UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FRAMEWORK PARA REPLICAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS

GABRIEL FELIPE CRISTOFOLINI

BLUMENAU
2014

2014/2-09

GABRIEL FELIPE CRISTOFOLINI

FRAMEWORK PARA REPLICAÇÃO DE DADOS EM DISPOSITIVO MÓVEIS

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Jacques Robert Heckmann, Mestre – Orientador

BLUMENAU
2014

2014/2-09

FRAMEWORK PARA REPLICAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS

Por

GABRIEL FELIPE CRISTOFOLINI

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Jacques Robert Heckmann, Mestre – Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB
Membro:	Prof. Francisco Adell Péricas, Mestre – FURB

Blumenau, 08 de dezembro de 2014.

Dedico este trabalho a minha família e meus amigos por todo o apoio que me deram durante esta dura jornada.

AGRADECIMENTOS

A Deus...

À minha família pelo apoio incondicional.

Aos meus amigos por terem me motivado nos momentos difíceis.

Ao meu orientador por acreditar na conclusão deste trabalho.

Aos meus amigos Marcelo Gonzaga e Guilherme Furtado pelo auxilio na elaboração da monografia.

Os ideais que iluminaram o meu caminho são a bondade, a beleza e a verdade.

Albert Einstein

RESUMO

Este trabalho apresenta a especificação e implementação de um *framework* que permite a replicação de dados entre dispositivos móveis e a manutenção do estado global das base de dados envolvidas nesse processo. O trabalho tem como objetivo disponibilizar uma série de classes e interfaces para que o desenvolvedor replique seus dados da forma mais abstrata possível, focando em suas regras de negócio e não na forma que a replicação deve ser feita. O *framework* trabalha com a topologia estrela, ou seja, todo os dispositivos se conectam a um nó central para transmitir suas alterações. No trabalho para a comunicação entre os dispositivos e o nó central é utilizado *webservice* SOAP. Por fim foi feita uma aplicação exemplo, uma agenda, utilizando a tecnologia definida e implementada por este trabalho. Nesta agenda foram implementadas as operações básicas disponibilizadas pelo *framework*

Palavras-chave: Replicação. *Framework*. Dispositivos móveis.

ABSTRACT

This paper presents the specification and implementation of a framework that enables the replication of data between the mobile devices and the maintaining of the global state of database involved in this process. The paper aims to provide a series of classes and interfaces so the developer can replicate his data in the most abstract way, focusing in his business rules, but not in the method that the replication should be made. The framework works with the star topology, that means that all the devices connect to a central node to transmit its changes. For the communication between the devices and the central node is used SOAP webservice. Eventually was made an application, for example, an agenda that uses a technology defined and implemented by this project. The basic operations were implanted in this agenda by the framework.

Keywords: Replication. Framework. Mobile devices.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

CVS – *Concurrent Version System*

DER – Diagrama Entidade Relacionamento

DNS – *Domain Name System*

HTTP – *Hypertext Transfer Protocol*

JSON – *JavaScript Object Notation*

SGBD – Sistema Gerenciador de Banco de Dados

SOAP – *Simple Object Access Protocol*

UML – *Unified Modeling Language*

WSDL – *Web Services Description Language*

SUMÁRIO

1 INTRODUÇÃO

Atualmente os dispositivos móveis, tais como celulares e *tablets*, vêm ganhando cada vez mais usuários (OLIVEIRA, 2011), desde o advento das redes móveis, as quais permitiram que as pessoas movimentem-se livremente utilizando seus dispositivos e compartilhando recursos na rede através de meios de comunicação sem fio. O tráfego de dados por esse meio não para de crescer, pois, segundo observa Cisco (2014, tradução nossa), “O tráfego mensal de dados móveis irá passar de 16 hexabytes em 2018”. O poder computacional dos dispositivos móveis também vem evoluindo, de forma a possibilitar o desenvolvimento de aplicações cada vez mais complexas (LUCIO, 2011).

A sociedade atual está vivendo na era da informação, onde os dados possuem cada vez mais valor para as organizações. Segundo Gurgel (2006, p. 1) “Atualmente, a informação tem sido considerada o principal bem que uma organização possui”. Nesse contexto, em que a informação tem se tornado um bem de suma importância para as organizações, essas as estão captando e distribuindo de diversos modos. Os dispositivos móveis podem auxiliar nessa tarefa, capturando os dados e auxiliando no seu processamento.

Estes dados podem ser organizados através do uso de softwares de banco de dados. Em computação móvel, devido a sua natureza distribuída, existem diversas formas de implementar as aplicações de banco de dados. Segundo Monteiro (2005, p. 1), “A computação móvel possibilita o desenvolvimento de novas e sofisticadas aplicações em banco de dados”.

Devido a esta natureza distribuída dos bancos de dados móveis, a replicação de dados torna-se necessária para aumentar a disponibilidade do sistema e melhorar o seu desempenho (MONTEIRO, 2005). Uma demanda inerente à replicação de dados é a sincronização dos mesmos entre os dispositivos de forma a manter os dados consistentes entre os nós sincronizados. Segundo Silva (2014, p. 2) “com o avanço das pesquisas na área de sistemas distribuídos, novos modelos para manter o estado global de um sistema de bases de dados distribuídos estão sendo propostos”. Grande parte das soluções existentes para a sincronização de dados é baseada nas plataformas *desktop*, porém para os dispositivos móveis não existe uma grande gama de soluções implementada. A falta de ferramentas ou de *frameworks* para realizar a sincronização de dados em ambientes móveis acaba fazendo com que o desenvolvedor tenha que desenvolver esta porção do software, manualmente.

Um aspecto importante e que deve ser considerado dentro da replicação de dados, é quais cópias terão permissão de gravar em quais cópias. Sobre esse aspecto existem duas abordagens. A primeira consiste de sistemas em que existe um único mestre (*single-master* ou

master-slave) em que são feitas as alterações e as mesmas são repassadas para os demais, que são denominados escravos. A segunda abordagem é a *multi-master*, na qual múltiplas réplicas submetem suas informações (MACEDO et al., 2008). Existem diversas formas para que dispositivos se interconectem dentro de um sistema distribuído (MACEDO et al., 2008). Uma delas é a topologia estrela, na qual existe um dispositivo central e os demais se conectam nele para se tornarem parte do sistema.

Diante do exposto, a especificação e desenvolvimento de um *framework* para replicação de dados é demonstrado neste trabalho. O principal intuito deste é replicar dados e manter as bases de dados consistentes e sincronizadas.

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver um *framework* para dispositivos móveis que facilite a replicação e a manutenção do estado global dos estados dentro de um sistema distribuído móvel.

Os objetivos específicos deste trabalho são:

a) implementar um algoritmo no *framework* de manutenção do estado global dos dados dentro das diferentes bases de dados de um sistema distribuído;
b) fornecer uma interface com métodos que façam a busca e a replicação de dados entre os dispositivos distribuídos;
c) fornecer suporte para a formação de sistemas distribuídos em um modelo *multi-master*, utilizando a topologia em estrela;
d) validar o uso do *framework* proposto através do desenvolvimento de uma aplicação distribuída simplificada.

1.2 ESTRUTURA

O trabalho foi dividido em quatro capítulos. No primeiro, consta uma introdução sobre o assunto. O segundo capítulo apresenta a fundamentação teórica necessária para o desenvolvimento do aplicativo. O terceiro capítulo traz informações sobre o desenvolvimento do trabalho, seus requisitos, sua especificação, sua implementação, operacionalidade e resultados. No quarto capítulo são relatadas as conclusões do trabalho e suas possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

A seguir será demonstrada a fundamentação teórica que foi utilizada para a elaboração deste trabalho. Na seção 2.1 é apresentado o assunto de replicação de dados para dispositivos móveis; na 2.2 é apresentada a tecnologia de replicação de dados; na 2.3 é apresentada a tecnologia SqlLite; na 2.4 é apresentado o sistema operacional Android e sua arquitetura; na 2.5 a tecnologia CVS; na 2.6 a tecnologia JSON; e por fim na seção 2.7 os trabalhos correlatos.

2.1 REPLICAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS

A replicação de dados é um processo que consiste em manter várias cópias de um mesmo dado, de forma a aumentar a disponibilidade do sistema e o seu desempenho (MONTEIRO et al., 2006). O aumento de disponibilidade acontece devido à existência de várias cópias do mesmo dado. Em caso de falha na conexão, a informação estará descentralizada. Já o aumento de desempenho ocorre devido ao fato de as cópias de dados estarem em diversos locais, permitindo assim que o sistema acesse o dado que está mais próximo, reduzindo o tempo de latência para acesso aos dados.

Existem duas abordagens para a replicação de dados: as otimistas e as pessimistas.

> Os algoritmos pessimistas coordenam as réplicas de forma síncrona durante os acessos e bloqueiam os outros usuários durante um *update*. Algoritmos otimistas permitem que os dados sejam acessados sem a necessidade de uma sincronização prévia, assumindo de forma "otimista" que eventuais problemas irão ocorrer muito raramente. (MACEDO et al.,2008, p. 3)

Estas duas abordagens acabam se diferenciando devido à forma que abordam o controle de concorrência. A técnica que obtém um melhor desempenho nos ambientes com baixa conectividade é a abordagem otimista, devido ao grande tempo de latência nesses ambientes (MACEDO et al., 2008). Qualquer sistema distribuído deve encontrar o equilíbrio entre disponibilidade e consistência (NODARI, 2007). Esse é o principal desafio dos algoritmos otimistas: gerenciar a concorrência de forma a manter as bases de dados consistentes. Alguns exemplos de utilização de algoritmos otimistas são as tecnologias DNS - *Domain Name System* ou no sistema de controle de versões CVS - *Concurrent Versions System* (NODARI, 2007). Uma das características que algoritmos otimistas normalmente implementam é a ordenação de comando. Devido à possibilidade de criação de comandos em locais independentes é necessária a ordenação para a possível detecção de conflitos e a eventual resolução dos conflitos detectados.

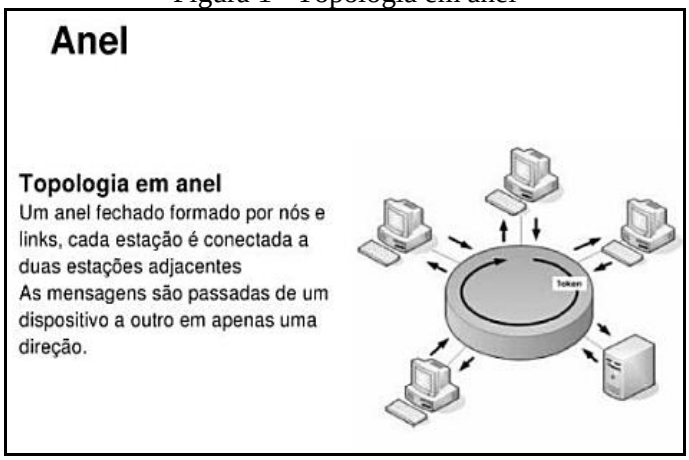
Referente a possíveis tipos de anomalias ou conflitos que podem ser detectados pode-se destacar alguns tipos que são recorrentes em execução de transações em sistemas

distribuídos. O primeiro deles é o de leitura suja, também conhecido como conflito de leitura de dados não efetivados. Este tipo de conflito ocorre quando uma transação t1 executa ações utilizando objetos que estão sendo alterados por outra transação t2, porém t2 ainda não foi efetivada e por tanto t1 está lendo objetos desatualizados (GEHRKE; RAMAKRISHNAN, 2008) O segundo tipo de conflito é referente à execução de leituras não repetíveis. basicamente o que ocorre nesse tipo de conflito é que 2 transações leem o mesmo recurso e executam a mesma operação, porém a efetivação de uma delas não permite que a outra seja efetivada. Um exemplo disso ocorre quando duas transações t1 e t2 leem o valor 1 de uma variável inteira sem sinal A e as duas decrementam este valor; a efetivação de uma das operações vai impedir a execução da outra (GEHRKE; RAMAKRISHNAN, 2008). Por fim, existe o conflito de sobrescrita de dados não efetivados: esse conflito é relativo a transações que sobrescrevem objetos de uso em comum. A transação t1 sobrescreve o valor de um objeto A que já foi alterado por uma transação t2, entretanto não levou em consideração o resultado da transação t2 (GEHRKE; RAMAKRISHNAN, 2008).

Na replicação de dados um aspecto a ser considerado é quais são as cópias que possuem permissão para gravar nas demais. Sistemas com um único mestre são chamados de *master-slave*. Nesse caso apenas uma das réplicas sofre alteração e a repassa para as outras. Sistemas com vários mestres são denominados *multi-master*. Nestes sistemas todos podem gravar em todos os outros, o que faz com que ocorra maior disponibilidade de dados, entretanto, aumenta o custo computacional para o gerenciamento dos dados (MACEDO et al., 2008).

O modo como as bases de dados se interconectam deve ser observado. Existem várias topologias para interconexão de dispositivos. Entre elas estão a topologia em anel, barramento e em estrela. Na topologia em anel cada dispositivo tem duas conexões, de forma que o último dispositivo que se conecta à rede possui uma conexão para o primeiro, formando um anel. A Figura 1 representa a estrutura de rede em anel.

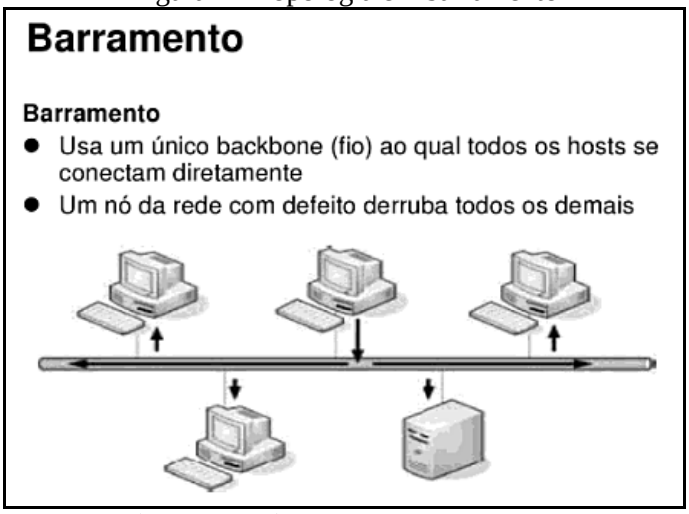
Figura 1 - Topologia em anel



Fonte: Ross (2008, p. 19).

Na topologia em barramento os dispositivos possuem duas conexões, entretanto o último não possui conexão com o primeiro. Nesta topologia caso um nó seja desconectado toda a rede é prejudicada. A Figura 2 representa a topologia citada.

