UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SYNCEASY – APLICATIVO PARA SINCRONIZAÇÃO DE ARQUIVOS ENTRE DISPOSITIVOS MÓVEIS E COMPUTADORES UTILIZANDO METADADOS

BERNARDO MARQUARDT MÜLLER

BLUMENAU
2011

2011/1-07

BERNARDO MARQUARDT MÜLLER

SYNCEASY – APLICATIVO PARA SINCRONIZAÇÃO DE ARQUIVOS ENTRE DISPOSITIVOS MÓVEIS E COMPUTADORES UTILIZANDO METADADOS

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. Mauro Marcelo Mattos, Dr. - Orientador

BLUMENAU
2011

2011/1-07

SYNCEASY – APLICATIVO PARA SINCRONIZAÇÃO DE ARQUIVOS ENTRE DISPOSITIVOS MÓVEIS E COMPUTADORES UTILIZANDO METADADOS

Por

BERNARDO MARQUARDT MÜLLER

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

Presidente:	Prof. Mauro Marcelo Mattos, Dr. – Orientador, FURB
Membro:	Prof. Francisco Adell Péricas, Prof. – Mestre, FURB
Membro:	Prof. Paulo Fernando da Silva, Prof. – Mestre, FURB

Blumenau, 28 de junho de 2011.

Dedico este trabalho à minha namorada por sempre estar ao meu lado, meus pais e meus irmãos por sempre me apoiarem e aos meus amigos pela grande amizade e incentivo.

AGRADECIMENTOS

Aos meus pais por sempre me apoiarem e nunca duvidarem da minha capacidade.

À minha namorada, pelas cobranças, por sempre estar ao meu lado, mesmo nos momentos mais difíceis. Por ser a pessoa que sempre me incentivou e que foi determinante para a conclusão deste trabalho.

Aos meus grandes amigos que sempre estiveram ao meu lado dando um empurrão a mais.

A minha líder Patrícia, pela compreensão e apoio.

Ao meu orientador Mauro, pelo seu grande auxílio e por ter feito parte desta empreitada.

Quem dorme, sonha. Quem trabalha, consegue.

Washington Sebastián “El Loco”Abreu Gallo

RESUMO

Este trabalho apresenta a especificação e o desenvolvimento de um aplicativo para sincronizar arquivos entre um celular e um computador. O trabalho é baseado na arquitetura cliente/servidor utilizando a arquitetura J2ME e J2EE e comunicação entre cliente e servidor foi utilizando *sockets*. Apresenta também a especificação e implementação de um algoritmo para a sincronização de arquivos.

Palavras-chave: J2ME. Sincronização de sistema de arquivos. Dispositivos móveis.

ABSTRACT

This work presents the specification and development of an application to synchronize files between a cell phone and a computer. This work is based on the client/server architecture using J2ME architecture and J2EE language and the communication between the client and server it was used sockets. It also presents the specification and implementation of an algorithm for the file synchronization.

Key-words: J2ME. System files synchronization. Mobile devices.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

CDC - *Connected Device Configuration*

CLDC - *Connected Limited Device Configuration*

CORBA - *Common Object Request Broker Architecture*

CPU – *Central Processing Unit*

DOS – *Disk Operating System*

FAT - *File Allocation Table*

IP – *Internet Protocol*

J2ME – Java Micro Edition

JEE – Java Enterprise Edition

KVM – *Kilobyte Virtual Machine*

MHZ – *Mega Hertz*

MIDP - *Mobile Information Device Profile*

NTFS - *New Technology File System*

PIM - *Personal Information Management*

ROM – *Read Only Memory*

RPC - *Remote Procedure Call*

XIP - *eXecutable In Place*

SUMÁRIO

1 INTRODUÇÃO

No mundo moderno tem-se produzido cada vez mais informações. O meio mais utilizado para se armazenar estas informações é através de diversos dispositivos de armazenamento digitais, dispensando-se cada vez mais o uso de meios analógicos, tais como o papel. Um conjunto de dados armazenados em meios digitais são agrupados no que é chamado de arquivo. Um arquivo então, nada mais é do que uma série de dados agrupados, seguindo uma regra estrutural para que seja possível armazená-los em um dispositivo de armazenamento e posteriormente efetuar diversas manipulações nesse, tais como ler, transferir para outro dispositivo de armazenamento ou apagar. Um arquivo sempre será criado por um aplicativo (CARVALHO, 2005).

Com o crescente avanço da tecnologia, os dispositivos móveis têm agregado novas funções, desde editar e criar documentos de texto, até gravar vídeos. As pessoas criam cada vez mais arquivos, ocasionando uma necessidade crescente por espaço de armazenamento em seus dispositivos móveis. Com isso, tem-se a necessidade de duplicar ou sincronizar um conjunto de arquivos criados num celular ou em um computador, para manter ambos com o mesmo conteúdo. A isso se dá o nome de *backup* ou replicação de dados.

Realizar a sincronização de uma grande quantidade de arquivos é um trabalho demorado se feito manualmente, principalmente quando alterações nos arquivos foram realizadas tanto na origem quanto no destino da replicação. A sincronização executada por uma ferramenta garante que nenhuma informação seja perdida neste processo. Para manter estes diretórios sempre com os mesmos dados, são utilizados softwares aplicativos que automatizam este processo, facilitando o gerenciamento de diversos diretórios.

Existem alguns softwares que se propõem a automatizar esta tarefa. Dentre os principais, pode-se citar o Dropbox (DROPBOX, 2010) e o ActiveSync (SOFTONIC, 2008).

De acordo com o exposto, este trabalho propõe desenvolver um software aplicativo, seguindo a arquitetura cliente/servidor, para facilitar a sincronização de diretórios de um celular com um computador.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um software aplicativo que permita realizar a sincronização de diretórios entre um celular e um computador.

Os objetivos específicos do trabalho são:

a) disponibilizar um algoritmo de sincronização e replicação de arquivos e pastas, baseado na arquitetura cliente/servidor;
b) disponibilizar um software aplicativo servidor, que será compatível com a plataforma Windows;
c) disponibilizar um software aplicativo cliente, que será compatível com a arquitetura J2ME.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está dividido em quatro capítulos. O primeiro capítulo contempla a introdução onde são apresentadas as razões e os objetivos que levaram ao desenvolvimento do presente trabalho. No segundo capítulo são apresentados as principais tecnologias e conceitos utilizados no trabalho, bem como os trabalhos correlatos. No terceiro capítulo são expostos o desenvolvimento e os resultados obtidos com o mesmo. No quarto capítulo são apresentadas as conclusões a respeito do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Nas seções seguintes serão apresentados alguns aspectos teóricos relacionados aos recursos e tecnologias utilizados no desenvolvimento deste trabalho. Primeiramente é apresentado o assunto metadados, seguido da apresentação da tecnologia J2ME, e posteriormente é apresentada a tecnologia J2EE. Na sequência é apresentado o assunto sistema de arquivos e a arquitetura cliente/servidor. Posteriormente são apresentadas considerações sobre algoritmos de sincronização e são relacionados os trabalhos correlatos.

2.1 METADADOS

Conforme IBGE (2008), metadados podem ser basicamente definidos como dados que descrevem os dados, ou seja, são informações úteis para identificar, localizar, compreender e gerenciar os dados.

Os metadados têm um papel muito importante na administração de dados, mas podem ser considerados de suma importância, pois é a partir deles que as informações serão processadas, atualizadas e consultadas (IMASTERS, 2003).

O uso de metadados se faz útil quando é necessário gerenciar um grande nível de informações, aonde não é preciso armazenar os dados em sua forma completa para o gerenciamento, guardando-se apenas informações relativas a eles, para que seja fácil encontrá-los.

Uma técnica utilizada para o gerenciamento de uma grande quantidade de arquivos através de metadados é utilizar um dicionário customizado baseado em chave dupla. A utilização de um dicionário customizado traz um ganho de velocidade na busca por arquivos a serem sincronizados, facilitando o gerenciamento dos dados que estão sendo manipulados (SYNCSHARP, 2010).

Este dicionário de dados é carregado com as informações mais relevantes dos arquivos que serão buscadas nos diretórios selecionados. Para que um arquivo possa ser identificado unicamente, pode-se selecionar algumas informações tais como o caminho relativo do arquivo, seu nome, data de última modificação do arquivo e o *hash* do mesmo. Estas

informações são consideradas metadados e são armazenadas no dicionário de dados para posterior manipulação.

