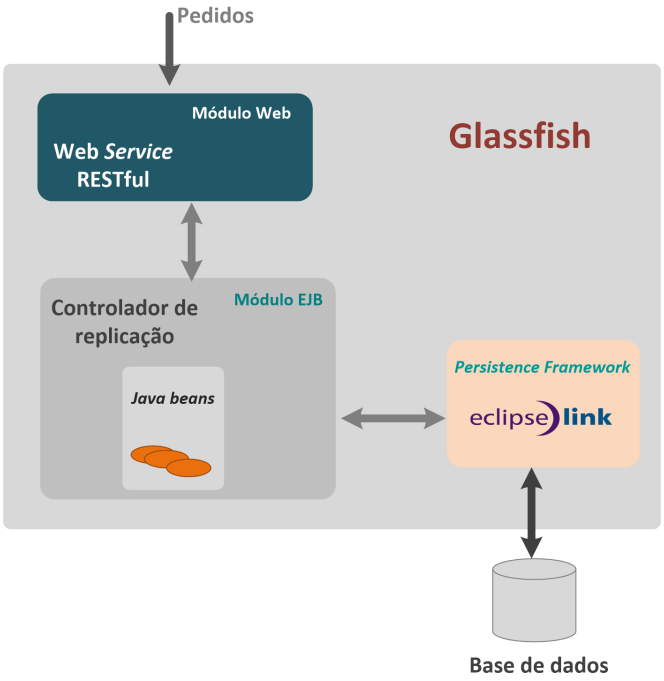


Figura 5.5: Arquitetura do sistema final.

Módulo *web*

O módulo *web* é responsável pela recepção de pedidos dos *sites* secundários. Este módulo foi configurado através da API Jersey disponibilizando um *web service* que é invocado pelos clientes (agentes de replicação dos *sites* secundários).

O tipo de *web service* utilizado é o RESTful. Este tipo de *web service* tem suporte para integração com o protocolo HyperText Transfer Protocol (HTTP) e permite que diferentes clientes possam invocar recursos configurados do lado do servidor de forma escalável [5].

Na Listagem 5.1 apresentamos um fragmento dos recursos disponibilizados através de RESTful. Estes recursos permitem que os agentes de replicação dos *sites* secundários comuniquem com o *site* principal. Neste fragmento temos dois recursos, o *post* e o *getSerie*:

- **Recurso *post*:** O recurso *post* é utilizado para receber pedidos nos quais os agentes de replicação enviam as atualizações. É do tipo HTTP POST. O conteúdo do pedido é do tipo *application/xml* recebendo um parâmetro (*@QueryParam*) com o nome *site_id* (identificador do *site* que enviou o pedido).
- **Recurso *getSerie*:** Através do recurso *getSerie*, os agentes de replicação requisitam

ao controlador de replicação um novo número de série. O recurso *getSerie* é do tipo HTTP GET. O conteúdo dos pedidos e das respostas é do tipo *application/json*. Este recurso recebe como parâmetro o identificador do *site* que enviou o pedido.

Listing 5.1: Excerto de recursos no módulo *web*

```
//metodo http
@POST
//caminho relativo do recurso
@Path("post")
//tipo de conteudo recebido
@Consumes({"application/xml"})
 public Response post(@QueryParam("site_id") int site_id, Document doc)


//metodo http
@GET
//caminho relativo do recurso
@Path("getserie")
//tipo de conteudo recebido
@Consumes({"application/json"})
//tipo de conteudo devolvido
@Produces({"application/json"})
    public String getSerie(@QueryParam("site_id") String site_id)
```

Módulo Enterprise JavaBeans (EJB)

O módulo Enterprise JavaBeans é utilizado para o processamento das atualizações. Neste módulo definimos os procedimentos a executar para a deteção/resolução de conflitos, a atualização da base de dados e a disponibilização das atualizações aos restantes *sites*.

A atualização da base de dados é feita através da *framework* EclipseLink que disponibiliza uma API para persistência de dados. Esta API permite criar entidades em Java associadas a entidades da base de dados. Cada entidade Java é guardada na base de dados utilizando um gestor de entidades (ver Apêndice C).

Na Listagem 5.2 apresentamos um excerto de uma entidade Java que representa uma tabela da base de dados. Esta tabela guarda as atualizações para os *sites* secundários, após a deteção/resolução de conflitos e atualização da base de dados no *site* principal.

Listing 5.2: Excerto de uma entidade Java

```
@Entity
@Table(name = "outgoingqueue")
@XmlRootElement

public class Outgoingqueue implements Serializable {
    @Size(max = 2147483647)
    @Column(name = "records")
    private String records;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "internalid")
    private Long internalid;

 //Construtor
 //Metodos
```

Cada linha iniciada pelo caractere @ é denominada anotação. Uma anotação permite criar um mapeamento entre uma entidade na base de dados e a entidade Java. Na Tabela 5.1 estão descritas as anotações utilizadas.

Tabela 5.1: Anotações utilizadas na Listagem 5.2

Anotação	Descrição
@Entity	Indica que se trata de uma Entidade Java.
@Table	Nome da tabela da base de dados.
@XmlRootElement	Anotação para serialização.
@Size	Tamanho máximo de um campo.
@Column	Coluna associada à variável.
@Id	Chave primária da entidade.
@GeneratedValue	Geração de valores para a chave primária.
@Basic	Campo do tipo primitivo.

5.3.1.2 Processamento de Pedidos

Como referimos na secção 5.2.1.4, o agente de replicação envia vários pedidos ao controlador de replicação. Descrevemos agora esses pedidos e o seu processamento por parte do controlador de replicação.

POST de atualizações

O POST de atualizações é utilizado pelo agente de replicação para enviar um conjunto de atualizações ao controlador de replicação. Na Figura 5.6 representamos o fluxo de processamento de um POST de atualizações proveniente de um *site* secundário.

O controlador de replicação processa um pedido POST de atualizações de cada vez. Para tal, o controlador de replicação verifica se já está a processar algum pedido POST de atualizações. Caso esteja, responde com uma mensagem indicando estado ocupado.

Caso o controlador de replicação esteja disponível, verifica a identidade do *site* secundário. As atualizações estão descritas em XML sendo necessário extrair as atualizações, tabela a tabela.

Cada atualização possui, para além dos dados, um identificador da tabela e a chave global do registo alterado. Depois da extração de atualizações procede-se à deteção e posterior resolução de conflitos.

Se existem conflitos, será feita a sua resolução registando num *log* as atualizações conflituosas e o resultado da sua resolução.

Após a deteção de conflitos, são carregadas as prioridades das tabelas. Cada tabela possui uma prioridade de processamento. De seguida, são carregadas as permissões que cada *site* secundário possui para receber atualizações.

Cada *site* secundário possui um conjunto de permissões guardadas na base de dados central. As permissões indicam ao controlador de replicação as atualizações que deve disponibilizar a cada *site* secundário.

A base de dados do *site* principal é atualizada com os registos de cada atualização. De seguida as atualizações são disponibilizadas aos *sites* secundários. A disponibilização das atualizações consiste em guardar na base de dados do *site* principal o conjunto de atualizações no formato XML, associada à informação do *site* secundário destino.