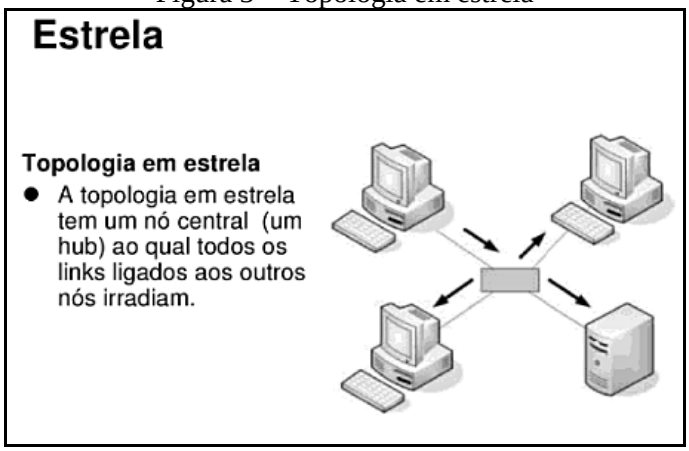
Figura 2 - Topologia em barramento



Fonte: Ross (2008, p. 19).

Por fim, tem-se a topologia em estrela na qual existe um nodo central que centraliza as requisições e as distribui, onde todos os dispositivos da rede se conectam ao nodo central. A Figura 3, exemplifica o funcionamento da topologia em estrela.

Figura 3 - Topologia em estrela



Fonte: Ross (2008, p. 20).

2.2 BANCO DE DADOS MÓVEIS

Bancos de dados são comumente utilizados em sistemas para guardar dados. Segundo Date (1991, p. 5) “O sistema de banco de dados é basicamente um sistema de manutenção de registros por computador – ou seja, um sistema cujo objetivo global é manter as informações e torná-las disponíveis quando solicitadas”. Já a computação móvel consiste em dispositivos heterogêneos conectados a uma rede sem fio. Isso introduziu uma nova realidade: o acesso aos dados em qualquer lugar que se esteja (RAINONE, 2014). Diante deste cenário, os bancos de dados precisaram evoluir de forma a atender esta demanda. Para atender essas novas necessidades, surgiram novos desafios.

Um dos desafios é a disseminação de dados, que consiste na entrega de dados para clientes a partir de um conjunto de produtores (RAINONE, 2014). No ambiente móvel são necessários mecanismos para garantir que a entrega de dados seja realizada com sucesso bem como mecanismos para controle das alterações, para que elas, mesmo em caso de desconexão, sejam posteriormente replicadas.

Outro ponto a ser observado é a heterogeneidade de ambientes. Isso faz com que os dados sejam armazenados de diferentes formas, dentro das diversas bases de dados que compõe o sistema distribuído. Deste modo, é necessária a criação de formas de transmissão de dados entre dispositivos, de modo que todos os dispositivos consigam compreender e armazenar as informações que estão sendo replicadas.

2.3 SQLITE

O SQLite é uma biblioteca desenvolvida em C ANSI, podendo ser utilizada em diversas plataformas (GONÇALVES, 2013). Ela é nativa para a plataforma Android, a qual

oferece uma série de bibliotecas para utilização dele nas aplicações desenvolvidas para a plataforma. O SQLite cria um banco de dados que é guardado localmente junto com a aplicação, em um arquivo com a extensão “.db” que pode atingir até 2 terabytes de tamanho.

Segundo Gonçalves (2013) algumas das características do SQLite são:

a) suporta o uso de transações;
b) não oferece integridade referencial (chaves estrangeiras);
c) SQLite dá suporte a maior parte dos comandos SQL 92.

O uso do SQLlite é recomendado para aplicações locais e que não possuem alta concorrência.

2.4 SISTEMA OPERACIONAL ANDROID

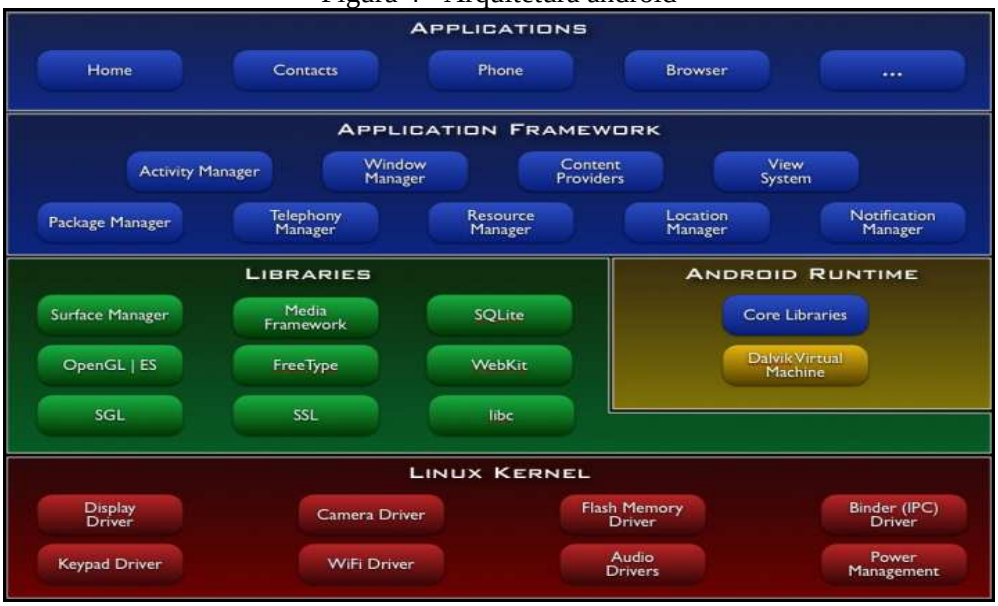
O Android é um sistema operacional desenvolvido pela Google e que é baseado em Linux. Sua primeira versão foi distribuída em 2007 e atualmente é mantido pela Open Handset Alliance (OHA), um grupo que congrega várias empresas de tecnologia, no qual a própria Google é um dos membros.

Segundo Hermann (2013) “A plataforma Android foi construída de forma a permitir que os desenvolvedores possam criar aplicações capazes de tirar o melhor proveito de tudo que um dispositivo móvel pode oferecer”. Tendo em vista essa facilidade, a plataforma Android despertou interesse de grandes corporações, as quais passaram a fabricar seus dispositivos com esse sistema operacional embutido. A plataforma Android é suportada por inúmeros dispositivos, desde celulares até *tablets*, e possui ampla penetração no mercado.

O Android é dividido em quatro camadas: *kernel* Linux, bibliotecas, *framework* de aplicação e aplicações (HERMANN, 2013).

A Figura 4 representa a arquitetura Android e suas diversas camadas. Como pode ser observado, o *kernel* do android é Linux e acima deste tem-se as camadas que provêm recursos e abstração para o desenvolvimento de aplicativos na plataforma.

Figura 4 - Arquitetura android



Fonte: Android developers (2014).

2.5 CONCURRENT VERSION SYSTEM (CVS)

O CVS é um versionador de código aberto que permite o trabalho em diversas versões de arquivos organizados em um diretório, mantendo seus históricos de alterações. Ele é especialmente útil para elaboração de documentos colaborativos, utilizando uma arquitetura cliente-servidor (FIGUEIREDO NETO, 2010).

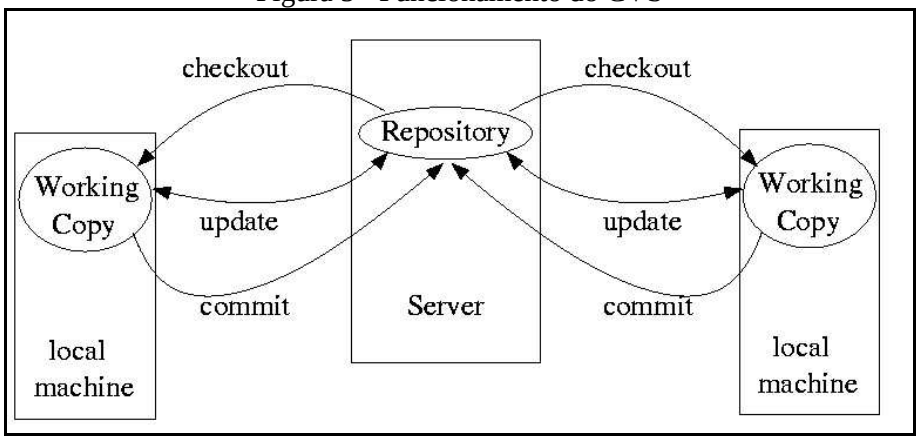
O seu funcionamento ocorre da seguinte forma. Um projeto controlado por CVS pode ser formado por diversos módulos, o que permite que os usuários editem módulos de maneira concorrente. Ao confirmar alterações e enviar os dados para o servidor, o servidor faz um *merge* das mesmas. Caso ocorra um conflito é necessário a intervenção humana para resolver a inconsistência. Caso a alteração seja bem sucedida, é incrementado o número de versão de cada arquivo envolvido (FIGUEIREDO NETO, 2010).

No CVS é possível que um cliente compare versões distintas de um arquivo, ou peça o histórico de versões, ou baixe alguma versão em especifico. Esta tecnologia também permite manter diversos estados de versão, ou seja, uma versão de software pode estar destinada à manutenção de software, enquanto outra simultaneamente pode estar sendo utilizada para implementação de novas funcionalidades (FIGUEIREDO NETO, 2010). O CVS utiliza 3 comandos básicos para a sincronização de seus arquivos: *commit*, *update* e *checkout*.

A Figura 5 representa o fluxo do CVS e como os arquivos são replicados entre o servidor e os clientes durante a execução sistema. O comando *checkout* refere-se à carga inicial de dados; o *update* consiste na atualização de uma versão já existente no cliente

enquanto o *commit* trata-se do envio das alterações ocorridas localmente no cliente para o servidor.

Figura 5 - Funcionamento do CVS



Fonte: Thecafetchno (2013).

2.6 JSON

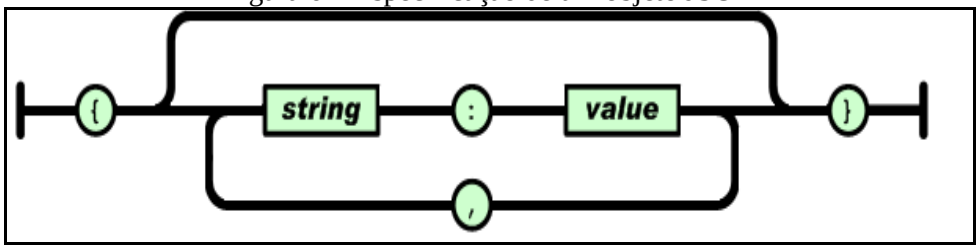
O JSON (*JavaScript Object Notation*) é uma formatação leve para trocas de dados. Ela é de fácil compreensão e totalmente independente de linguagem (JSON, 2010). O JSON possui estruturas que permitem representar objetos e listas.

> JSON está constituído em duas estruturas: uma coleção de pares nome/valor. Em várias linguagens, isto é caracterizado como um object, record, struct, dicionário, hash table, keyed list, ou arrays associativas. Uma lista ordenada de valores. Na maioria das linguagens, isto é caracterizado como uma array, vetor, lista ou sequência. (JSON, 2010)

As estruturas em JSON são representadas utilizando-se uma combinação de valores com os separadores, assim podendo representar estruturas de armazenamento de dados. Abaixo serão demostradas as diferentes estruturas que a notação JSON pode representar.

Na notação JSON, um objeto é um conjunto desordenado de pares nome/valor. Um objeto começa com chave de abertura e termina com chave de fechamento. Cada nome é seguido por dois pontos e os pares nome/valor são seguidos por vírgula (JSON, 2010). A Figura 6 exemplifica a estrutura.

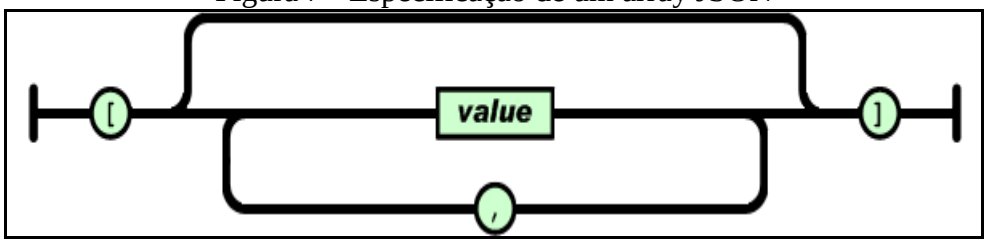
Figura 6 - Especificação de um objeto JSON



Fonte: Json (2010).

Um *array* é uma coleção de valores ordenados. O *array* começa com um colchete de abertura e termina com colchete de fechamento. Os valores são separados por vírgula (JSON, 2010). A Figura 7 exemplifica a estrutura.

Figura 7 - Especificação de um array JSON



Fonte: Json (2010)

A expressão *value* das figuras acima pode representar uma *String*, um número, um valor lógico, um objeto ou um outro *array*.

2.7 TRABALHOS CORRELATOS

Os três trabalhos a seguir demonstram a replicação de dados aplicada a diferentes ambientes. No primeiro trabalho apresentado ela é aplicada no ambiente móvel e nos outros dois ao ambiente convencional.

2.7.1 Um mecanismo para a consistência de dados replicados em computação móvel

Esse trabalho apresenta um novo protocolo para a replicação e a consistência de dados e a convergência de diversas réplicas das bases de dados para um estado final convergente (MONTEIRO et al., 2006). Basicamente o trabalho propõe um protocolo (*peer-to-peer*) em que os dispositivos servidores e móveis se conectem de forma intermitente. A replicação de dados ocorre entre todos os servidores. Dessa forma as aplicações conseguem se comunicar com qualquer servidor que consigam acessar e fazer as operações necessárias. Nesse protocolo réplicas que estão em servidores distintos podem conter conteúdos distintos em um determinado instante no tempo, entretanto as réplicas convergem para atingir uma consistência eventual.

O protocolo proposto por esse trabalho foi projetado para trabalhar com bases de dados replicadas e com dispositivos móveis fracamente conectados. O protocolo utiliza a arquitetura *peer-to-peer* onde servidores e clientes são conectados de modo intermitentes uns aos outros. Segundo o trabalho, para maximizar a performance e a disponibilidade de dados, cada banco de dados é integralmente replicado em um conjunto de servidores. Ainda segundo Monteiro et al. (2006) “Quando uma atualização ocorre na base de dados local, se a mesma fosse um

mestre em ambiente síncrono, uma conexão seria imediatamente realizada nas bases secundárias e as informações sobre a atualização seriam a elas transmitidas".

2.7.2 Replicação assíncrona entre bancos de dados médicos distribuídos

Trata-se de uma *engine* para fazer replicação de dados entre bancos de dados distribuídos, especificamente banco de dados voltados a telemedicina (MACEDO et al., 2008). Nesse contexto a replicação de dados torna-se importante para que exames realizados em regiões diferentes consigam ser acessados por médicos.

Nesse trabalho abordou-se conceitos de replicação parcial e técnicas de replicação de base de dados otimista, devido à grande distância das bases de dados situadas em todo o país. No trabalho em questão utilizou-se internamente a Postgres Replication (PGR) uma ferramenta que permite que sejam especificados subconjuntos de tabelas das bases de dados a serem monitoradas (MACEDO et al., 2008).