Feita a varredura dos diretórios de origem e tendo sido o dicionário de metadados criado, uma busca é iniciada no diretório de destino para que se possa verificar através da comparação do *hash* dos arquivos de origem (que estão armazenados nos metadados do dicionário em forma de lista) e de destino, quais os dados a serem copiados, quais arquivos foram renomeados e quais foram apagados, tanto para o diretório de origem, quando para o de destino, efetuando assim a sincronização entre os diretórios de forma simples e rápida.

Efetuando-se o casamento de um dicionário de metadados da origem, com um dicionário de metadados do destino, através de métodos que efetuem buscas nestes dicionários, usando-se como parâmetro de busca e comparação informações como o *hash* de um arquivo e o nome do mesmo, faz com que a busca por arquivos a serem sincronizados seja acelerada devido a pouca quantidade de verificações necessárias para garantir que um arquivo deva ser sincronizado.

2.2 J2ME

> Originalmente a tecnologia Java ME foi criada com objetivo para trabalhar com restrições associadas com a construção de aplicações para dispositivos de pequeno porte. Por este propósito a Sun definiu as bases para a tecnologia Java ME se enquadrar em ambientes limitados e possibilitar a criação de aplicações Java rodando em dispositivos de pequeno porte com memória limitada, vídeo e pouco poder de processamento. (SUN MICROSYSTEMS, 2010, tradução nossa).

Segundo Caldeira (2002, p. 33), o J2ME é dividido em configurações e perfis. Estes oferecem informações sobre as APIs de diferentes famílias de equipamentos.

De acordo com Sun Microsystems (2010), a plataforma J2ME foi dividida em duas configurações, uma para dispositivos móveis com baixo poder de processamento e uma para dispositivos móveis com maior poder de processamento, descritas a seguir:

a) *Connected Device Configuration* (CDC): esta configuração é voltada para dispositivos com no mínimo 512KB de memória para executar a máquina Java, 256KB de memória dinâmica para armazenar variáveis, além de recursos de conectividade de rede e largura de banda possivelmente persistente e alta;
b) *Connected Limited Device Configuration* (CLDC): ela é voltada para dispositivos com 128KB de memória para executar a máquina Java, 32 KB de memória

> dinâmica para armazenar variáveis, possuindo interface restrita com o usuário, baixo poder de processamento, normalmente alimentado por bateria, conectividade de rede geralmente sem fio, largura de banda baixa e com acesso intermitente. Ela define uma máquina virtual Java reduzida, chamada de KVM e um conjunto também reduzido de APIs.

Um perfil atende às demandas específicas de uma certa família de dispositivos, enquanto uma configuração visa aparelhos que possuem recursos de hardware semelhantes. Um perfil é definido para dispositivos que executam tarefas semelhantes.

O MIDP foi o primeiro perfil definido para a plataforma J2ME, sendo lançado em novembro de 1999. Esse perfil foi implementado sobre a configuração CLDC. Também para a configuração CDC foi desenvolvido um perfil, o qual ficou conhecido como *Foundation Profile* (VALENTE et al., 2003, p. 2). O MIDP é o componente que, em termos de software, representa as limitações impostas por uma configuração. O perfil MIDP fornece funcionalidades para o desenvolvimento de aplicativos de uma ampla variedade de celulares, utilizando para isso, a configuração CLDC.

2.3 JEE

> *Java Platform, Enterprise Edition* (Java EE) é o padrão da indústria para o desenvolvimento de aplicações portáteis, robustas, escaláveis e seguras no lado servidor. Construído nos sólidos fundamentos do *Java Standard Edition* (Java SE), o *Java EE* fornece serviços *web*, modelo de componentes, gerenciamento e APIs[1] (*Application Programming Interface*) de comunicação que o fizeram o padrão da indústria para a implementação da arquitetura orientada a serviços e aplicativos web 2.0. (SUN MICROSYSTEMS, 2009, tradução nossa).

De acordo com Prado (2009), Java é uma linguagem de programação voltada para o desenvolvimento de aplicações que rodem em uma série de plataformas, seja Windows, Linux, Unix, Solaris ou Mac, assim como em dispositivos móveis como telefones celulares, PDAs e mainframes. Graças a esta versatilidade, a linguagem Java conta com três conhecidos ambientes de desenvolvimento:

a) Java Standard Edition (JSE): ambiente de desenvolvimento mais utilizado, sendo

[1] API: de Application Programming Interface (ou Interface de Programação de Aplicativos) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por programas aplicativos.

utilizado em Servidores e PCs;

b) JEE: voltada para redes, internet e intranets;
c) J2ME: ambiente para desenvolvimento de aplicações para dispositivos móveis como PDA´s e celulares.

A plataforma JEE difere-se da plataforma JSE pela adição de bibliotecas que fornecem funcionalidade para implementar software distribuído, tolerante a falhas e multicamada, baseada amplamente em componentes modulares executando em um servidor de aplicações. A plataforma JEE é considerada um padrão de desenvolvimento, já que o fornecedor de software nesta plataforma deve seguir determinadas regras se quiser declarar os seus produtos como compatíveis com JEE. Ela contém bibliotecas desenvolvidas para o acesso a base de dados, RPC, CORBA, etc. Devido a estas características a plataforma é utilizada principalmente para o desenvolvimento de aplicações corporativas (SUN MYCROSYSTEMS, 2009).

2.4 SISTEMA DE ARQUIVOS

De acordo com Tanenbaum e Woodhull (2000, p. 27), o sistema de arquivos é a forma que o sistema operacional usa para representar a informação em um espaço de armazenamento, apresentando um modelo abstrato e independente de dispositivos. Neste modelo são implementadas chamadas de sistema para as operações de criar, remover, ler e escrever arquivos.

O sistema de arquivos é uma parte fundamental do sistema operacional, pois fornece uma visão abstrata dos dados persistidos, além de ser responsável pelo serviço de nomes, acesso a arquivos e organização geral.

A maioria dos sistemas de arquivos possui o conceito de diretório, que consiste em uma forma de organizar e agrupar arquivos. As mesmas chamadas de sistema utilizadas para operações com arquivos também são aplicadas aos diretórios. Neste modelo, a estrutura hierárquica é formada e pode ser constituída por inúmeros níveis, a partir de um diretório raiz.

Carvalho (2005, p. 3) explica que a maioria dos arquivos armazenados em um sistema de arquivos possui um nome e um caminho, utilizados para a identificação única em tal sistema. Um caminho representa um nó de uma estrutura de diretórios. A localização de um

arquivo nesta estrutura parte da raiz, percorrendo os nós correspondentes aos diretórios até encontrar a informação desejada.

Uma estrutura hierárquica busca organizar informações em uma ordem lógica ou de importância. Normalmente apresenta uma estrutura de árvore onde cada nó da estrutura tem apenas um correspondente superior. Na gerência de arquivos, a estrutura hierárquica é geralmente utilizada e nela são os diretórios e arquivos que representam os elementos da estrutura. (TANENBAUM; WOODHULL, 2000, p. 27).

Dentre os principais sistemas de arquivos para computadores pode-se citar:

a) FAT: trata-se de um sistema de arquivos que funciona com base em uma espécie de tabela que indica onde estão os dados de cada arquivo. Esse esquema é necessário porque o espaço destinado ao armazenamento é dividido em blocos, e cada arquivo gravado pode ocupar vários destes, mas não necessariamente de maneira sequencial: os blocos podem estar em várias posições diferentes. Assim, a tabela acaba atuando como um "guia" para localizá-los (ALECRIM, 2011);
b) NTFS: desenvolvido para superar as limitações do sistema FAT, o NTFS utiliza algumas estruturas em 64 bits. Oferecendo melhor segurança e recuperação a dados, a NTFS também oferece um nível de armazenamento muito maior que FAT (MORIMOTO, 2007).

No que diz respeito a sistemas de arquivo de celulares, Jipping (2007, p. 188) diz que sistemas operacionais de celulares têm muitos dos requisitos dos sistemas operacionais desktop. A maioria são implementados em ambientes de 32 bits, a maioria permite que os usuários atribuam nomes arbitrários aos arquivos e permitem armazenar vários arquivos que necessitam de algum tipo de estrutura organizada. Isso significa que um sistema de arquivos hierárquico baseado em diretório é desejável. Se os sistemas de telefonia móvel não têm uma mídia removível, então qualquer sistema de arquivos poderiam ser utilizados. Nos sistemas que usam memória *flash*, há circunstâncias especiais a considerar. A memória *flash* não pode simplesmente substituir a memória já gravada. Ele deve primeiro apagar a memória gravada e depois. Além disso bytes individuais não podem ser apagadas, blocos inteiros devem ser eliminados de uma vez.

