UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FRAMEWORK DE REPLICAÇÃO DE DADOS COM CRIPTOGRAFIA SIMÉTRICA UTILIZANDO PUSH NOTIFICATION PARA ANDROID

FERNANDO KLOCK

BLUMENAU
2011

2011/2-09

FERNANDO KLOCK

FRAMEWORK DE REPLICAÇÃO DE DADOS COM CRIPTOGRAFIA SIMÉTRICA UTILIZANDO PUSH NOTIFICATION PARA ANDROID

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Dalton Solano dos Reis , M.Sc. - Orientador

BLUMENAU
2011

2011/2-09

FRAMEWORK DE REPLICAÇÃO DE DADOS COM CRIPTOGRAFIA SIMÉTRICA UTILIZANDO PUSH NOTIFICATION PARA ANDROID

Por

FERNANDO KLOCK

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

Presidente:	Prof. Dalton Solano dos Reis, Mestre – Orientador, FURB
Membro:	Prof. Aurélio Faustino Hoppe, Mestre – FURB
Membro:	Prof. Paulo Cesar Rodacki Gomes, Doutor – FURB

Blumenau, 12 de dezembro de 2011

Dedico este trabalho a todos os amigos e familiares, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, que sempre esteve presente.

Ao meu orientador, Dalton Solano dos Reis, por ter acreditado na conclusão deste trabalho.

Falar obscuramente qualquer um sabe; com clareza, raríssimos.

Galileu Galilei

RESUMO

Este trabalho apresenta um *framework* para replicação de dados com a utilização de criptografia simétrica e o conceito de *push notification* para dispositivos móveis da plataforma Android. Com a utilização do *framework* na criação de aplicações móveis o desenvolvedor tem a possibilidade de replicar os dados do banco SQLite entre um dispositivo Android e um servidor remoto. O sentido de replicação e a chave da criptografia simétrica poderão ser configurados de acordo com a necessidade da aplicação. A utilização do *push notification* de dados está disponível no *framework* desenvolvido para o servidor, tendo como objetivo inicial alertar o dispositivo móvel para eventuais atualizações. Com a utilização desta tecnologia o dispositivo não necessita requisitar o servidor diversas vezes procurando atualizações, sendo que para isto seriam gastos muito processamento e bateria do aparelho.

Palavras-chave: Android. Criptografia. Push. Replicação.

ABSTRACT

This paper presents a framework for data replication with the use of symmetric cryptography and the concept of push notification mobile platform Android. Using the framework in creating mobile applications developer has the ability to replicate data between a database SQLite Android device and a remote server. The direction of replication and the symmetric encryption key can be configured according to application needs. The use of the push notification data is available on the framework developed for the server, with the initial aim to alert the mobile device to any updates. Using this technology, the device does not need to order the server several times looking for updates, and it would cost to much processing and battery of the device.

Key-word: Android. Cryptography. Push. Replication.

LISTA DE ILUSTRAÇÕES

Figura 1 – Push no Android 20
Figura 2 – Camadas da plataforma Android 21
Figura 3 – Funcionamento do DBMoto 22
Quadro 1 – Comparação entre os trabalhos correlatos 24
Figura 4 - Diagrama de casos de uso 26
Figura 5 - Diagrama de pacotes do dispositivo 32
Figura 6 - Diagrama de classe do pacote `com.google.android.c2dm` 33
Figura 7 - Diagrama de classe do pacote `br.com.fernandoklock.replication` 34
Figura 8 - Diagrama de classe do pacote `br.com.fernandoklock.crypto` 35
Figura 9 - Diagrama de classe do pacote `br.com.fernandoklock.database` 36
Figura 10 - Diagrama de pacotes do servidor 37
Figura 11 - Diagrama de classe do pacote `br.com.fernandoklock.database` 38
Figura 12 - Diagrama de classe do pacote `br.com.fernandoklock.replication` 39
Figura 13 - Diagrama de atividades de notificações 40
Figura 14 - Diagrama de seqüência da replicação de dados 41
Quadro 10 – Fragmento do manifesto para indicar a classe receptora de notificação 43
Quadro 11 – Fragmento do manifesto de permissão da aplicação para notificação 43
Quadro 12 – Fragmento de código para disparo de registro de notificação 44
Quadro 13 - Recebimento de notificação na classe `C2DMBroadcastReceiver` 44
Quadro 14 - Tratamento das mensagens recebidas na classe `C2DMBaseReceiver` 45
Quadro 15 – Geração de notificação Android 45
Quadro 16 – Requisitando token de autenticação no servidor C2DM. 46
Quadro 17 – Envio de notificação push para dispositivo 47
Quadro 18 - Utilização da criptografia simétrica AES 48
Quadro 19 - Trecho de código da implementação *socket* do servidor 49
Quadro 20 – Trecho de código da implementação *socket* no dispositivo 50
Quadro 21 - Análise para execução do comando *update* 52
Figura 15 - Localização das bases de dados SQLite no projeto 53
Quadro 22 - Classe `AtualizaDB` que estende a classe abstrata `DBManipulate` 54
Quadro 23 - Instância das classes de parametrização do *framework* 55

Figura 16 - Tela do menu da aplicação 56
Figura 17 - *Log* de erros da Replicação 56
Figura 18 - Telas da aplicação teste do *framework* 57
Figura 19 - Registrar presença para um aluno 57
Figura 20 - Tela preferências para replicação de dados 58
Quadro 24 - Registro de presença do aluno no servidor 59
Figura 21 - Recebimento da notificação no dispositivo 59
Figura 22 - Desempenho da replicação de dados 61
Figura 23 - Desempenho da criptografia simétrica 63

LISTA DE TABELAS

Tabela 1 – Desempenho da transmissão de dados no dispositivo 60
Tabela 2 – Desempenho da criptografia simétrica 62

LISTA DE SIGLAS

ADT - Android *Development Tools*

API - *Application Programming Interface*

ASCII - *American Standard Code for Information Interchange*

C2DM - *Cloud to Device Message*

CPU - *Central Processing Unit*

HTTP - *Hyper Text Transfer Protocol*

I/O - *Input/Output*

Java SE - Java *Standard Edition*

JDBC - Java *Data Base Connectivity*

JSON - *JavaScript Object Notation*

KB - *Kilobytes*

OHA - *Open Handset Alliance*

ORB - *Object Request Broker*

PDA - *Personal Digital Assistants*

REST - *REpresentational State Transfer*

RPC - *Remote Procedure Call*

SDK - *Software Developer Kit*

SGBD - Sistema de Gerenciamento de Banco de Dados

SGBDH - Sistema de Gerenciamento de Banco de Dados Heterogêneos

SOA - *Service Oriented Architecture*

SOAP - *Simple Object Access Protocol*

SQL - *Structured Query Language*

TCP/IP - *Transmission Controle Protocol / Internet Protocol*

UDP - *User Datagram Protocol*

UML - *Unified Modeling Language*

URL - *Universal Resource Locator*

UTF-8 - *Unicode Transformation Format 8-bit*

XML - *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO **14**
1.1 OBJETIVOS DO TRABALHO 15
1.2 ESTRUTURA DO TRABALHO 15
2 FUNDAMENTAÇÃO TEÓRICA **16**
2.1 CRIPTOGRAFIA SIMÉTRICA 16
2.2 SQLITE 17
2.3 SOCKET 18
2.4 PUSH NOTIFICATION 19
2.5 PLATAFORMA ANDROID 20
2.6 TRABALHOS CORRELATOS 22
2.6.1 DBMoto 22
2.6.2 Heros 23
2.6.3 Tabela comparativa entre os trabalhos correlatos 24
3 DESENVOLVIMENTO DO FRAMEWORK **25**
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 25
3.2 ESPECIFICAÇÃO 26
3.2.1 Casos de Uso 26
3.2.1.1 Visualizar notificações 27
3.2.1.2 Configurar parâmetros de conexão 27
3.2.1.3 Configurar parâmetros de replicação 28
3.2.1.4 Consumir e processar transações 28
3.2.1.5 Enviar e finalizar transações 29
3.2.1.6 Retornar erros SQL 30
3.2.1.7 Iniciar servidor de replicação 31
3.2.1.8 Notificar dispositivo 31
3.2.2 Diagrama de classe 31
3.2.2.1 Diagrama de classes do dispositivo 32
3.2.2.1.1 Pacote `com.google.android.c2dm` 32
3.2.2.1.2 Pacote `br.com.fernandoklock.replication` 33
3.2.2.1.3 Pacote `br.com.fernandoklock.crypto` 35
3.2.2.1.4 Pacote `br.com.fernandoklock.database` 36

3.2.2.2 Diagrama de classes do servidor 37
3.2.2.2.1 Pacote `br.com.fernandoklock.database` 37
3.2.2.2.2 Pacote `br.com.fernandoklock.replication` 38
3.2.3 Diagrama de atividades 40
3.2.4 Diagrama de seqüência 41
3.3 IMPLEMENTAÇÃO 42
3.3.1 Técnicas e ferramentas utilizadas 42
3.3.2 Preparando ambiente Android para *push* 43
3.3.3 Envio de notificações para o dispositivo Android pelo servidor 46
3.3.4 Criptografia simétrica AES 48
3.3.5 Utilização do *socket* para transferência de dados 49
3.3.6 Gerenciamento da replicação de dados 51
3.3.7 Operacionalidade da implementação 52
3.4 RESULTADOS E DISCUSSÃO 60
4 CONCLUSÕES 64
4.1 EXTENSÕES 65
REFERÊNCIAS BIBLIOGRÁFICAS 66

1 INTRODUÇÃO

Os dispositivos móveis estão entrando no cotidiano da sociedade apresentando várias facilidades que, até então, eram disponíveis apenas para computadores *desktop*, como transferências interbancárias e compra de produtos. O aumento no uso destes dispositivos, principalmente *smartphones*, trouxe algumas preocupações com relação à segurança dos dados manipulados. Ataques de cibercriminosos vêm crescendo devido a grande adesão de dispositivos móveis em ambientes coorporativos, combinada à infra-estrutura frágil dos telefones celulares e ao lento avanço em relação à segurança (INFOMONEY, 2011). Sendo assim, como garantir a segurança dos dados manipulados nos dispositivos móveis? Uma solução para garantir a segurança nestes casos é o uso da criptografia.

A criptografia pode ser aplicada nos vários níveis de interação do sistema. Um deles é o nível de replicação de dados, onde as informações são transferidas entre o cliente e o servidor como cópia de dados. Esta cópia é acionada por uma transação executada em um Sistema de Gerenciamento de Banco de Dados (SGBD) *source*, replicando para um ou mais *targets*, de forma que a informação seja consistente em todas as bases (HOPKINS; THOMAS, 1998). As vantagens de assegurar a segurança neste nível são: sigilo, integridade e autenticação.

Diante do exposto, disponibilizou-se um *framework* para a plataforma de desenvolvimento Android *Software Developer Kit* (SDK), que agrega características necessárias para garantir a segurança da informação e que possibilita ao desenvolvedor abstração na replicação de dados entre o dispositivo e o servidor. A replicação utiliza como repositório de dados o SQLite, uma biblioteca que disponibiliza a modelagem de dados através do padrão *Structured Query Language* (SQL). O diferencial da aplicação é a utilização do *push*[1], servindo como mensageiro para alertar novas atualizações nas bases de dados. Após o recebimento da notificação via *push notification*, o dispositivo móvel e o servidor irão estabelecer uma comunicação utilizando a arquitetura de comunicação *socket*. Os dados transferidos utilizam a criptografia simétrica, tornando a comunicação cliente/servidor segura.

---

[1] *Push* é um serviço onde os servidores notificam as aplicações móveis, a fim de atualizar a ferramenta ou dados do usuário (CODE, 2011a).

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *framework* de replicação de dados com criptografia simétrica utilizando *push notification* para o Android.

Os objetivos específicos do trabalho são:

a) disponibilizar a replicação de dados entre o dispositivo móvel e o servidor utilizando a biblioteca SQLite;
b) aplicar criptografia simétrica com a utilização de chaves de 128 bits;
c) definir a interface de comunicação entre o dispositivo e o servidor;
d) validar o *framework* criando um aplicativo exemplo que utilize a replicação de dados.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está apresentada em quatro capítulos, onde o segundo capítulo contém a fundamentação teórica necessária para o entendimento deste trabalho.

O terceiro capítulo apresenta como foi desenvolvida a ferramenta para as plataformas Android e Java *Stardard Edition* (Java SE), os casos de uso da aplicação, diagramas de classe e a especificação que define a ferramenta. No terceiro capítulo são apresentadas também, as partes principais da implementação juntamente com resultados e discussões que aconteceram durante o desenvolvimento do projeto.

Por fim, o quarto capítulo refere-se às conclusões do presente trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está organizado em 7 seções: a primeira trata da criptografia simétrica, a segunda descreve o funcionamento da biblioteca SQLite, a terceira apresenta o REST, a quarta o *socket*, a quinta mostra o funcionamento do *push notification*, a sexta a plataforma Android e a última traz os trabalhos correlatos ao trabalho proposto.

2.1 CRIPTOGRAFIA SIMÉTRICA

> O termo criptografia surgiu da fusão das palavras gregas "kryptós" e "gráphein", que significam "oculto" e "escrever", respectivamente. Trata-se de um conjunto de conceitos e técnicas que visam codificar uma informação de forma que somente o emissor e o receptor possam acessá-la, evitando que um intruso consiga interpretá-la. (ALECRIM, 2005).

Cifragem é o processo pelo qual a criptografia torna dados incompreensíveis, isso é, codificados ou secretos, normalmente chamados de texto cifrado. Já o processo inverso é a decifragem, em que dados são decodificados para possibilitar sua leitura e normalmente chamados de texto claro (CARVALHO, 2001, p. 52). Para a cifragem e decifragem de uma informação é necessário o uso de chaves. As chaves são um conjunto de bits baseado em um determinado algoritmo capaz de criptografar um conjunto de informações. Se o receptor da mensagem usar uma chave incompatível com a chave do emissor, não conseguirá extrair a informação (ALECRIM, 2005).

Existem tamanhos diferentes de chaves a serem utilizadas na criptografia, como por exemplo, 8 bits e 128 bits. Quanto mais bits forem utilizados, mais segura será a criptografia. Caso um algoritmo utilize chaves de 8 bits, apenas 256 chaves poderão ser usadas na decodificação. Porém, se for usado 128 bits existirá uma grande quantidade de combinações deixando a informação criptografada mais segura (ALECRIM, 2005).

Há maneiras diferentes de cifrar uma informação sendo uma delas a criptografia de chave simétrica. Este estilo de criptografia realiza a cifragem e a decifragem de uma informação por meio de algoritmos que utilizam a mesma chave, garantindo sigilo na transmissão de dados (ALECRIM, 2005). Os algoritmos atuais são especializados para serem executados por computadores ou por alguns dispositivos de hardware. Na maioria das

aplicações, a criptografia é realizada através de software de computador. De modo geral os algoritmos simétricos são executados muito mais rapidamente que os assimétricos, pois no assimétrico o emissor deve criar uma chave de codificação e envia-la ao receptor (NUNES, 2007).

Existem vários algoritmos de criptografia simétrica disponibilizados comercialmente, dentre eles citam-se o *Data Encryption Standard* (DES) e o *Advanced Encryption Standard* (AES). O DES é um tipo de cifra de bloco de 64 bits, ou seja, um algoritmo que torna uma *string* de tamanho fixo de um texto plano e a transforma, através de uma série de complicadas operações, em um texto cifrado do mesmo tamanho. Já o AES, predecessor do DES, é o algoritmo padrão adotado pelo governo dos Estados Unidos e tem as seguintes características: cifra em blocos de 128 bits usando chaves de criptografia com 128, 192 e 256 bits, é implementado tanto em hardware como em software e apresenta um algoritmo de variação recursiva do DES (TECNOLOGIADAREDE, 2010).

2.2 SQLITE

O SQLite é uma biblioteca que implementa uma *engine* de banco de dados SQL. Diferentemente da maioria dos outros bancos de dados, o SQLite não tem um processo de servidor separado, pois a leitura e escrita dos dados são feitas diretamente em arquivos comuns. Toda a estrutura de SQL da biblioteca, como as tabelas, índices, *triggers* e *views* estão contidos em um único arquivo no disco. O formato do arquivo de banco de dados é multi-plataforma, ou seja, pode ser utilizado em sistemas 32 bits e 64 bits, bem como para as plataformas Android e iOS (SQLITE, 2009).

A biblioteca apresenta algumas características relevantes quanto à utilização de hardware nos dispositivos. Com todos os recursos habilitados, seu tamanho pode ser inferior a 300 *kilobytes* (KB), dependendo das configurações de otimização do compilador. Se os recursos opcionais são omitidos, o tamanho pode ser reduzido a menos de 180 KB. Além disso, a biblioteca pode ser utilizada em um espaço mínimo de *stack* (4 KB) e de *heap* (100 KB). Estas características tornam SQLite uma ótima escolha para dispositivos de memória limitada, como celulares e *Personal Digital Assistant* (PDA) (SQLITE, 2009).

A base do código SQLite é mantida por uma equipe internacional de desenvolvedores que trabalham em tempo integral. Os desenvolvedores continuam expandindo as capacidades

da biblioteca, além de aumentar sua confiabilidade e desempenho, mantendo a compatibilidade. A cada lançamento o produto é cuidadosamente testado com relação a erros de *Input/Output* (I/O), falhas nos sistemas ou falhas de energia, diminuindo possíveis erros durante a execução (SQLITE, 2009).