Por fim, é alterado o estado do controlador de replicação para *livre* e retornada uma resposta de sucesso HTTP 200 *OK*, ao *site* que enviou o pedido.

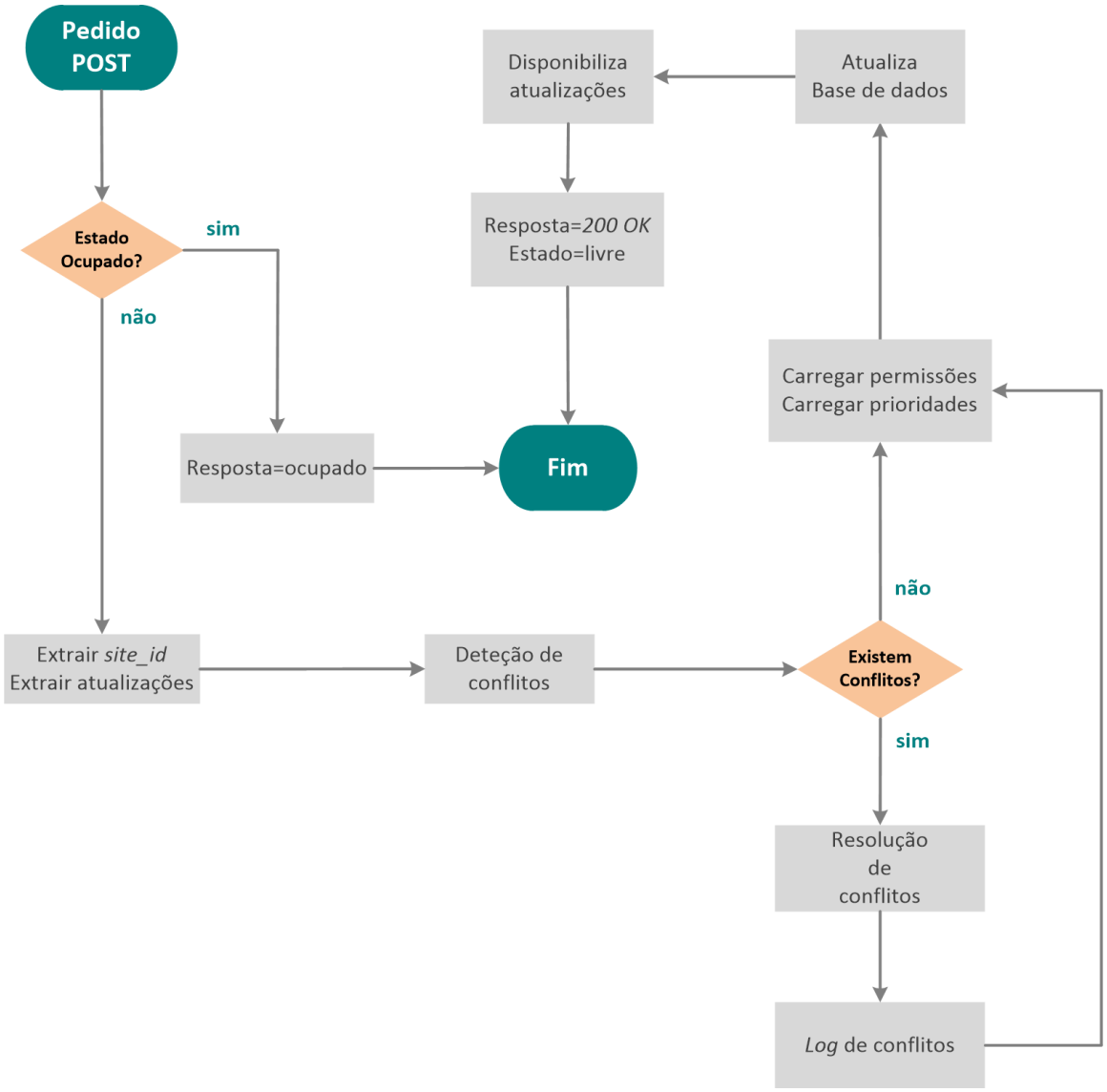


Figura 5.6: Fluxograma de pedido *post*.

GET de atualizações

O GET de atualizações é enviado pelo agente de replicação ao controlador de replicação, para obter atualizações dos dados.

Quando o controlador de replicação processa atualizações vindas dos *sites* secundários, estas são de imediato disponibilizadas aos restantes *sites*. Essas atualizações são guardadas na base de dados do *site* principal na tabela *outgoingqueue*.

O fluxograma da Figura 5.7 mostra o processamento de um pedido vindo de um *site* secundário a requisitar atualizações.

Assim que o controlador de replicação recebe um pedido GET de atualizações, extrai o identificador do *site* que está a requisitar as atualizações. Com base nesse identificador executa uma *query* à base de dados de modo a obter obter as atualizações pendentes para esse *site*. Caso não existam atualizações, o controlador de replicação responde com uma mensagem HTTP *204 No Content* e termina a sua execução.

Se existirem atualizações pendentes será feita uma verificação para determinar o seu número. Caso exista apenas uma, o controlador de replicação envia na resposta o ficheiro XML e um parâmetro (*flag last*) a indicar que é a última (*last=true*). Caso exista mais do que uma, a *flag* terá o valor *false*.

GET série e *update checkout*

Os pedidos de requisição de um novo número de série (GET série) e de atualização do estado de uma atualização pendente (*update checkout*), são utilizados para controlo da replicação.

O pedido GET série é utilizado para obtenção de um novo número de série, logo que o agente de replicação inicia a sua execução. Para isso, o controlador de replicação mantém na base de dados do *site* principal um número que incrementa à medida que são pedidos novos números de série.

No fluxograma da Figura 5.8 encontra-se representado o processamento de um pedido GET série. Após recebido um pedido a solicitar um novo número de série, o controlador de replicação extrai o identificador do *site*. De seguida obtém o número de série corrente guardado na base de dados principal, retornado-o como resposta. Por fim, é incrementado

o número guardado na base de dados principal e registado, num histórico, o número de série obtido pelo *site secundário*.

O pedido *update checkout* destina-se a enviar ao *site* principal *feedback* acerca do processamento de um conjunto de atualizações recebido num pedido GET.

A Figura 5.9 mostra o processamento de um pedido *update checkout*. Quando o controlador de replicação recebe um pedido deste tipo, extrai o identificador da atualização e marca-a como enviada.