2.4.1 Sistema Operacional Symbian

De acordo com Gubian e Savoldi (2009), a versão atual do sistema operacional Symbian é a segunda geração de um sistema denominado EPOC Kernel Architecture (EKA2) construído originalmente na década de 80 (primeiramente em 8 e posteriormente 16 bits) para dispositivos Psion (*personal organizers* ou PDAs).

O sistema operacional é organizado de forma modular, como pode ser observado na figura 1, onde as funcionalidades são providas através de blocos separados (e não como uma estrutura monolítica).

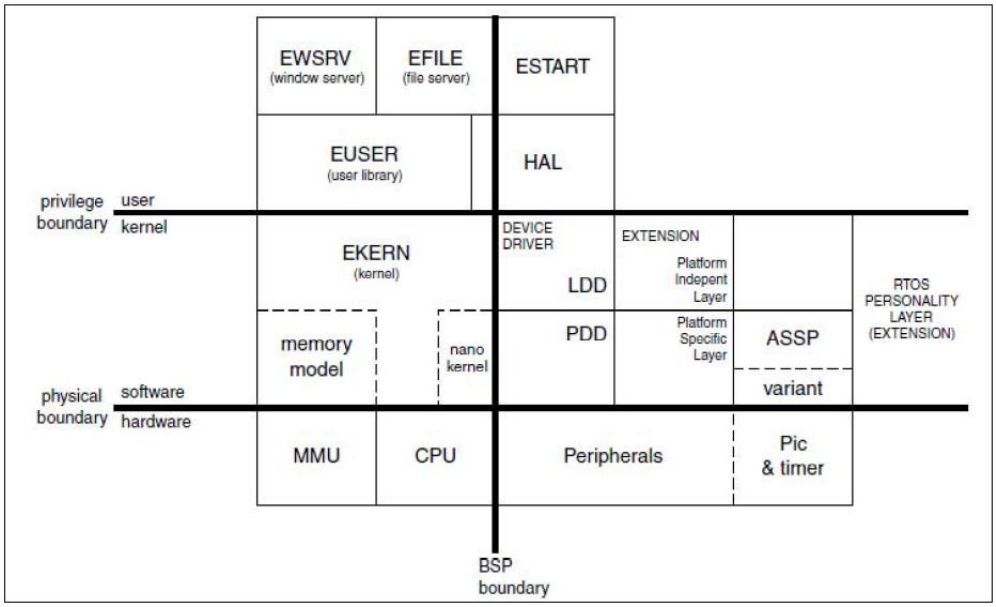

Figura 1 – Arquitetura do Symbian OS

Gubian e Savoldi (2009) dizem ainda que embora seja monousuário o sistema é multitarefa preemptivo baseado em prioridades, sendo capaz de chavear o tempo de CPU entre múltiplas *threads* o que dá ao usuário do dispositivo móvel a impressão de que múltiplas aplicações estão executando ao mesmo tempo. O sistema aloca o tempo de CPU baseado na prioridade das *threads*. O *kernel real-time* estabelece que os serviços são limitados pelo tempo com garantia de cumprimento em tempo conhecido.

O núcleo do sistema é construído como um *microkernel* sobre um *nanokernel* de tempo real. O nanokernel é responsável por prover primitivas para sincronização rápida entre processos, *timers*, disparo inicial de interrupções e escalonamento de tarefas.

O núcleo foi concebido para suportar a pilha de sinalização das funções de telefone (GSM/GPRS/UMTS) e a parte de gerenciamento de informações pessoais (PIM) em um único processador.

O Symbian foi concebido para ser executado em pequenos computadores baseados em bateria. Apesar disto, ele roda em CPUs de 32 bits com menos de 400 MHZ tais como: XScale e Arm9 e Arm11 (mais recentes) e as antigas Arm7.

2.4.1.1 Sistema de arquivos Symbian

Em um *smartphone* baseado em Symbian, o sistema de arquivos pode ser acessado por meio de um componente chamado de "file Server" ou F32 o qual gerencia cada dispositivo orientado a arquivos. Este componente prove acesso a arquivos, diretórios e drives (através de mapeamento). O F32 usa uma arquitetura cliente/servidor o que permite receber e processar requisições relacionadas a arquivos de múltiplos clientes (GUBIAN, SAVOLDI, 2009).

Ele é capaz de tratar diferentes sistemas de arquivos como FAT (usados para discos removíveis) através de componentes plugáveis. Além disso, ele suporta um máximo de 26 unidades, cada uma das quais sendo identificada com uma letra seguida de dois pontos (ex. A:), notação esta semelhante aquela usada no Microsoft DOS.

O dispositivo ROM, onde o *firmware* reside, é sempre designado com a letra Z:. Neste dispositivo residem os arquivos executáveis e os dados, os quais são chamados de XIP tendo em vista que os mesmos são disparados diretamente sem a necessidade do processo de carga de programas do disco para a memória comum em sistemas operacionais desktop.

Além disso, o *firmware*, ou imagem ROM, é geralmente armazenado em memória flash (também conhecido como EEPROM) que é uma memória não volátil que pode ser programada e apagada eletronicamente.

O drive C: é destinado como drive default para usuários, os quais podem ser mapeados no mesmo chip de memória flash do firmware enquanto os demais dispositivos removíveis recebem a designação a partir da letra D: (GUBIAN, SAVOLDI, 2009).

2.5 CLIENTE/SERVIDOR

De acordo com Kioskea (2009), em uma arquitetura cliente/servidor, as máquinas clientes (ligadas a uma rede) contatam o servidor a fim de estabelecer uma conexão para que

o cliente possa acessar algum recurso ou serviço disponibilizado pelo servidor. Cada instância de um cliente pode enviar requisições de dado para algum dos servidores conectados e esperar pela resposta. Por sua vez, algum dos servidores disponíveis pode aceitar tais requisições, processá-las e retornar o resultado para o cliente.

A característica de cliente-servidor descreve a relação de programas em um aplicativo. O componente de servidor fornece uma função ou serviço a um ou muitos clientes, que iniciam os pedidos de serviços. Funções como troca de *e-mail*, acesso à internet e acessar banco de dados, são construídos com base no modelo cliente-servidor.

A tecnologia cliente/servidor é uma arquitetura na qual o processamento da informação é dividido em módulos ou processos distintos. Um processo é responsável pela manutenção da informação (servidores) e outros responsáveis pela obtenção dos dados (os clientes). Os processos cliente enviam pedidos para o processo servidor, e este por sua vez processa e envia os resultados dos pedidos. Geralmente, os serviços oferecidos pelos servidores dependem de processamento específico que só eles podem fazer. O processo cliente, por sua vez, fica livre para realizar outros trabalhos. A interação entre os processos cliente e servidor é uma troca cooperativa, em que o cliente é o ativo e o servidor reativo, ou seja, o cliente requisita uma operação, e neste ponto o servidor processa e responde ao cliente (ORFALI et al, 1994, p. 13-14).

2.6 ALGORITMOS DE SINCRONIZAÇÃO

De acordo com Ferreira e Souza (2009) o uso de computadores móveis está ganhando muita popularidade. O número de usuários com tais computadores está crescendo e a tendência é que isto continue no futuro, quando o número de clientes móveis deverá superar em muito o número de clientes tradicionais em ambientes corporativos. Aplicações executadas em clientes móveis requisitam informações conectando-se aleatoriamente aos repositórios de dados. Clientes móveis constituem uma nova carga de trabalho e exibem padrões de acesso diferentes daqueles vistos nos tradicionais sistemas cliente-servidor de arquivos.

Alan et al. (2005) afirmam que a estratégia de replicação otimista está sendo mais bem aceita, visto que uma baixa consistência nos arquivos pode ser aceita em troca de uma

possiblidade de se atualizar os arquivos enquanto desconectado de uma rede. Estas atualizações descoordenadas devem ser posteriormente sincronizadas combinando automaticamente atualizações não conflitantes e relatando os conflitos na atualização. A estratégia de replicação otimista consiste em o usuário alterar seus arquivos e diretórios desconectado da rede, fornecendo ao usuário uma maior mobilidade e liberdade para alterar seus arquivos, para que a sincronização dos arquivos possa ser realizada somente quando o usuário achar necessário fazê-la.