2.3 SOCKET

Segundo Macoratti (2004), *socket* pode ser entendido como uma porta de um canal de comunicação que permite a um processo executando em um computador enviar ou receber mensagens para ou de outro processo que pode estar sendo executado no mesmo computador ou em um computador remoto.

Para que exista comunicação em rede, faz-se necessário o uso de um protocolo de comunicação. Um protocolo funciona como linguagem entre computadores em rede e para validar esta comunicação, é preciso que os computadores utilizem o mesmo protocolo. Existem diversos tipos de protocolos. As aplicações de s*ocket* são feitas baseadas no *Transmission Control Protocol / Internet Protocol* (TCP/IP) ou *User Datagram Protocol* (UDP) (MACORATTI, 2004).

O *socket* representa um ponto de conexão para uma rede TCP/IP, muito semelhante aos soquetes elétricos encontrados em casas, que fornecem um ponto de conexão para os aparelhos eletrodomésticos. Quando dois computadores querem manter uma conversação, cada um deles utiliza um *socket*. Um computador é chamado servidor, pois ele abre um *socket* e presta atenção nas conexões, e outro computador denomina-se cliente, chamando o *socket* servidor para iniciar a conexão (FRACO; SILVA, 2009).

Segundo Franco e Silva (2009), existem dois tipos de troca de mensagens via *socket*: a orientada a conexão e a não orientada a conexão. Na comunicação orientada a conexão, TCP ou *Datastream*, uma aplicação servidora disponibiliza um canal ao qual aplicações clientes se conectarão para trocarem dados. Uma aplicação servidora pode ter conexões com vários clientes. Já na comunicação não orientada a conexão, UDP ou *Datagram*, não é estabelecida uma conexão entre a aplicação cliente e a servidora. A aplicação servidora estabelece uma porta que estará disponível para que qualquer cliente possa enviar dados. A comunicação é unidirecional e não há garantias de que os dados realmente chegarão.

2.4 PUSH NOTIFICATION

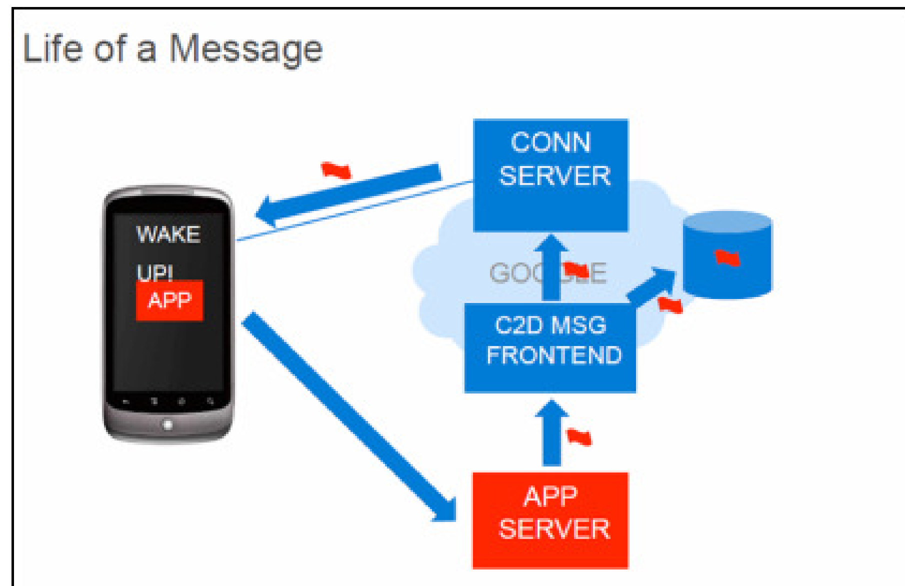
A maioria dos aplicativos desenvolvidos para a plataforma Android utiliza a internet para manter o dispositivo atualizado. Tradicionalmente, os aplicativos utilizam *polling* para recuperar as informações, como um cliente de *e-mail* que acessa o servidor em intervalos de tempo. O problema é que o *polling* gera um desgaste desnecessário dos recursos, pois utiliza processamento para buscar algo que normalmente ainda não existe. Isso é um problema, em especial, para os dispositivos móveis, pois consome uma preciosa banda de rede e a vida útil da bateria (EMIDIO, 2010).

Para as aplicações móveis que precisam utilizar os dados de um servidor remoto é mais eficiente as informações serem enviadas para o dispositivo móvel, ao invés de ficar várias vezes requisitando o servidor. A aplicação pode simplesmente aguardar o servidor informar quando ocorre uma mudança de dados (LANE, 2009).

Na versão 2.2 do Android a Google disponibilizou *Cloud to Device Message* (C2DM), cujo objetivo é facilitar o envio de dados dos servidores às aplicações em dispositivos Android. O serviço fornece um mecanismo simples e leve onde os servidores notificam as aplicações móveis, a fim de atualizar a ferramenta ou dados do usuário. O serviço C2DM controla todos os aspectos de enfileiramento e entrega de mensagens para o aplicativo em execução no dispositivo de destino (CODE, 2011a).

Existem algumas limitações do C2DM documentas pela Google que devem ser abordadas. A mensagem enviada do servidor para o dispositivo móvel pode conter apenas 1.024 bytes e o servidor remetente é limitado para o envio de mensagens, sejam elas globais ou para um dispositivo em específico. Já o número máximo de mensagens que podem ser enviadas não foi mencionado pela Google (CODE, 2011b).

A Figura 1 apresenta o funcionamento do C2DM.



Fonte: AMU Team (2010).

Figura 1 – Push no Android

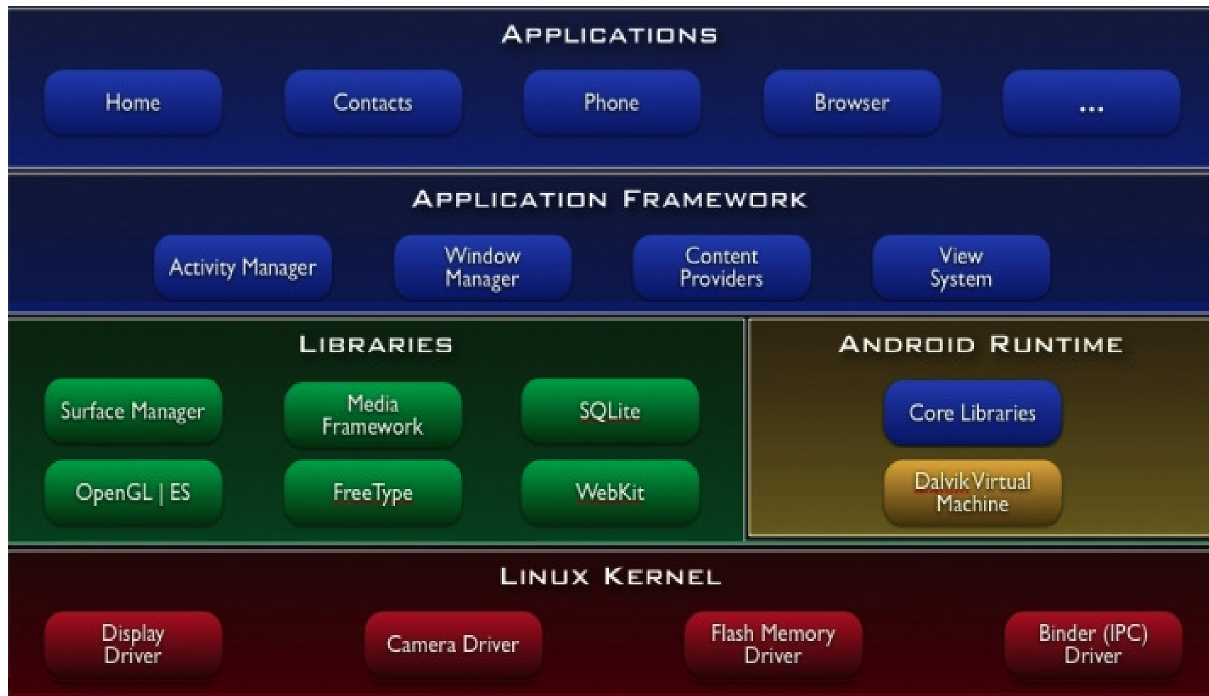
Conforme AMU Team (2010), o ciclo de vida do C2DM pode ser divido em duas etapas:

a) habilitando C2DM: a *APPlication* (APP) do dispositivo registra-se com o Google, obtendo um IDentificador (ID). Após isso, a aplicação envia o ID de registro para o *APPlication Server* (APP SERVER);
b) enviando mensagem: o APP SERVER envia uma mensagem ao servidor *Cloud to Device Message Front End* (C2D MSG FRONTEND). O servidor C2D MSG FRONTEND armazena a mensagem caso o dispositivo móvel esteja inativo, senão a mensagem é enviada ao dispositivo. A APP do dispositivo processa a mensagem recebida através do *CONNection Server* (CONN SERVER).

2.5 PLATAFORMA ANDROID

Segundo Faria (2008), Android é uma plataforma de código aberto para dispositivos móveis criado pela Google em parceria com a *Open Handset Alliance* (OHA). O kit de desenvolvimento provê ferramentas e chamadas via *Application Programming Interface* (API) para o desenvolvimento de aplicativos baseados na linguagem Java. Os principais recursos dessa plataforma móvel são a máquina virtual otimizada, o navegador integrado e a biblioteca SQLite.

A Figura 2 apresenta as camadas da plataforma Android.



Fonte: Faria (2008).

Figura 2 – Camadas da plataforma Android

Conforme Faria (2008), a arquitetura é dividida em cinco camadas, as quais são:

a) Linux *kernel*: responsável pelos serviços e segurança, além dos gerenciamentos de memória, processos, rede e *drivers*;
b) *libraries*: carrega um conjunto de bibliotecas C/C++ utilizadas pelo sistema. Estão incluídas neste conjunto a biblioteca C padrão (`Libc`), além de funções para navegadores web, de aceleração de hardware, renderização 3D e funções de acesso a biblioteca SQLite;
c) Android *runtime*: é uma instância da máquina virtual *Dalvik* criada para as aplicações executadas no Android. A *Dalvik* é uma máquina virtual com melhor desempenho e maior integração com a nova geração de hardware, otimizada para consumo mínimo de memória, bateria e *Central Processing Unit* (CPU);
d) *application framework*: camada onde estão todas as APIs e recursos utilizados pelos aplicativos, como classes visuais e provedor de conteúdo, além dos gerenciamentos de recursos, notificação e pacotes;
e) *applications*: encontram-se todos os aplicativos do Android, como cliente de *e-mail*, contatos, navegador web, entre outros.

2.6 TRABALHOS CORRELATOS

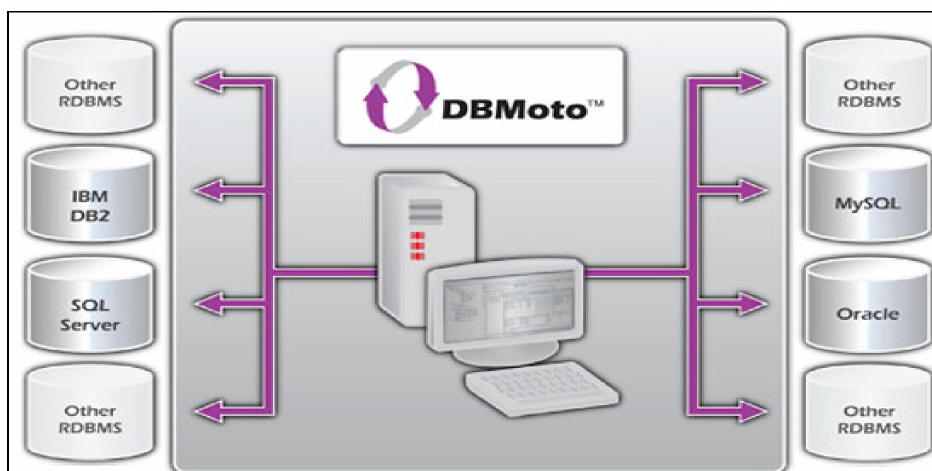
Algumas ferramentas possuem comportamentos semelhantes ao trabalho proposto. Entre elas, citam-se a ferramenta DBMoto (HIT SOFTWARE, 2010) e o *framework* Heros (UCHÔA; MELO, 1999).

2.6.1 DBMoto

O DBMoto apresenta uma solução de replicação de dados para servidores e possíveis estações de trabalho que necessitem contingência de informações (HIT SOFTWARE, 2010).

Para efetuar a replicação cada comando transacional executado no banco de dados de *source* é gerenciado pelo DBMoto, para que o dado possa ser transferido com êxito ao banco de dados *target*. Como o DBMoto opera com replicação entre banco de dados heterogêneos, a ferramenta apresenta um interpretador SQL para traduzir os comandos específicos de cada tipo de banco de dados para o padrão SQL *American National Standards Instituite* (ANSI) (HIT SOFTWARE, 2010).

A Figura 3 ilustra o funcionamento do DBMoto, sendo que as diferentes bases de dados em ambos os lados demonstram a heterogeneidade e o microcomputador ao centro o motor de funcionamento da aplicação, cujo objetivo é interpretar os comandos SQL e comunicar-se com os SGBD.



Fonte: HIT Software (2010).

Figura 3 – Funcionamento do DBMoto

Segundo o HIT Software (2010), a ferramenta apresenta três estados de funcionamento:

a) *refresh*: lê os dados de um SGBD, conforme regras definidas pelo administrador da aplicação e escreve o resultado na base de dados de destino;
b) *mirroring*: refere-se a espelhamento, executando replicação em tempo real, baseando-se nos registros de transações (*logs*);
c) *synchronization*: permite manter as bases de origem e destino sincronizadas.

Além dos estados de funcionamento, a aplicação inclui características como: alta confiabilidade e rápida implementação (HIT SOFTWARE, 2010).

2.6.2 Heros

O Departamento de Informática da Pontifícia Universidade Católica (PUC), Rio de Janeiro, desenvolveu um *framework* para Sistema Gerenciador de Bancos de Dados Heterogêneos (SGBDH), chamado Heros.

> O framework HEROS tem por objetivo possibilitar a instanciação de softwares (SGBDHs) que permitam que um conjunto de sistemas de bancos de dados heterogêneos, cooperativos mas autônomos, sejam integrados de tal forma em uma federação, que consultas e atualizações possam ser realizadas, de forma transparente à localização dos dados, aos caminhos de acesso e a qualquer heterogeneidade ou redundância que exista. (UCHÔA; MELO, 1999, p. 3).

O Heros disponibiliza para comunicação em experiências já comprovadas, componentes que interagem com os protocolos *Remote Procedure Call* (RPC), *Object Request Broker* (ORB), JSON, *socket* e chamada direta. No entanto, podem ser definidos outros tipos de comunicação como o *Distributed Component Model* (DCOM) da Microsoft (UCHÔA; MELO, 1999, p. 5).

Segundo Uchôa e Melo (1999, p. 4), o *framework* Heros é composto por seis *frameworks*:

a) interface usuário: responsável pela comunicação do Heros com o usuário final. Realiza analise sintática das solicitações, detectando erros de sintaxe ou referências inválidas a objetos do esquema;
b) esquema: gerencia as coleções de objetos que podem ser tabelas, índices, visões e procedimentos armazenados;
c) consulta: produz o plano de execução otimizado, com base nos planos de

execuções e de dados adicionais dos SGBDH componentes do Heros;

d) transação: submete consultas ou solicitações de execução dos SGBDH componentes do Heros;
e) comunicação: responsável pela realização física da comunicação entre os SGBDH;
f) regra: responsável por prover ao Heros o comportamento ativo, isto é, a capacidade de reagir automaticamente às ocorrências de eventos.

2.6.3 Tabela comparativa entre os trabalhos correlatos

A Quadro 1 demonstra de forma resumida as principais características entre os trabalhos correlatos, tendo como base os critérios mais importantes extraídos a partir dos conceitos descritos. As informações foram dispostas em colunas, representando na vertical os trabalhos analisados e as linhas apresentam características de cada sistema, indicando semelhanças e/ou diferenças.

características / trabalhos analisados	DBMoto	Heros
heterogeneidade	X	X
diferentes protocolos de comunicação	-	X
replicação através de *refresh*	X	X
replicação através de *mirroring*	X	-
replicação através de *synchronization*	X	-

Quadro 1 – Comparação entre os trabalhos correlatos

Podemos observar que os trabalhos correlatos estudados apresentam a heterogeneidade para replicação entre banco de dados distintos, entretanto apenas o DBMoto apresenta técnicas de replicação mais sofisticadas como o *mirroring* e o *synchronization*.

Apesar dos relatos indicarem que os sistemas apresentados atingem uma alta compatibilidade com os SGDB atuais, nenhum deles apresenta a replicação de dados para as plataformas de dispositivos móveis, como Android e o IOS.

3 DESENVOLVIMENTO DO FRAMEWORK

Este capítulo demonstra o desenvolvimento da ferramenta construída. São apresentados os principais requisitos, especificação, implementação, operacionalidade e os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Para atender os objetivos da utilização da replicação com criptografia simétrica e o uso de *push notification*, os *frameworks* do dispositivo e do servidor foram construídos a partir dos seguintes Requisitos Funcionais (RF):

a) permitir replicar a base de dados SQLite do dispositivo para o servidor e do servidor para o dispositivo (RF01);
b) permitir receber notificações no dispositivo com alerta de dados a serem replicados do servidor (RF02);
c) permitir o envio de notificação do servidor para o dispositivo (RF03);
d) permitir represar dados no dispositivo e no servidor quando os mesmos não tiverem acesso a web (RF04);
e) permitir criptografar dados para serem replicados (RF05);
f) permitir descriptografar dados recebidos (RF06);
g) permitir alterar o tamanho de chave para criptografia (RF07).