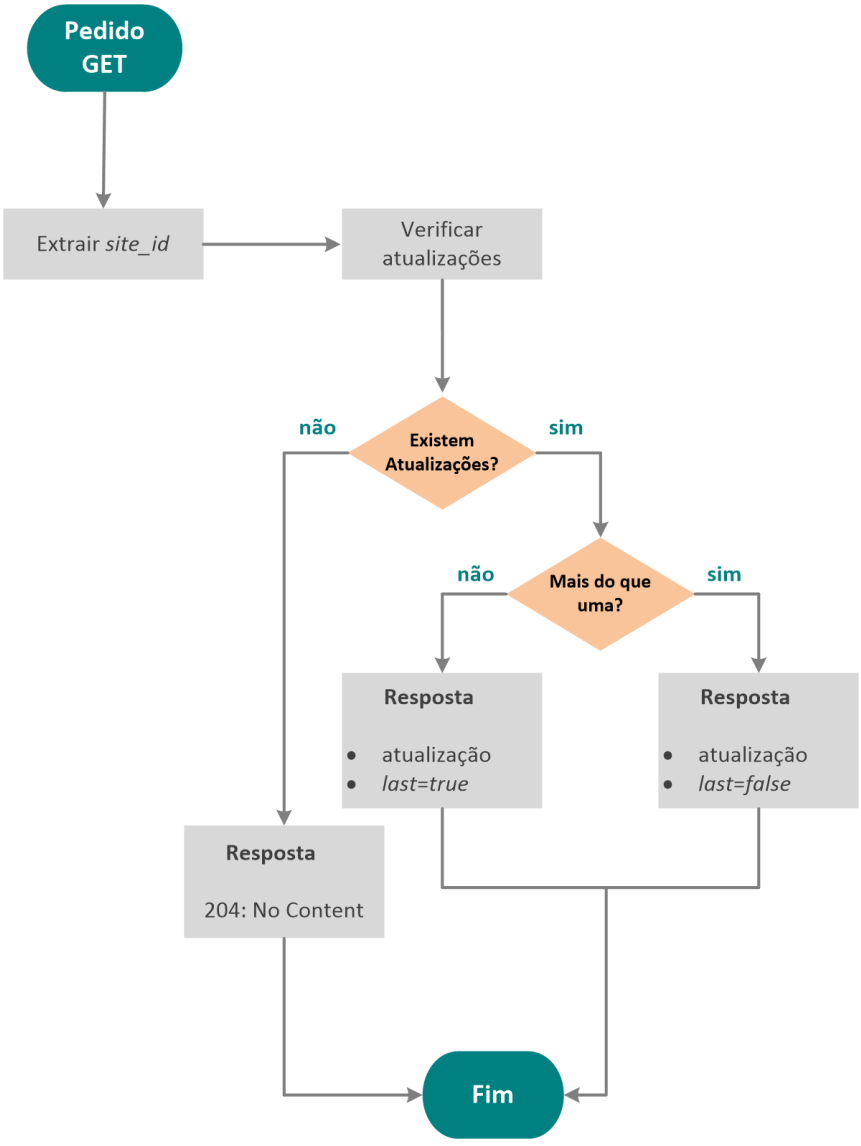


Figura 5.7: Fluxograma de pedido *get*.

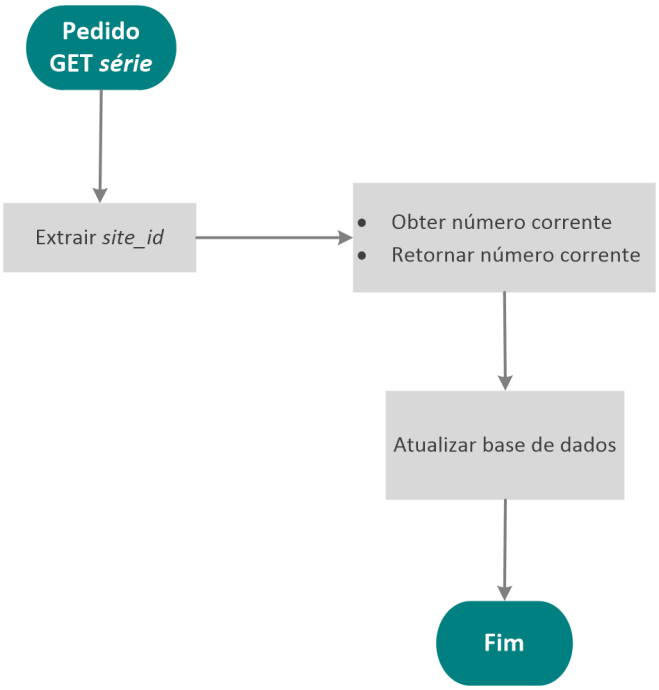


Figura 5.8: Fluxograma de pedido *get* série.

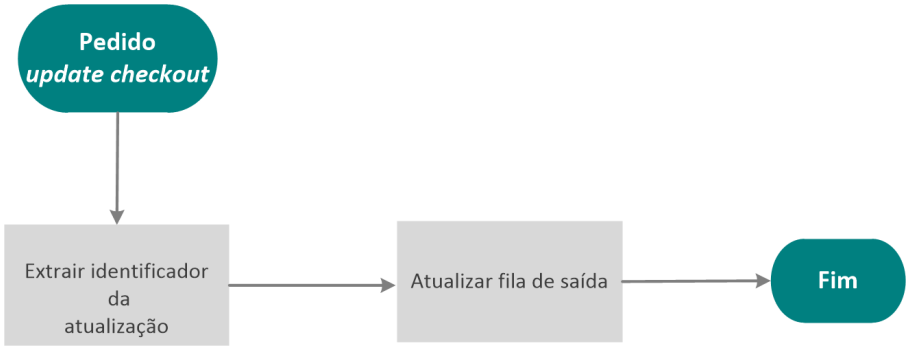


Figura 5.9: Fluxograma de pedido *update checkout*.

5.3.1.3 Processamento de Atualizações

Cada atualização recebida pelo controlador de replicação contém os campos:

- ***idext*:** chave primária global do registo;
- ***lch*:** identificador do *site* que originou a alteração;
- ***serierep*:** número de série da atualização;

- ***tabela*:** nome da tabela a que respeita a alteração;
- ***dados*:** registo dos dados alterados.

O processamento das atualizações é feito de forma hierárquica para manter a integridade referencial. Considere o seguinte exemplo representado na Figura 5.10 em que:

- A tabela *A* referencia *B* e *D*;
- A tabela *B* referencia *C*;

As atualizações da tabela *C* são as primeiras a ser processadas para que, quando se processar atualizações da tabela *B*, os registos em *C* referenciados pela tabela *B* já estejam guardados. Da mesma forma, as atualizações das tabelas *B* e *D* são processadas antes das atualizações da tabela *A*.

Esta ordem de processamento é garantida atribuindo uma prioridade de processamento a cada tabela. Esta prioridade permite criar uma árvore em que as folhas têm a maior prioridade e a raiz a menor.

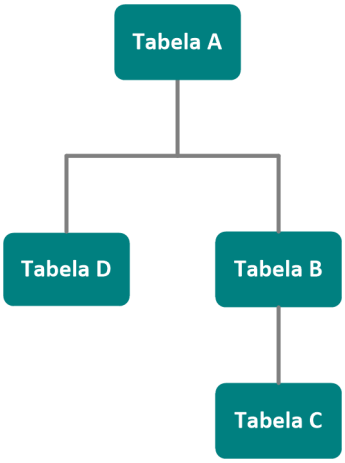


Figura 5.10: Hierarquia de processamento de atualizações.

5.3.1.4 Deteção e Resolução de Conflitos

Um conflito dá-se quando um *site* secundário tentar alterar um registo que foi alterado por outro *site*. Isto significa que o *site* que está a tentar alterar o registo tem uma atualização pendente.

Quando é detetado um conflito são utilizadas heurísticas para a sua resolução. As heurísticas são uma forma de solucionar um problema, neste caso um conflito, de forma a que a sua resolução produza uma solução aceitável para o sistema.