No que se refere à sincronização, Suel e Memon (2002, p. 12) identificam que alguns problemas devem ser observados quanto à sincronização de arquivos de usuário, dentre os quais pode-se citar:

a) definição de parâmetros para verificar se um arquivo precisa ser sincronizado;
b) velocidade do meio de comunicação entre as partes a serem sincronizadas;
c) quantidade de arquivos a serem sincronizados;
d) método de transferência de arquivo, se feita em partes do arquivo ou o arquivo completo.

Dentre os diversos algoritmos de sincronização existentes, Suel e Memon (2002, p. 13-14) citam o utilizado na ferramenta *rsync*, por ser muito empregado para sincronização em sistemas Unix. O algoritmo de nome homônimo à ferramenta se mostra eficiente quando o cliente e o servidor possuem alguma versão do arquivo a ser sincronizado, pois este quebra o arquivo de origem em vários pedaços e gera uma pequena assinatura de cada pedaço. Então, envia pela rede estas assinaturas e, fazendo o mesmo do outro lado, descobre quais pedaços faltam no arquivo de destino para torná-lo igual ao de origem.

Syncsharp (2010) propõe um algoritmo de sincronização baseado no conceito de metadados criados a partir de uma descrição de um dicionário, onde o algoritmo cria os metadados a partir das informações coletadas através da varredura dos diretórios a serem sincronizados alimentando o dicionário.

O dicionário consiste numa classe Java, que contém uma lista onde serão armazenados os arquivos e diretórios que estão sendo gerenciados pela ferramenta, utilizando o conceito de chave dupla. No construtor da classe do dicionário, serão passados por parâmetro o caminho relativo do arquivo ou diretório a ser incluído no dicionário, representado pelo primeiro parâmetro, o *hash* do arquivo no segundo parâmetro e um objeto do tipo `FileUnit` passado no terceiro parâmetro, que é um objeto que servirá para armazenar informações relativas ao arquivo ou diretório, tais como data de criação e modificação, caminho relativo e tamanho.

Este algoritmo mostra-se eficiente para gerenciar grandes quantidades de arquivos, transferindo-os por completo quando se faz necessária a replicação do arquivo (SYNCSHARP, 2010).

2.7 TRABALHOS CORRELATOS

Como base para o trabalho proposto, pôde-se observar trabalhos semelhantes, dentre os quais foram selecionados o Dropbox (DROPBOX, 2010) e o ActiveSync (MICROSOFT, 2007), que são descritos a seguir.

2.7.1 Dropbox

"Dropbox é um serviço de armazenamento *online* onde você compartilha e salva imagens, vídeos, documentos, música e tudo o que quiser" (SOFTONIC, 2010).

O aplicativo Dropbox surgiu para servir como opção de *backup* e compartilhamento *online* entre computadores. Com ele é possível fazer *backup* de arquivos que serão hospedados nos servidores do Dropbox, podendo o usuário instalar o aplicativo em diversos computadores que serão conectados à conta do mesmo, mantendo sempre a sincronia entre eles através dos servidores do Dropbox.

De acordo com Dropbox (2010), estão disponíveis aplicativos para as plataformas móveis iPhone e Android, sendo que os outros sistemas operacionais somente poderão ter acesso aos arquivos do usuário hospedados nos servidores do Dropbox através de uma interface web.

2.7.2 ActiveSync

Para manter o celular atualizado e sincronizado com o computador de mesa, para

plataformas móveis *Windows Mobile*, a principal solução é o ActiveSync. Este é o programa oficial da Microsoft responsável pela comunicação rápida e eficiente entre o celular e o computador (SOFTONIC, 2008).

Com o ActiveSync é possível sincronizar contatos, mensagens de texto, fotos, agenda, imagens, músicas com um computador de mesa, e quando o usuário estiver longe do seu computador, poderá sincronizar os contatos, agenda, *e-mails* e tarefas com o servidor do Microsoft Exchange via internet (MICROSOFT, 2007).

Segundo Softonic (2008), o ActiveSync assume outras funções além de manter uma cópia de segurança de tudo o que a pessoa tem dentro do celular. Gerencia a instalação de novos programas e aplicações no dispositivo, administra diferentes contas de usuários e configura as conexões externas.

3 DESENVOLVIMENTO

Nesta seção é descrito o desenvolvimento do aplicativo de sincronização de arquivos entre cliente e servidor para celular, utilizando para isto a tecnologia J2ME.

Para tanto, esta seção descreve também os principais requisitos que o aplicativo deve atender bem como descrever os passos e os resultados obtidos com a implementação. Na seção abaixo são apresentados os requisitos do trabalho desenvolvido.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Serão disponibilizadas as seguintes funcionalidades:

a) permitir o espelhamento completo de um diretório, juntamente com os seus subdiretórios (Requisito Funcional - RF);
b) permitir que sejam selecionados vários diretórios para serem sincronizados (RF);
c) permitir a geração de um *log* com a informação dos arquivos comparados a serem atualizados (RF);
d) sincronizar bidirecionalmente um dispositivo móvel e um computador (RF);
e) versionar arquivos com mesmo nome para evitar perda (RF);
f) identificar arquivos duplicados (RF);

Os Requisitos Não Funcionais (RNF) são:

a) utilizar a linguagem JEE na implementação do servidor;
b) utilizar a linguagem J2ME para a implementação do cliente;
c) utilizar a plataforma móvel *Symbian* para a validação do protótipo.

3.2 ESPECIFICAÇÃO

Para fazer a especificação do servidor e do cliente do aplicativo, foram utilizados diagramas da UML. Os diagramas escolhidos para melhor apresentar as funcionalidades do

sistema são: diagrama de casos de uso, diagrama de sequência, diagrama de classes e diagrama de disposição. A ferramenta adotada para fazer estes diagramas foi o Enterprise Architect.

3.2.1 Diagrama de disposição

Segundo Menezes (2004, p. 176), um diagrama de disposição modela o inter-relacionamento entre recursos de infra estrutura, de rede ou artefatos de sistema. Este diagrama é normalmente usado para representar servidores.

O diagrama de disposição apresentado na figura 2 representa a arquitetura da aplicação. A aplicação servidora é executada sobre um microcomputador PC enquanto que a aplicação cliente é executada em um celular. A comunicação entre elas é feita utilizando-se socket sobre o protocolo TCP/IP.

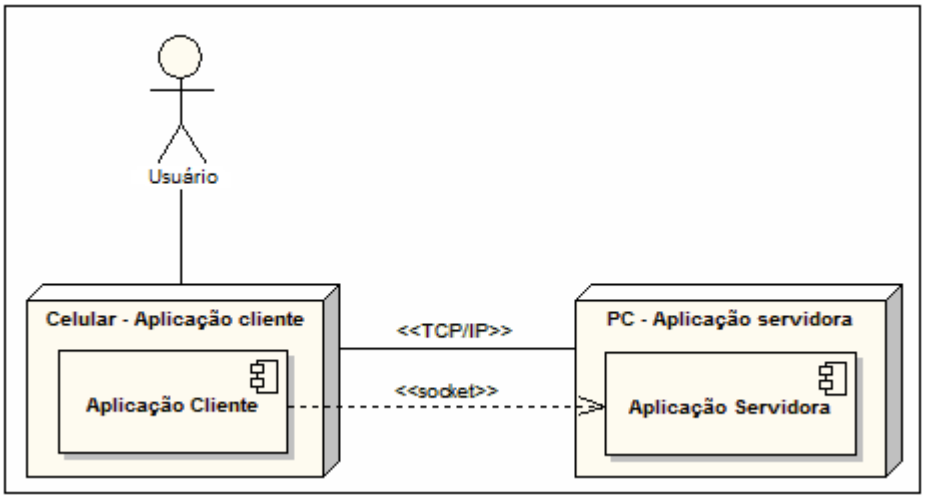

Figura 2 – Diagrama de disposição do sistema

3.2.2 Diagrama de casos de uso

Segundo Debone (2004), um caso de uso descreve um objetivo que um ator externo tem com o sistema. Um ator pode ser um elemento humano ou não que interage com o sistema. O ator se encontra fora do escopo de atuação do sistema, enquanto o conjunto de casos de uso forma o escopo do sistema.

Na figura 3, observa-se o diagrama de caso de uso da ferramenta, onde o usuário

executa o caso de uso no módulo servidor.