Os *frameworks* atendem os seguintes Requisitos Não-Funcionais (RNF):

a) o sistema será desenvolvido na linguagem Java (RNF01);
b) o sistema utilizará a biblioteca SQLite (RNF02);
c) o sistema deve ser compatível com o sistema operacional Android 2.2 ou posterior (RNF03).

3.2 ESPECIFICAÇÃO

A especificação foi realizada a partir da ferramenta Enterprise Architect, utilizando a linguagem de modelagem *Unified Modeling Language* (UML). Para a modelagem foram utilizados casos de uso, diagramas de classe, atividade e seqüência.

3.2.1 Casos de Uso

Nesta sessão são descritos os casos de uso de todos os recursos do *framework*. Foram identificados três atores principais. O primeiro deles, o `Servidor`, que disponibiliza o *listener* para a conexão com o dispositivo. O segundo ator, o `Usuário Dispositivo`, representa o utilizador do sistema, interagindo através do recebimento de notificações. O terceiro ator, o `Sistema utilizador do framework`, faz o uso das ferramentas disponibilizadas para possibilitar o funcionamento da replicação de dados no Android. Na Figura 4 é apresentado o diagrama de casos de uso do *framework*.

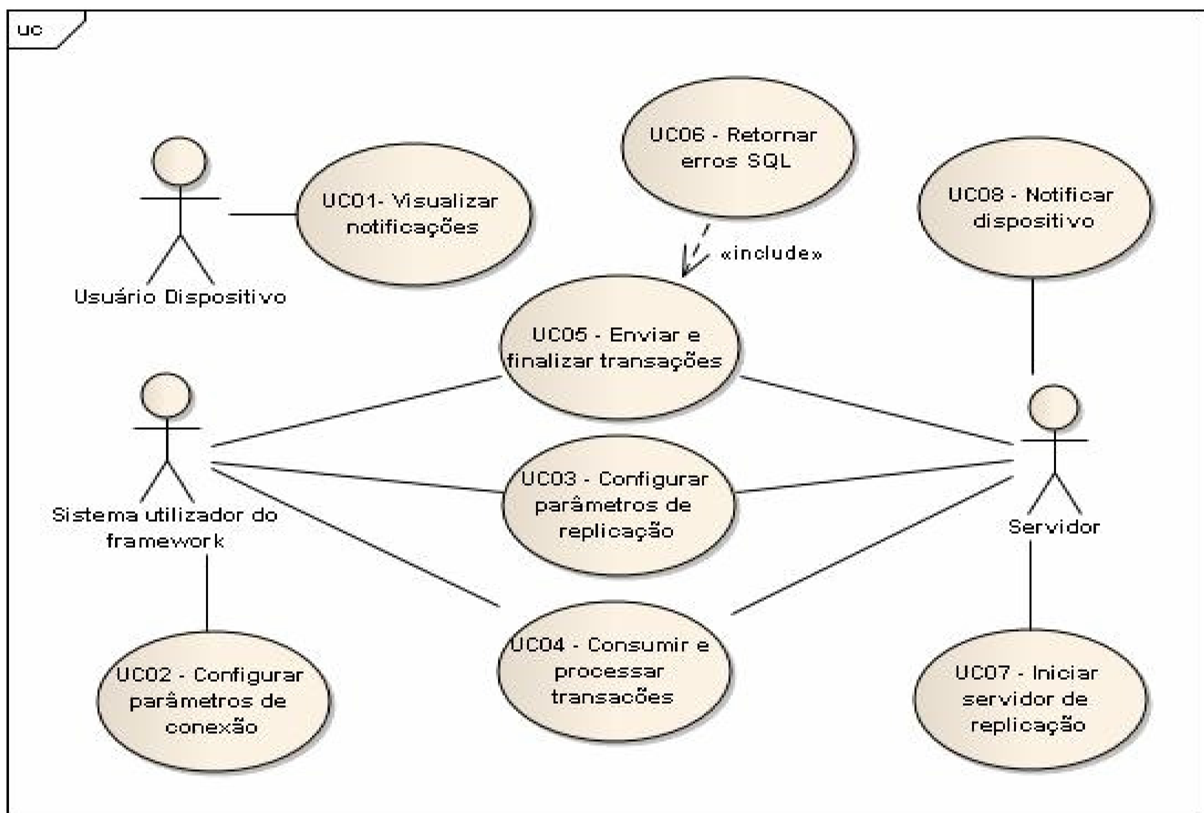


Figura 4 - Diagrama de casos de uso

3.2.1.1 Visualizar notificações

Este caso de uso descreve como o `Usuário Dispositivo` interage com o dispositivo no recebimento de notificações. Detalhes deste caso de uso estão descritos no Quadro 2.

UC01 – Visualizar Notificações	
Descrição	Permite ao `Usuário Dispositivo` visualizar notificações no dispositivo.
Pré-condições	Aplicação com o *framework* deve estar instalada no dispositivo.
Cenário principal	1. Sistema recebe notificações pela web; 2. `Usuário Dispositivo` clica na mensagem recebida pela notificação no menu superior do dispositivo; 3. Sistema processa a notificação recebida iniciando a replicação de dados.
Exceção 01	No passo 1 se o dispositivo apresentar erros de comunicação com a web, o mesmo não receberá a notificação.
Exceção 02	No passo 2 se o `Usuário Dispositivo` clicar no botão `Limpar` nas notificações, o sistema não faz o processamento da solicitação recebida.
Pós-condições	Notificação recebida e encaminhada para replicação de dados.

Quadro 2 - Caso de uso UC01

3.2.1.2 Configurar parâmetros de conexão

Este caso de uso descreve como são configurados os parâmetros de conexão do dispositivo com o servidor. Detalhes deste caso de uso estão descritos no Quadro 3.

UC02 – Configurar Parâmetros de Conexão	
Descrição	Permite ao `Sistema utilizador do framework` configurar parâmetros de conexão como nome do servidor de destino e porta de conexão.
Pré-condições	Dispositivo e `Servidor` devem ter acesso a web.
Cenário principal	1. O `Sistema utilizador do framework` informa nome do `Servidor` e porta de conexão através da classe `ParameterReplication`; 2. Sistema armazena as configurações; 3. O `Sistema utilizador do framework` solicita conexão via *socket* ao `Servidor`.
Exceção 01	No passo 3 se a conexão com o `Servidor` falhar, o *framework* não irá fixar as configurações repassadas.
Pós-condições	Configurações de conexões com o `Servidor` estabelecidas.

Quadro 3 - Caso de uso UC02

3.2.1.3 Configurar parâmetros de replicação

Este caso de uso descreve como são configurados os parâmetros de replicação do *framework*. Detalhes deste caso de uso estão descritos no Quadro 4.

UC03 – Configurar parâmetros de replicação	
Descrição	Permite ao `Sistema utilizador do framework` / `Servidor` configurar a criptografia e o sentido da replicação.
Pré-condições	Aplicação com o *framework* deve estar instalada no dispositivo.
Cenário principal	1. O `Sistema utilizador do framework` / `Servidor` informa a chave de criptografia através da classe `ParameterReplication`; 2. Sistema armazena a chave informada; 3. O `Sistema utilizador do framework` / `Servidor` informa se será configurado como *source* ou *target* na comunicação. 4. Sistema configura a replicação para o modo escolhido
Exceção 01	No passo 1 se a chave de criptografia informada for diferente entre dispositivo e `Servidor`, o sistema não conseguira descriptogrfocar a solicitação recebida.
Exceção 02	No passo 3 se o `Sistema utilizador do framework` e o `Servidor` forem configurados como *source* ou *target*, a replicação de dados não irá funcionar.
Pós-condições	Parâmetros de replicação configurados.

Quadro 4 - Caso de uso UC03

3.2.1.4 Consumir e processar transações

Este caso de uso descreve como as transações são consumidas e processadas durante a replicação de dados. Detalhes deste caso de uso estão descritos no Quadro 5.

UC04 – Consumir e processar transações	
Descrição	Permite ao `Sistema utilizador do framework / Servidor` consumir e processar transações recebidas do banco de dados *source*.
Pré-condições	1. `Servidor` deve estar ativo; 2. Parâmetros de conexão devem estar configurados; 3. Parâmetros de replicação devem estar configurados.
Cenário principal	1. O `Sistema utilizador do framework` no dispositivo solicita o envio da replicação de dados; 2. O *framework* do `Servidor` recebe a solicitação e estabelece a comunicação; 3. O `Sistema utilizador do framework / Servidor` configurado como *source* inicia a transação de replicação; 4. O `Sistema utilizador do framework / Servidor` configurado como *target* recebe a transação e aplica no banco de dados; 5. O `Sistema utilizador do framework / Servidor` configurado como *target* retorna a confirmação da replicação de dados; 6. O `Sistema utilizador do framework / Servidor` configurado como *source* encerra a replicação.
Exceção 01	No passo 4 se ocorrer erros ao aplicar a transação na base *target*, o *framework* irá retornar os registros com problemas para a base *source*.
Exceção 02	No passo 5 se o *framework* não receber resposta do sucesso da replicação, a transação continuará pendente.
Pós-condições	Recebimento da replicação de dados efetuado.

Quadro 5 - Caso de uso UC04

3.2.1.5 Enviar e finalizar transações

Este caso de uso descreve como são enviadas e finalizadas as transações da base de dados *source* para *target*. Detalhes deste caso de uso estão descritos no Quadro 6.

UC05 – Enviar e finalizar transações	
Descrição	Permite ao `Sistema utilizador do framework / Servidor` enviar e finalizar transações.
Pré-condições	1. `Servidor` deve estar ativo; 2. Parâmetros de conexão devem estar configurados; 3. Parâmetros de replicação devem estar configurados.
Cenário principal	1. O `Sistema utilizador do framework / Servidor` executa transação SQL (*insert* , *update* e *delete*); 2. O *framework* adiciona as transações no banco de dados *after-image*; 3. O `Sistema utilizador do framework / Servidor` solicita o inicio da replicação de dados através do método `startReplication()`; 4. O *framework* do dispositivo se conecta no servidor e executa a replicação; 5. O `Sistema utilizador do framework / Servidor` recebe a confirmação da replicação
Exceção 01	No passo 2 se ocorrer algum erro de sintaxe no comando SQL, o *framework* não irá inserir as transações no *after-image*.
Exceção 02	No passo 5 se o *framework* não receber resposta do sucesso da replicação, a transação continuará pendente.
Pós-condições	Envio da replicação de dados efetuado.

Quadro 6 - Caso de uso UC05

3.2.1.6 Retornar erros SQL

Este caso de uso descreve como são retornados os erros SQL ocorridos na replicação de dados. Detalhes deste caso de uso estão descritos no Quadro 7.

UC06 – Retornar erros SQL	
Descrição	Permite ao `Sistema utilizador do framework / Servidor` que sejam retornados os erros SQL da replicação.
Pré-condições	1. `Servidor` deve estar ativo; 2. Parâmetros de conexão devem estar configurados; 3. Parâmetros de replicação devem estar configurados; 4. Transações consumidas e processadas.
Cenário principal	1. O `Sistema utilizador do framework / Servidor` solicita os registros que apresentaram erros através do método `getReplicationError()`; 2. O *framework* retorna os registros que apresentaram erros na replicação; 3. O `Sistema utilizador do framework / Servidor` trata os erros conforme suas regras de negócio.
Pós-condições	Retorna os erros obtidos durante o sincronismo dos bancos.

Quadro 7 - Caso de uso UC06

3.2.1.7 Iniciar servidor de replicação

Este caso de uso descreve como é iniciado o servidor de replicação. Detalhes deste caso de uso estão descritos no Quadro 8.

UC07 – Iniciar servidor de replicação	
Descrição	Permite ao `Servidor` que o *listener* de replicação de dados seja iniciado.
Pré-condições	Parâmetros de replicação devem estar configurados;
Cenário principal	1. O `Servidor` faz a leitura dos parâmetros de replicação; 2. O `Servidor` solicita a ativação do *listener* através do método `startListenerReplication()` ; 3. O *framework* inicia os serviços para replicação de dados.
Pós-condições	Iniciado o servidor de replicação.

Quadro 8 - Caso de uso UC07

3.2.1.8 Notificar dispositivo

Este caso de uso descreve como o `Servidor` notifica os dispositivos. Detalhes deste caso de uso estão descritos no Quadro 9.

UC08 – Notificar dispositivo	
Descrição	Permite ao `Servidor` que sejam notificados os dispositivos.
Pré-condições	1. `Servidor` deve estar ativo; 2. Parâmetros de conexão devem estar configurados; 3. Parâmetros de replicação devem estar configurados.
Cenário principal	1. O `Servidor` solicita o envio de notificações; 2. O *framework* envia a notificação através do método *sendMessage()*; 3. O `Servidor` recebe a confirmação do envio.
Exceção 01	No passo 3 caso ocorra algum erro de comunicação com o servidor da Google no envio da notificações, o *framework* não irá enviar as notificações.
Pós-condições	Envia notificações para os dispositivos.

Quadro 9 - Caso de uso UC08

3.2.2 Diagrama de classe

A seguir serão mostrados os diagramas de classe, primeiramente uma abordagem na

diagramação do *framework* no dispositivo, em seguida o servidor.

3.2.2.1 Diagrama de classes do dispositivo

Neste item serão descritos os pacotes componentes do *framework* desenvolvido para o dispositivo. Na Figura 5 é possível visualizar os pacotes que compõem a aplicação.

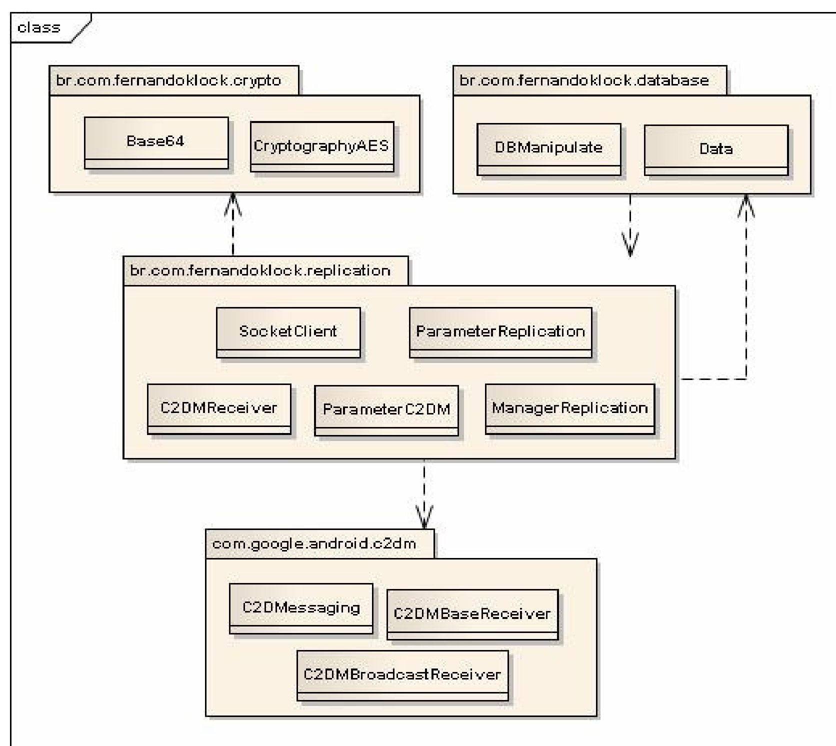


Figura 5 - Diagrama de pacotes do dispositivo

3.2.2.1.1 Pacote `com.google.android.c2dm`

Este pacote é responsável por prover os recursos necessários para o funcionamento do *push*. A Figura 6 mostra com mais detalhes o pacote `com.google.android.c2dm`.

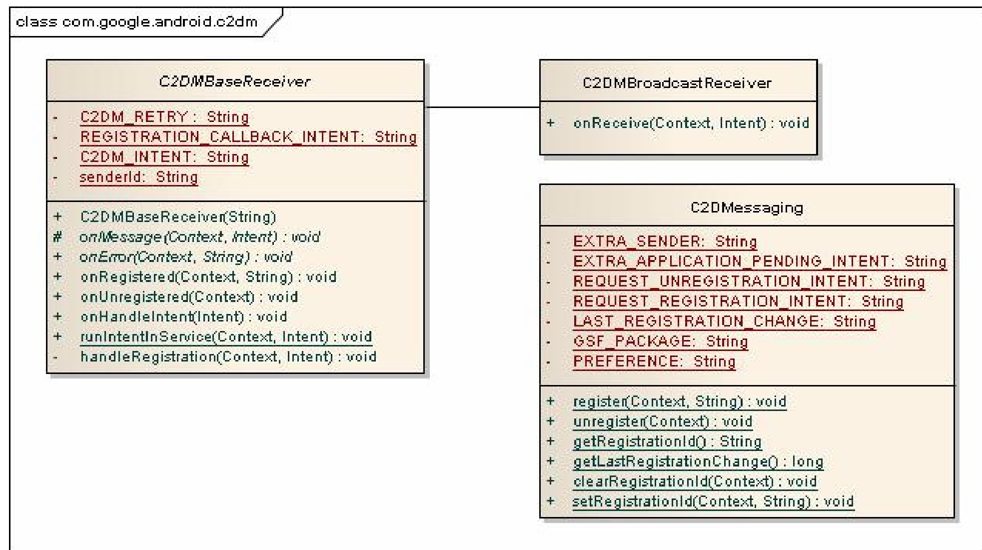


Figura 6 - Diagrama de classe do pacote `com.google.android.c2dm`

A classe `C2DMessaging` é responsável pelo registro da aplicação no servidor Google onde deixa a aplicação elegível a receber notificações externas. Esta solicitação é gerada através do método `register()`, obtendo uma identificação. Para a remoção deste registro no servidor, o utilizador do *framework* deverá invocar o método `unregister()`.