2.7.3 Sincronização de bancos de dados distribuídos utilizando s*napshot isolation*

O artigo apresenta como é o comportamento de um algoritmo de replicação de dados utilizando o conceito de *snapshot isolation* em um ambiente distribuído. O conceito de *snapshot isolation* é que as transações possuam imagens do banco de dados, para que, em caso de falha, restaurem a imagem anterior da tabela em questão (SILVA, 2014). Ainda segundo Silva (2014, p.2) "Um sistema distribuído de bases de dados, não deixa de ser visto sobre alguns aspectos como um SGBD local, ou seja, este deve respeitar a principal propriedade que define um SGBD, conhecida como ACID". Nesse trabalho demonstra-se como seria o comportamento de um algoritmo de replicação que utiliza os *snapshots* para garantir a ACID. A ACID consiste em 4 propriedades que garantem a consistência de uma transação são elas: atomicidade, consistência, isolamento e durabilidade.

2.8 PRODUTOS CORRELATOS

Nesta seção serão apresentados alguns produtos comerciais correlatos ao trabalho apresentado.

2.8.1 OBJECTMMRS – Replicação de Banco de Dados

É uma ferramenta desenvolvida pela empresa sistemas object para replicação de dados que suporta projetos de replicação unidirecional (*master-slave*) ou bidirecional (*multi-master*)

de banco de dados desenvolvida para que as organizações não fiquem mais dependentes de um servidor central único de banco de dados.

Os bancos de dados suportados por essa ferramenta são:

a) Oracle (8.x, 9.x, 10.x, 11, XE);
b) DB2 IBM (8.x, 9.x, Express);
c) PostgreSQL (7.4, 8.x,9.x);
d) Firebird (1.5, 2.x);
e) MS SQLServer 7, 2000, 2005, 2008, Express;
f) Sybase 10 ou superior;
g) Informix 11.5 ou superior;
h) MySQL 5 ou superior;
i) SQLite.

2.8.2 Informatica – Data Replication

É um software de replicação transacional heterogênea em tempo real que é altamente dimensionável, confiável e fácil de configurar. Utiliza a captura de dados alterados com base em registros para minimizar o impacto nos sistemas-fonte. É fornecida a captura de alterações transacionais em tempo real e aplicação dessas a outros bancos de dados, data warehouses ou arquivos de texto de formato portátil. Permite o registro de dados transacionais alterados em um banco de dados, sistema operacional e ambiente de hardware heterogêneos, eliminando a necessidade de uma janela de lote enquanto fornece e carrega informações em tempo real para armazenamentos de dados operacionais.

Além disso, o software oferece um *console* para gerenciamento da arquitetura envolvida na replicação de dados.

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas do desenvolvimento da aplicação. A seção 3.1 traz os requisitos principais a serem abordados pelo trabalho. Na seção 3.2 consta a especificação, com modelos e diagramas UML. A implementação, com técnicas, ferramentas utilizadas e detalhes sobre a operacionalidade da implementação estão na seção 3.3. A seção 3.4 mostra os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Seguem os requisitos que são implementados por esse trabalho.

a) o *framework* deverá disponibilizar um método de inclusão de dados (Requisito Funcional - RF);
b) o *framework* deverá disponibilizar um método para atualizar dados (RF);
c) o *framework* deverá disponibilizar um método de remoção de dados (RF);
d) o *framework* deverá disponibilizar para os dispositivos um método para que eles mesmos sincronizem sua base de dados (RF);
e) deverá ser implementado um serviço para sincronização dos dados e gerenciamento de conflitos (RF);
f) o nodo central deverá possuir um console que permita verificar os dispositivos inscritos;(RF)
g) deverá ser disponibilizado um *webservice* para os dispositivos móveis incluírem-se dentro do sistema (RF);
h) o *framework* e suas dependências deverão ser implementadas em Java (Requisito Não Funcional - RNF);
i) o *framework* deverá suportar a plataforma Android (RNF).

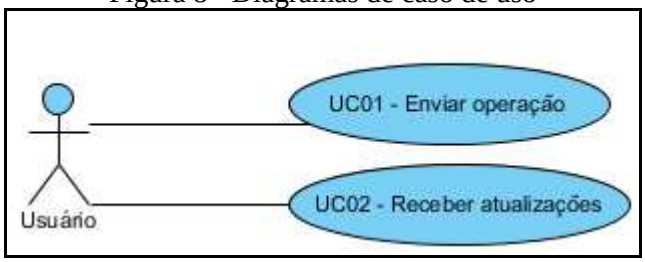
3.2 ESPECIFICAÇÃO

Esta seção mostra os diagramas de classes, diagramas de caso de uso e atividade. Para o desenvolvimento destes diagramas foi usado o Visual Paradigm Standart Edition versão 11.1.

3.2.1 Diagrama de caso de uso

Nesta seção são descritos os casos de uso do *framework*. Por se tratar de um *framework*, toda ação será disparada por um usuário, que está utilizando um sistema que utiliza o *framework* para replicação dos dados. A figura 8 exibe o diagrama de caso de uso.

Figura 8 - Diagramas de caso de uso



O caso de uso UC01 é executado por um usuário que gera uma ação que resulta em um envio de operação para o *framework*. Este caso de uso está descrito no Quadro 1.

Quadro 1- UC01 Enviar operação

UC01 Enviar operação	
Descrição	Este caso de uso tem como objetivo enviar uma operação para o servidor e verificar se a mesma pode ser executada, utilizando as regras do *framework*.
Ator	Usuário
Pré-condição	Os bancos de dados envolvidos devem conter as tabelas básicas para o funcionamento do *framework*.
Pós-condição	O nó ponta e o central são sincronizados
Cenário principal	1. Usuário executa uma ação que gera uma operação no sistema. 2. O sistema, através do *framework*, solicita uma ação no banco de dados servidor. 3. A ação é incluída em uma fila de envio. 4. É verificada a existência de uma conexão do dispositivo com alguma rede. 5. Caso exista a conexão todas as operações pendentes são enviadas. 6. O *framework* valida cada operação recebida e executa a mesma. 7. É executado o caso de UC02 para atualizar a base de dados cliente com os dados do servidor.
Cenário alternativo 2	1. Após o passo 3 é verificado que não existe conexão. 2. O processo é finalizado e a ação continua pendente na fila de envio.
Cenário alternativo 3	1. Após o passo 5 do cenário principal é verificado que o objeto enviado está desatualizado 2. É disparado o evento para resolução de conflitos no cliente. 3. É retornada uma exceção própria do *framework*, juntamente com o objeto que foi enviado e o que está no servidor. 4. É executado o caso de UC02.
Cenário alternativo 4	1. Após o passo 5 do cenário principal é verificado que a operação é de alteração ou exclusão. 2. O registro não existe mais na base de dados 3. É retornada uma exceção do *framework* juntamente com o objeto enviado 4. É executado o caso de uso UC02.
Cenário alternativo	1. Após o passo 6 do cenário principal a operação não pôde ser executada com sucesso no banco de dados do servidor. 2. É retornada a exceção do banco de dados para o cliente juntamente com o objeto que foi enviado. 3. É disparado o evento para resolução de conflitos no cliente. 4. É executado o caso de uso UC02.

O caso de UC02 representa o envio de uma solicitação para recebimento dos dados do nó central e a aplicação deles em um nó na ponta da estrela. O quadro 2 apresenta o detalhamento do mesmo.

Quadro 2 – UC02 Receber atualizações

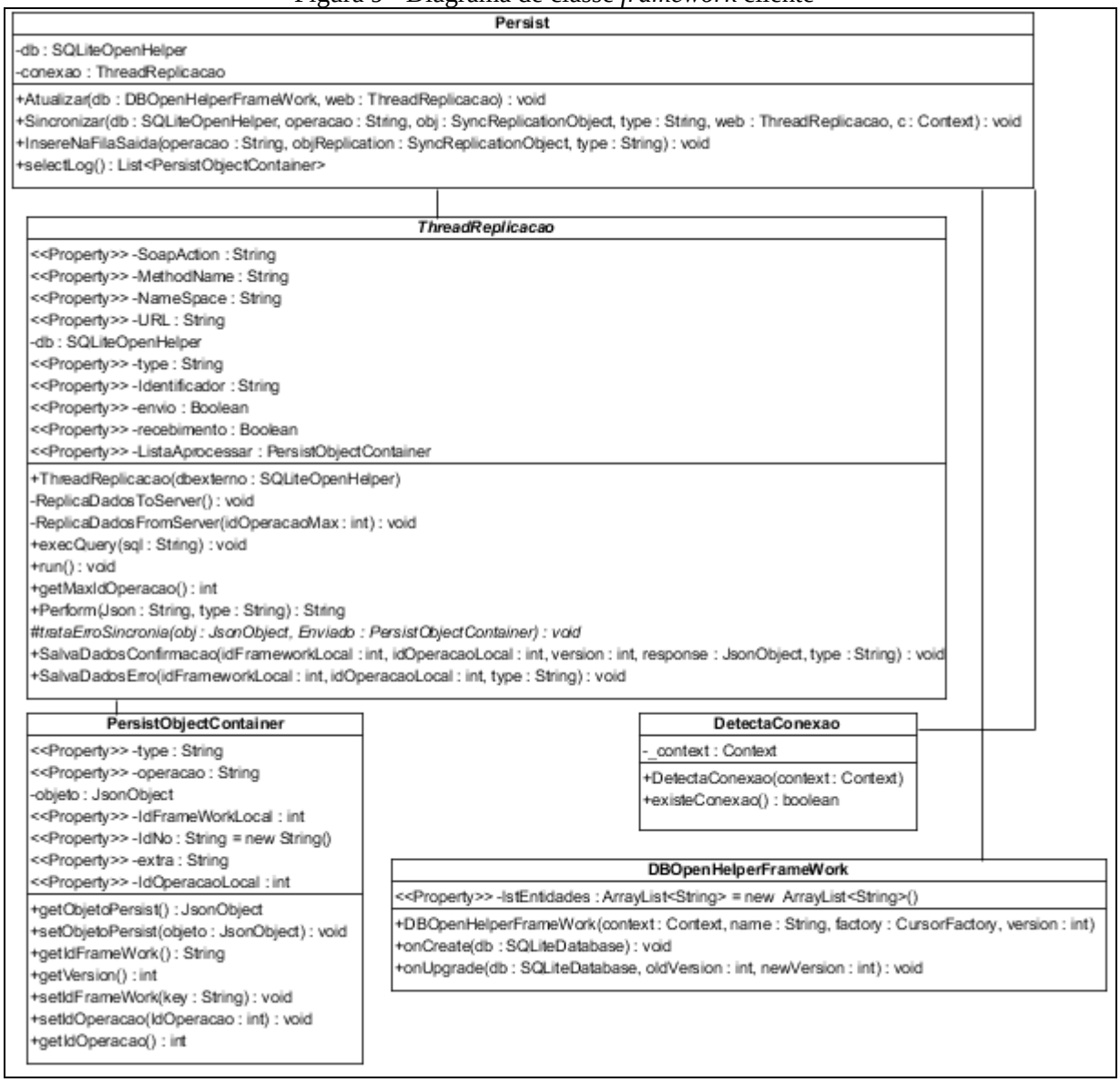
UC02 Receber atualizações	
Descrição	Esse caso de uso tem como objetivo que todas as operações executadas no nó central sejam replicadas para o banco de dados local do cliente
Ator	Usuário
Pré-condição	O banco de dados deve conter as tabelas básicas para o funcionamento do *framework*.
Pós-condição	O banco de dados cliente atualizado com as mesmas operações realizada no banco de dados servidor
Cenário principal	1. Usuário gera uma ação que necessite a sincronização dos dados do dispositivo com os do nó central 2. O sistema faz a requisição através do *framework* das operações que foram executadas no nó central. 3. O *framework* verifica se existe conexão do dispositivo com alguma rede externa. 4. Caso exista uma conexão a requisição é enviada. 5. No nó central são recuperadas todas as operações que foram executadas pelos demais dispositivos no banco de dados central e ainda não foram replicadas no nó que solicitou a replicação. 6. As operações recuperadas são executadas no cliente pelo *framework*.
Cenário alternativo 1	1. Após o passo 3 do cenário principal é verificado que a conexão não existe. 2. A solicitação é abandonada.

3.2.2 Diagramas de classes

Nesta seção serão exibidos 4 diagramas de classes: um para o *framework* cliente, um para o uso do *framework* no aplicativo cliente, outro para o *framework* servidor e, por fim, um sobre o uso do *framework* na parte servidor.

Na figura 9 é exibido o diagrama de classes do *framework* cliente.