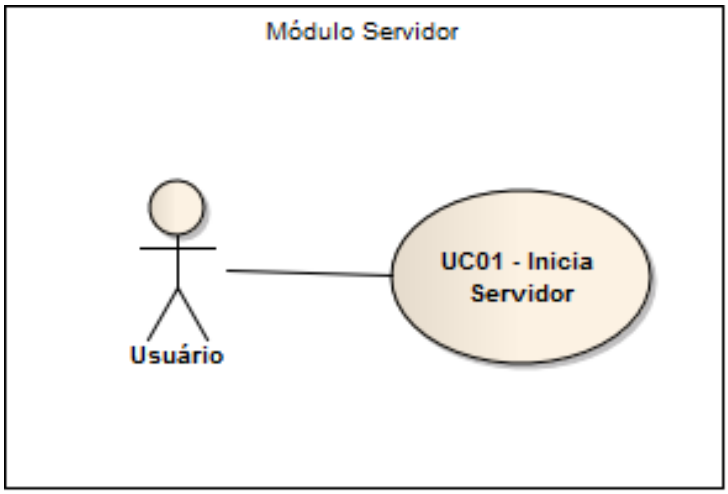

Figura 3- Caso de uso do módulo servidor executado pelo usuário

No quadro 1 é descrito o caso de uso que diz respeito à interação do usuário com o módulo servidor, para iniciar o servidor.

UC01 – Inicia Servidor
Descrição: Permite ao usuário iniciar o servidor da aplicação, selecionando para tanto um diretório.
Ator Principal: Usuário
Cenário Principal: Iniciar Servidor
1. o sistema apresenta a tela de entrada ao usuário;
2. o usuário clica no botão "Seleciona Diretório";
3. o sistema apresenta uma janela para o usuário selecionar um diretório;
4. o usuário seleciona o diretório e clica no botão "Selecionar";
5. o usuário clica no botão "Iniciar Servidor";
6. o sistema inicia um serviço que aguarda uma conexão do cliente.

Cenário de exceção
1. no passo 5, se o usuário não tiver seleciona anteriormente um diretório, o sistema apresentará uma mensagem de erro;
2. no passo 5, caso exista alguma outra instância do servidor já iniciado, apresentará uma mensagem de erro.

Pré-condições
1. nenhuma outra instância do servidor deve estar rodando nesta mesma máquina.

Pós-condições
1. o servidor é iniciado.

Quadro 1 - Descrição do caso de uso "`UC01-Inicia Servidor`"

Na figura 4 é apresentado os casos de uso do módulo servidor executados pelo módulo cliente.

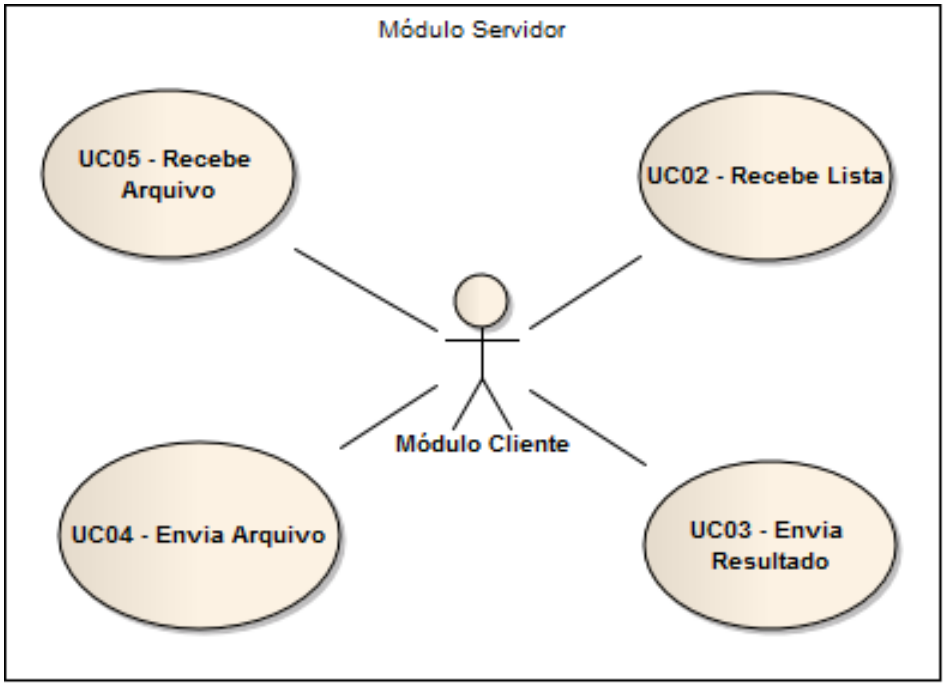

Figura 4 – Casos de uso do módulo servidor executados pelo módulo cliente

UC02 – Recebe Lista

Descrição: Permite que o módulo cliente receba a lista com os metadados do servidor.

Ator Principal: Módulo Cliente

Cenário Principal: Recebe Lista

1. o módulo servidor envia o arquivo contendo os metadados;
2. o módulo cliente recebe o arquivo contendo os metadados;

Pré-condições

1. o módulo cliente deve ter uma conexão estabelecida com o módulo servidor;

Pós-condições

1. o módulo cliente grava em arquivo as informações recebidas.

Quadro 2 – Caso de uso "UC02 – Recebe Lista"

UC03 – Envia Resultado

Descrição: Permite que o módulo cliente receba uma lista com o resultado do casamento das listas de metadados.

Ator Principal: Módulo Cliente

Cenário Principal: Envia resultado

1. o módulo servidor envia o arquivo contendo o caminho relativo dos arquivos e diretórios a serem copiados para o servidor, e os arquivos e diretórios a serem recebidos pelo cliente;
2. o módulo cliente recebe o arquivo contendo os caminhos dos arquivos e diretórios a serem sincronizados;

Pré-condições

1. o módulo cliente deve ter uma conexão estabelecida com o módulo servidor;

Pós-condições

1. o módulo cliente grava em arquivo as informações recebidas.

Quadro 3 – Caso de uso "UC03 – Envia Resultado"

UC04 – Envia Arquivo
Descrição: Permite que o módulo cliente receba um arquivo do módulo servidor.
Ator Principal: Módulo Cliente
Cenário Principal: Envia arquivo
1. o módulo servidor envia um arquivo para o módulo cliente;
2. o módulo cliente recebe o arquivo e grava no sistema de arquivos;

Pré-condições
2. o módulo cliente deve ter uma conexão estabelecida com o módulo servidor;

Pós-condições
2. o arquivo é armazenado no sistema de arquivos do módulo cliente.

Quadro 4 – Caso de uso "UC04 – Envia arquivo"

UC05 – Recebe Arquivo
Descrição: Permite que o módulo servidor receba um arquivo do módulo cliente.
Ator Principal: Módulo Cliente
Cenário Principal: Recebe arquivo
3. o módulo cliente envia um arquivo para o módulo servidor;
4. o módulo servidor recebe o arquivo e grava no sistema de arquivos;

Pré-condições
3. o módulo cliente deve ter uma conexão estabelecida com o módulo servidor;

Pós-condições
3. o arquivo é armazenado no sistema de arquivos do módulo servidor.

Quadro 5 – Caso de uso "UC05 – Recebe arquivo"

Na figura 5 é apresentado o diagrama de casos de uso executados pelo usuário no m

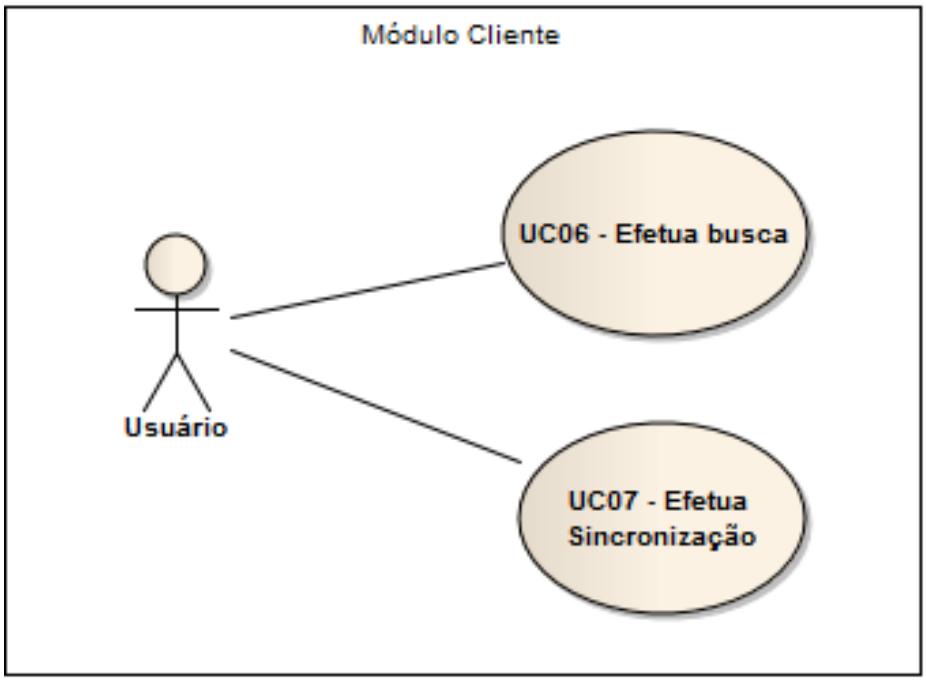

Figura 5 – Diagrama de caso de uso executado pelo usuário no módulo cliente

No quadro 6 é descrita a interação do usuário com o módulo cliente para que seja efetuada a busca.