A classe `C2DMBroadcastReceiver` recebe as mensagens C2DM no método `onReceive()`, e repassa a solicitação para o método `runIntentService()` da classe `C2DMBaseReceiver`. Já a classe `C2DMBaseReceiver` é uma classe abstrata responsável por manipular o corpo das mensagens recebidas via *push*, verificando se é uma mensagem de registro, uma mensagem de remoção de registro ou uma mensagem de notificação. Para o recebimento de uma mensagem de registro a classe solicita o método `onRegistered()` e para remoção de registro o `onUnregistered()`, sendo que os mesmos irão executar na classe que herda a `C2DMBaseReceiver`. Caso a mensagem recebida seja apenas de notificação, a classe repassa para o método `onMessage`().

3.2.2.1.2 Pacote `br.com.fernandoklock.replication`

Pacote responsável por sincronizar as bases de dados *source* e *target*. As classes componentes deste pacote fazem a conexão entre o *framework* do dispositivo e do servidor, a replicação de dados entre as bases e parametrização dos serviços utilizados para a replicação. A Figura 7 mostra com mais detalhes o pacote `br.com.fernandoklock.replication`.

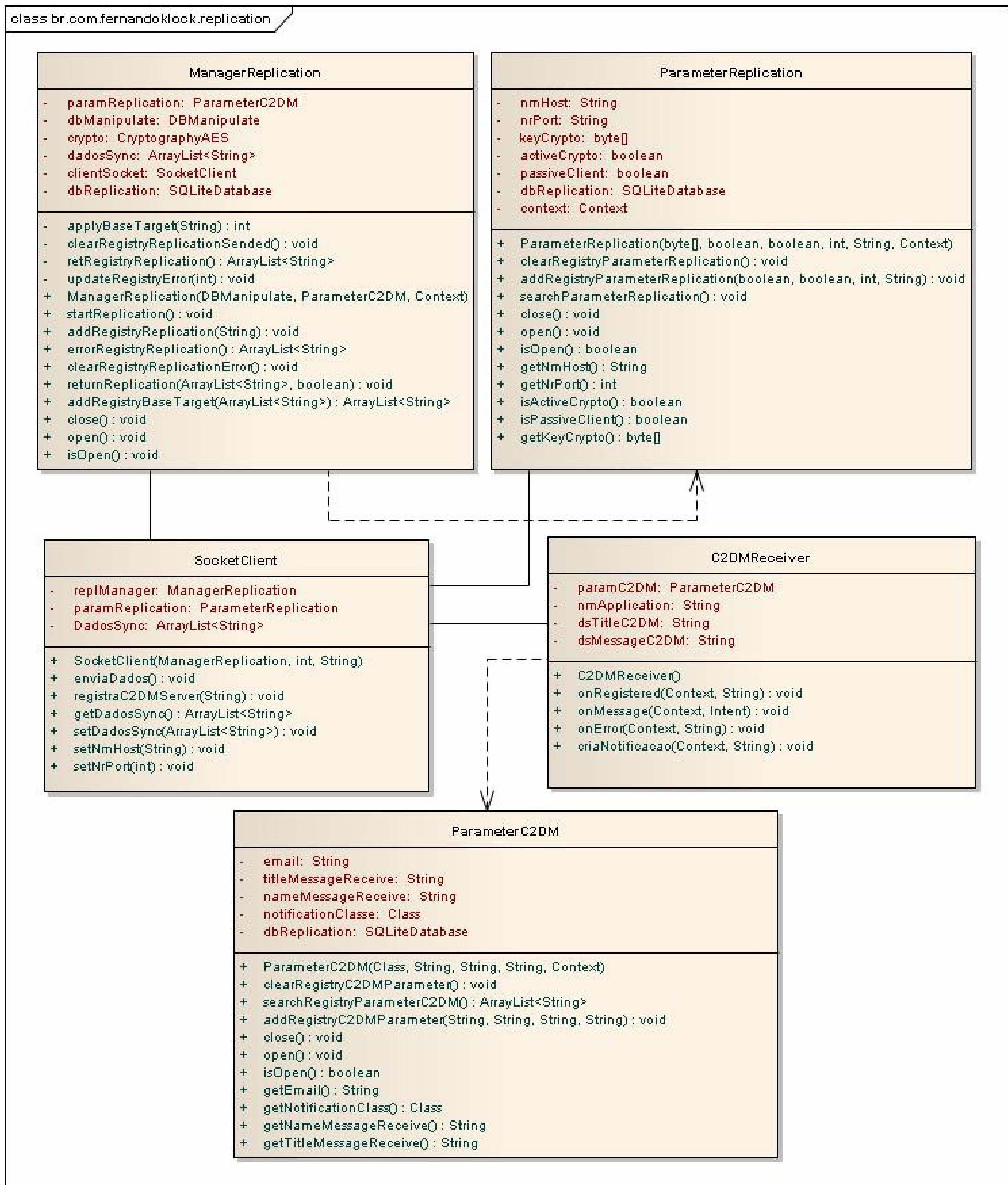


Figura 7 - Diagrama de classe do pacote `br.com.fernandoklock.replication`

A classe `ManagerReplication` faz o controle da replicação de dados e aplica os dados repassados entre o servidor e o dispositivo. Além disso, esta classe armazena todas as transações efetuadas no banco SQLite do dispositivo através do método `addRegistryReplication()`, para que os dados possam ser represados quando o dispositivo não está conectado na web.

A classe `ParameterReplication` é responsável por armazenar as configurações de replicação informadas pelo utilizador do *framework*. Já a classe `SocketClient` faz a conexão

do dispositivo com o servidor para que a replicação de dados seja efetuada. Todos os dados replicados através do *socket* são recebidos da classe `ManagerReplication`.

A classe `C2DMReceiver` é responsável por manipular as mensagens recebidas pelo servidor da Google.

Quando o registro é feito no servidor Google é reservado uma espécie de *ticket* que é enviado para o dispositivo confirmando o registro. Nesse momento a classe `C2DMReceiver` intercepta o registro através do método `onRegistered()` e envia o *ticket* recebido para o servidor, que o mantém para futuras notificações. Além do método `onRegistered()`, a classe herda os métodos `onMessage()`, `onError()` e `onUnregistered()` da classe `C2DMBaseReceiver`. No recebimento da mensagem pelo método `onMessage()` são geradas as notificações, chamando o método `criaNotificacao()`, para que as mesmas possam ser visualizadas pelo usuário da aplicação.

A classe `ParameterC2DM` é responsável por armazenar as configurações para criar a notificação no dispositivo. Os métodos `getNameMessageReceive()` e `getTitleMesageReceive()` armazenam a mensagem e o titulo, respectivamente, da notificação criada.

3.2.2.1.3 Pacote `br.com.fernandoklock.crypto`

Este pacote contém as classes responsáveis por aplicar a criptografia nos dados enviados e recebidos no dispositivo. A Figura 8 mostra com mais detalhes o diagrama de classes do pacote `br.com.fernandoklock.crypto`.

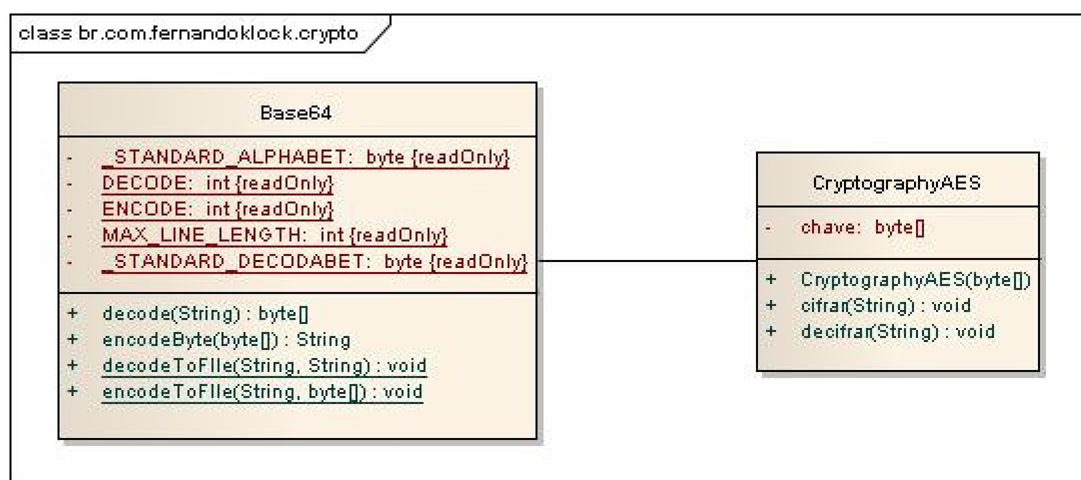


Figura 8 - Diagrama de classe do pacote `br.com.fernandoklock.crypto`

A classe `Base64` é responsável por padronizar a formatação dos dados, transformando

os bytes cifrados e decifrados em texto para ser transmitido.

A classe `CryptographyAES` é responsável por cifrar e decifrar os dados, utilizando o algoritmo de criptografia simétrica chamado de AES.

3.2.2.1.4 Pacote `br.com.fernandoklock.database`

Este pacote é responsável por manipular os dados solicitados pelo utilizador do *framework* no banco de dados da aplicação. A Figura 9 mostra com mais detalhes o pacote `br.com.fernandoklock.database.`

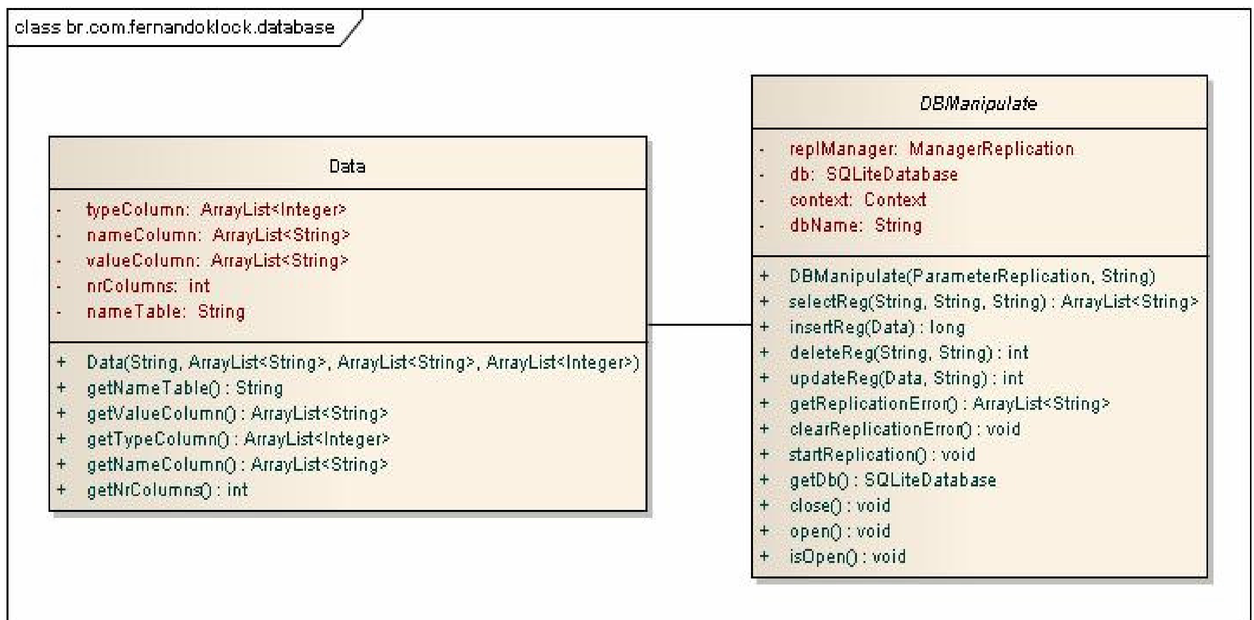


Figura 9 - Diagrama de classe do pacote `br.com.fernandoklock.database`

A classe `Data` é responsável por padronizar os dados que serão repassados para atualização e inserção de registros no banco SQLite. Cada registro da base de dados é representado por um objeto `Data` no *framework*.

A classe `DBManipulate` é uma classe abstrata na qual o desenvolvedor irá utilizar para manipular as *queries* e transações no banco de dados, através dos métodos `selectReg()`, `insertReg()`, `updateReg()` e `deleteReg()`. Além disso, a classe fornece os métodos para iniciar a replicação e recuperar possíveis erros ocorridos durante o sincronismo das bases.

3.2.2.2 Diagrama de classes do servidor

Neste item serão descritos os pacotes componentes do *framework* desenvolvido para o servidor. Grande parte das funcionalidades desenvolvidas no dispositivo estão implementadas no servidor, entretanto sua codificação apresenta algumas variâncias. O pacote `br.com.fernandoklock.crypto` não será abortado nesta sessão, pois as classes são idênticas as desenvolvidas no dispositivo que já foi apresentado anteriormente. Na Figura 10 é possível visualizar os pacotes que compõem a aplicação.

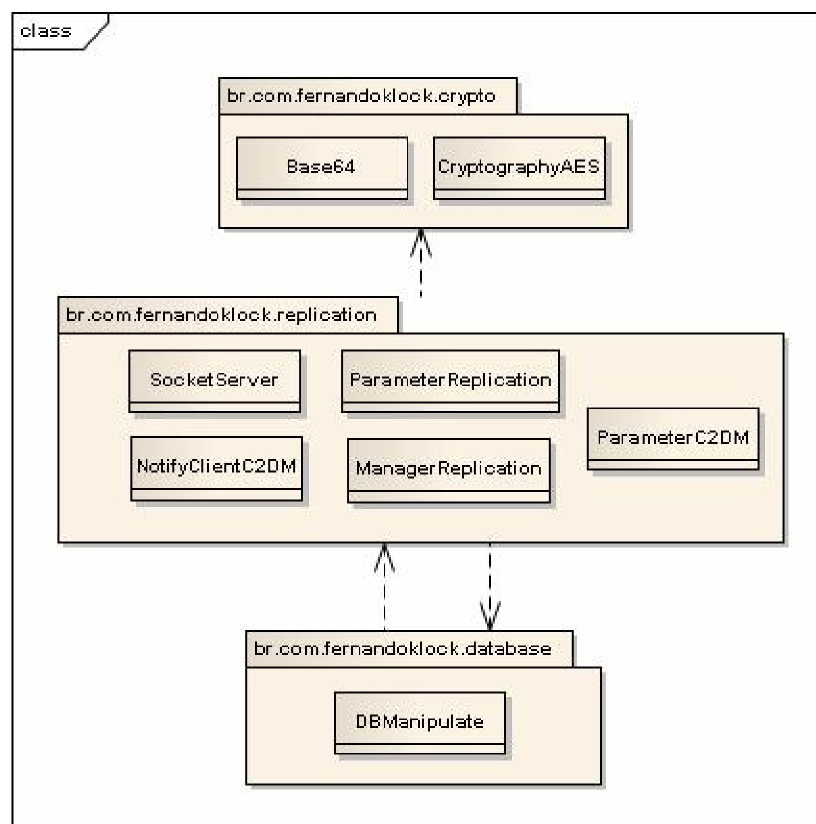


Figura 10 - Diagrama de pacotes do servidor

3.2.2.2.1 Pacote `br.com.fernandoklock.database`

Este pacote é responsável por manipular os dados do banco SQLite do usuário no servidor via *Java DataBase Connectivity* (JDBC).

A Figura 11 mostra com mais detalhes o pacote `br.com.fernandoklock.database`.

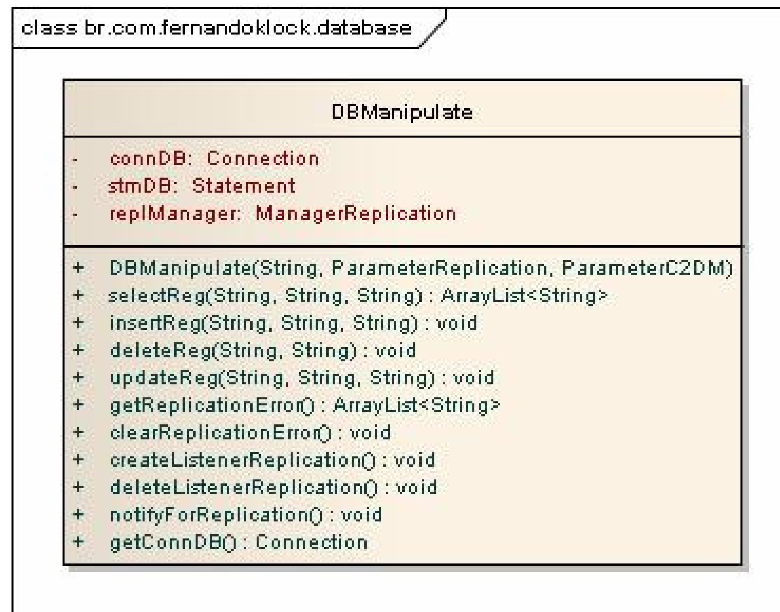


Figura 11 - Diagrama de classe do pacote `br.com.fernandoklock.database`

A classe abstrata `DBManipulate` prove os recursos necessários ao desenvolvedor para acessar a base de dados, através dos métodos `selectReg()`, `insertReg()`, `updateReg()` e `deleteReg()`. Além disso, através dela o desenvolvedor poderá solicitar a abertura do *listener* no servidor, com o método `createListenerReplication()` e solicitar o envio de notificações para o dispositivo, com o método `notifyForReplication()`.