Neste projeto são utilizadas duas heurísticas baseadas em prioridades:

1. **Prioridade por *site*:** dadas duas atualizações sobre um mesmo registo originadas em *sites* diferentes, será escolhida a atualização gerada pelo *site* que criou o registo. A informação sobre quem cria um registo está guardada na chave global de cada registo.

2. **Prioridade por influência:** é utilizada quando duas alterações sobre o mesmo registo são feitas por *sites* diferentes em que nenhum deles foi o "criador" do registo. Será escolhida a atualização gerada pelo *site* que tem maior influência no sistema. A influência de cada *site* é determinada pelo número de vezes que um *site* envia atualizações ao *site* principal. Maior número de sincronizações representa uma maior influência no sistema.

Estas heurísticas são hierárquicas, ou seja, apenas se recorre à segunda quando a primeira não se aplica.

5.3.2 Base de Dados Principal

A base de dados do *site* principal é constituída por tabelas das bases de dados dos *sites* secundários e por tabelas para controlo de replicação.

A Figura 5.11 mostra o modelo relacional das tabelas utilizadas para controlo de replicação:

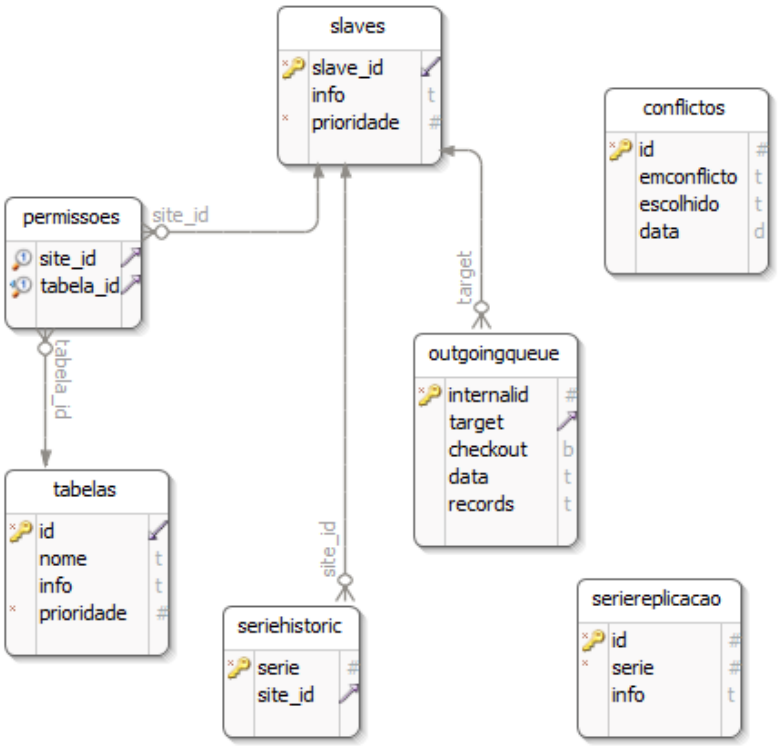


Figura 5.11: Tabelas para controlo de replicação.

- **Tabela *slaves*:** armazena informação acerca dos *sites* secundários, nomeadamente o identificador de cada *site* e o número de vezes que sincronizou informação com o *site* principal;

- **Tabela *seriereplicacao*:** guarda o número de série de replicação do sistema;

- **Tabela *tabelas*:** nome das tabelas que estão atualmente a ser sincronizadas. Para cada tabela é guardada uma prioridade de processamento;

- **Tabela *seriehistoric*:** regista o histórico de números de série que cada *site* secundário recebeu;

- **Tabela *permissoes*:** nome das tabelas de que cada *site* recebe atualizações;

- **Tabela *conflictos*:** é guardado o *log* de todos os conflitos detetados e o resultado da sua resolução;

- **Tabela *outgoingqueue*:** armazena as atualizações para os *sites* secundários. As atualizações são guardadas no formato XML.

5.4 Testes à Comunicação entre Agente de Replicação e Controlador de Replicação

Nesta secção é testada a comunicação entre os agentes e controlador de replicação. São descritas mensagens trocadas durante a sincronização e o seu conteúdo. Para isso, foi utilizado o *software* Curl [4]. O Curl permite efetuar pedidos HTTP ao módulo de comunicação do controlador de replicação. Através deste método é possível testar os pedidos criados pelos agentes de replicação verificando as respostas obtidas.

Pedido GET série

A Listagem 5.3 mostra um exemplo de um pedido GET série. Na linha 6, temos o tipo do pedido: HTTP *GET*; e o Uniform Resource Locator (URL) do recurso. No URL, é passado como parâmetro o identificador do *site*; neste caso o *site id* é 2.

A partir da linha 11 temos a resposta do controlador de replicação. O controlador de replicação retornou uma resposta HTTP *200 OK*. O seu conteúdo é do tipo JSON (linha 15). Os dados recebidos encontram-se na linha 19, onde se verifica que o novo número de série recebido do controlador de replicação é 769.

Listing 5.3: Exemplo de pedido *get* série

```
1  >curl -vv "http://192.168.70.71:8080/TactisRepMaster/rest/replicationv2/getserie?
       site_id=2"
2  * About to connect() to 192.168.70.71 port 8080 (#0)
   *   Trying 192.168.70.71...
4  * connected
   * Connected to 192.168.70.71 (192.168.70.71) port 8080 (#0)
6  > GET /TactisRepMaster/rest/replicationv2/getserie?site_id=2 HTTP/1.1
   > User-Agent: curl/7.28.1
8  > Host: 192.168.70.71:8080
   > Accept: */*
10 >
   < HTTP/1.1 200 OK
12 < X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.2.
   2 Java/Oracle Corporation/1.7)
14 < Server: GlassFish Server Open Source Edition 3.1.2.2
   < Content-Type: application/json
16 < Transfer-Encoding: chunked
   < Date: Mon, 13 May 2013 15:24:55 GMT
18 <
   {"serie":769}
20 * Connection #0 to host 192.168.70.71 left intact
```

```
* Closing connection #0
```

Pedido POST de atualizações

O pedido POST de atualizações, representado na Listagem 5.4, consiste no envio de atualizações ao controlador de replicação. Neste pedido incluímos as atualizações e o tipo de conteúdo do pedido. Na linha 1 utilizamos a opção *-H* para adicionar o tipo de conteúdo ao cabeçalho do pedido HTTP (neste caso *application/xml*). Através da opção *-D* enviamos um conjunto de atualizações no formato XML. O conjunto de atualizações utilizado tem as seguintes características:

- **Estrutura:** a estrutura utiliza o formato XML. Neste formato é definido um elemento raiz (elemento *Results*) que possui um atributo *site* com o valor 3. Este atributo identifica o *site* a que pertence o conjunto de atualizações. Cada elemento *Tabela* representa uma atualização sobre uma tabela da base de dados. Neste exemplo existem duas alterações sobre as tabelas *Dadospessoais* e *Pessoa* (linhas 3 e 13 respetivamente). Um elemento *Tabela* tem vários sub-elementos *campo*;
- **Conteúdo:** o conteúdo é representado pelos elementos *campo*. Cada um destes elementos representa uma coluna da tabela a que está associado. Um elemento *campo* é constituído por três atributos: *mime* (tipo de dados que a coluna representa), *nome* (nome da coluna) e o *ref* (indica se a coluna referencia uma tabela externa);
- **Referências:** as referências são guardas no atributo *ref* do elemento *campo*. Este atributo representa o nome da tabela externa referenciada por uma coluna. Na linha 8 temos um exemplo de uma coluna com o nome *razaovis* que referencia a tabela *razoesvisita*. A chave primária global é *5@3*.