UC06 – Efetua busca
Descrição: Permite que seja efetuada a busca no diretório selecionado do cliente.
Ator Principal: Usuário
Cenário Principal: Efetua busca
5. o sistema apresenta a tela de entrada ao usuário;
6. o usuário clica na opção "Menu";
7. o usuário clica na opção "Seleciona Local";
8. o sistema apresenta os diretórios existentes no cliente;
9. o usuário posiciona a seleção em algum diretório;
10. o usuário seleciona a opção "Seleciona Pasta";
11. o usuário seleciona a opção "Seleciona Local";
12. o usuário seleciona a opção "Retorna";
13. o sistema apresenta a tela de entrada com o diretório selecionado;
14. o usuário seleciona a opção "Efetua Busca";

Pré-condições
2. o sistema cliente deve estar instalado no celular do usuário;

Pós-condições
1. o sistema efetua a busca por arquivos e diretórios no diretório selecionado.
2. o sistema cria um arquivo texto com informações dos diretórios e arquivos encontrados.

Quadro 6 – Descrição do caso de uso "UC06-Efetua busca"

No quadro 7 é descrita o procedimento para que seja iniciada a sincronização entre o cliente e o servidor.

UC07 – Efetua sincronização
Descrição: Permite que seja efetuada sincronização dos arquivos presentes no diretório do cliente, com os arquivos no diretório selecionado no servidor.
Ator Principal: Usuário
Cenário Principal: Efetua sincronização
1. o usuário informa o endereço ip do servidor;
2. o usuário seleciona o tipo de sincronização a ser realizada;
3. o usuário seleciona a opção "Menu";
4. o usuário seleciona a opção "Sincroniza".

Pré-condições
1. deverá ter sido efetuado o procedimento "Efetua busca";
2. o servidor deve ter sido iniciado;

Pós-condições
1. o sistema efetua a sincronização dos arquivos presentes no diretório selecionado com o servidor.

Quadro 7 – Descrição do caso de uso "UC07-Efetua sincronização"

3.2.3 Diagrama de classes

Nos diagramas de classes aqui apresentados, foram relacionados os principais métodos

e atributos das classes disponibilizadas para um melhor entendimento.

A figura 6 apresenta o diagrama de classes da aplicação cliente do aplicativo SyncEasy.

Dentre as classes disponíveis, pode-se citar que a classe `ClienteGUI` é responsável por apresentar a interface gráfica ao usuário. A classe `FileBrowser` é a que apresenta ao usuário de maneira gráfica, uma lista com os diretórios e arquivos presentes no sistema de arquivos do dispositivo, para que o usuário possa selecionar um diretório a ser sincronizado. A classe `ArrayList` foi incluída porque a plataforma J2ME não dá suporte oficial a objetos deste tipo. Como o uso destes objetos é facilitado pelos seus métodos disponíveis, esta implementação voltada à plataforma J2ME foi incluída.

A classe `BuscaCliente` efetua a busca de arquivos e diretórios com base no atributo `diretorioSelecionado` que guarda o diretório raiz que deverá ser sincronizado, gerando um arquivo texto com os metadados dos arquivos e diretórios encontrados.

A classe `ArquivoCliente` possui a modelagem dos atributos a serem guardados e verificados, baseado nos metadados colhidos na classe `BuscaCliente`.

Para gerenciar o fluxo de execução do aplicativo cliente, foi criada a classe `Cliente`, que executa uma *thread* para gerenciar o *socket* de conexão com o servidor.

A plataforma J2ME não dá suporte para que seja feito *hash* de arquivo, então para solucionar este problema foram incluídas as classes `MD5` e `MD5State`. Esta biblioteca é uma adaptação da implementação de uma rotina de *hash* MD5 para J2EE (MACINTA, 2011). Entretanto, como a solução que foi portada para a plataforma J2ME não está otimizada para fazer *hash* de arquivo, somente é possível ser feito *hash* de um *array* de *bytes*, o que limita o tamanho do arquivo a ser trabalhado nesta rotina a um tamanho máximo de aproximadamente 1 *megabyte*.

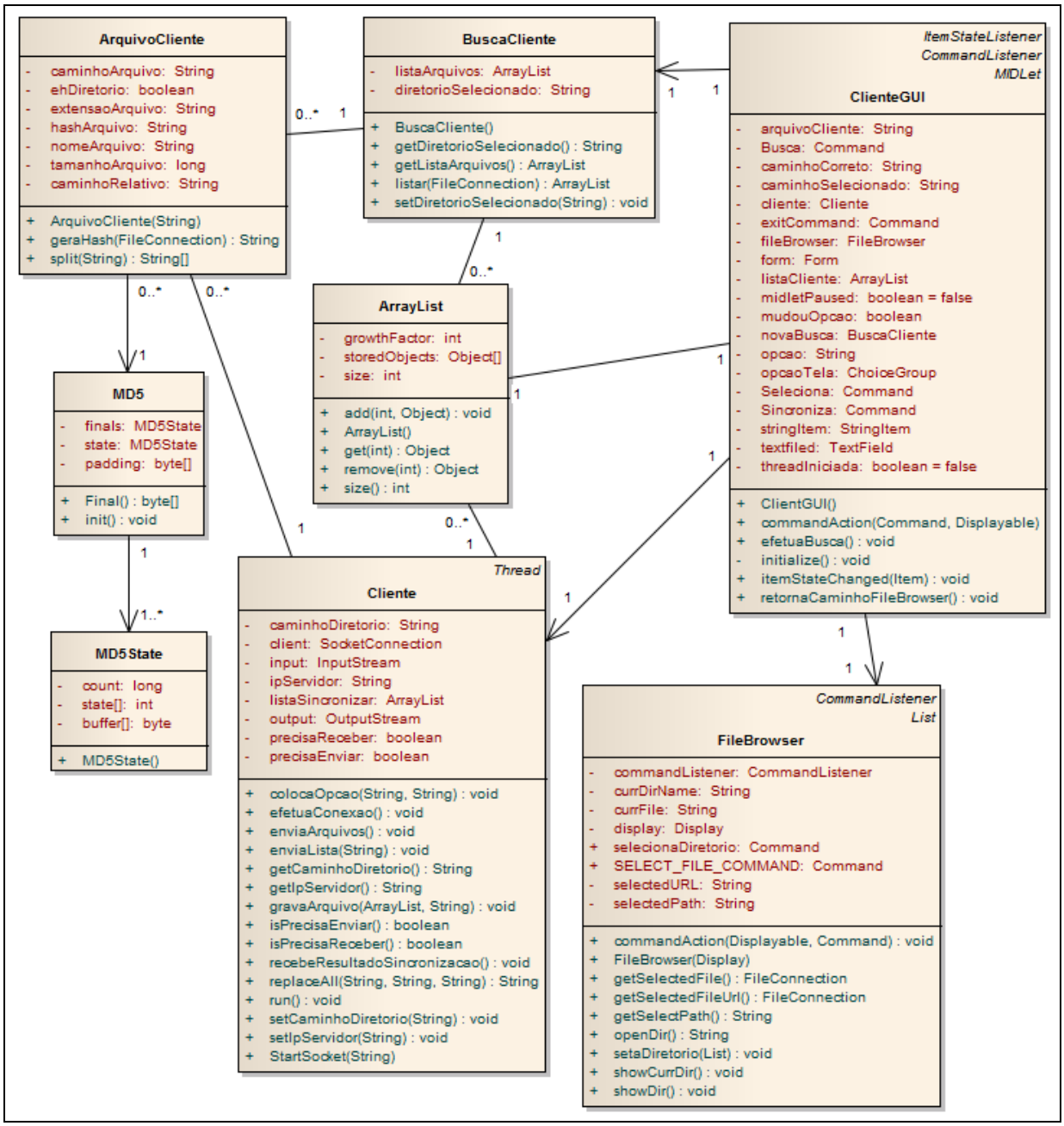

Figura 6 – Diagrama de classes da aplicação cliente

Na figura 7 é apresentado o diagrama de classes da aplicação servidora. Nesta aplicação a classe `ServidorGUI` é responsável por criar a tela da aplicação e apresentar ao usuário. A classe `ServidorConexao` contém os métodos de conexão com o cliente via *socket*. Esta também possui os métodos responsáveis pelas funcionalidades da aplicação servidora. Para gerenciar a *thread* e iniciar o processo de conexão, a classe `ServidorConexaoThread` é utilizada, bem como esta classe gerencia o fluxo principal de execução desta aplicação. A classe `Busca` responsável por varrer o diretório selecionado na interface, armazenando os metadados dos arquivos e diretórios, e posteriormente gravando estas informações em um arquivo texto. A classe `ArquivoCliente` possui a modelagem dos atributos a serem