3.2.2.2.2 Pacote `br.com.fernandoklock.replication`

Este pacote é responsável por gerenciar a replicação de dados do servidor, solicitar as notificações aos dispositivos, parametrizar as configurações de replicação e das notificações. A Figura 12 mostra com mais detalhes o pacote `br.com.fernandoklock.replication`.

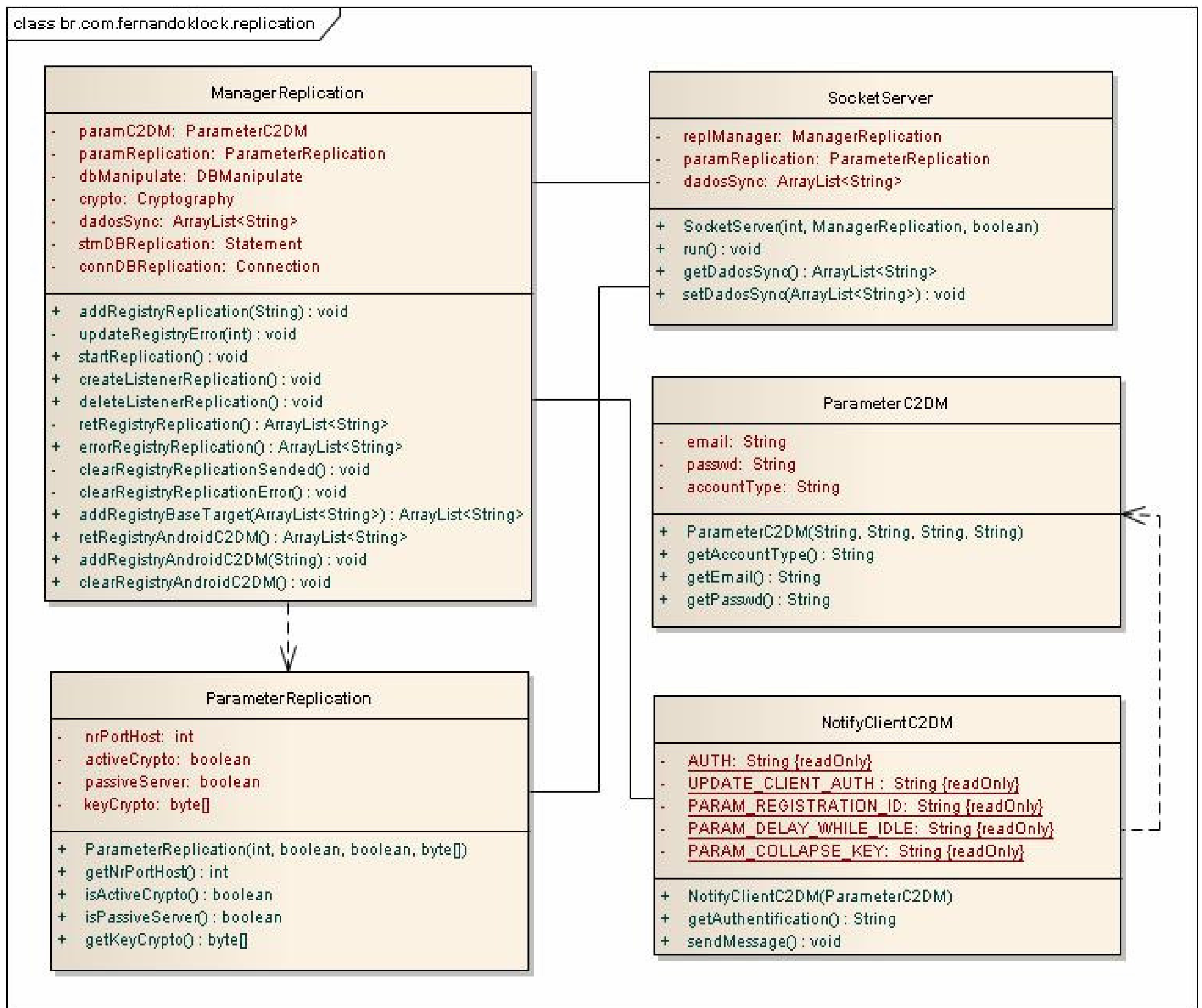


Figura 12 - Diagrama de classe do pacote `br.com.fernandoklock.replication`

A classe `ManagerReplication` faz o controle da replicação de dados através da troca de *tags* entre os *hosts* participantes do sincronismo e aplica os comandos SQL transmitidos nas base de dados. Além disso, a classe gerencia a *thread* que inicia o servidor de replicação, através do método `createListenerReplication()`, possibilitando a conexão do dispositivo para efetivar a replicação de dados. Os métodos `retRegistryC2DM()`, `addRegistryC2DM()` e `clearRegistryC2DM()` manipulam o banco de dados do *framework* para gerenciar o recebimento do *ticket* C2DM, que o dispositivo envia ao servidor para solicitar as notificações. Já a classe `ParameterReplication` é responsável por armazenar as configurações de replicação informadas pelo utilizador do *framework*.

A classe `ParameterC2DM` é responsável por armazenar as configurações para que o servidor possa enviar notificações aos dispositivos. Outra classe relacionada é a `NotifyClientC2DM`, cujo objetivo é notificar os dispositivos com o envio da mensagem via *push*, assim o servidor informa o aplicativo de que há dados a serem sincronizados entre os

bancos. O método `sendMessage()` faz uma requisição HTTP ao servidor da Google.

A classe `SocketServer` é uma *thread* que recebe e envia os dados transferidos do dispositivo para o servidor. Todas as informações recebidas nesta classe são repassadas para a classe `ManagerReplication`, que irá aplicar a requisição solicitada.

3.2.3 Diagrama de atividades

A Figura 13 mostra o diagrama de atividades da rotina completa de notificações do servidor para o dispositivo.

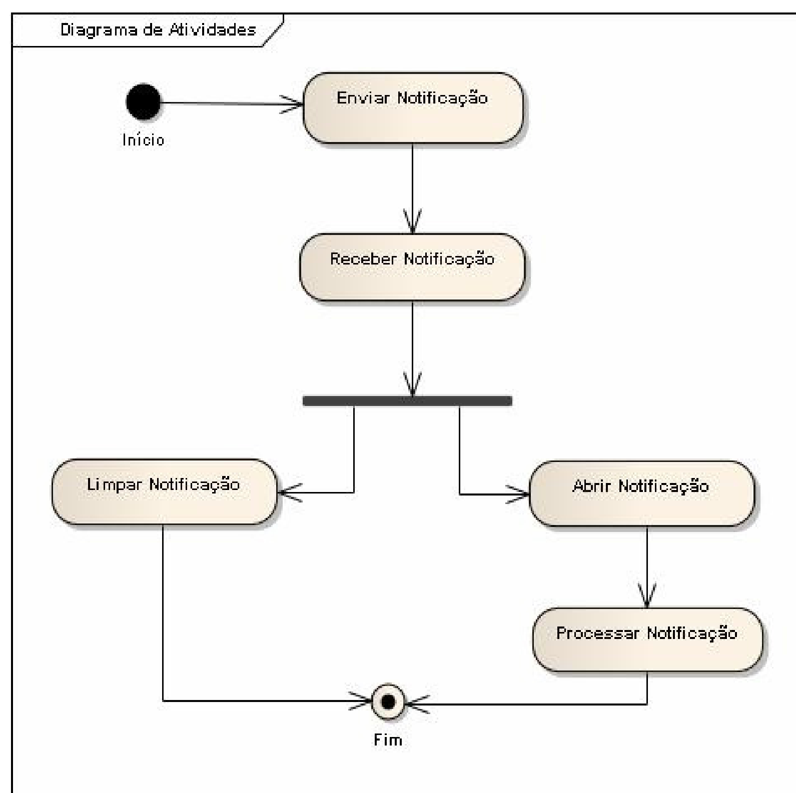


Figura 13 - Diagrama de atividades de notificações

A primeira operação é o envio de notificação por parte do servidor, onde é solicitado ao servidor C2DM da Google a notificação ao dispositivo móvel. Após isto, o dispositivo móvel recebe a notificação e apresenta na aba de notificações do Android o evento. O usuário do dispositivo pode optar em limpar a notificação, finalizando o processo, ou abrir a notificação onde será solicitado o processamento da mensagem para a aplicação responsável pelo evento.

3.2.4 Diagrama de seqüência

O diagrama de seqüência da Figura 14 mostra a interação do `Sistema utilizador do framework` com o *framework* na replicação de dados do dispositivo, nos casos de uso `UC04` e `UC05`.

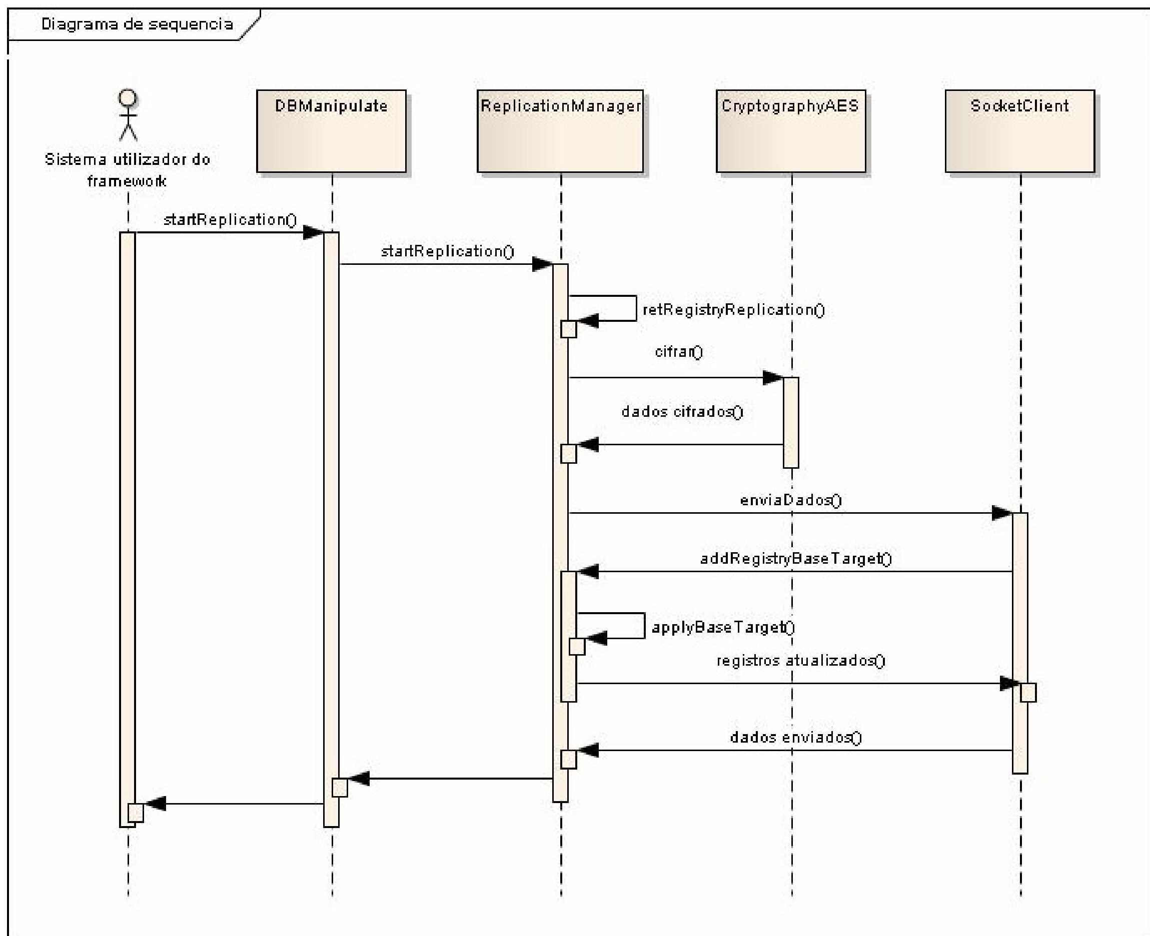


Figura 14 - Diagrama de seqüência da replicação de dados

O sincronismo entre as bases de dados é iniciado pelo `Sistema utilizador do framework` através do método `startReplication()` da classe `DBManipulate`, que faz a interface do desenvolvedor com o *framework*. Após a solicitação da replicação, a classe `RelicationManager` recupera os registros a serem replicados no *after-image*, com o método `retRegistryReplication()`, e solicita a cifragem dos mesmos através do método `cifra()` na classe `CryptographyAES`. Com isso, temos os dados cifrados e prontos a serem enviados via socket.

Como o dispositivo móvel é um cliente *socket*, o envio dos dados é solicitado para a classe `SocketClient`. Todas as chamadas para a comunicação entre os *frameworks* são iniciadas pela classe `ManagerReplication`. O método `enviaDados()` faz o envio dos

comandos SQL ao servidor, já os métodos `addRegistryBaseTarget()` e `ApplyBaseTarget()` são responsáveis por aplicar os registros SQL no banco de dados local utilizado pelo *framework*.

3.3 IMPLEMENTAÇÃO

A seguir são descritas as técnicas e ferramentas utilizadas na implementação, bem como detalhes das principais classes e rotinas implementadas durante o desenvolvimento da aplicação.

3.3.1 Técnicas e ferramentas utilizadas

O desenvolvimento do sistema proposto foi efetuado em duas partes, paralelas e interligadas por um serviço *socket*. As funcionalidades propostas foram desenvolvidas em um aplicativo cliente, a ser executado na plataforma Android. A parte que atende ao servidor foi desenvolvida em Java SE, utilizando um *server socket* para comunicar-se com aplicação do dispositivo.

O desenvolvimento da replicação de dados usando *push notification* foi desenvolvido na linguagem Java com a API de desenvolvimento do Android na versão 2.2, também conhecida por Froyo (ver seção 2.5). O ambiente de desenvolvimento utilizado foi o Eclipse com o conjunto de *plugins* do Android *Development Tools* (ADT). Para a execução e a depuração da aplicação também foi utilizado um dispositivo *smartphone* da fabricante Motorola chamado Milestone que possui o sistema operacional Android Froyo.

Para o *framework* do servidor foi utilizado o banco SQLite através da interface JDBC para os controles internos da aplicação e para utilização do banco de dados local do usuário. Nas requisições *push* foi utilizado o protocolo HTTP com a URL de requisição da Google.

3.3.2 Preparando ambiente Android para *push*

O arquivo `AndroidManifest.xml` possui informações necessárias para as aplicações Android executarem. Para utilizar o serviço de *push notification* da Google, a aplicação deve manter um serviço de registro no servidor da Google. Este serviço indica ao servidor que a aplicação está apta para receber notificações. No Quadro 10 é exibido trecho de código XML que informa pelo *manifest* qual serviço faz o papel de receptor de notificações.

```
<receiver android:name="com.google.android.c2dm.C2DMBroadcastReceiver"
          android:permission="com.google.android.c2dm.permission.SEND">
 <intent-filter>
   <action android:name="com.google.android.c2dm.intent.RECEIVE" /
   <category android:name="br.com.fernandoklock.replication" />
 </intent-filter>
 <intent-filter>
   <action android:name="com.google.android.c2dm.intent.REGISTRATION"/>
   <category android:name="br.com.fernandoklock.replication" />
 </intent-filter>
</receiver>
```

Quadro 10 – Fragmento do manifesto para indicar a classe receptora de notificação

Após indicar para o Android qual será a classe que consome as notificações, deve ser indicado que a aplicação tem permissão para registrar e receber as mensagens, e também que pode manipular a mensagem. O Quadro 11 mostra como são declaradas as permissões no manifesto através das *tags uses-permission*, sendo que cada uma delas representa um tipo de permissão distinta.

```
<!-- Somente esta aplicação pode receber as
     mensagens e registrar o resultado -->
<permission
  android:name="br.com.fernandoklock.replication.permission.C2D_MESSAGE"
  android:protectionLevel="signature"
/>
<uses-permission
  android:name="br.com.fernandoklock.replication.permission.C2D_MESSAGE"
/>
<!--Esta aplicação tem permissão para registrar e
    receber notificações -->
<uses-permission
  android:name="com.google.android.c2dm.permission.RECEIVE"
/>
```

Quadro 11 – Fragmento do manifesto de permissão da aplicação para notificação

Após configurada, a aplicação deve iniciar o processo de registro no servidor C2DM da Google para receber um *ticket* que representa o dispositivo na Google e enviar esse *ticket* ao servidor, assim o servidor pode enviar futuras notificações. O Quadro 11 mostra o trecho

de código onde é iniciado o serviço de registro do dispositivo, sendo que o parâmetro `senderId` representa a conta do utilizador do *framework* na Google e o objeto `Context` representa o pacote no qual a aplicação está rodando.

```
public static void register(Context context, String senderId) {
 Intent registrationIntent = new Intent(REQUEST_REGISTRATION_INTENT);
 registrationIntent.setPackage(GSF_PACKAGE);
 registrationIntent.putExtra(EXTRA_APPLICATION_PENDING_INTENT,
             PendingIntent.getBroadcast(context, 0, new Intent(), 0));
 registrationIntent.putExtra(EXTRA_SENDER, senderId);
 context.startService(registrationIntent);
}
```

Quadro 12 – Fragmento de código para disparo de registro de notificação

Com o serviço de registro iniciado, entra em ação a classe que faz o papel de receptora, `C2DMBroadcastReceiver`, definida no manifesto. Essa classe intercepta pelo método `onReceive()` todos os dados enviados do servidor da Google para aplicação. No Quadro 13 segue o trecho de código que captura as mensagens enviadas para o dispositivo. Todos os dados recebidos são encaminhados para a classe `C2DMBaseReceiver` que faz o tratamento especifico de cada solicitação.

```
public final void onReceive(Context context, Intent intent){
 C2DMBaseReceiver.runIntentInService(context, intent);
 setResult(Activity.RESULT_OK, null /* data */, null /* extra */);
}
```

Quadro 13 - Recebimento de notificação na classe `C2DMBroadcastReceiver`

Após o recebimento das mensagens, a classe `C2DMBroadcastReceiver` faz a chamada do método `onHandleIntent()` na classe `C2DMBaseReceiver`. No Quadro 14 segue o trecho de código que faz o tratamento das mensagens recebidas. Dentro do processamento temos: processamento de notificação recebido através da *tag* `C2DM_INTENT` e processamento de registro de notificação recebido através das *tags* `C2DM_RETRY` / `REGISTRATION_CALLBACK_INTENT`. O registro de notificação é responsável por enviar para o servidor o *ticket* do servidor Google que deve ser usado para envio de notificações.

```
public final void onHandleIntent(Intent intent) {
  try {
       Context context = getApplicationContext();
       if (intent.getAction().equals(REGISTRATION_CALLBACK_INTENT)) {
           handleRegistration(context, intent);
       } else if (intent.getAction().equals(C2DM_INTENT)) {
                 onMessage(context, intent);
       } else if (intent.getAction().equals(C2DM_RETRY)) {
                 C2DMessaging.register(context, senderId);
       }
  } finally {
       mWakeLock.release();
  }
```

Quadro 14 - Tratamento das mensagens recebidas na classe `C2DMBaseReceiver`

O processo de registro retorna um *ticket* que deve ser redirecionado para o servidor que gerará notificações. O processo de envio para o servidor é feito pela classe `SocketClient` no método `registraC2DMServer()` através de uma conexão *socket*.