Figura 9 - Diagrama de classe *framework* cliente

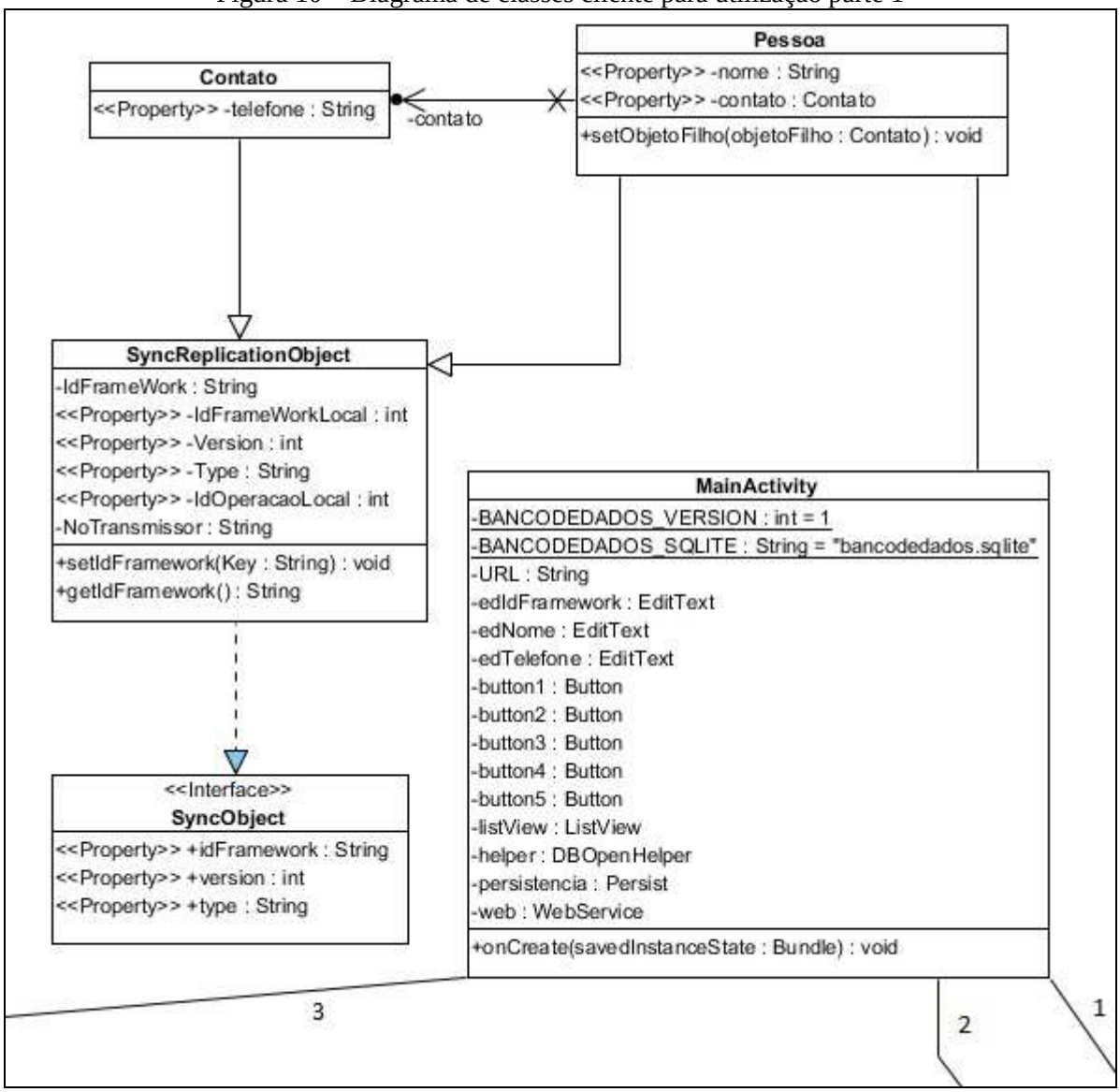


O diagrama de classes representa as entidades envolvidas para o envio e recebimento das operações do cliente. A classe `Persist` é a responsável pela recepção dos parâmetros da aplicação cliente através dos métodos `Atualizar` e `Sincronizar` No método `Sincronizar` é inserida a operação em uma fila, através do método `InsereNaFilaSaidaOperacao`. Após isso, essa classe utiliza o método `existeConexao` da classe `DetectarConexao` para verificar se existe alguma conexão com a internet, caso a mesma exista, é disparada a *thread* de replicação passando os parâmetros necessário para o envio dos dados e execução dos comandos recebidos do nó central. No caso do método `atualizar` a parte operacional é a mesma. A principal diferença é que não é enviada nenhuma operação para o nó central, somente é recebido o retorno e atualizada a base local.

A classe `ThreadReplicacao` é responsável pelo envio de dados e execução dos comandos recebidos do nó central. O método `run` da *thread* primeiro envia os dados que foram colocados para envio pela classe `persist`, caso o servidor retorne algum erro é disparado o método `trataErroSincronizacao`. Em um segundo momento é enviada uma requisição para que sejam retornadas todas as operações que foram feitas no servidor por outros dispositivos. A classe `ThreadReplicacao` executa-as no banco de dados local.

As figuras 10 e 11 representam a utilização do *framework* em um aplicativo cliente.

Figura 10 – Diagrama de classes cliente para utilização parte 1

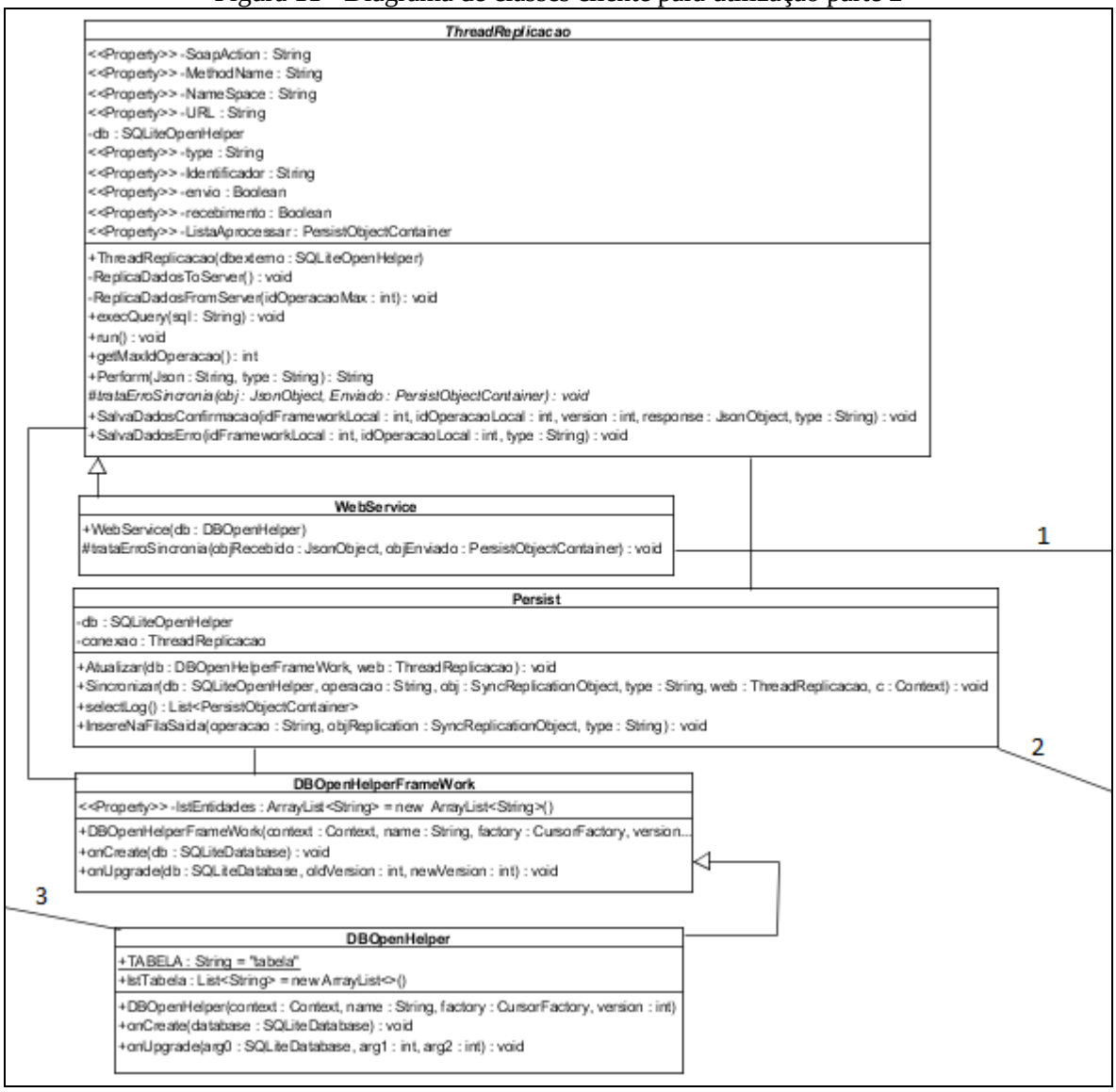


A Figura 10 representa a relação das classes que terão seus objetos persistidos. Todos devem herdar da classe `SyncReplicationObject` para assim serem persistidos no servidor. Esta classe contém as propriedades básicas para o funcionamento do *framework*.

A figura 11 apresenta as classes exemplos `DBOpenHelper` que herda da classe `DbOpenHelperFramework`. Essa classe é a responsável pela persistência na base local.

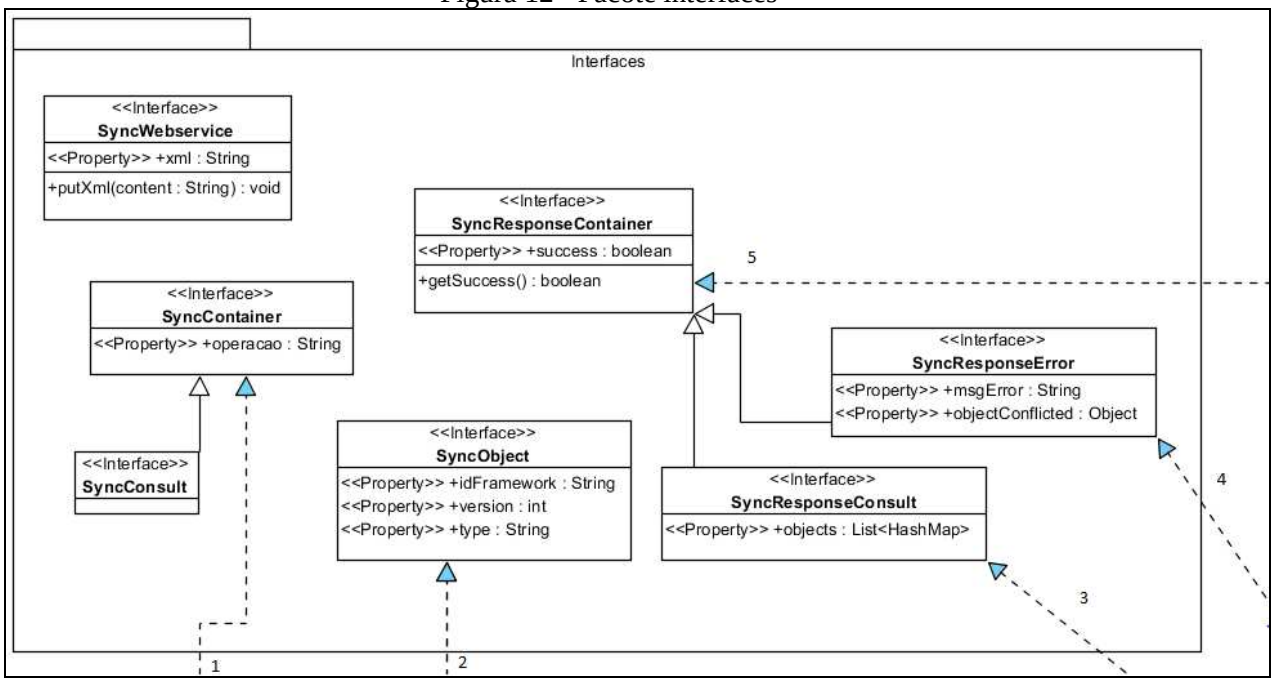
Enquanto isso, a classe exemplo `WebService` herda de `ThreadReplicacao` e é a responsável pelo tratamento de erros de sincronia através do método `trataErroSincronizacao`.

Figura 11 - Diagrama de classes cliente para utilização parte 2



A figura 12 exibe as interfaces criadas para a comunicação entre o cliente e o servidor. Essas interfaces são utilizadas na conversão do JSON para objetos tanto na parte cliente quanto na parte servidor. A mesma pode ser separada em 4 grupos, as que derivam de `SyncResponseContainer` que são as responsáveis pela transmissão dos retornos do servidor para o cliente, as que derivam de `SyncContainer` que são responsáveis por encapsular as operações a serem realizadas no servidor, a classe `SyncObject` responsável por definir a maneira que um objeto replicável deve ser transmitido e pôr fim a `SyncWebservice` responsável pela definição de como um WebService deve ser implementado.

Figura 12 - Pacote interfaces



As figuras 13 e 14 mostram objetos que implementam as interfaces de comunicação do pacote de interfaces. Esses objetos são utilizados tanto na parte cliente quanto no servidor. A única exceção a esta regra é a classe `SyncReplication` que está na figura 13. Ela é a responsável pela separação inicial das operações enviadas para o servidor.

Figura 13 – Diagrama de classes de comunicação para comunicação cliente

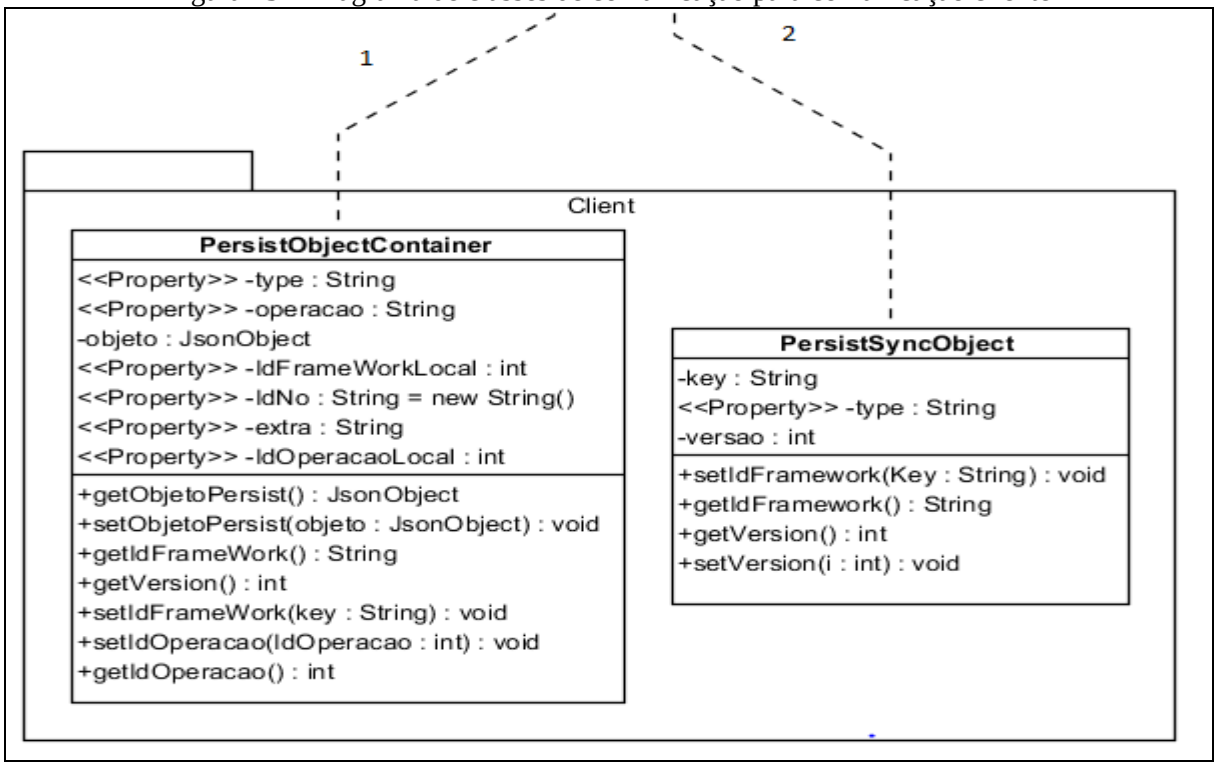
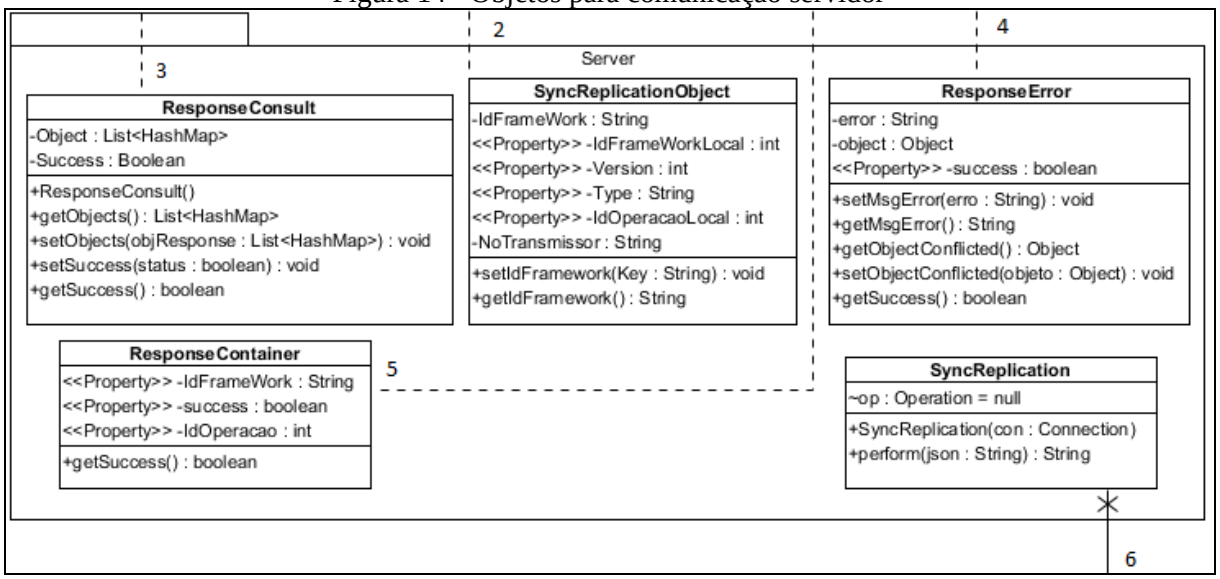


Figura 14 - Objetos para comunicação servidor



As figuras 15 e 16 representam os pacotes que aplicam as regras de negócios na parte server. Além disso, existem duas classes de funções úteis para a conversão de operações JSON para sql: `Utils` e `BaseClient`.