Em cada atualização são utilizados três campos para controlo de replicação (linhas 9, 10 e 11 para a tabela *Dadospessoais* e 19, 20 e 21 para *Pessoa*).

A linha 29 indica o tipo do pedido: HTTP POST. No endereço do recurso é passado o identificador do *site*.

A resposta do controlador de replicação encontra-se a partir da linha 37. O controlador de replicação responde com a mensagem HTTP *100 Continue* para que o cliente envie o resto do pedido, seguida de uma mensagem de sucesso (linhas 37 e 38 respetivamente).

Listing 5.4: Exemplo de pedido *post* de atualizações

```
curl -vv -X POST -H "Content-Type: application/xml" -d
<Results site="3">
    <Tabela nome="Dadospessoais">
     <campo mime="text" nome="pessoa" ref="pessoa">9@3</campo>
     <campo mime="text" nome="morada" ref="">Resende</campo>
     <campo mime="text" nome="nacionalidade" ref="">Portuguesa</campo>
     <campo mime="text" nome="entidade1" ref="">SNS</campo>
     <campo mime="text" nome="razaovis" ref="razoesvisita">5@3</campo>
     <campo mime="integer" nome="lch" ref="">3</campo>
     <campo mime="bigint" nome="serierep" ref="">780</campo>
     <campo mime="text" nome="idext" ref="">9@3</campo>
    </Tabela>
    <Tabela nome="Pessoa">
     <campo mime="timestamp" nome="datacriacao" ref="">2013-04-17T16:16:11.598</campo
         >
     <campo mime="text" nome="nome" ref="">Dina Raquel</campo>
     <campo mime="date" nome="birth" ref="">1992-03-29T00:00:00.000</campo>
     <campo mime="text" nome="telemovel" ref="">989957841</campo>
     <campo mime="text" nome="site" ref="site">1@3</campo>
     <campo mime="integer" nome="lch" ref="">3</campo>
     <campo mime="bigint" nome="serierep" ref="">780</campo>
     <campo mime="text" nome="idext" ref="">9@3</campo>
    </Tabela>
</Results>

  http://192.168.70.71:8080/TactisRepMaster/rest/replicationv2/post?site_id=3
  *  About to connect() to 192.168.70.71 port 8080 (#0)
  *   Trying 192.168.70.71...
  * connected* Connected to 192.168.70.71 (192.168.70.71) port 8080 (#0)
  > POST /TactisRepMaster/rest/replicationv2/post?site_id=3 HTTP/1.1
  > User-Agent: curl/7.28.1
  > Host: 192.168.70.71:8080
  > Accept: */*
  > Content-Type: application/xml
  > Content-Length: 4381
  > Expect: 100-continue
  >
  < HTTP/1.1 100 Continue
  < HTTP/1.1 200 OK
  < X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.2.2
      Java/Oracle Corporation/1.7)
  < Server: GlassFish Server Open Source Edition 3.1.2.2
  < Content-Length: 0
  < Date: Mon, 13 May 2013 16:22:52 GMT
  <* Connection #0 to host 192.168.70.71 left intact
  * Closing connection #0
```

Pedido *get* de atualizações

O pedido de atualizações inclui o identificador do *site* que inicia o pedido, tal como é apresentado na linha 6 da Listagem 5.5. Nesta mesma linha está identificado o tipo do pedido (HTTP GET) e o endereço do recurso *getupdates*.

Após verificar se existem atualizações, o controlador de replicação responde com uma mensagem de sucesso HTTP *200 OK* (linha 11). A resposta incluí o tipo de conteúdo (*application/xml* na linha 17) e um conjunto de atualizações no formato XML.

Nas linhas 15 e 16, temos dois parâmetros recebidos no cabeçalho da resposta, os campos *last* e *xml_id*. O campo *last* é utilizado para informar o agente de replicação se a atualização presente no corpo da resposta é a última disponível. Neste caso, o valor *true* significa que não existem mais atualizações. O segundo campo, *xml_id*, é o identificador do conjunto de atualizações. Este identificador é enviado ao controlador de replicação após o processamento das atualizações.

Listing 5.5: Exemplo de pedido *get* de atualizações

```
1  >curl -vv "http://192.168.70.71:8080/TactisRepMaster/rest/replicationv2/getupdates?
       site_id=2"
2  * About to connect() to 192.168.70.71 port 8080 (#0)
   *   Trying 192.168.70.71...
4  * connected
   * Connected to 192.168.70.71 (192.168.70.71) port 8080 (#0)
6  > GET /TactisRepMaster/rest/replicationv2/getupdates?site_id=2 HTTP/1.1
   > User-Agent: curl/7.28.1
8  > Host: 192.168.70.71:8080
   > Accept: */*
10 >
   < HTTP/1.1 200 OK
12 < X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.2.
   2 Java/Oracle Corporation/1.7)
14 < Server: GlassFish Server Open Source Edition 3.1.2.2
   < last: true
16 < xml_id: 1648
   < Content-Type: application/xml
18 < Transfer-Encoding: chunked
   < Date: Mon, 13 May 2013 15:35:15 GMT
20 <
   <Updates site="2">
22     <Tabela nome="Pessoa">
         <campo mime="text" nome="telemovel" ref="">989957841</campo>
24       <campo mime="date" nome="birth" ref="">1992-03-29T00:00:00.000</campo>
         <campo mime="timestamp" nome="datacriacao" ref="">2013-04-17T16:16:11.598</
             campo>
26       <campo mime="integer" nome="lch" ref="">3</campo>
```

```
        <campo mime="text" nome="site" ref="site">1@3</campo>
28      <campo mime="text" nome="idext" ref="">9@3</campo>
        <campo mime="bigint" nome="serierep" ref="">767</campo>
30      <campo mime="text" nome="nome" ref="">Dina Raquel</campo>
      </Tabela>
32    <Tabela nome="Dadospessoais">
        <campo mime="text" nome="nacionalidade" ref="">Portuguesa</campo>
34      <campo mime="text" nome="morada" ref="">Resende</campo>
        <campo mime="text" nome="razaovis" ref="razoesvisita">5@3</campo>
36      <campo mime="integer" nome="lch" ref="">3</campo>
        <campo mime="text" nome="entidade1" ref="">SNS</campo>
38      <campo mime="text" nome="idext" ref="">9@3</campo>
        <campo mime="bigint" nome="serierep" ref="">767</campo>
40      <campo mime="text" nome="pessoa" ref="pessoa">9@3</campo>
      </Tabela>
42  </Updates>
  * Connection #0 to host 192.168.70.71 left intact
44  * Closing connection #0
```

Pedido *update checkout*

A Listagem 5.6 exemplifica o envio de *feedback* por parte de um agente de replicação ao controlador de replicação após processamento de um conjunto de atualizações.