O consumo de notificações no dispositivo gera um alerta na área de notificações, mostrando que o servidor deseja sincronizar dados com a aplicação. No Quadro 15 é demonstrado o trecho de código de geração de notificação para Android. Através dos objetos `Notification` e `NotificationManager` são repassados os parâmetros necessários para a notificação como o titulo, descrição e frame que deverá ser aberto, representados respectivamente por `dsTitleC2DM`, `dsMessageC2DM` e `nmAplication`.

```
private void criaNotificacao(Context ctx, String evento) {
 NotificationManager notificationManager = (NotificationManager)
                     ctx.getSystemService(Context.NOTIFICATION_SERVICE);
 Notification notificacao = new Notification(R.drawable.icon,
                             this.dsTitleC2DM,System.currentTimeMillis());
 notificacao.flags |= Notification.FLAG_INSISTENT;
 notificacao.flags |= Notification.FLAG_AUTO_CANCEL;

 Class classe = null;
 try {
      classe = Class.forName(this.nmAplication);
 } catch (ClassNotFoundException e) {
          e.printStackTrace();
          return;
 }
 PendingIntent p = PendingIntent.getActivity(ctx, 0, new
                  Intent(ctx.getApplicationContext(),classe), 0);
 notificacao.setLatestEventInfo(ctx, this.dsTitleC2DM,
                                     this.dsMessageC2DM + evento, p);
 notificacao.vibrate = new long[] { 100, 1000, 1000, 1000 };
 notificationManager.notify(R.string.app_name, notificacao);
}
```

Quadro 15 – Geração de notificação Android

3.3.3 Envio de notificações para o dispositivo Android pelo servidor

No servidor foi implementado o envio de notificações, conforme regra da replicação de dados definida para o *framework*, pré-requisito para o envio é possuir o *ticket* do dispositivo. Isto é feito apenas porque o envio é solicitado para um servidor da Google de C2DM. Com o *ticket*, recebido pelo dispositivo, o servidor precisa primeiramente de uma sessão autenticada no servidor Google. No Quadro 16 é demonstrado um trecho de código de captura de sessão com servidor Google de C2DM. Dentro da lista de objetos `NameValuePair` são repassados os parâmetros de autenticação, solicitados via HTTP.

```
public String getAuthentification() {

 HttpClient client = new DefaultHttpClient();
 HttpPost post =
         new HttpPost("https://www.google.com/accounts/ClientLogin");
 try {
      List<NameValuePair> nameValuePairs =
                          new ArrayList<NameValuePair>(1);
      nameValuePairs.add(new BasicNameValuePair("Email",
                          this.paramC2DM.getEmail()));
      nameValuePairs.add(new BasicNameValuePair("Passwd",
                          this.paramC2DM.getPasswd()));
      nameValuePairs.add(new BasicNameValuePair("accountType",
                          this.paramC2DM.getAccountType()));
      nameValuePairs.add(new BasicNameValuePair("source",
                          "Google-cURL-Example"));
      nameValuePairs.add(new BasicNameValuePair("service", "ac2dm"));

      post.setEntity(new UrlEncodedFormEntity(nameValuePairs));
      HttpResponse response = client.execute(post);

      BufferedReader rd = new BufferedReader(
              new InputStreamReader(response.getEntity().getContent()));

      String line = "";
      while ((line = rd.readLine()) != null) {
            if (line.startsWith("Auth=")) {
                return line.substring(5, line.length());
            }
      }
 } catch (Exception e) {
         return "";
 }

 return "";
}
```

Quadro 16 – Requisitando token de autenticação no servidor C2DM.

Com a autenticação recebida, agora o servidor pode mandar uma notificação para o dispositivo. O Quadro 17 mostra o envio de notificações aos dispositivos, sendo que para cada

identificador recebido do dispositivo móvel o servidor faz uma solicitação HTTP com o Google C2DM e o envia através da *tag* PARAM_REGISTRATION_ID.

```
public void sendMessage() throws Exception {
 ArrayList<String> authDispositivos =
                            this.replManager.retRegistryAndroidC2DM();

 if (authDispositivos == null)
     return;

 for (String auth_android : authDispositivos) {
      System.out.println("Started");
      String auth_key = getAuthentification();

      if (auth_key.trim().equalsIgnoreCase("")) {
          throw new Exception("Nao foi possivel autenticar" +
                              " no servidor da google.");
      }

      StringBuilder postDataBuilder = new StringBuilder();
      postDataBuilder.append(PARAM_REGISTRATION_ID).append("=").
                      append(auth_android);
      postDataBuilder.append("&").append(PARAM_COLLAPSE_KEY).
                      append("=").append("0");
      postDataBuilder.append("&").append("data.payload").append("=").
                      append(URLEncoder.encode("Lars war hier", UTF8));

      byte[] postData = postDataBuilder.toString().getBytes(UTF8);

      URL url = new URL("http://android.clients.google.com/c2dm/send");
      HttpURLConnection conn = (HttpURLConnection)url.openConnection();
      conn.setDoOutput(true);
      conn.setUseCaches(false);
      conn.setRequestMethod("POST");
      conn.setRequestProperty("Content-Type",
                              "application/x-www-form-urlencoded");
      conn.setRequestProperty("Content-Length",
                             Integer.toString(postData.length));
      conn.setRequestProperty("Authorization",
                              "GoogleLogin auth=" + auth_key);

      OutputStream out = conn.getOutputStream();
      out.write(postData);
      out.close();
```

Quadro 17 – Envio de notificação push para dispositivo

No envio da notificação o *ticket* é informado no parâmetro PARAM_REGISTRATION_ID e para poder se comunicar respeitando a segurança do servidor da Google, é enviado o token de autorização onde deve ser requisitado junto ao servidor como já explicado anteriormente.

3.3.4 Criptografia simétrica AES

Através da criptografia simétrica AES o *framework* do dispositivo e do servidor cifram e decifram os dados transferidos a fim de estabelecer um padrão de segurança no sincronismo das bases de dados. A chave de criptografia deve ser idêntica no servidor e no dispositivo, sendo que a mesma deve ser informada no momento em que a classe `ParameterReplication` é instanciada. O tamanho da chave de criptografia utilizada pode variar de acordo com o nível de segurança que o desenvolvedor deseja utilizar na aplicação, aplicando ao algoritmo chaves de 128, 192 ou 256 bits. A aplicação utiliza a API *javax.cripto* e não apresenta um mecanismo para transferência das chaves entre os integrantes da replicação.

Ao cifrar ou decifrar algum dado o *framework* utiliza a classe `Base64` para padronizar o texto a ser transferido e recebido, com objetivo de não apresentar problemas com diferentes padrões de *string*, como *Unicode Transformation Format 8-bit* (UTF-8) e *American Standard Code for Information Interchange* (ASCII). O Quadro 18 mostra com mais detalhes a utilização da criptografia simétrica AES na aplicação. Através do objeto `SecretKeySpec` são repassadas a chave de criptografia e a *string* contendo o algoritmo de criptografia que será utilizado. Após isto é instanciado o objeto `Cipher` que fará a cifragem/decifragem dos dados com a chamada do método `doFinal()`.

```
public static String cifrar(String value) throws Exception {
 String retorno = null;
 SecretKeySpec spec = new SecretKeySpec(chave, "AES");
 AlgorithmParameterSpec paramSpec = new IvParameterSpec(new byte[16]);
 Cipher cifra = Cipher.getInstance("AES/CBC/PKCS5Padding");
 cifra.init(Cipher.ENCRYPT_MODE, spec, paramSpec);

 byte[] cifrado = cifra.doFinal(value.getBytes());
 retorno = Base64.encodeBytes(cifrado);
 return retorno;
}

public static String decifrar(String cifra) throws Exception {
 String retorno = null;
 SecretKeySpec skeySpec = new SecretKeySpec(chave, "AES");
 AlgorithmParameterSpec paramSpec = new IvParameterSpec(new byte[16]);

 byte[] decoded = Base64.decode(cifra);
 Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
 cipher.init(Cipher.DECRYPT_MODE, skeySpec, paramSpec);
 retorno = new String(cipher.doFinal(decoded));
 return retorno;
}
```

Quadro 18 - Utilização da criptografia simétrica AES

A mesma implementação de criptografia utilizada no dispositivo móvel foi disponibilizada para o servidor, entretanto é o servidor que define qual o nível de segurança utilizada no canal de comunicação. Caso o *framework* do dispositivo informe um tamanho de chave diferente do servidor, a replicação não poderá ser estabelecida. Esta funcionalidade foi disponibilizada para que o utilizador do *framework* possa garantir um nível de segurança da replicação de dados no dispositivo móvel.

3.3.5 Utilização do *socket* para transferência de dados

Os *frameworks* do dispositivo e do servidor utilizam a tecnologia *socket* para criar o canal de comunicação na replicação dos bancos de dados. O servidor conta com uma *thread* na classe `SocketServer`, responsável por prover os acessos ao dispositivo. Além de receber os dados de sincronismos das bases, o *socket* recebe as requisições de registros *push notification* dos dispositivos, para que o servidor possa enviar as notificações. O Quadro 19 mostra um trecho de código da implementação *socket* no servidor.

```
serverSocket = echoServer.accept();
is = new DataInputStream(serverSocket.getInputStream());
os = new DataOutputStream(serverSocket.getOutputStream());

ArrayList<String> dadosLidos = new ArrayList<String>();
String dadoLido = is.readUTF();

if (dadoLido.equals(this.paramReplication.tamanhoChave())) {
     os.writeUTF("OK");
} else {
     os.writeUTF("NOK");
     is.close();
     os.close();
     continue;
}

dadoLido = is.readUTF();

if ((dadoLido.equals("BeginReplicationSource") &&
      this.paramReplication.isPassiveServer()       ) ||
    (dadoLido.equals("BeginReplicationTarget")  &&
     !this.paramReplication.isPassiveServer())      )      {
     os.writeUTF("OK");
} else {
     os.writeUTF("NOK");
     is.close();
     os.close();
     continue;
}
```

Quadro 19 - Trecho de código da implementação *socket* do servidor

Já no dispositivo a utilização do *socket* funciona como cliente, sendo que o mesmo se conecta no servidor e efetua a transferência dos dados. Durante todas as transferências de dados, o dispositivo e o servidor trocam *tags* para informar o tipo de solicitação de replicação, como por exemplo, a *tag* `BeginReplicationSource` informando que uma requisição para transação de um banco *source* está sendo solicitada. Além disso, na utilização da classe `SocketClient` são geradas exceções através do recebimento das *tags* `NOK`, informando que os *frameworks* estão com a mesma configuração de sincronismo ou que as chaves de criptografias são distintas. O Quadro 20 mostra o trecho de código da implementação *socket* no dispositivo.

```
SocketAddress sockaddr = new InetSocketAddress(addr,
                             this.paramReplication.getNrPortHost());
smtpSocket = new Socket();
smtpSocket.connect(sockaddr, 2000);

os = new DataOutputStream(smtpSocket.getOutputStream());
is = new DataInputStream(smtpSocket.getInputStream());

...

dadoLido = is.readUTF();

if (dadoLido.equals("NOK")) {
    throw new Exception("Chaves de criptografia incompativeis!");
}

if (!this.paramReplication.isPassiveClient()) {
    dadosSync = this.replManager.retDataSync();
    os.writeUTF("BeginReplicationSource");
    dadoLido = is.readUTF();

    if (dadoLido.equals("NOK")) {
        throw new Exception("Server e Client de replicacao"
                            " estão configurados como SOURCE!");
    }

    for (String dadoSync : this.dadosSync) {
         os.writeUTF(dadoSync);
    }

    os.writeUTF("EndOfTransmission");

    ...

    this.replManager.returnReplication(errorReplication, dadosLidos);
}
```

Quadro 20 – Trecho de código da implementação *socket* no dispositivo

3.3.6 Gerenciamento da replicação de dados

O *framework* criado disponibiliza ao desenvolvedor a escolha de qual será o sentido da replicação de dados entre os equipamentos contidos no canal de comunicação *socket*. Tanto o dispositivo móvel como o servidor podem ser utilizados como base *source* e *target*. Quando um dos *frameworks* é configurado para agir como *source* o mesmo habilita as funções de *after-image*, que funciona como uma base de dados de transações já efetivadas durante a utilização do banco local. Ao executar uma rotina de inserção de dados, por exemplo, as informações são armazenadas no banco local e no banco *after-image* da replicação, para que possam ser replicadas na base *target*. Quando o sistema utilizador do *framework* solicita a replicação de dados, o banco *source* faz a leitura dos dados existentes no *after-image* e retorna para a classe `ManagerReplication`, que encaminha as solicitações ao envio por *socket*.

A base de dados *target* somente recebe as transações executadas no banco *source* e as aplica. No caso em que o banco *target* for o dispositivo móvel, os comandos SQL são divididos em *tokens* para serem inseridos no objeto *ContentValues* do Android. Isto se faz necessário, pois os comandos *delete*, *update* e *insert* apresentam modos distintos de implementação. Sendo assim, cada comando enviado via *socket* é analisado e encaminhado para sua respectiva operação no banco de dados. O Quadro 21 mostra a analise para a execução do comando *update*, sendo que cada token da *string* recebida é verificado para montar os parâmetros que serão repassados ao método `update()`.

```
while (tokens.hasMoreTokens()) {

        String token = tokens.nextToken();

        if (Pattern.compile("update", Pattern.CASE_INSENSITIVE)
                                    .matcher(token).find()) {
            continue;
        }

        if (Pattern.compile("set", Pattern.CASE_INSENSITIVE)
                                    .matcher(token).find()) {
            clausula_set = true;
            continue;
        }

        if (Pattern.compile("where", Pattern.CASE_INSENSITIVE)
                                    .matcher(token).find()) {
            clausula_where = true;
            clausula_set = false;
            continue;
        }

        if (clausula_set) {
            set += " " + token;
            continue;
        }

        if (clausula_where) {
            where += " " + token;
            continue;
        }
        tabela = token;
}

ContentValues cv = new ContentValues();
String vet_column_alter[] = set.split(",");

for (int i = 0; i < vet_column_alter.length; i++) {
     String vet_update[] = vet_column_alter[i].split("=");
     cv.put(vet_update[0].trim(), vet_update[1].trim());
}

result = this.dbManipulate.getDb().update(tabela, cv, where,
                                                new String[] {});
```

Quadro 21 - Análise para execução do comando *update*

Caso ocorram erros durante as atualizações dos registros, os mesmos são retornar a base *source* para que possam ser adicionados ao *log* de erros.

3.3.7 Operacionalidade da implementação

Para demonstrar a operacionalidade do *framework* foi desenvolvida uma aplicação de demonstração para o dispositivo móvel e para o servidor, a fim de explorar todas as

funcionalidades disponibilizadas. Serão apresentadas imagens da aplicação no simulador utilizado para o desenvolvimento.

Antes de iniciar o desenvolvimento da aplicação, o desenvolvedor deverá importar sua base de dados no projeto dos *frameworks*, tanto no dispositivo móvel quanto no servidor. Além de sua base de dados, o desenvolvedor deverá copiar também a base de dados `ReplicationDB`, que faz os controles para a replicação de dados. A Figura 15 mostra o local onde as bases de dados deverão estar para o funcionamento do *framework*.