Figura 15 - Pacote ServerRules

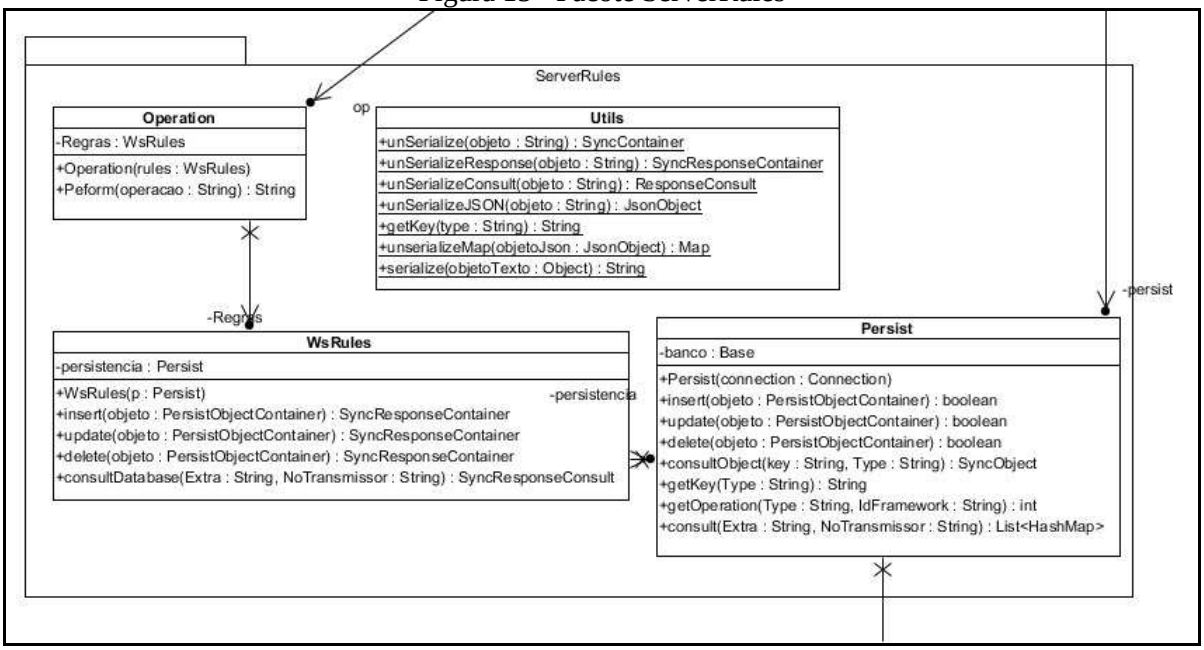
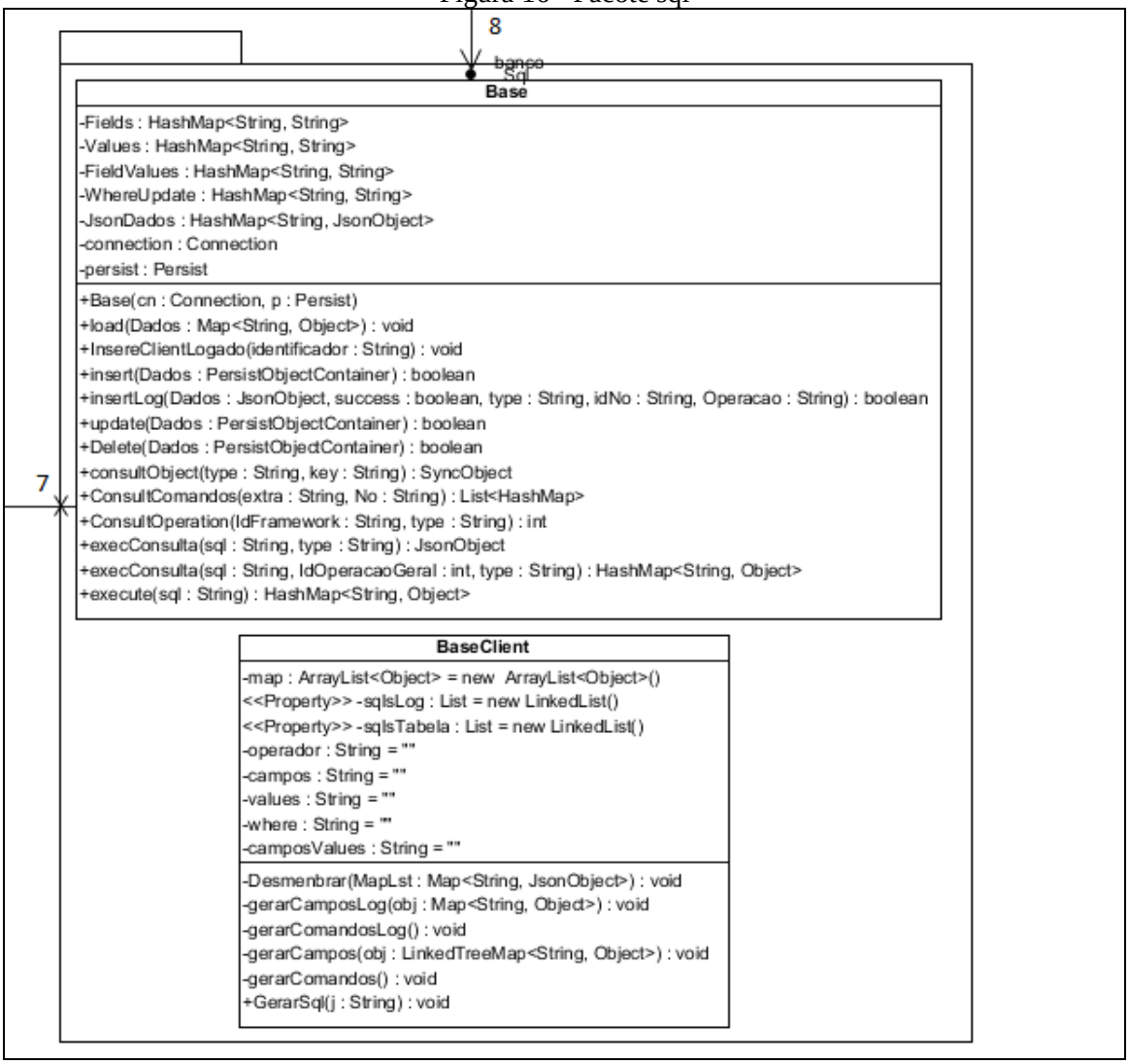
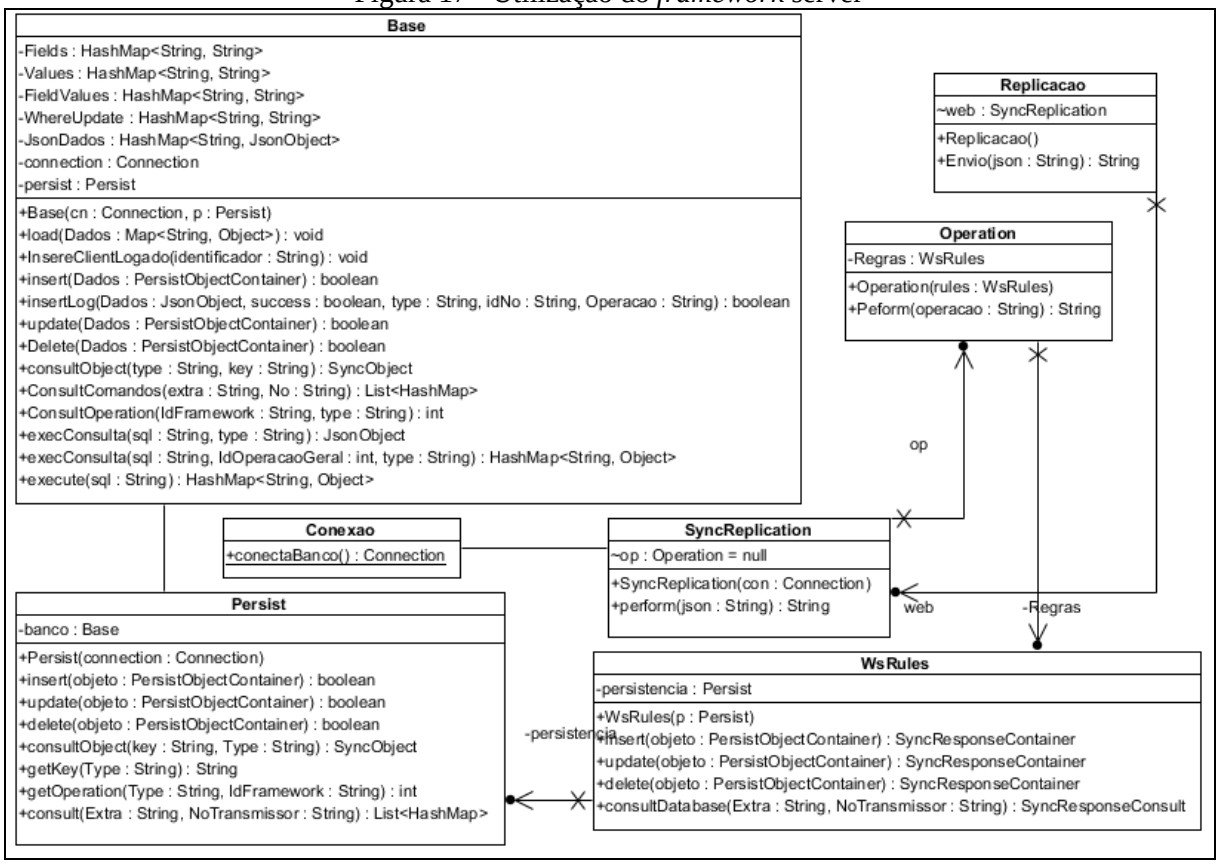


Figura 16 - Pacote sql



Por fim, na figura 17, é demonstrada a criação da classe exemplo de utilização do *framework*, a qual é chamada de `replicacao` e a sua relação com as classes do *framework*.

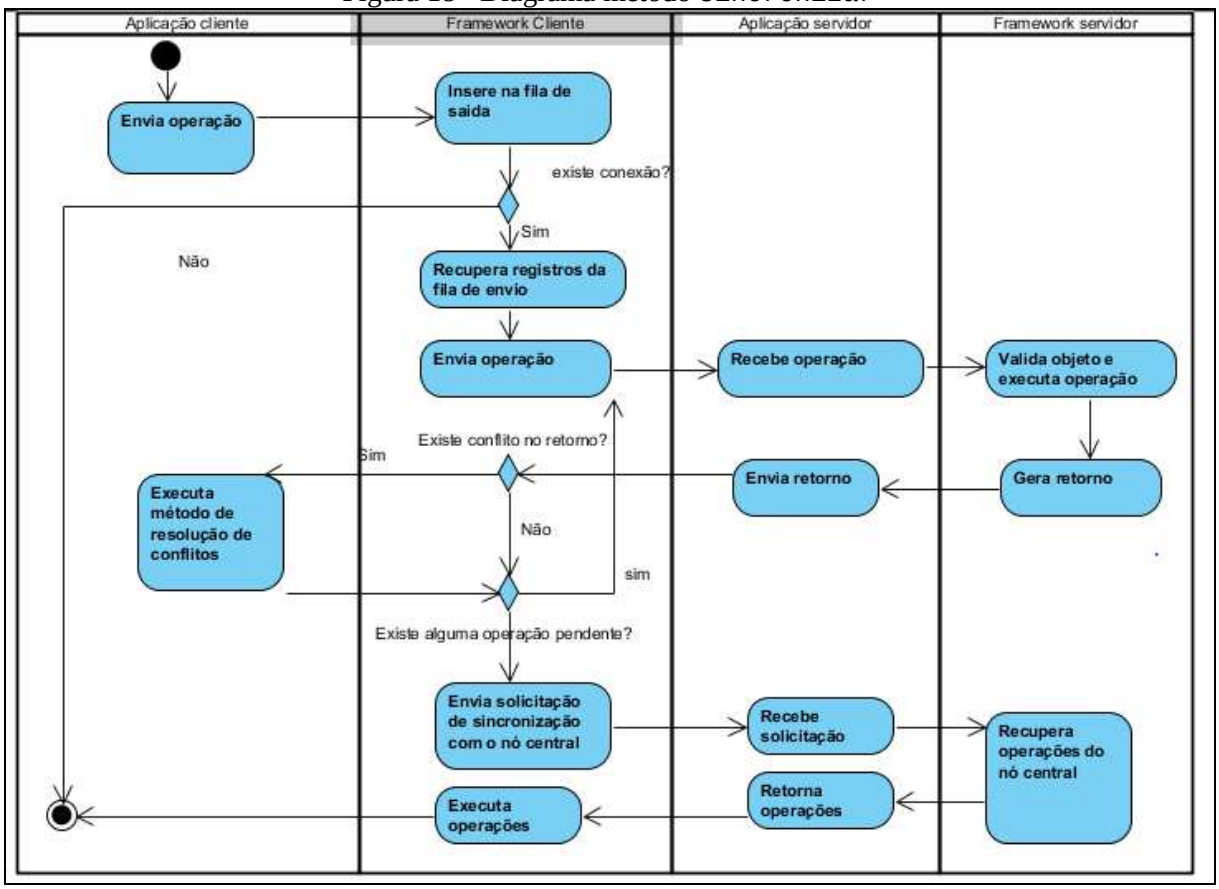
Figura 17 - Utilização do *framework* server



3.2.3 Diagramas de atividades

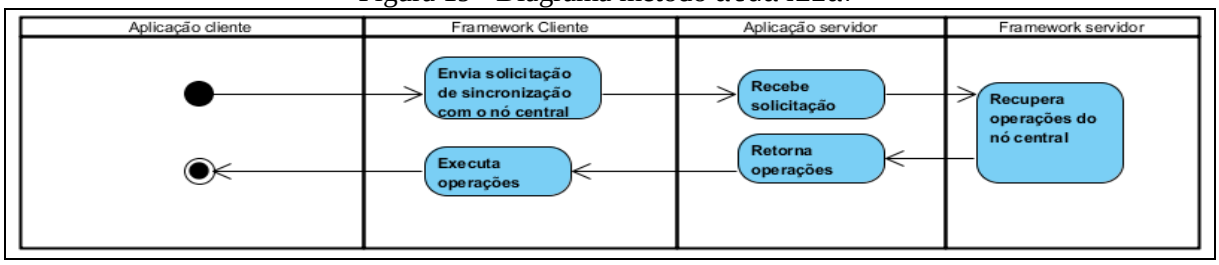
Na figura 18 é demonstrado o fluxo do método `Sincronizar` da classe `persist`. Este é o método responsável pelo envio de uma operação para o servidor. Em um segundo momento ele solicita o recebimento das operações do nó central e aplica as mesmas na base local.