No pedido são adicionados o tipo dos dados e o conteúdo (linha 1 e 2). O tipo dos dados é *application/json*. Como conteúdo, é enviado o identificador do conjunto de atualizações recebidas no *site* secundário (neste caso o identificador é 1647).

A linha 7 indica o tipo do pedido (HTTP POST) e o caminho para o recurso (*update-checkout*). Após o processamento do pedido, o controlador de replicação retorna uma mensagem de sucesso HTTP *200 OK* (linha 15).

Listing 5.6: Exemplo de pedido *update checkout*

```
>curl -vv -X POST -H "Content-Type: application/json" -
d " {"update_id":1647}" http://192.168.70.71:8080/TactisRepMaster/rest/replicationv2/
    updatecheckout
* About to connect() to 192.168.70.71 port 8080 (#0)
*   Trying 192.168.70.71...
* connected
* Connected to 192.168.70.71 (192.168.70.71) port 8080 (#0)
> POST /TactisRepMaster/rest/replicationv2/updatecheckout HTTP/1.1
> User-Agent: curl/7.28.1
> Host: 192.168.70.71:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 17
>
* upload completely sent off: 17 out of 17 bytes
< HTTP/1.1 200 OK
< X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.2.
2 Java/Oracle Corporation/1.7)
< Server: GlassFish Server Open Source Edition 3.1.2.2
< Content-Length: 0
< Date: Mon, 13 May 2013 16:39:56 GMT
<
* Connection #0 to host 192.168.70.71 left intact
* Closing connection #0
```

5.5 Resumo

Neste capítulo descrevemos a arquitetura e as opções de implementação de um sistema de replicação constituído por agentes e por um controlador de replicação. Os agentes são aplicações desenvolvidas na linguagem Java. Cada agente envia e pede atualizações ao controlador de replicação utilizando o *standard* XML para a transferência de registos alterados. As mensagens para controlo de replicação utilizam o formato JSON. Em cada agente de replicação existe um módulo de comunicação desenvolvido através da API Jersey.

Um ficheiro XML fornece ao agente de replicação a configuração das tabelas e colunas a replicar em cada *site* secundário.

O controlador de replicação utiliza a tecnologia Java EE e consiste numa aplicação *web* num servidor aplicacional Glassfish. Este controlador recebe as atualizações vindas dos agentes de replicação através de um módulo de comunicação desenvolvido e configurado com o *web service* RESTful. As atualizações são processadas, sendo detetados e resol-

vidos eventuais conflitos. A resolução de conflitos é feita através de heurísticas baseadas em prioridades. O resultado da resolução de um conflito é sempre registado num *log*. O modelo de persistência dos dados adotado para atualização da base de dados do *site* principal é o JPA.

Dada a complexidade do servidor Noviserver, os agentes de replicação não são uma camada intermédia entre o Noviserver e a base de dados. São uma aplicação independente do sistema Novigest que apenas comunica com o Noviserver através de um módulo desenvolvido neste servidor para a recepção dos registos alterados no formato XML.

No *site* principal, as atualizações são processadas de forma hierárquica por forma a garantir a consistência das chaves externas entre as tabelas replicadas. No entanto, este protótipo não prevê relações cíclicas entre as tabelas. Ou seja, dadas duas tabelas *A* e *B*, se a tabela *A* tem uma chave externa para *B*, então *B* não pode conter uma chave externa para *A*.

Dado que a sincronização de alterações é feita periodicamente, quando alguém num *site* secundário está a fazer alterações a um conjunto de registos, os registos podem "subitamente" ser alterados pelo agente de replicação modificando os dados que estavam a ser alterados localmente. Isto sucede quando o agente pede atualizações ao controlador de replicação e as passa ao Noviserver. Neste caso, as alterações ainda não guardadas na base de dados são perdidas.

O protocolo para sincronização de alterações com o *site* principal implementa dois pedidos de envio e de requisição de alterações. O agente de replicação envia os registos que foram alterados no seu *site* secundário, antes de pedir atualizações, ao *site* principal. Isto permite que as alterações possam ser processadas de forma a que sejam detetados eventuais conflitos no controlador de replicação. Caso o agente pedisse primeiro as alterações ao *site* principal, estas poderiam gerar conflito com os registos guardados localmente.

Capítulo 6

Conclusões e Trabalho Futuro

O projeto descrito neste relatório tinha como objetivo principal permitir que, dado um sistema *multi-site* onde a base de dados está armazenada no *site* principal, os restantes *sites* continuassem a operar quando não existe ligação ao *site* principal. Para isso, migrámos a base de dados do *site* principal para uma base de dados distribuída onde cada *site* secundário contém uma réplica local da base de dados.

Idealmente, a base de dados deveria ser fragmentada pelos vários *sites*, tornando cada *site* responsável pelo fragmento que armazena, evitando conflitos. Como não foi possível fragmentar a base de dados devido à sua complexidade e à arquitetura do Noviserver, optamos pela replicação da base de dados. Fizemos uma análise a diferentes projetos para replicação de bases de dados, verificando que nenhum deles satisfaz os requisitos definidos inicialmente.

Dadas as características do sistema pretendido — suportar o modo *offline*, suportar sincronização de alterações, minimizar alterações ao sistema Novigest e à base de dados, ser configurável, ser multiplataforma, ser escalável e utilizar tecnologias *open-source* — propusemos uma solução baseada em agentes de replicação. Nesta solução, cada *site* secundário possui uma réplica da base de dados principal, um servidor Noviserver e um agente de replicação. O *software* Novigest pode operar localmente sem estar dependente da ligação à Internet. As atualizações das bases de dados do *sites* secundários são detetadas por um agente de replicação com base num identificador associado aos dados. O identificador é gerido pelo *site* principal e é utilizado pelo Noviserver sempre que este altera os dados. O agente de replicação guarda o histórico de identificadores já sincronizados com o controlador de replicação (localizado no *site* principal).

No mecanismo de deteção e resolução de conflitos executado no *site* principal existirão alterações que serão descartadas pelas heurísticas utilizadas. Os dados eliminados do sistema, são guardados num *log* que é armazenado na base de dados do *site* principal. Visto que a resolução de conflitos é automática, pode ser necessária a recuperação de dados excluídos. Para isso, o *log* permite registar os dados em conflito e o resultado da resolução do conflito.

Na implementação do protótipo final, são utilizadas duas chaves primárias em cada registo guardado nas bases de dados a replicar: uma chave local para garantir integridade referencial entre as tabelas guardadas localmente e uma chave global que permite identificar inequivocamente um registo em qualquer *site* do sistema.

A solução proposta não garante, simultaneamente, que (1) as bases de dados estejam mutuamente consistentes a cada instante, (2) os dados estejam sempre disponíveis e (3) a falha de uma parte do sistema não afete todo o sistema. No entanto, assim que é feita a sincronização de todas as réplicas, assegura-se consistência mútua. Para isso, existe um mecanismo de deteção e resolução centralizado de conflitos baseado em duas heurísticas: prioridade por *site* e prioridade por influência. Os dados estão sempre disponíveis localmente e a falha de um *site* não afeta os restantes.

Trabalho Futuro

A solução apresentada poderá incluir novas funcionalidades num futuro próximo, tanto no controlo da sincronização como na sua configuração.