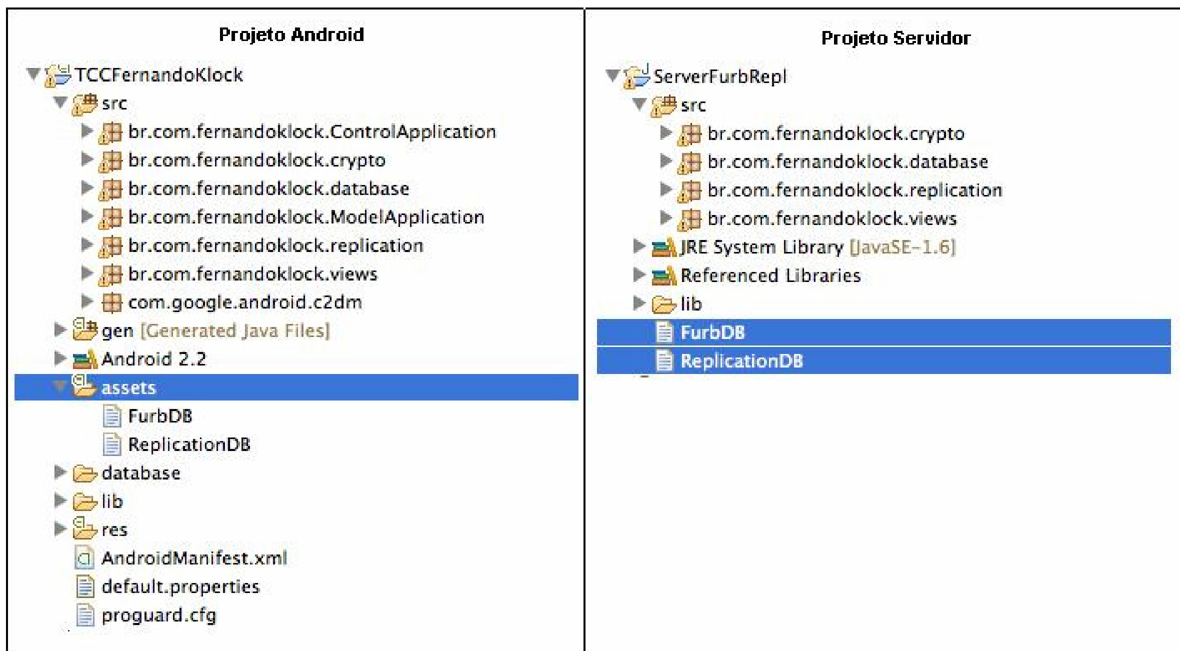


Figura 15 - Localização das bases de dados SQLite no projeto

Para melhor entendimento da utilização do *framework*, será detalhada a implementação realizada na aplicação de teste. Tanto no dispositivo como no servidor o desenvolvedor deverá criar uma classe que irá estender a classe abstrata `DBManipulate`, contendo os métodos necessário para utilização da replicação de dados e da base de dados SQLite. O Quadro 22 mostra a classe `AtualizaDB` criada para estender a classe `DBManipulate`.

```java
public class AtualizaDB extends DBManipulate {

    public AtualizaDB(String dbName, Context context,
            ParameterReplication paramReplication) throws IOException {
            super( dbName,context, paramReplication);
    }

    @Override
    public int deleteReg(String table, String where)
                                    throws Exception {
            return super.deleteReg(table, where);
    }

    @Override
    public long insertReg(Data data) throws Exception {
            return super.insertReg(data);
    }

    @Override
    public int updateReg(Data data, String where) throws Exception {
            return super.updateReg(data, where);
    }

    @Override
    public ArrayList<String> selectReg(String tables, String projection,
                                    String where) throws Exception {
            return super.selectReg(tables, projection, where);
    }
```

Quadro 22 - Classe `AtualizaDB` que estende a classe abstrata `DBManipulate`

Após criar a classe, basta o sistema instanciar as classes `ParameterReplication` e `ParameterC2DM`, para que sejam definidos os parâmetros de utilização do *framework*. Com isto todas as chamadas de métodos utilizados na replicação e manipulação dos dados podem ser feitos através da classe que estende a `DBManipulate`. O Quadro 23 mostra o trecho de código onde são instanciadas as classes para utilização do *framework*.

```
ParameterReplication paramReplication = null;
AtualizaDB db = null;
ParameterC2DM paramC2DM = null;

try {
      paramC2DM = new ParameterC2DM(this, "klockfernando@gmail.com",
                                  "Framework FURBRepl",
                                  "Mensagem do framework FURBRepl",
                                  this.getClass());
} catch (IOException e) {
         e.printStackTrace();
}

try {
      paramReplication = new ParameterReplication(this,
                           "192.168.1.100", 15001, true,
                           true, new byte[] { 0x00, 0x01, 0x02, 0x03,
                                              0x04, 0x05, 0x06, 0x07,
                                              0x08, 0x09, 0x0a, 0x0b,
                                              0x0c, 0x0d, 0x0e, 0x0f });
} catch (IOException e1) {
        e1.printStackTrace();
}

try {
      db = new AtualizaDB("FurbDB", this, paramReplication);
} catch (IOException e) {
        e.printStackTrace();
}
```

Quadro 23 - Instância das classes de parametrização do *framework*

Sabendo como utilizar o *framework* na codificação, agora serão apresentadas as telas da aplicação teste implementada. A aplicação console criada para o servidor será utilizada nesta demonstração como receptora dos dados para o Android e não será apresentada em sua totalidade, pois contém as mesmas funcionalidades utilizadas no dispositivo, com exceção do envio das notificações.

A aplicação no dispositivo foi implementada de forma a possuir uma tela inicial de menu que permite restaurar as configurações padrões do *framework*, verificar o log de erros de transação e entrar no sistema que utiliza o *framework*. No momento que a aplicação é aberta a tela de menu possui três botões, conforme pode ser configura na Figura 16.

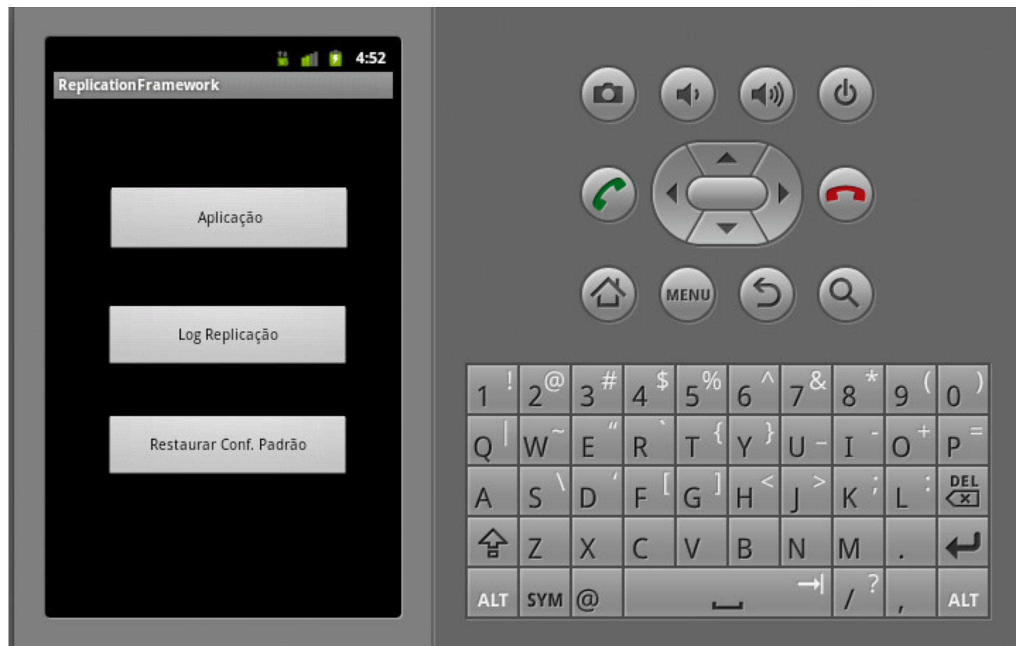


Figura 16 - Tela do menu da aplicação

Quando o botão `Restaurar Conf. Padrão` é pressionado o sistema confirma se o usuário deseja restaurar as configurações de replicação e o banco de dados utilizado pelo sistema. Neste momento o *framework* faz a copia dos bancos que estão no diretório padrão da aplicação, conforme apresentado anteriormente na Figura 26.

Através do botão `Log Replicação` o *framework* apresenta os erros SQL que ocorreram durante a replicação de dados, obtidos através do método `getReplicationError()` da classe `DBManipulate`. Estes erros podem ocorrer quando o dispositivo está configurado como banco de dados *source* e o servidor como *target*. A Figura 17 mostra erros ocorridos durante a replicação de dados.

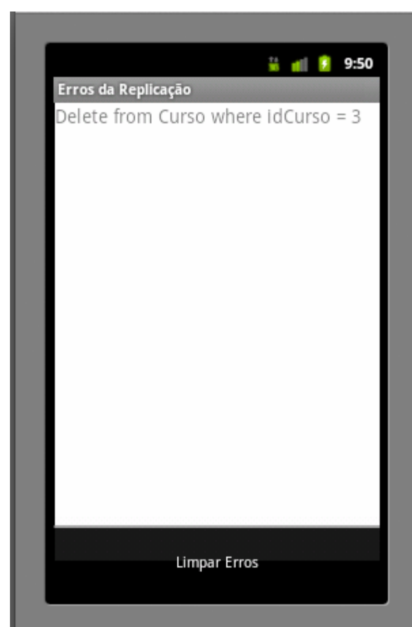


Figura 17 - *Log* de erros da Replicação

O botão `Aplicação` apresenta o sistema utilizador do *framework*, onde são manipulados os dados no banco de dados SQLite. Através desta aplicação as transações são registradas no gerenciador interno de replicação, a fim de replicar os dados. Para manipular os registros o desenvolvedor utilizará os métodos `selectReg()`, `updateReg()`, `deleteReg()` e `insertReg()` da classe `DBManipulate`. A Figura 18 mostra as telas da aplicação criada para utilizar o *framework*.

Figura 18 - Telas da aplicação teste do *framework*

Ao entrar na aplicação podem-se iniciar os testes de replicação com a utilização do *framework*. Como a aplicação trata-se de um sistema de chamada para acadêmicos, é registrada uma presença para um aluno, conforme Figura 19.

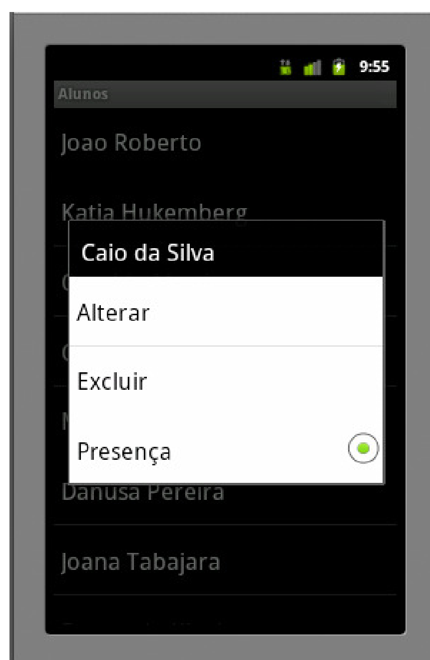


Figura 19 - Registrar presença para um aluno

Após isto, o utilizador do *framework* poderá acionar a replicação de dados através do método `startReplication()`, que estará disponível na classe que irá estender a `DBManipulate`. No caso da aplicação implementada no dispositivo, foi utilizada uma tela de preferências onde são passados todos os parâmetros necessários para o sincronismo, conforme Figura 20. Assim, a classe `ParameterReplication` receberá como parâmetro os dados informados na tela abaixo. Neste caso, a aplicação do dispositivo está configurada como base *source*.

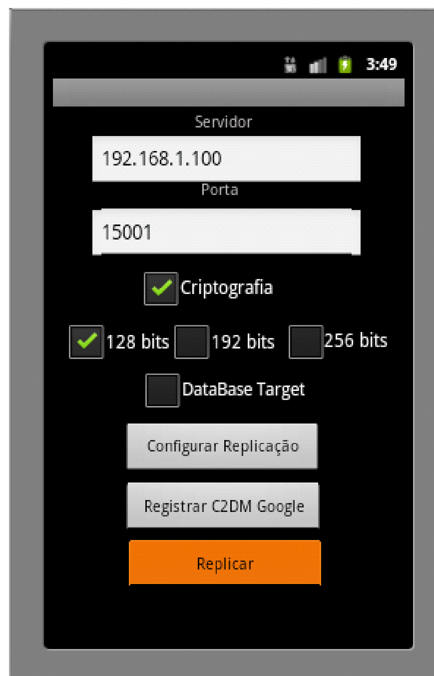


Figura 20 - Tela preferências para replicação de dados

Ao iniciar a replicação no dispositivo o servidor deverá estar com *server socket* habilitado para receber dados. Neste caso, o desenvolvedor da aplicação no servidor deverá iniciar a *thread* que aguarda solicitações de replicação, através do método `createListenerReplication()` da classe `DBManipulate`.

Com a replicação efetuada, o servidor da replicação recebe os dados e registra na sua base de dados. No Quadro 24 pode ser vista a presença registrada para o aluno no servidor, através da aplicação console.

```
 ------ Server FurbRepl ------

1 - Notifica dispositivo
2 - Inicia listener
3 - Lista Cursos
4 - Lista Disciplinas
5 - Lista Alunos
6 - Inserir Curso
7 - Alterar Curso
8 - Excluir Curso
9 - Inserir Disciplina
10 - Alterar Disciplina
11 - Excluir Disciplina
12 - Inserir Aluno
13 - Alterar Aluno
14 - Excluir Aluno
15 - Inserir Aluno em Disciplinas
16 - Alterar Aluno em Disciplinas
17 - Excluir Aluno em Disciplinas
18 - Lista Erros
19 - Limpa Erros
20 - Alterar Sentido da Replicacao (SOURCE / TARGET)
21 - Alterar chave de Criptografia
22 - Listar Alunos com Presença
23 - Limpar Registros C2DM
24 - Sair

Opcao: 22

ID da Disciplina:
4

Aluno           Data da Presença
Caio da Silva ; 16-10-2011
```

Quadro 24 - Registro de presença do aluno no servidor

Nos casos em que o servidor utiliza o *framework* como replicador de dados *source*, existe a opção de notificar os dispositivos alertando que existem dados a serem replicados. Para utilizar esta funcionalidade, o desenvolvedor do aplicativo deverá chamar o método `notifyForReplication()` da classe `DBManipulate` no servidor. Com isso o dispositivo não terá um consumo excessivo de bateria, pois não precisará verificar constantemente possíveis atualizações. A Figura 21 mostra o recebimento da notificação no dispositivo.

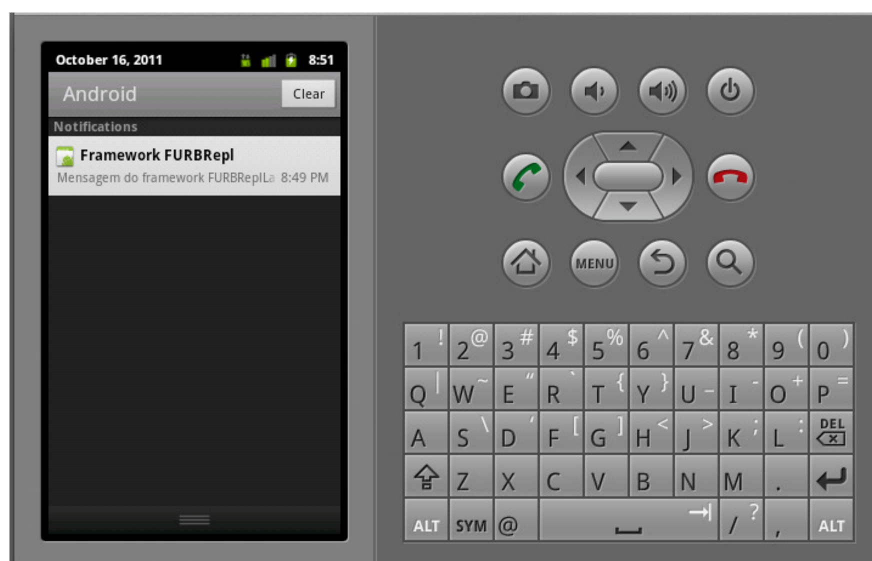


Figura 21 - Recebimento da notificação no dispositivo

3.4 RESULTADOS E DISCUSSÃO

Este trabalho apresentou o desenvolvimento de um *framework* para utilização na replicação de dados com criptografia simétrica e *push notification* em Android. Permitindo através do dispositivo que o desenvolvedor de uma aplicação tenha a possibilidade de transferir ou receber os dados de sua aplicação para um servidor. Também foi desenvolvido um *framework* que será utilizado no servidor, para prover os acessos necessários ao dispositivo.

Para realizar os testes desenvolveu-se um aplicativo que se beneficiou do *framework*, para simular o processo de replicação de dados entre o dispositivo e o servidor. Durante os testes os equipamentos estavam alocados em uma rede privada utilizando o protocolo TCP. A comunicação entre os *frameworks* foi estabelecida através da arquitetura *socket*, sendo necessário abrir uma porta para agir como *listener* no servidor, recebendo os dados enviados pelo dispositivo. Com a porta aberta no servidor o *framework* poderá apresentar uma vulnerabilidade, pois rastreando a mesma cibercriminosos podem, eventualmente, invadir o servidor e obter dados sigilosos. Entretanto, se a infra-estrutura do local onde o servidor está hospedado disponibilizar um serviço de *firewall*[2] este problema pode ser contornado aplicando regras de segurança sobre a rede.

Para obter dados com relação à velocidade da replicação, foram efetuados alguns testes que buscam obter o tempo de sincronismo com a utilização do *framework*. Os dados extraídos com os testes representam todo o fluxo de replicação efetuado por bases de dados *source* e *target*. A Tabela 1 mostra o desempenho da transmissão de dados no dispositivo.