Figura 18 - Diagrama método `sincronizar`



Na figura 19, tem-se o fluxo do método `atualizar`, utilizado para recebimento de operações do servidor e a execução dessas no aplicativo cliente. A diferença deste método para o método `sincronizar` é que neste não é necessário o envio de uma operação para o servidor, ou seja, é somente uma forma de buscar operações.

Figura 19 - Diagrama método `atualizar`

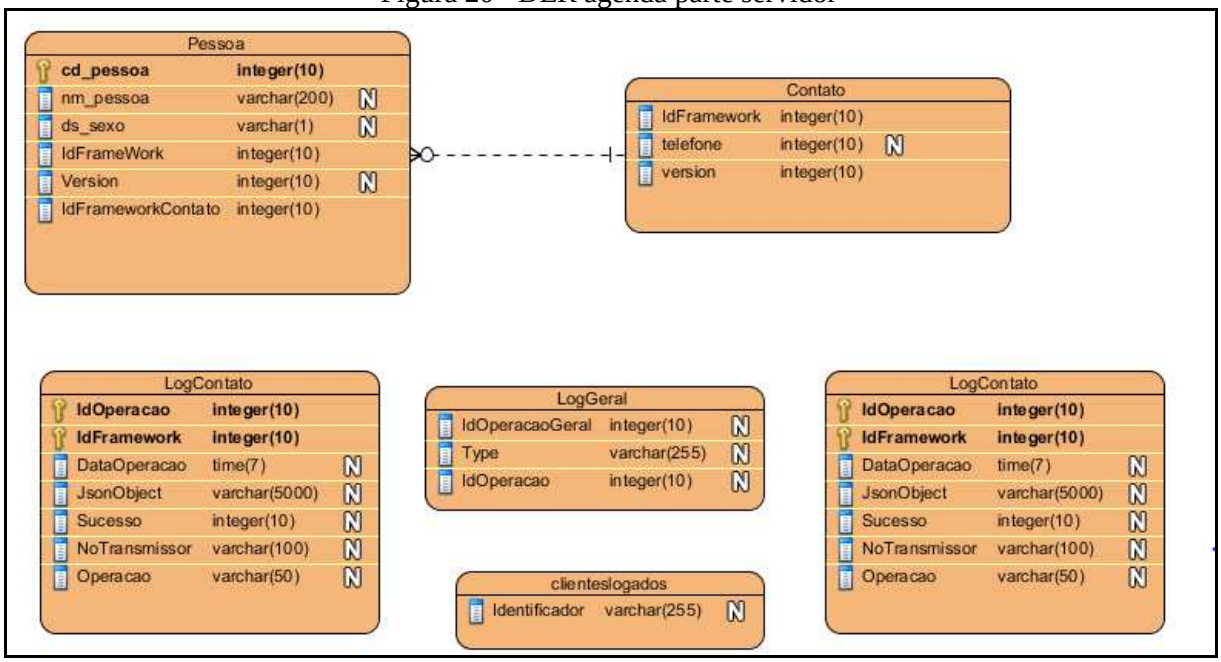


3.2.4 Diagramas de Entidade Relacionamento (DER)

Neste capitulo serão exibidos dois diagramas: um primeiro referente às estruturas de um aplicativo exemplo de agenda o segundo da parte servidor.

A figura 20 mostra o DER de uma agenda em sua parte servidor.

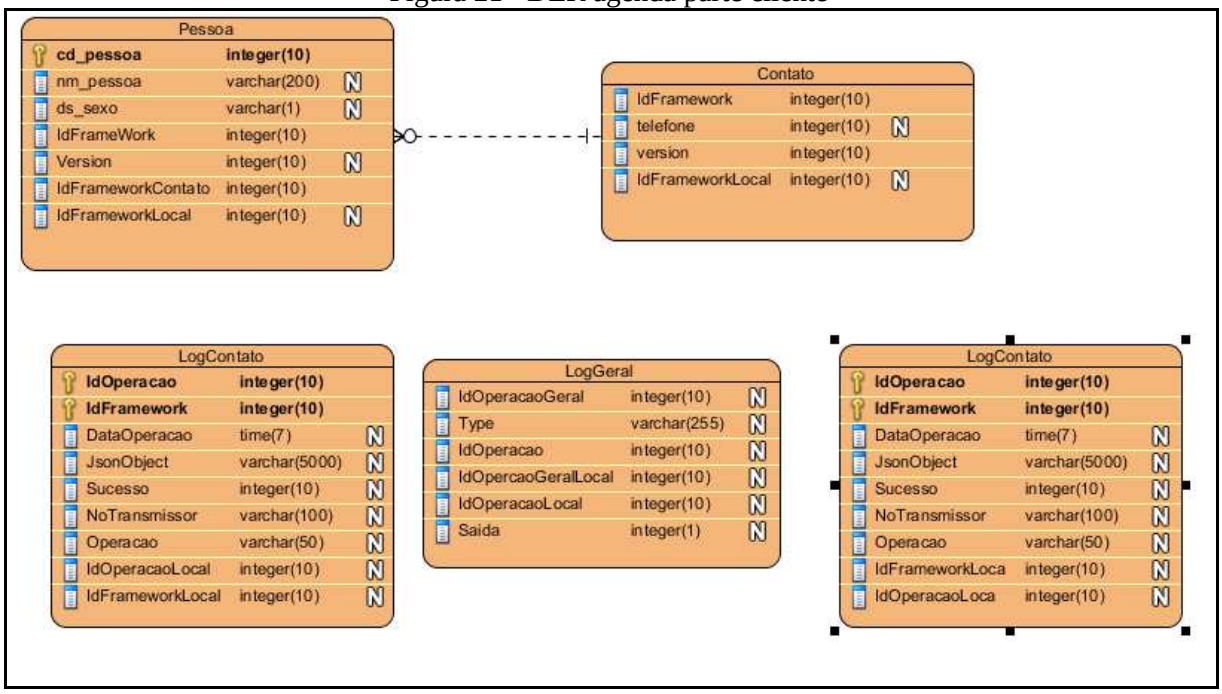
Figura 20 - DER agenda parte servidor



Na figura 20 é possível observar que algumas estruturas são necessárias para o funcionamento do *framework*. Para cada tabela do sistema, deve existir uma correspondente tabela de log para que nela sejam armazenadas as operações que foram executadas na tabela original. Também é necessária uma tabela chamada `LogGeral` que serve para armazenar a sequência dos comandos que foram executadas na base em questão. A tabela `clienteslogados` serve para armazenar quais dispositivos se conectaram ao *framework*. Nas tabelas do sistema é necessário adicionar o campo `IdFrameWork`, que serve como identificador do registro para o *framework*. Cada relacionamento do sistema também deve ser mapeado por este campo. Como na figura 20, onde existe o campo `IdFrameworkContato` na tabela pessoa mapeando a ligação entre as 2 entidades.

A estrutura da base cliente é bem similar à do servidor com pequenas diferenças. A figura 21 mostra a estrutura da base cliente.

Figura 21 - DER agenda parte cliente



As principais diferenças dos dois diagramas são a presença dos campos, que servem para controle do *framework* localmente e o campo saída na tabela de `logGeral` que serve para indicar se uma operação foi recebida ou enviada ao nó central.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas, detalhando algumas das principais rotinas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Foi utilizada a linguagem de programação Java, tanto no aplicativo cliente quanto na parte servidor. Foram utilizados recursos do Android SDK para a implementação da parte cliente do *framework*. Como banco de dados foi utilizado o MySql na parte servidor e o SqlLite na parte cliente, já que o mesmo é nativo da plataforma Android. O servidor de aplicações utilizado na aplicação exemplo foi o GlassFish Server 4.

3.3.1.1 Comunicação entre cliente e servidor

A comunicação entre o cliente e o servidor ocorre via *WebService* SOAP. Este método é simples e de fácil integração com o Android. Foi utilizada a biblioteca ksoap2, que não é nativa do Android, mas oferece uma série de facilidades para o desenvolvimento da

comunicação entre um dispositivo Android e um *WebService*. Na figura 22 tem-se um exemplo da utilização desta biblioteca dentro do *framework*.

Figura 22 – Uso do KSoap2 dentro do *framework*

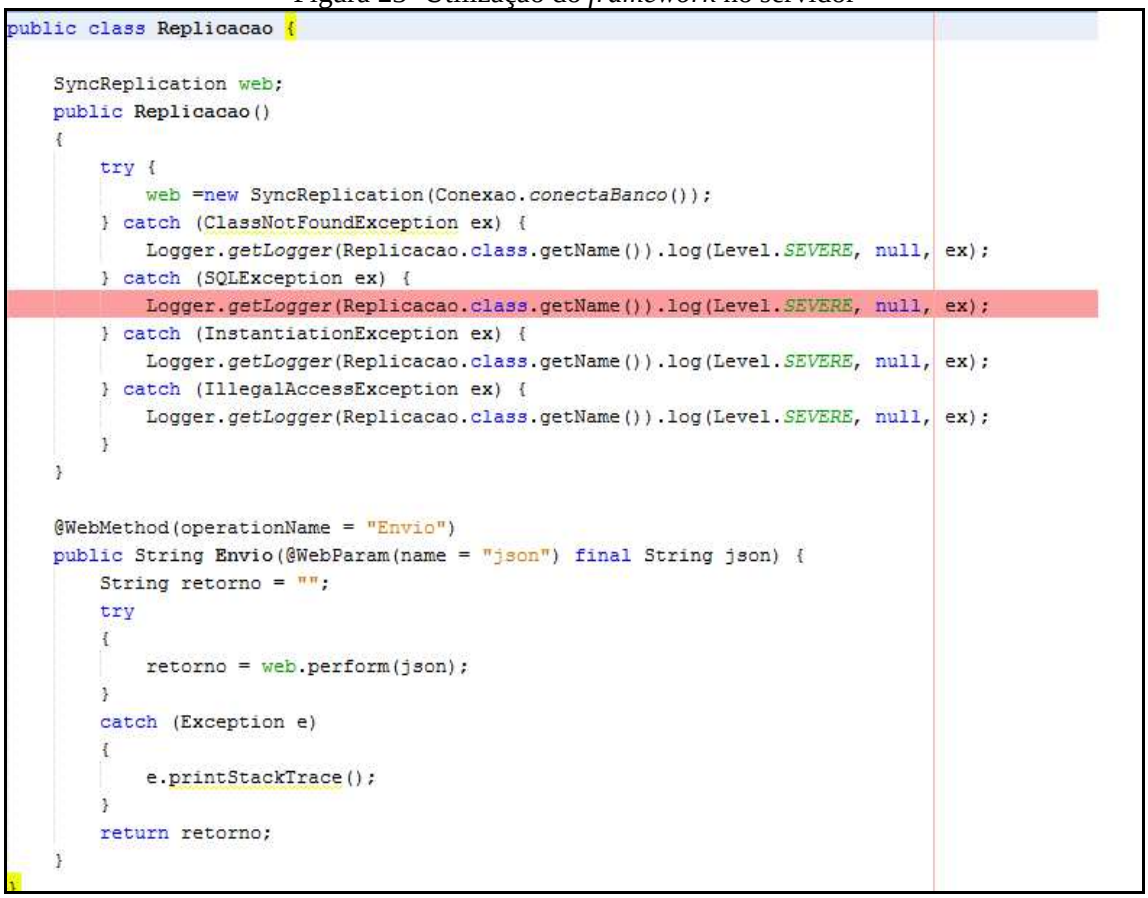
```
public final String Perform(String Json, String type) {
    SoapObject request = new SoapObject(NameSpace, MethodName);
    request.addProperty("json", Json);
    SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
    envelope.setOutputSoapObject(request);
    HttpTransportSE androidHttpTransport = new HttpTransportSE(getURL());

    try {
        androidHttpTransport.call(SoapAction, envelope);
        return  envelope.getResponse().toString();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "";
}
```

No código apresentado na figura 22, é feita a conexão com um WebService e a transmissão de um pacote SOAP contendo a propriedade JSON e o recebimento do pacote de retorno do servidor.

3.3.2 Utilização do *framework* na parte servidor

Para utilização do *framework* na parte servidor é necessário criar um método no *webservice* chamado `Envio`. Este método deve ser responsável pelo recebimento do JSON, pelo repasse dele para a classe `SyncReplication` e o retorno do objeto resultante da operação que foi executada. A Figura 23 exemplifica o uso do *framework*.

Figura 23- Utilização do *framework* no servidor



```
public class Replicacao {

    SyncReplication web;
    public Replicacao()
    {
        try {
            web =new SyncReplication(Conexao.conectaBanco());
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(Replicacao.class.getName()).log(Level.SEVERE, null, ex);
        } catch (SQLException ex) {
            Logger.getLogger(Replicacao.class.getName()).log(Level.SEVERE, null, ex);
        } catch (InstantiationException ex) {
            Logger.getLogger(Replicacao.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IllegalAccessException ex) {
            Logger.getLogger(Replicacao.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @WebMethod(operationName = "Envio")
    public String Envio(@WebParam(name = "json") final String json) {
        String retorno = "";
        try
        {
            retorno = web.perform(json);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return retorno;
    }
}
```

Conforme pode ser visto, para instanciar a classe `SyncReplication`, é necessário criar uma conexão com o banco de dados e passá-la como parâmetro para o construtor da classe. Na figura 24 está o exemplo de implementação da criação de conexão com o banco de dados.