O protótipo implementado não suporta o tratamento de dados apagados, uma vez que o Novigest não apaga dados do sistema (apenas os torna inativos). No entanto, este sistema pode ser alterado para garantir esse tipo de operações, bastando criar um histórico das chaves primárias globais e tabelas dos registos apagados. Deste modo, o agente de replicação poderia enviar ao *site* principal esse registo.

Um *site* secundário, sem ligação à Internet, pode demorar algum tempo a comunicar com o *site* principal. Isto pode causar, por exemplo, duplicação de fichas de pacientes, visto que estes podem entretanto ir a uma clínica situada num *site* diferente. Isso seria resolúvel se todas as fichas de pacientes contivessem um dado único, como o número de cidadão ou número fiscal. Dado que estes dados são facultativos na versão atual do sistema Novigest, numa versão futura convém torná-los obrigatórios para permitir ao *site* principal gerir as fichas duplicadas.

No *site* principal o registo de conflitos está atualmente a ser guardado numa tabela da base de dados em formato XML. Uma ferramenta interessante seria a criação de uma plataforma para que o administrador das clínicas pudesse interagir com este registo, quer para acompanhar ou anular alguns resultados do gestor de conflitos. Esta seria uma forma de melhorar as heurísticas utilizadas de momento.

Apêndice A

Função de *Trigger*

Listing A.1: Função PL/SQL para replicação de dados pessoais

```
CREATE OR REPLACE FUNCTION function_replicate_dadospessoais()
  RETURNS trigger AS
$BODY$
DECLARE
 current_datetime timestamp:=LOCALTIMESTAMP;
 source_site integer;
 mycode text;
 triger text:='replicate_dadospessoais';
 checkin boolean:=false;
    mycode:=NEW.dbkey||'@'||source_site;
BEGIN

 SELECT site_id INTO source_site from location;
 IF tg_op = 'INSERT' THEN
     INSERT INTO replication.dadospessoais(siteID, evento, triggerid, checkin,
         modification_date, new_dbkey, new_class, mycode)
     VALUES (source_site,tg_op,triger,checkin,current_datetime,NEW.dbkey,NEW.class,
         mycode);
     Return NEW;
  END IF;

  IF tg_op = 'UPDATE' THEN
  INSERT INTO replication.dadospessoais(siteID, evento, triggerid ,checkin,
      modification_date, old_dbkey, old_class, old_pessoa, old_morada, old_localidade
      , old_email, (...), new_dbkey, new_class, new_pessoa, new_morada,
      new_localidade, new_email,(...))
  VALUES(source_site, tg_op, triger, checkin, current_datetime, OLD.dbkey, OLD.class,
       OLD.pessoa, OLD.morada, OLD.localidade, OLD.email, (...), NEW.dbkey, NEW.class
      , NEW.pessoa, NEW.morada, NEW.localidade, NEW.email,(...));
 RETURN new;
  END IF;
END;
```

```
$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
ALTER FUNCTION function_replicate_dadospessoais()
  OWNER TO postgres;
```

Na Listagem A.1 está representada a função *funtion_replicate_dadospessoais* invocada por um *trigger* .

Inicialmente são declaradas algumas variáveis utilizadas no processo de replicação como: a data e hora a que foi despertado o *trigger*, o código do *site* (guardado na tabela *location* da base de dados) e uma chave primária composta pelo código do *site* e pela chave primária local.

Através da variável *tg_op* é possível verificar o evento que desencadeou o *trigger*. No código são considerados eventos de *insert* e *update*. Visto que o Noviserver nunca elimina dados, não ocorrem eventos *delete*.

Caso se trata de um evento *insert*, o Noviserver tem uma forma particular de lidar com a criação de registos. Nessa instrução SQL, o Noviserver apenas cria uma entrada contendo a chave primária e um identificador que relaciona o tipo de registo com as classes Java. De seguida é feito um *update* inserindo os campos com os respetivos dados. Assim, nesta função, procede-se do mesmo modo e, é guardada a instrução de *insert* seguida de um *update*.

Note-se que são sempre guardados os dados antes e depois de serem alterados, utilizando os *correlation names OLD* e *NEW* na tabela *dadospessoais* do esquema *replication*.

Apêndice B

Configuração do Servidor Glassfish

A configuração representada na Listagem B.1 corresponde ao ficheiro *web.xml* utilizado pelo servidor Glassfish. O *alias TactisRepMaster* define caminho de acesso ao servidor. Este *alias* é declarado no elemento XML *<display-name>*. De seguida, é configurada a utilização da API Jersey para configuração do Web *service REST*. Por fim, é indicado o padrão URL para definir a localização dos recursos, elemento *<url-pattern>*.

Através desta configuração, o caminho para os recursos a serem acedidos pelos agentes de replicação é: *http://endereçoIP:8080/TactisRepMaster/rest/*. A este endereço basta concatenar o caminho relativo dos recursos.

Listing B.1: Ficheiro de configuração do servidor

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee␣http://java.sun.com/
             xml/ns/javaee/web-app_2_5.xsd"
         id="WebApp_ID"
         version="2.5">

   <display-name>TactisRepMaster</display-name>
  <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet
            -class>
    <init-param>
       <param-name>com.sun.jersey.config.property.packages</param-name>
       <param-value>service</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
```

```
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>

</web-app>
```

A Listagem B.2 apresenta o ficheiro *glassfish-resources.xml*. Neste ficheiro está configurado um recurso que representa o acesso à base de dados do *site* principal. Neste recurso são parametrizados alguns valores para a criação de uma *pool* de ligações à base de dados.

Listing B.2: Configuração de recursos no Glassfish

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD␣GlassFish␣Application␣Server␣3.1␣
    Resource␣Definitions//EN" "http://glassfish.org/dtds/glassfish-resources_1_5.dtd"
    >

<resources>
    <jdbc-connection-pool allow-non-component-callers="false" associate-with-thread="
        false" connection-creation-retry-attempts="0" connection-creation-retry-
        interval-in-seconds="10" >
        <property name="serverName" value="localhost"/>
        <property name="portNumber" value="5432"/>
        <property name="databaseName" value="masterreplication"/>
        <property name="User" value="dap"/>
        <property name="Password" value="foobar"/>
        <property name="URL" value="jdbc:postgresql://localhost:5432/
            masterreplication"/>
        <property name="driverClass" value="org.postgresql.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource enabled="true" jndi-name="jdbc/masterreplication" object-type="
        user" pool-name="post-gre-sql_masterreplication_dapPool"/>
</resources>
```

Apêndice C

Gestor de Entidades

A utilização da *framework* EclipseLink permite o uso de JPA. O gestor de entidades (*EntityManager*) permite apagar, modificar ou guardar entidade na base de dados. Na Listagem C.1 é apresentado um exemplo da criação de um gestor de entidades.

Listing C.1: Criação de gestor de entidades

```
@PersistenceContext(unitName = "TactisRepMasterPU")
    private EntityManager em;
```

Antes da criação da variável *em* que representa o gestor de entidades, é definido o contexto do mesmo. O contexto define: (1) o conjunto de entidades Java criadas para mapeamento das tabelas na base de dados e (2) a ligação à base de dados. Podem ser definidos vários contextos no ficheiro de configuração *persistence.xml*, representado Na Listagem C.2.