guardados e verificados, baseado nos metadados colhidos na classe `Busca`. Na classe `Dicionario` estão descritos os métodos responsáveis por armazenar os metadados de forma que seja possível encontrar um objeto da classe `Arquivo` baseado no seu *hash* ou caminho relativo. A classe `Gerador` é responsável por efetuar o casamento das listas do cliente e do servidor, gerando assim uma lista com os arquivos a serem sincronizados.

Figura 7 – Diagrama de classes da aplicação servidora

3.2.4 Diagrama de sequência

O diagrama de sequência apresentado na figura 8 representa o caso de uso UC01-Inicia Servidor, o qual descreve os passos de inicialização do servidor, aonde o usuário seleciona um diretório e inicia o servidor que ficará aguardando por conexões provenientes do cliente.

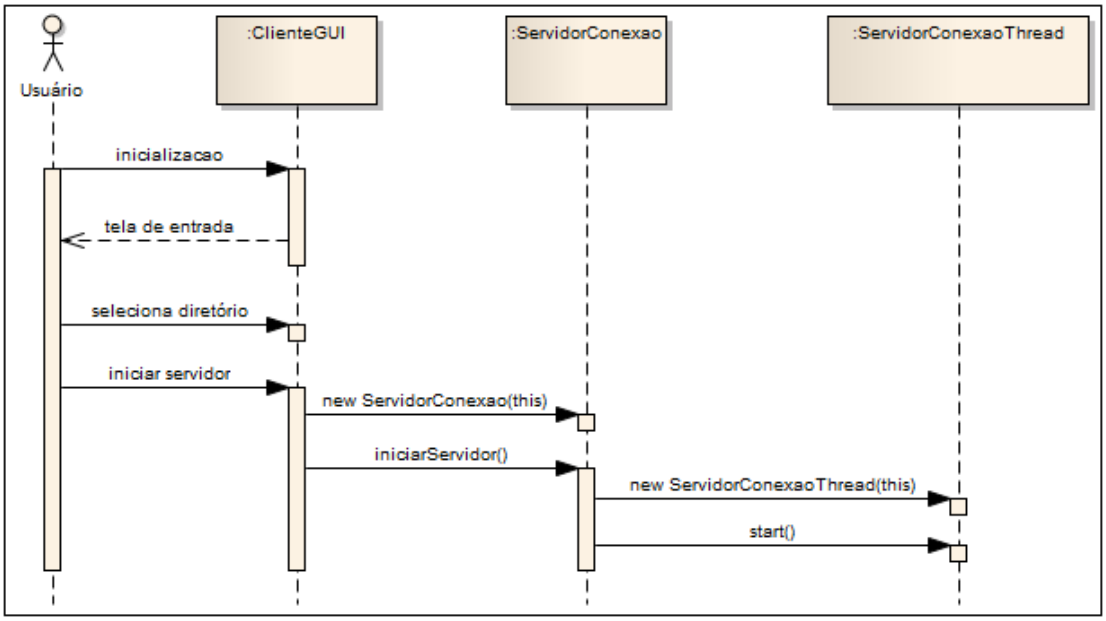

Figura 8 – Diagrama de sequência referente ao caso de uso UC01

A figura 9 apresenta o fluxo realizado para que seja possível que o usuário inicie o procedimento de busca do sistema, aonde o usuário deve selecionar o diretório em que deverá ser feita a busca e iniciar o processo da busca.

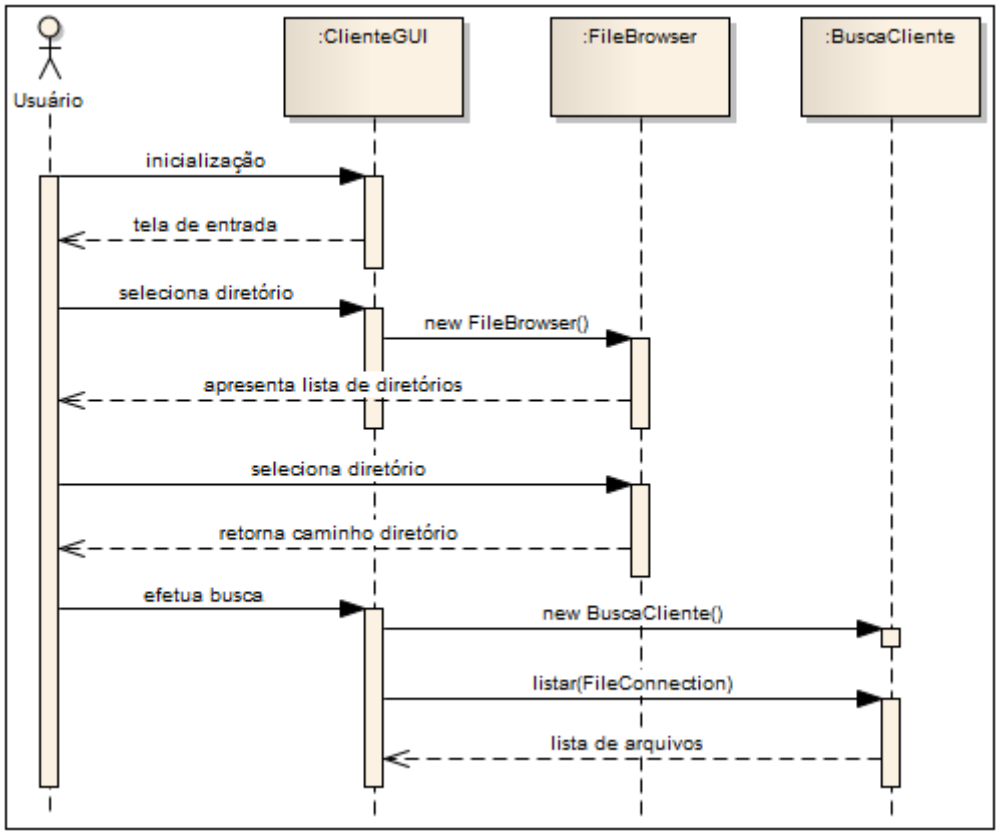

Figura 9 – Diagrama de sequência do caso de uso UC02-Efetua busca

A figura 10 apresenta o diagrama de sequência referente ao fluxo do caso de uso UC03-Efetua Sincronização, onde o usuário informa o endereço IP do servidor onde deseja se conectar e o tipo de sincronização desejada. Após isto o usuário seleciona a opção sincronizar para que a conexão com o servidor seja feita e a troca de arquivos entre o servidor e o cliente seja iniciada.