Figura 24 - Classe de conexão

```
public class Conexao {


    public static Connection conectaBanco() throws ClassNotFoundException, SQLException, InstantiationException, IllegalAcces
        Class.forName("com.mysql.jdbc.Driver").newInstance();
        System.out.println("Driver carregado com sucesso!");
        String connectionUrl = "jdbc:mysql://localhost/teste?user=root&password=root";
        Connection con = DriverManager.getConnection(connectionUrl, "root", "root");
        return con;
    }
}
```

3.3.3 Utilização do *framework* no aplicativo cliente

Neste capítulo serão demonstrados quais são os passos para utilização do *framework* em sua parte cliente.

3.3.3.1 Definição das entidades que serão replicadas

Para definir quais entidades serão replicadas é necessária a criação de uma subclasse a paritr de `DbOpenHelperFramework` e criar uma lista com os nomes das entidades que se deseja que replicar. Isso é necessário para que o *framework* crie as tabelas básicas para o seu funcionamento. Na figura 25 tem-se o exemplo de como definir os objetos que serão replicados.

Figura 25 - Criação da base cliente

```
public void onCreate(SQLiteDatabase database) {
    final StringBuilder sql = new StringBuilder();
    sql.append("CREATE TABLE pessoa ( ");
    sql.append("IdOperacaoLocal INTEGER,");
    sql.append("IdFrameworkLocal TEXT,");
    sql.append("idframework TEXT,");
    sql.append("version  INTEGER, ");
    sql.append("cd_pessoa INTEGER,");
    sql.append("nm_pessoa TEXT,");
    sql.append("ds_sexo   TEXT,");
    sql.append("idframeworkContato TEXT ");
    sql.append(")");
    database.execSQL(sql.toString());
    final StringBuilder sql2 = new StringBuilder();
    sql2.append("CREATE TABLE Contato (   ");
    sql2.append("IdOperacaoLocal INTEGER, ");
    sql2.append("IdFrameworkLocal TEXT,   ");
    sql2.append("idframework text, ");
    sql2.append("Version     text, ");
    sql2.append("telefone    text  ");
    sql2.append(")");
    database.execSQL(sql2.toString());
    ArrayList<String> lst = new ArrayList<String>();
    lst.add("pessoa");
    lst.add("contato");
    super.setLstEntidades(lst);
    super.onCreate(database);
}
```

Na figura 25 existe um exemplo da criação de 2 tabelas no Android e a inscrição delas no *framework* através do método `setLstEntidades`, passando uma lista de entidades que vão ser replicadas pelo framework.

3.3.3.2 Criação das classes para replicação

Para uma entidade ser persistida através do *framework* é necessário que a classe que representa esta entidade seja subclasse de `SyncReplicationObject`. A figura 26 mostra a classe exemplo `Pessoa`.

Figura 26 - Objeto `pessoa`

```
public class Pessoa extends SyncReplicationObject {

    @SerializedName("nm_pessoa")
    private String nome;

    @SerializedName("Contato")
    private Contato contato;

    public String getNome() {
        return nome;
    }
    public void setNome(String some) {
        this.nome = some;
    }

    public Contato getContato() {
        return contato;
    }

    public void setObjetoFilho(Contato objetoFilho) {
        this.contato = objetoFilho;
    }
}
```

As *annotations* `@SerializedName` servem para definir o nome do campo a ser utilizado quando esse for serializado pela classe `GSON`.

3.3.3.3 Criação da classe para tratamento de conflitos

A classe para gerenciar os conflitos deve ser subclasse de `ThreadReplicacao` e sobreescrever método `trataErroSincronia`, que é disparado pelo *framework* cada vez que algum erro de sincronia ocorrer. O desenvolvedor que utiliza o *framework* pode escolher o método que irá utilizar para tratar o erro que ocorreu. Na classe exemplo só é gerada uma mensagem de *log*, entretanto o desenvolvedor pode desenvolver toda uma lógica própria para resolver o conflito. Essa também é a classe responsável pela comunicação com o servidor. A figura 27 demonstra a implementação desta classe.

Figura 27 - Tratamento de conflitos

```
package com.example.testeweb20;

import android.util.Log;

public class WebService extends ThreadReplicacao {

    public WebService(DBOpenHelper db) {
        super(db);
    }

    @Override
    protected void trataErroSincronia(JsonObject objRecebido, PersistObjectContainer objEnviado) {
        // TODO Auto-generated method stub
        Log.i("", "Erro de sincronia");
    }

}
```

3.3.3.4 Configuração do Web Services Description Language (WSDL)

No aplicativo cliente é necessário configurar o endereço do WSDL do nó central, bem como os dados básicos para comunicação com o *webservice* que está no servidor. A figura 28 mostra o modo como essa configuração deve ser feita. A classe em que essa configuração deve ser realizada é a subclasse herdeira de `ThreadReplicacao`.

Figura 28 - Configuração do nó central

```
web3.setURL(URL);
web3.setSoapAction("http://Replicacao.me.org/Replicacao/");
web3.setNameSpace("http://Replicacao.me.org/");
web3.setMethodName("Envio");
web3.setIdentificador("Local2");
```

Além dos dados básicos para a conexão com o *webservice*, é necessario criar um identificador único para o cliente que está conectando ao servidor. Este identificador serve para a identificação das ações que por ele são replicadas para o servidor.

3.3.3.5 Envio de operações

Nessa seção será demonstrado como implementar as ações básicas incluir, alterar, excluir e atualizar. As chamadas das operações de *insert*, *update* e *delete* precisam de seis parâmetros:

a) `helper`, que é uma instância de `DbOpenHelperFramework`;
b) uma String, que define a operação;
c) o objeto a ser replicado;
d) o nome da entidade a ser replicada;
e) a instância de uma classe que herde de `ThreadReplicacao`;
f) o `appliccationContext`.

Nas figuras 29,30 e 31 são demonstradas as chamadas das operações *insert*, *update* e *delete*.

Figura 29 - Operação de *insert*

```
persistencia.Sincronizar(helper, "Insert", c, "contato", web2,getApplicationContext());
```

Figura 30 - Operação de *update*

```
persistencia.Sincronizar(helper, "Update", p, "pessoa", web2, getApplicationContext());
```

Figura 31 - Operação de *delete*

```
persistencia.Sincronizar(helper, "Delete", p, "pessoa", web2,getApplicationContext());
```

A operação de atualizar não envia objeto algum para o servidor, apenas atualiza a base local com os comandos que estão no nó central e que ainda não foram replicados. A chamada do método `Atualizar` necessita de 2 parâmetros:

a) `helper` que é uma instância de `DbOpenHelperFramework;`

b) uma classe que herde de `ThreadReplicacao`.

A figura 32 demonstra a chamada do método `Atualizar`.

Figura 32 - Operação de atualização

```
persistencia.Atualizar(helper, webConsulta);
```

3.3.4 Utilização do JSON para a serialização de objetos

Para a transmissão dos objetos através do *framework* é realizada a serialização desses através do JSON. Para tal efeito, neste trabalho foi utilizada a biblioteca GSON, fornecida pelo Google, a qual facilita a serialização de objetos. Os principais objetivos do projeto GSON são fornecer métodos toJson e fromJson, suportar a Java Generics, não necessitar que objetos sejam modificados para serem convertidos para JSON e suportar objetos complexos (NETO, 2012).

A figura 33 exemplifica o uso da biblioteca GSON para recuperar um objeto Java.

Figura 33 – Recuperação de objeto Java via biblioteca GSON

```
public static ResponseConsult unSerializeConsult(String objeto)
{
    Gson gson = new GsonBuilder().create();
    ResponseConsult objetoRestaurado = gson.fromJson(objeto, ResponseConsult.class);
    return objetoRestaurado;
}
```

Nesse caso o GSON irá retornar o objeto `ResponseConsult`, instanciado a partir da *String* JSON que foi passada no segundo parâmetro. A figura 34 exemplifica o processo inverso, ou seja, a serialização de um objeto utilizando a biblioteca.

Figura 34 - Serialização JSON

```
public static String serialize(Object objetoTexto)
{
    if (objetoTexto != null)
    {
        Gson gson = new GsonBuilder().create();
        String jsonSerializado = gson.toJson(objetoTexto);
        return jsonSerializado;
    }
    return "";
}
```

3.3.4.1 Detecção de conflitos pelo servidor.

Nessa seção será mostrada a lógica da parte de controle de conflitos. Existem três tipos de exceção que são detectados pelo nó central:

a) registro inexistente;
b) registro desatualizado;
c) operação que não pôde ser executada no banco de dados.

A detecção de registro inexistente ocorre nas operações de *update* e *delete*. Basicamente o algoritmo verifica se o registro enviado existe na base de dados servidor. Para recuperar esse objeto é utilizado o campo `IdFramework`. Para a detecção de registro desatualizado é utilizado o campo `version`. Caso o registro enviado tenha uma `version` menor que o existente uma exceção é retornada. Já quando a operação não pôde ser executada no banco de dados o *framework* retorna um erro genérico.

Nas figuras 35 e 36 é exibida a lógica do *update* para controle de conflitos.

Figura 35 - Resolução de conflitos no *update* - parte 1

```
public final SyncResponseContainer update(PersistObjectContainer objeto)
{
    String idFramework = objeto.getIdFrameWork();
    String type = objeto.getType();
    SyncObject obj = persistencia.consultObject(idFramework,type);
    if ( obj != null && obj.getVersion() > 0)
    {

        if (obj.getVersion() < objeto.getVersion())
        {
            if ((persistencia.update(objeto)))
            {
                ResponseContainer result = new ResponseContainer();
                result.setSuccess(true);
                result.setIdFrameWork(objeto.getIdFrameWork());
                result.setIdOperacao(objeto.getIdOperacao());

                return result;
            }
            else
            {
                ResponseError result = new ResponseError();
                result.setSuccess(false);
                result.setMsgError("Erro no update");
                return result;
            }
        }
```

Figura 36 - Resolução de conflitos no *update* - parte 2

```
        else
        {
            SyncResponseError result = new ResponseError();
            result.setSuccess(false);
            result.setObjectConflicted(persistencia.consultObject(objeto.getIdFrameWork(),objeto.getType()));
            result.setMsgError("Conflito");
            return result;
        }
    }
    else
    {
        SyncResponseError result = new ResponseError();
        result.setSuccess(false);
        result.setObjectConflicted(persistencia.consultObject(objeto.getIdFrameWork(),objeto.getType()));
        result.setMsgError("Registro inexistante");
        return result;
    }
}
```

A lógica de detecção de conflitos do *framework* é similar ao algoritmo do CVS, especificamente na forma de versionar os registros individualmente, utilizando um campo `version` para guardar a versão.

Outro ponto a ser destacado é que na operação de *insert* só são detectados conflitos referentes ao banco de dados pois não há sentido em verificar a versão de um objeto que ainda não existe no banco de dados. A figura 37 mostra o algoritmo do *insert*.

Figura 37 - Resolução de conflitos no *insert*

```
public final SyncResponseContainer insert(PersistObjectContainer objeto)
{
    JsonObject j = objeto.getObjetoPersist();
    Map campoValor = unserializeMap(j);
    if (persistencia.insert(objeto))
    {
        ResponseContainer result = new ResponseContainer();
        result.setSuccess(true);
        result.setIdFrameWork(objeto.getIdFrameWork());
        result.setIdOperacao(objeto.getIdOperacao());
        return result;
    }
    else
    {
        SyncResponseError result = new ResponseError();
        result.setSuccess(false);
        result.setObjectConflicted(objeto);
        result.setMsgError("Erro no insert");
        return result;
    }


}
```

3.3.4.2 Serialização das operações no servidor

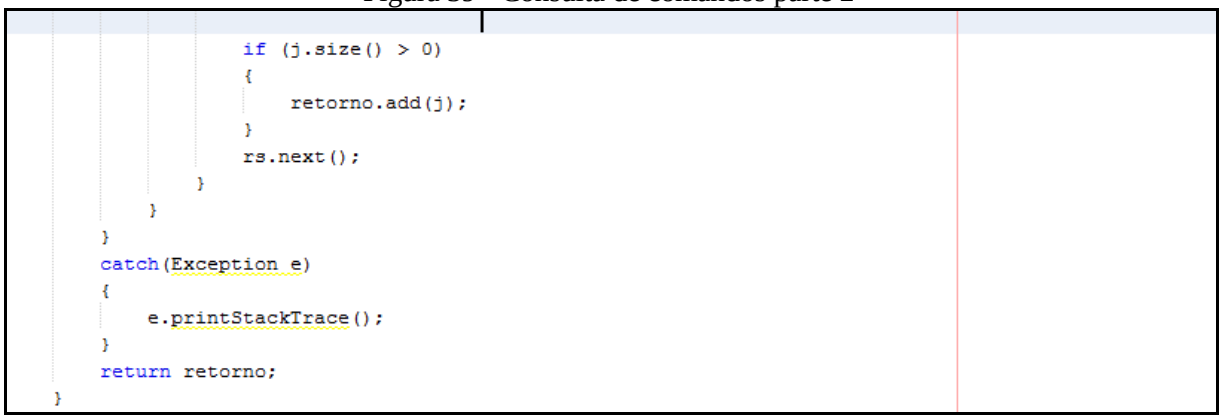
Quando um cliente solicita ao servidor as operações que foram executadas por outros nós, é necessário que as operações sejam armazenadas em uma lista para futura serialização e retorno para o cliente.

As figuras 38 e 39 mostram a lógica deste processo.

Figura 38 - Consulta de comandos parte 1

```
public List<HashMap> ConsultComandos(String extra, String No)
{

    List<HashMap> retorno = new ArrayList<HashMap>();
    try
    {
        String sqlPrincipal = " "+
        "SELECT                 "+
        "   *                   "+
        "FROM                   "+
        "   loggeral            "+
        "WHERE                  "+
        "   IdOperacaoGeral >  "+extra+" "+
        "ORDER BY IdOperacaoGeral ";
        Statement stm = connection.createStatement();
        ResultSet rs = stm.executeQuery(sqlPrincipal);
        rs.first();
        if (rs.getRow() != 0)
        {
            while (!rs.isAfterLast())
            {
                String sql =
                        " SELECT        " +
                        "   *           " +
                        " FROM          " +
                        "   Log"+rs.getString("type")+
                        " WHERE     "+
                        "  NoTransmissor <> '"+No+"' " +
                        "  AND IdOperacao = "+rs.getInt("IdOperacao");
                HashMap<String,Object>  j = execConsulta(sql,rs.getInt("IdOperacaoGeral"),rs.getString("type"));
```

Figura 39 - Consulta de comandos parte 2



```
                    if (j.size() > 0)
                    {
                        retorno.add(j);
                    }
                    rs.next();
                }
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        return retorno;
    }
```

A lógica de retorno é a seguinte: são retornadas todas as operações que não foram enviadas pelo nó cliente que solicitou a atualização, ordernadas pelo campo `idOperacaoGeral`. Esta ordernação é importante pois no cliente é necessário que as operações sejam executadas na mesma ordem que foram executadas no nó central.