Listing C.2: Configuração de contextos JPA

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="
    http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.
    com/xml/ns/persistence␣http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
    ">

  <persistence-unit name="TactisRepMasterPU" transaction-type="JTA">
    <jta-data-source>jdbc/masterreplication</jta-data-source>
    <class>Entidades2.Dadospessoais</class>
    <class>Entidades2.Empresa</class>
    <class>Entidades2.Entidade</class>
    <class>Entidades2.Ingoingqueue</class>
    <class>Entidades2.Pessoa</class>
    <class>Entidades2.Razoesvisita</class>
    <class>Entidades2.Seriereplicacao</class>
```

```
    <class>Entidades2.Site</class>
    <class>Entidades2.Slaves</class>
    <class>Entidades2.Tabelas</class>
    <class>Entidades2.Outgoingqueue</class>
    <class>Entidades2.Seriehistoric</class>
    <class>Entidades2.Conflictos</class>
    <class>Entidades2.Conveniopaciente</class>
    <class>Entidades2.PessoaConvenios</class>
    <class>Entidades2.Consulta</class>
    <class>Entidades2.Tratamentos</class>
    <class>Entidades2.ConsultaTratamentos</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
    </properties>
  </persistence-unit>
</persistence>
```

Neste caso é utilizado o contexto *TactisRepMasterPU*. O acesso à base de dados é definido no elemento *<jta-data-source>*. Este acesso encontra-se configurado no servidor Glassfish como um recurso: *jdbc/masterreplication*. O restante conteúdo do ficheiro define o nome das entidades Java.

Ao longo do projeto foram utilizados vários métodos disponibilizados pela API JPA do EclipseLink. Estes métodos permitem a utilização do gestor de entidades. A Listagem C.3 apresenta um exemplo da utilização desses mesmos métodos através de duas funções: *getUpdates* e *saveObjects*.

Na função *getUpdates*, é criada uma *query* utilizando a linguagem de *querys* JPA que tem uma sintaxe ligeiramente diferente da linguagem SQL. Nesta função, obtêm-se todos os registos pendentes na tabela *outgoingqueue*. Os registos são ordenados por ordem decrescente através da coluna *internalid*. Por fim, é retornada a lista dos registos.

Na função *saveObjects*, é recebida uma lista de entidades Java. Para cada entidade, é obtido o identificador da entidade e é utilizado o método *find* do gestor de entidades para ver se o registo já existe na base de dados. Caso exista, é feita uma atualização ao registo correspondente através do método *merge*. Se o registo ainda não existe, será criado na base de dados utilizando o método *persist*.

Listing C.3: Excerto de utlização do gestor de entidades

```
public List<Outgoingqueue> getUpdates() {
    String query = "SELECT␣o␣FROM␣Outgoingqueue␣o␣WHERE␣o.checkout=:checkout␣
        order␣by␣o.internalid␣desc";
    Query q = em.createQuery(query).setParameter("checkout", false);
    List<Outgoingqueue> res = q.getResultList();
```

```
    return res;
    }



 public void saveObjects(List list) {
        Iterator<Update> i = list.iterator();
        while (i.hasNext()) {
            Update u = i.next();
            String idext = u.getIdext();
            Object exist = em.find(u.getClass(), idext);
                if (exist != null) {
                    em.merge(u);
                } else {
                    em.persist(u);
                }
            }
}
```

Referências

[1] About Postgresql. http://www.postgresql.org/about/. Consultado em Janeiro de 2013.

[2] About the Eclipse Foundation. http://www.eclipse.org/org/#about. Consultado em Abril de 2013.

[3] Bucardo Overview. http://bucardo.org/wiki/Bucardo/Documentation/Overview. Consultado em Novembro de 2012.

[4] Curl. http://curl.haxx.se/. Consultado em Maio de 2013.

[5] Deciding Which Type of Web Service to Use. http://docs.oracle.com/javaee/6/tutorial/doc/gjbji.html. Consultado em Abril de 2013.

[6] Extensible Markup Language (XML). http://www.w3.org/XML/. Consultado em Abril de 2013.

[7] Glassfish Server Open Source Edition. https://glassfish.java.net/public/faq/GF_FAQ_2.html. Consultado em Abril de 2013.

[8] Hot Standby. http://www.postgresql.org/docs/9.2/static/hot-standby.html. Consultado em Dezembro de 2012.

[9] Introducing JSON. http://www.json.org/. Consultado em Abril de 2013.

[10] Introduction to XML. http://www.w3schools.com/xml/xml_whatis.asp. Consultado em Abril de 2013.

[11] Jersey. https://jersey.java.net/. Consultado em Abril de 2013.

[12] JPA Performance Benchmark (JPAB). http://www.jpab.org/EclipseLink/PostgreSQL/server/Hibernate/PostgreSQL/server.html. Consultado em Abril de 2013.

[13] Log-Shipping Standby Servers. `http://www.postgresql.org/docs/9.1/static/warm-standby.html`. Consultado em Dezembro de 2012.

[14] Pgcluster: System Composition. `http://pgcluster.projects.pgfoundry.org/`. Consultado em Novembro de 2012.

[15] Pgpool-ii: Project Overview. `http://wiki.postgresql.org/wiki/Pgpool-II`. Consultado em Novembro de 2012.

[16] Postgresql 8.3.23 Documentation. `http://www.postgresql.org/docs/8.3/static/high-availability.html`. Consultado em Dezembro de 2012.

[17] Postgresql 9.2.3 Documentation. `http://www.postgresql.org/docs/9.2/static/different-replication-solutions.html`. Consultado em Dezembro de 2012.

[18] Slony Overview. `http://wiki.postgresql.org/wiki/Slony`. Consultado em Novembro de 2012.

[19] SQL Triggers. `http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/triggers.htm#g1041767`. Consultado em Fevereiro de 2013.

[20] Symmetricds User Guide. `http://www.symmetricds.org/doc/3.3/html-single/user-guide.html`. Consultado em Dezembro de 2012.

[21] Timestamp Definition. `http://www.linfo.org/timestamp.html`. Consultado em Fevereiro de 2012.

[22] What is the Document Object Model. `http://www.w3.org/TR/DOM-Level-2-Core/introduction.html`. Consultado em Abril de 2013.

[23] Write-Ahead Logging (wal). `http://www.postgresql.org/docs/8.1/static/wal-intro.html`. Consultado em Dezembro de 2012.

[24] T.M. Connolly and C.E. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Number vol. 1 in International computer science series. Addison-Wesley, 2005.

[25] C. Coronel, S.A. Morris, and Peter Rob. *Database Systems: Design, Implementation, and Management*. Course Technology Ptr, 2012.

[26] R. Elmasri and S. Navathe. *Fundamentals of Database Systems, Sixth Edition*. Addison-Wesley, 2010.

[27] B. Kemme, R.J. Peris, and M. Patiño-Martínez. *Data Replication*. Synthesis Lectures on Data Management Series. Morgan & Claypool, 2010.

[28] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Computer science. Springer New York, 2011.

[29] S.K. Rahimi and F.S. Haug. *Distributed Database Management Systems: A Practical Approach*. Wiley, 2010.

[30] C. Ray. *Distributed Database Systems*. Pearson Education, 2009.

[31] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. Connect, learn, succeed. McGraw-Hill Education, 2010.