Tabela 1 – Desempenho da transmissão de dados no dispositivo

DESEMPENHO DA TRAMISSÃO DE DADOS (EM SEGUNDOS)				
Quantidade de registros replicados	**Base *source* sem criptografia**	**Base *source* com criptografia 128 bits**	**Base *target* sem criptografia**	**Base *target* com criptografia 128 bits**
100	0,18	1,03	3,31	4,71
500	1,96	12,70	16,82	20,22
1000	3,61	22,34	35,63	41,51
3000	9,96	27,12	100,32	122,67

Após a execução dos testes com volumes de dados distintos verificou-se que a maior perda de processamento está vinculada á manipulação dos registros dentro do dispositivo

móvel. Nos casos em que o aparelho utiliza a replicação como banco de dados *target*, o tempo total de sincronismo é alto, pois a execução de transações dentro do banco SQLite no Android é lenta. Entretanto, quando o banco de dados está configurado como *source* e com a criptografia habilitada, o maior custo de processamento está vinculado à criptografia, sendo que todos os comandos SQL enviados e recebidos são cifrados e decifrados. A Figura 22 demonstra graficamente o desempenho da replicação de dados.

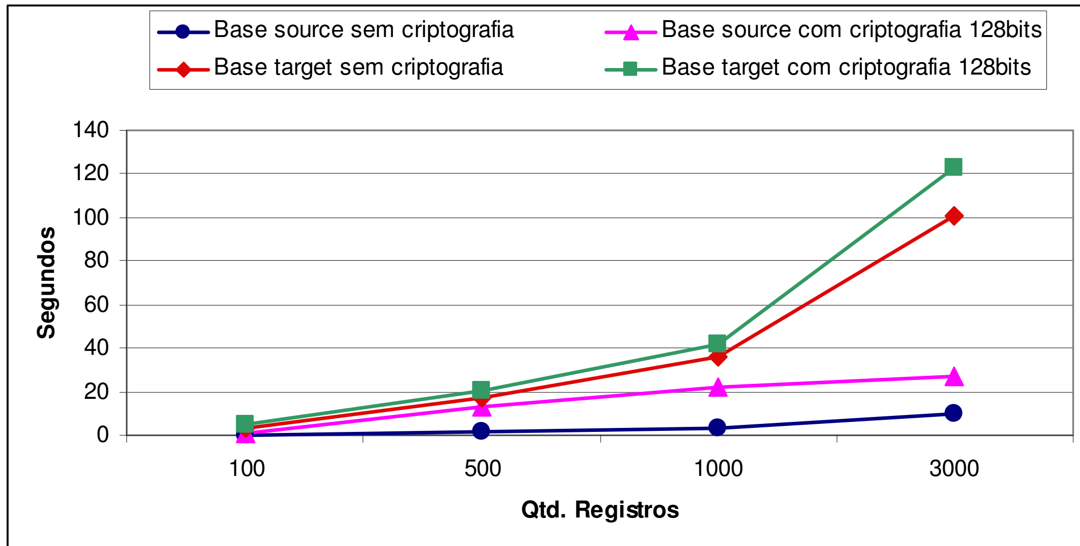


Figura 22 - Desempenho da replicação de dados

O trabalho desenvolvido tinha como objetivo inicial disponibilizar a arquitetura REST como padrão de comunicação entre o servidor e o dispositivo. Para isto, seria necessário criar um servidor *Service Oriented Architecture* (SOA) dentro do *framework*, para prover os recursos de acesso do dispositivo. Cada requisição enviada para o serviço necessitaria de um retorno, pois caso algum registro apresentasse problema ao ser aplicado em uma das bases, o *framework* necessitaria de um registro no seu *log* de erros da replicação. Sendo assim, várias requisições POST ou GET seriam transferidas entre os equipamentos, podendo apresentar uma lentidão na replicação. Com a implementação *socket* apresentada, uma conexão é efetuada fazendo com que o dispositivo transfira e receba todos os dados necessários, sem a espera de requisições *web*.

O utilização da criptografia para obter a segurança da informação nos dados replicados apresentou dificuldades na implementação. Os dados replicados via *socket* apresentavam problemas no momento de serem decifrados, pois o dado transmitido não tinha um formato padrão, como UTF-8 e ASCII. Para resolver este problema foi disponibilizada a classe

---

[2] *Firewall* é uma barreira de proteção que controla o tráfego de dados entre computadores e a internet (ALECRIM, 2004).

`Base64`, que transforma as informações em um texto padrão para que não ocorram problemas na utilização da criptografia.

O trabalho proposto tinha como objetivo inicial disponibilizar a criptografia simétrica utilizando apenas chaves de 128 bits. Entretanto, o *framework* fornece a possibilidade do desenvolvedor escolher sua chave de criptografia variando o tamanho entre 128, 192 e 256 bits. Quanto maior for tamanho da chave utilizada maior será o nível de segurança, contudo o desempenho do dispositivo será comprometido, pois o custo computacional de processamento será maior. A Tabela 2 mostra o desempenho no uso da criptografia simétrica no dispositivo.

Tabela 2 – Desempenho da criptografia simétrica

DESEMPENHO DA CRIPTOGRAFIA SIMÉTRICA (EM SEGUNDOS)			
Quantidade de registros criptografados	**Chave de 128 bits**	**Chave de 192 bits**	**Chave de 256 bits**
100	0,64	0,76	0,82
500	4,91	5,57	6,73
1000	10,24	12,36	14,54
3000	19,71	22,01	26,89

Mesmo isolando o algoritmo de criptografia para criar as estatísticas, foi verificado que o mesmo pode sofrer impacto no seu desempenho, pois existem aplicações/instruções que estão rodando em *background* no Android.

A diferença de tempo entre os diferentes tamanhos de chaves da criptografia apresentou uma perda considerável, porém percebe-se que quanto maior o volume de dados maior é a perda no desempenho. Replicar grandes volumes de dados com criptografia de 256 bits pode ser inviável, dependendo do nível de agilidade que se deseja ter na aplicação que irá utilizar o *framework*. Essa lentidão pode ser vista na replicação com cinco mil registros, na qual apresenta uma diferença de, aproximadamente, sete segundos comparando 128 com 256 bits. Sendo assim, a melhor prática é enviar constantemente pequenas transações, pois são mais ágeis e mais confiáveis de serem submetidas. A Figura 23 demonstra graficamente a utilização da criptografia simétrica.

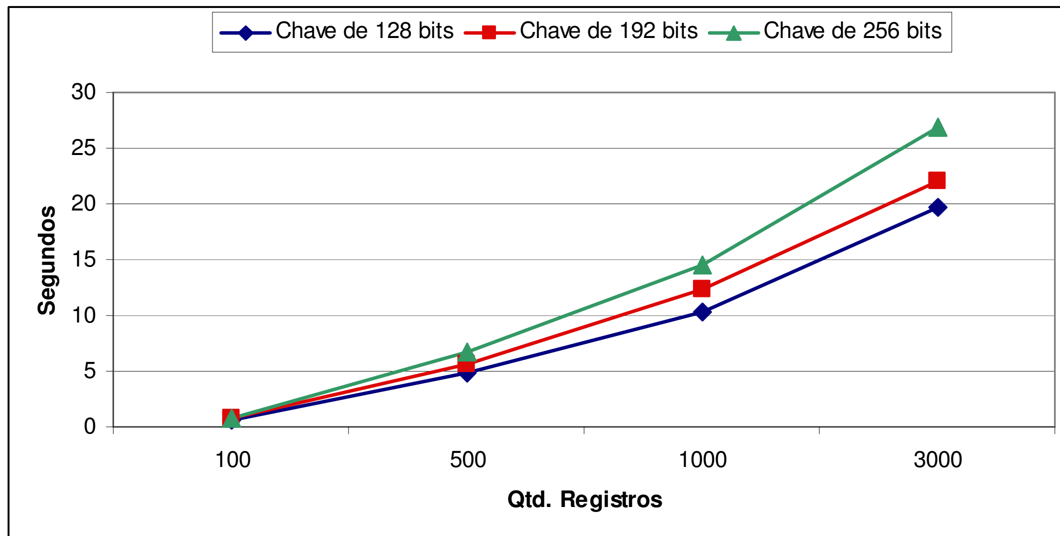


Figura 23 - Desempenho da criptografia simétrica

Outro grande problema encontrado foi na manipulação dos registros para o banco SQLite no Android. Cada comando transacional SQL executado no banco, como por exemplo um *update*, apresenta um modo distinto de implementação. Quando a base de dados do servidor envia um comando SQL para o dispositivo o mesmo é divido em *tokens*, para que assim os dados possam ser adicionados dentro de um objeto *ContentValues*, que manipula as colunas no SQLite. Sendo assim, no *framework* do Android foi necessário disponibilizar para classe `ManagerReplication` um método privado chamado `applyBaseTarget()`, cujo objetivo é traduzir os comandos SQL.

Este trabalho também teve como desafio a utilização do *push notification* para os dispositivos. Sendo limitado o uso de *push*, somente para aparelhos que possuíssem a versão 2.2 do Android ou posterior. Apesar de depender de servidor externo da Google, o *push* se demonstrou muito seguro na entrega da notificação ao dispositivo. Todo o teste de solicitação de notificação enviado pelo *framework* do servidor foi entregue com sucesso ao dispositivo, facilitando as atualizações no banco de dados do aparelho.

O estudo em questão não disponibilizou estatísticas quanto à utilização de memória e processamento na execução do *framework*. Entretanto, estes dados ajudariam a demonstrar o quão performático seria a implementação de uma aplicação que irá se beneficiar do *framework* criado.

4 CONCLUSÕES

Devido a grande adesão de dispositivos móveis e a utilização dos mesmos em operações que necessitem de sigilo da informação, como a compra de produtos, a segurança nestes aparelhos tornou-se uma ponto crucial para o desenvolvimento de softwares. Buscando diminuir a vulnerabilidade nestes dispositivos, desenvolveu-se um *framework* de replicação de dados que apresenta características necessárias para garantir a segurança na comunicação cliente/servidor.

O *framework* foi desenvolvido para a plataforma Android e tem como principal objetivo a replicação de dados entre um dispositivo móvel e um servidor, sendo que para isto foi utilizada a biblioteca SQLite como repositório de dados. A replicação conta com o uso da criptografia simétrica, possibilitando uma transmissão de dados com sigilo. Outro item importante é a utilização do *push*, pois se ocorrer alguma atualização na base de dados do servidor o dispositivo será notificado para que haja a transmissão de dados. Esta solução faz com que o dispositivo móvel gaste menos processamento e bateria, pois o mesmo não precisa ficar requisitando o servidor para verificar se existem novas atualizações.

O presente trabalho demonstra também a utilização da plataforma Android como cliente de uma arquitetura cliente-servidor. Para isso foi utilizado um servidor *socket* que recebe as requisições do dispositivo, estabelecendo a replicação de dados entre os equipamentos. O sincronismo entre as bases de dados pode ocorrer tanto do servidor para o dispositivo como do dispositivo para o servidor. Isto é possível devido à utilização de *tags* que são enviadas entre os *frameworks*, informando qual ação deverá ser escolhida após o recebimento da requisição. Sendo assim, o *framework* configurado para ser base *target* irá receber atualizações e o configurado como base *source* irá enviar atualizações. O trabalho proposto não faz o sincronismo em tempo real das bases de dados, pois isso causaria um desgaste excessivo de bateria e de processamento do aparelho.

Verificou-se a existência de aplicações que são correlatas ao trabalho desenvolvido, entretanto as ferramentas DBMoto e Heros não disponibilizam replicação para dispositivos móveis. Foram incorporadas funcionalidades básicas existentes nos trabalhos correlatos, como a implementação do *refresh* entre bases apresentado no DBMoto.

4.1 EXTENSÕES

Como sugestões de extensões para a continuidade do presente trabalho, tem-se:

a) disponibilizar a segurança com a utilização de criptografia assimétrica;
b) disponibilizar diferentes algoritmos de criptografia para que o utilizador do *framework* possa escolher. O *framework* desenvolvido disponibiliza somente o algoritmo AES;
c) disponibilizar a replicação de dados entre bancos heterogêneos;
d) disponibilizar um algoritmo de sincronismo entre os bancos de dados de modo a garantir que não haja erros na replicação;
e) disponibilizar replicação de dados entre dispositivos móveis, sem o uso do servidor;
f) disponibilizar a replicação de dados através de Objetos Java entre os *framework*. O *framework* desenvolvido disponibiliza a replicação através de *strings* que contém os comandos SQL;
g) disponibilizar métodos para extrair estatísticas quanto a utilização de memória e processamento no dispositivo Android.

REFERÊNCIAS BIBLIOGRÁFICAS

ALECRIM, Emerson. **Criptografia**. [S.l.], 2005. Disponível em: <http://www.infowester.com/criptografia.php>. Acesso em: 17 mar. 2011.

______. **Firewall**: conceitos e tipos. [S.l.], 2004. Disponível em: <http://www.infowester.com/firewall.php>. Acesso em: 13 nov. 2011.

AMU TEAM. **Android 2.2 cloud to device messaging**. [S.l.], [2010]. Disponível em: <http://www.androidmeup.com/articles/android-22-froyo-cloud-to-device-messaging-c2dm>. Acesso em: 21 mar. 2011.

BERENGUEL, André L. A. et al. Arquitetura AAA em sistema web baseados em REST. **Global Science and Tecnology**, Rio Verde, v. 1, n. 1, p. 1–7, dez./mar. 2008. Disponível em: <www.cefetrv.edu.br/periodicos/index.php/gst/article/download/10/3>. Acesso em: 30 mar. 2011.

CARVALHO, Daniel B. **Segurança de dados com criptografia**: métodos e algoritmos. 2. ed. Rio de Janeiro: Book Express, 2001.

CODE. **Android cloud to device messaging framework**. [S.l.], [2011a?]. Disponível em: <http://code.google.com/intl/pt-BR/android/c2dm/>. Acesso em: 20 mar. 2011.

______. **Limitations.** [S.1.], [2011b?]. Disponível em: <http://code.google.com/intl/pt-BR/android/c2dm/#limitations>. Acesso em: 24 mar. 2011.

EMIDIO, Jonatas. **Android cloud to device messaging.** [S.l.], [2010]. Disponível em: <http://androidpack.blogspot.com/2010/09/c2dm-android-cloud-to-device-messaging.html>. Acesso em: 17 mar. 2011.

FARIA, Alessandro O. **Programe seu Android.** [S.l.], [2008]. Disponível em: <http://www.linuxmagazine.com.br/images/uploads/pdf_aberto/LM43_73-77.pdf>. Acesso em: 19 mar. 2011.

FRANCO, Carla E. C.; SILVA, Daniel. Aplicação de socket em java para monitoramento de processos em estações de trabalho. In: SIMPÓSIO DE EXCELÊNCIA EM GESTÃO E TECNOLOGIA, 7., 2005, Resende. **Anais eletrônicos...** Resende: Dom Bosco, 2005. p. 1-8. Disponível em: <http://www.aedb.br/seget/artigos05/342_ARTIGO_SeGET%20(19.09.2005).pdf>. Acesso em: 05 nov. 2011.

HIT SOFTWARE. **Replicação de dados.** [S.l.], [2010]. Disponível em: <http://www.hitsw.com/localized/portuguese/products_services/dbmoto/dbmoto.html>. Acesso em: 21 mar. 2011.

HOPKINS, Matt; THOMAS, Dunstan. **Replicação de dados com InterBase/Firibird.** [S.l.], [1998]. Disponível em: <http://www.comunidadefirebird.org/cflp/downloads/CFLP_T022.PDF>. Acesso em: 18 mar. 2011.

INFOMONEY. **Dispositivos móveis serão principais alvos de ataques em 2011.** [S.l.], [2011]. Disponível em: <http://economia.uol.com.br/ultimas-noticias/infomoney/2011/01/05/dispositivos-moveis-serao-principais-alvos-de-ataques-virtuais-em-2011.jhtm>. Acesso em: 17 mar. 2011.

LANE, Dale. **Push notifications for mobile.** [S.l.], [2009]. Disponível em: <http://dalelane.co.uk/blog/?p=938>. Acesso em: 18 fev. 2011.

MACORATTI, José C. **Usando a comunicação Cliente – Servidor com sockets.** [S.l.], [2004]. Disponível em: <http://imasters.com.br/artigo/2414/dotnet/usando-a-comunicacao-cliente-servidor-com-sockets>. Acesso em: 20 out. 2011.

NUNES, Délio S. **Criptografia simétrica.** [S.l.], [2007]. Disponível em: <http://www.gta.ufrj.br/grad/07_2/delio/Criptografiasimtrica.html>. Acesso em: 12 maio 2011.

NUNES, Sergio; DAVID, Gabriel. **Uma arquitectura web para serviços web.** Porto, [2005]. Disponível em: <http://repositorio-aberto.up.pt/handle/10216/281>. Acesso em: 31 mar. 2011.

SQLITE. **About SQLite.** [S.l.], [2009?]. Disponível em: <http://www.sqlite.org/about.html>. Acesso em: 18 mar. 2011.

TECNOLOGIADAREDE. **Algoritmos de criptografias**. [S.l], [2010]. Disponível em: < http://tecnologiadarede.webnode.com.br/news/noticia-aos-visitantes/>. Acesso em: 16 dez. 2011.

UCHÔA, Elvira M. A.; MELO, Rubens N. Integração de sistemas de bancos de dados heterogêneos usando frameworks. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 14., 1999, Florianópolis. **Anais eletrônicos...** Florianópolis: UFSC, 1999. p. 1-13. Disponível em: <http://www.inf.ufsc.br/sbbd99/anais/SBBD-Completo/32.pdf>. Acesso em: 15 mar. 2011.