O cliente recebe essas operações e as executa conforme a figura 40 demonstra.

Figura 40 - Método `ReplicaDadosFromServer`

```
private void ReplicaDadosFromServer(int idOperacaoMax)
{

    PersistObjectContainer persist2 = new PersistObjectContainer();

    persist2.setOperacao("Consult");
    persist2.setObjetoPersist(Utils.unSerializeJSON(Utils.serialize(new SyncReplicationObject())));
    persist2.setIdNo(Identificador);
    persist2.setType(type);
    persist2.setExtra(String.valueOf(idOperacaoMax));

    BaseClient bc = new BaseClient();
    try
    {

        bc.GerarSql(Perform(Utils.serialize(persist2), type) );
        List sqlsLog = bc.getSqlsLog();
        for (Object object : sqlsLog) {
            execQuery((String) object);
        }
        List sqlBanco =  bc.getSqlsTabela();
        for (Object object : sqlBanco) {
            execQuery((String) object);
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

O método `ReplicaDadosFromServer` executa as operações que são retornadas pelo servidor. Para transformar o retorno JSON em SQL é utilizada a classe auxiliar `BaseClient`, que gera tanto os SQLs para as tabelas de logs quanto para as tabelas originais.

3.3.5 Operacionalidade da implementação

A seguir será demonstrada uma aplicação exemplo que foi criada para mostrar as funcionalidades do *framework*. O aplicativo trata-se de uma agenda simplificada, em que são alimentadas duas tabelas: uma de pessoas e outra de contatos. Neste aplicativo foram exploradas todas as funcionalidades que o *framework* disponibiliza para o aplicativo cliente.

Foram criadas cinco ações no aplicativo são elas: inserir, alterar, excluir, listar e atualizar. As três primeiras ações são as básicas de envio de dados para o *framework*. A ação de listar recupera um registro do banco de dados local, a ação atualizar busca os comandos pendentes do servidor e os replica na base local.

Assim que o aplicativo é iniciado, é necessário parametrizá-lo a respeito da URL do nó central. A figura 41 mostra a configuração inicial de um aplicativo cliente para um *webservice* que está em rede local.

Figura 41 - Configuração do *webservice*

A figura 42 mostra o aplicativo com um registro já recuperado através do *framework.*

Figura 42 – Registro Recuperado no aplicativo exemplo

Na figura 43 é demonstrada a sequência de transferência de JSON pelo cliente e retornada pelo servidor para que a inclusão do registro mostrado na figura 42 fosse concluída com sucesso.

Figura 43 - Sequências JSON transferidas

{"type":"contato","IdNo":"Local2","operacao":"Insert","objeto":{"telefone":"123","Type":"Contato","IdOperacaoLocal":3,"IdFrameWorkLocal":3,"Version":1,"IdFrameWork":""},"IdOperacaoLocal":3,"IdFrameWorkLocal":3}

{"success":true,"IdFrameWork":"1414952333473contato","IdOperacao":8}

{"type":"pessoa","IdNo":"Local2","operacao":"Insert","objeto":{"Contato":{"telefone":"123","IdFrameWork":"1414952333473contato","Type":"Contato","IdOperacaoLocal":3,"IdFrameWorkLocal":3,"Version":1},"nm_pessoa":"testejacques","Type":"Pessoa","IdOperacaoLocal":3,"IdFrameWorkLocal":7,"Version":1,"IdFrameWork":""},"IdOperacaoLocal":3,"IdFrameWorkLocal":7}

{"success":true,"IdFrameWork":"1414952493871pessoa","IdOperacao":9}

A figura 43 mostra a sequência de operações que resulta em uma inclusão de contatos bem sucedida. O primeiro JSON enviado é referente ao registro na tabela contato. Após, é recebido um retorno confirmando que o registro foi enviado com sucesso, juntamente com o `IdFramework` e o `IdOperacao` resultante dessa ação no *framework*. Na sequência foi enviado o objeto `Pessoa`. Nele foi feita a ligação com o objeto contato recém criado. Novamente foi transmitido um retorno com sucesso para o cliente e nele estão contidas as informações de `IdFrameWork` e `IdOperacao`.

3.4 RESULTADOS E DISCUSSÃO

Através do *framework* desenvolvido juntamente com o aplicativo exemplo de utilização, o trabalho alcançou seus objetivos que eram replicar os dados dentro de uma arquitetura estrela e fornece suporte para sincronização dos dados enviados.

A utilização do *framework* ficou separada em duas partes: a primeira voltada para aplicação cliente e a segunda para um servidor centralizador. Essas duas partes operam em conjunto para garantir a replicação de dados por toda a rede montada.

A comunicação entre as duas partes é feita através de um *webservice* SOAP. Seria interessante estudar outra forma de comunicação, pois esta ainda se trata de uma comunicação pesada e que necessita de uma estrutura para envio e recebimento. Poderia ser utilizada o protocolo HTTP nativo sem a necessidade de toda uma estrutura de webservice no meio do caminho, haja visto que para o *framework* funcionar bastaria o envio de operações via JSON e o retorno das respostas também via JSON.

Durante os testes, o *framework* se mostrou adequado para detectar três formas de conflito entre o nó ponta da estrela e o nó central, a saber: um registro que foi enviado e não estava atualizado, uma operação enviada que afetava um registro que não estava mais no banco de dados e uma operação que não pôde ser executada no banco por restrições do próprio banco. Foram realizados testes localmente no servidor. Sem interferência da infraestrutura de rede, o sistema em sua parte servidor pode processar até 407 operações por minuto.

No quadro 3 foi feito um quadro comparativo entre o framework desenvolvido e os trabalhos correlatos abordados, abordando 4 pontos: a topologia de rede suportada, o suporte para dispositivos móveis, o tipo de algoritmo utilizado e customização do tratamento de conflitos.

Quadro 3 - Quadro comparativo com os trabalhos correlatos

	Framework implementado	MACEDO et al.(2008)	MONTEIRO et al. (2006)	SILVA (2014)
Topologia de rede suportada	Estrela	Peer-to-peer	Independente de topologia	Independente de topologia
Suporta dispositivos móveis	X	X	-	-
Tipo de algoritmo	Otimista	Otimista	Otimista	Pessimista
Customização do tratamento de conflitos	X	-	-	-

Conforme pode ser observado o único trabalho que utilizou algoritmos pessimistas foi o Silva (2014). Isso deve-se ao fato de que o trabalho dele foi desenvolvido para ambientes de alta conectividade. Os demais trabalhos utilizaram algoritmos otimistas. Tanto o trabalho desenvolvido neste TCC quanto o Macedo et al. (2008) suportam a utilização de dispositivos móveis, utilizando topologias diferenciadas. Um aspecto diferenciado deste trabalho em relação aos demais é a liberdade no tratamento de conflitos que o usuário do *framework* pode adotar.

4 CONCLUSÕES

Os objetivos deste trabalho foram atingidos com sucesso. O *framework* possui uma lógica para manutenção dos estados globais das bases de dados de um sistema distribuído, fornecendo um conjunto de métodos que facilitam a replicação de dados entre os dispositivos envolvidos. Fornece uma estrutura para um modelo multi-master aliado a uma topologia em estrela. Foi desenvolvido um aplicativo de teste para demonstrar que estas funcionalidades estão de acordo com a especificação.

O *framework* permite que sejam replicadas operações de um nó cliente para um nó servidor e que as operações replicadas do nó servidor sejam replicadas para todos os nós clientes. Isso faz com que as bases envolvidas no processo convirjam para um estado de sincronia global, apesar de em um certo momento, nem todas as bases apresentarem os mesmos valores.

O algoritmo para a manutenção do estado global das bases, ou seja, o algoritmo que detecta os possíveis conflitos que irão fazer com que a bases de dados não possam ser replicadas adequadamente age no nó central. Caso algum conflito seja detectado, cabe ao aplicativo cliente resolver o conflito e reenviar o comando para o servidor.

A topologia *multi-master* em estrela foi implementada de forma que os dispositivos que estão nas pontas se conectem a um *webservice q*ue está no nó central. Isto foi demonstrado através da criação de uma aplicação cliente que envolve um aplicativo cliente instalado em um dispositivo Android e um nó central que utiliza o *framework* para a implementação de um *webservice*. A velocidade de processamento foi considerada aceitável em testes locais, utilizando apenas o algoritmo sem considerar questões de influência de rede. O processamento do servidor chegou a 407 operações por minuto, utilizando objetos da classe exemplo `pessoa`. A máquina utilizada para testes foi um *notebook* Dell com 8 gigabytes de memória RAM e processador intel i7.

Para a comunicação entre os dispositivos foi utilizada a tecnologia de *webservice* SOAP. Essa camada trata-se de um meio intermediário para a comunicação com o servidor, haja vista que para que a lógica do framework funcione é necessário apenas que exista um meio de comunicação para a transmissão dos objetos JSON.

Conforme as pesquisas realizadas, a replicação e sincronia de dispositivos móveis é um trabalho que envolve diversas técnicas, métodos e algoritmos. Neste *framework* foi utilizado um algoritmo positivo no qual conflitos detectados são reportados para quem os gerou. O

framework abstrai para o desenvolvedor uma série de validações e métodos de forma a deixar para ele apenas a implementação do método para resolver os conflitos detectados pela camada servidor.

4.1 EXTENSÕES

Algumas da possíveis extensões para este trabalho são:

a) implementar a comunicação utilizando somente o protocolo HTTP sem a utilização de um *webservice* intermediário;
b) criptografar as informações trafegadas;
c) implementar suporte a vários fornecedores de banco de dados na parte servidor;
d) suportar a replicação de dados em dispositivos móveis em outras arquiteturas ou topologias.

REFERÊNCIAS

ANDROID DEVELOPERS. **Android**: an open handset alliance project. [S.l.], 2014. Disponível em: <http://code.google.com/android>. Acesso em: 16 nov. 2014.

CISCO. Cisco visual networking index: G**lobal mobile data traficc forecast update, 2013-2018**. Estados Unidos da América, 2014. Disponível em: <http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html>. Acesso em: 01 abr. 2014.

DATE, C. J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro : Campus, 1991.

FIGUEIREDO NETO, Mário Lemes de. **CVS instalação e configuração.** 2010. Disponível em: < http://www.devmedia.com.br/cvs-instalacao-e-configuracao/2801>. Acesso em: 16 nov. 2014.

GEHRKE, Johannes; RAMAKRISHNAN, Raghu. **Sistemas de gerenciamento de bancos de dados.** 4. ed. Tradução Célia Taniwake. São Paulo: Mcgraw, 2008.

GONÇALVES, Eduardo Corrêa. **SQLite, muito prazer!** 2013. Disponível em: <http://www.devmedia.com.br/sqlite-muito-prazer/7100>. Acesso em: 16 nov. 2014.

GURGEL,Giovane M. M. **O valor estratégico das informações**. In: SIMPEP, 13.,2006 , Bauru. **Anais…** Bauru: 2006. p. 1-10. Disponível em: <http://issuu.com/aninhaquase32/docs/o_valor_estrat__gico_da_informa____>. Acesso em: 01 abr. 2014

HERMANN, Jonatas D. **Protótipo de um visualizador volumétrico de imagens DICOM na android**. 2013. 58 f. Trabalho de conclusão de curso (Bacharelado em computação) - FURB, Blumenau, Brasil.

JSON. **Introdução ao JSON**. [2010]. Disponível em: <http://www.json.org/json-pt.html>. Acesso em: 16 nov. 2014.

LUCIO, Diego R. **Um aplicativo para dispositivos móveis voltado para usuários de transporte público**. 2011. 56 f. Artigo(Bacharelado em computação) - UTFPR, Campo Mourão, Brasil. Disponível em:<http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/247/1/CM_COINT_2011_2_03.pdf > Acesso em: 19 de maio. 2014

MACEDO, Douglas et al. **Replicação assíncrona entre bancos de dados médicos Distribuídos**. 2008. 8 Artigo(Bacharelado em computação) - UFSC, Santa Catarina. Disponível em: <http://www.inf.ufsc.br/erbd2008/artigos/8.pdf>. Acesso em: 01 abr. 2014

MONTEIRO, José M. **Consistência de dados em dispositivos móveis**. 2005. 12 Artigo(Bacharelado em computação) - PUC, Rio de Janeiro, Brasil. Disponível em: <http://www.inf.ufsc.br/~frank/BDD/Artigos/Consistencia_BDD_Moveis.pdf>. Acesso em: 01 abr. 2014

MONTEIRO, José M. et al. **Um mecanismo para a consistência de dados replicados em computação móvel.** 2006. 25 f. Artigo(Bacharelado em computação) - PUC, Rio de Janeiro, Brasil. Disponível em: <ftp://ftp.inf.puc-rio.br/pub/docs/techreports/06_21_monteiro.pdf> Acesso em: 03 abr. 2014

NETO, Marin. **JSON fácil em java com gson !**, 2012. Disponível em: <http://blog.globalcode.com.br/2012/02/json-facil-em-java-com-gson.html> . Acesso em 16 nov. 2014

NODARI, Alexandre L. **Replicação otimista de dados estruturados em redes peer-to-peer**. 2007. 109 f. Dissertação (Mestrado em Informática) - Curso de Pós-Graduação em Informática. Pontifícia Universidade Católica do Paraná, Paraná, Curitiba.

OLIVEIRA, Thiago F. **Desenvolvimento de aplicação para auditoria de processos empresariais para dispositivos móveis**. 2011. 65 - FATECSJC, São José de Campos, Brasil. Disponível em: <http://fatecsjc.edu.br/trabalhos-de-graduacao/wp-content/uploads/2012/03/Desenvolvimento-de-Aplica%C3%A7%C3%A3o-para-Dispositivos-M%C3%B3veis.pdf> . Acesso em: 19 de maio. 2014

RAINONE, Flávia. **Bancos de dados móveis**. 2014. 19 f. Artigo(Bacharelado em computação) - USP, Brasil. Disponível em: <http://grenoble.ime.usp.br/movel/bdmoveisflavia.pdf> . Acesso em: 05 de abr. 2014

ROSS, Julio. **Redes de computadores**. 1. ed. Rio de Janeiro: Editora Antenna Edições Técnicas , 2008.

SILVA, Fabricio S. **Sincronização de banco de dados distribuídos utilizando snapshot isolation**. 2014. 6 f. Artigo(Bacharelado em computação) - UFPR, Curitiba,Brasil Disponível em: <http://www.inf.ufpr.br/aldri/disc/propostas/Fabricio_faseII.pdf >. Acesso em: 01 abr. 2014

THECAFETHECNO. **An introduction to concurrent versions system (CVS)** 2013. Disponível em: < http://thecafetechno.com/tutorials/cvs/an-introduction-to-concurrent-versions-system-cvs/>. Acesso em: 03 abr. 2014.