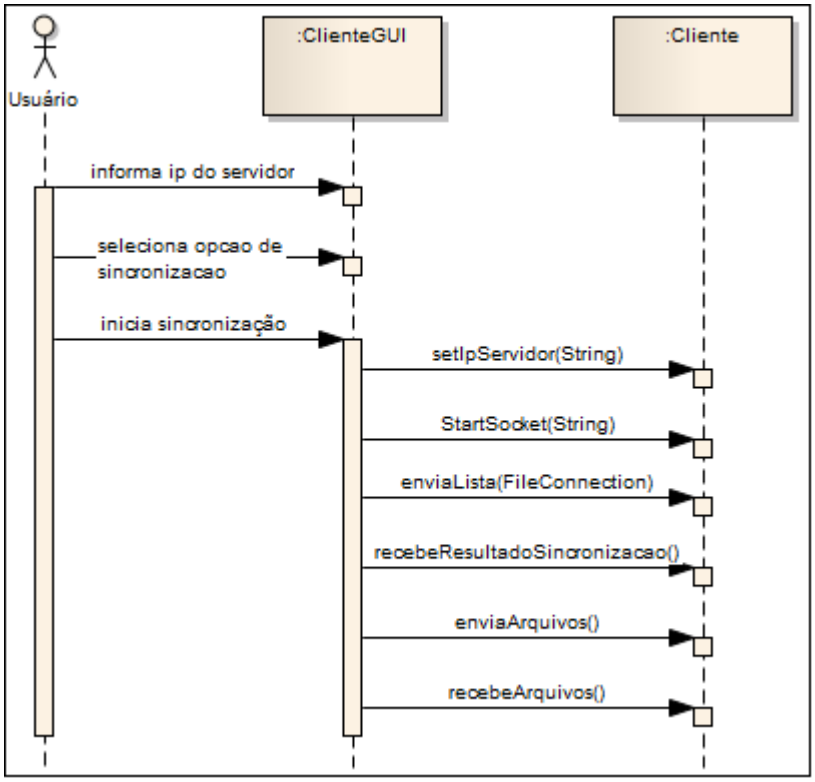

Figura 10 – Diagrama de sequência do caso de uso UC03-Efetua sincronização

3.3 IMPLEMENTAÇÃO

A seguir são apresentadas as técnicas e ferramentas utilizadas e a operacionalidade da implementação, bem como alguns trechos de códigos relevantes no sistema desenvolvido.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento das aplicações, optou-se por utilizar a ferramenta Netbeans 6.9.1 devido à estabilidade e suporte no desenvolvimento de aplicações MIDP.

Para a comunicação entre o cliente e o servidor, utilizou-se *socket*. O quadro 8 representa o método de inicialização do servidor de conexão. Para efeito de testes, utilizou-se a porta 22285, para aguardar o pedido de conexão do cliente.

```
public void iniciarServidor() {
    try {
         this.welcomeSocket = new ServerSocket(22285);
         this.servidorThread = new ServidorConexaoThread(this);
         this.servidorThread.start();
         this.setServidorIniciado(true);
      } catch (IOException ex) {
        System.out.println("ServidorConexao - Ocorreu um Erro: " +
ex.toString());
      }
}
```

Quadro 8 – Método de inicialização do servidor

No cliente, o método responsável pela conexão com o servidor é apresentado no quadro 9. O cliente abre um canal de comunicação com o servidor através de um *socket* através do comando `Connector.open`. Após a conexão, são criados os canais de comunicação com o cliente, um para receber e outro para enviar dados, `input` e `output` respectivamente.

```
public void efetuaConexao() {
      try {
         client = (SocketConnection)
Connector.open("socket://"+getIpServidor()+":22285",Connector.READ_WRITE);
         output = client.openOutputStream();
         input = client.openInputStream();
      } catch (IOException ex) {
         System.out.println("Erro ao conectar: "+ex.getMessage());
      }
   }
```

Quadro 9 – Método que efetua conexão do cliente com o servidor

No quadro 10 é apresentado o método do cliente que efetua a busca no diretório selecionado na interface do usuário. Este método recebe por parâmetro uma instância da classe `FileConnection` que pode ser tanto um diretório, quanto um arquivo. Em ambos os casos, o método irá criar um objeto da classe `Arquivo` para armazenar as informações referentes a este. Este objeto será adicionado a uma lista onde serão guardados todos os arquivos e diretórios encontrados no nível e nos subníveis do diretório selecionado. Se o parâmetro passado for um diretório, o método irá executar recursivamente, como pode ser observado na chamada `listar(filecon)`, até que todos os diretórios que estejam em subníveis do diretório atual sejam listados. Quando todos estes diretórios tiverem sido percorridos, o método irá adicionar na lista os arquivos encontrados. O mesmo conceito de

busca de arquivos e diretórios foi utilizado no lado do servidor.

```
public ArrayList listar(FileConnection arquivo) throws IOException {
      String caminhoRelativo = new String();
      int tamanhoDiretorioSelecionado = 0;
      boolean diretorio = arquivo.isDirectory();
      if(!arquivo.isHidden()) {
         if (diretorio) {
            ArquivoCliente novoArquivo = new
ArquivoCliente(arquivo.getURL());
            tamanhoDiretorioSelecionado =
this.getDiretorioSelecionado().length() - 1;
            caminhoRelativo = novoArquivo.getCaminhoArquivo() ;
            caminhoRelativo =
caminhoRelativo.substring(tamanhoDiretorioSelecionado);
            novoArquivo.setCaminhoRelativo(caminhoRelativo);
            listaArquivos.add(novoArquivo);
            FileConnection lista = arquivo;
            Enumeration e = lista.list("*", false);
            while (e.hasMoreElements()) {
               String proximo = (String)e.nextElement();
               String listaa = lista.getURL().concat(proximo);
               FileConnection fileCon = (FileConnection)
Connector.open(listaa, Connector.READ);
               listar(fileCon);
               fileCon.close();
               lista.close();
            }
         }  else {
            ArquivoCliente novoArquivo = new
ArquivoCliente(arquivo.getURL());
            tamanhoDiretorioSelecionado =
this.getDiretorioSelecionado().length() - 1;
            caminhoRelativo = novoArquivo.getCaminhoArquivo() ;
            caminhoRelativo =
caminhoRelativo.substring(tamanhoDiretorioSelecionado);
            novoArquivo.setCaminhoRelativo(caminhoRelativo);
            listaArquivos.add(novoArquivo);
           }
     }
    return(listaArquivos);
  }
```

Quadro 10 – Método de busca na aplicação cliente

As informações que foram recolhidas dos arquivos e diretórios podem ser chamadas de metadados, pois estas informações fazem referência a um elemento único. Estes metadados serão posteriormente repassados ao servidor e organizados de certa forma para facilitar o casamento das informações do cliente e servidor.

Após a busca ter sido efetuada no cliente, o método `gravaArquivo` é iniciado. Este método, apresentado no quadro 11, cria um arquivo texto com as principais informações dos arquivos e diretórios que foram salvos na busca, e o grava no diretório selecionado na interface. Este método também está adaptado na classe `ServidorConexao`, para que as

informações dos arquivos do servidor também sejam persistidas em arquivo texto.

```
public void gravaArquivo(String caminhoSelecionado, ArrayList
listaCliente) {
      try {
         FileConnection fc = (FileConnection)
Connector.open(caminhoSelecionado+"lista.bea",Connector.READ_WRITE);
         if (!fc.exists()) {
            fc.create();
         } else {
            fc.delete();
            fc.create();
         }
         OutputStream out = fc.openOutputStream();
         for(int i = 0; i < listaCliente.size(); i++)
            try {
               ArquivoCliente arquivo =
(ArquivoCliente)listaCliente.get(i);
               String linha = (arquivo.getNomeArquivo()+";"+

String.valueOf(arquivo.getTamanhoArquivo())+";"+
                              arquivo.getCaminhoArquivo()+";"+

String.valueOf(arquivo.getDataUltimaModificacao())+";"+
                              arquivo.getExtensaoArquivo()+";"+
                              arquivo.getHashArquivo()+";"+
                              arquivo.getCaminhoRelativo()+";");
               if((i + 1) < listaCliente.size())
                  linha = linha.concat("\n");
               linha = replaceAll(linha, "/", "\\");
               byte[] novoByte = linha.getBytes();
               out.write(novoByte);
            } catch (Exception e) {
               e.printStackTrace();
            }
         out.flush();
         out.close();
         fc.close();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
```

Quadro 11 – Método que grava as informações dos arquivos em disco

Quando o usuário opta por iniciar a sincronização dos arquivos entre o cliente e o servidor, o cliente envia para o servidor o arquivo texto com os metadados dos seus arquivos a serem sincronizados, para que o servidor realize o casamento destas informações com as informações presentes do servidor.

Optou-se por apresentar somente o método responsável pelo casamento dos metadados referente à opção de sincronização bidirecional, aonde deseja-se que o cliente e o servidor possuam os mesmos arquivos e diretórios. Este método pode ser observado no Apêndice A.

No quadro 13, pode-se observar os principais atributos da classe `Dicionário`, aonde serão carregados os principais metadados para que a busca seja efetuada.

Para cada arquivo presente na lista de arquivos do celular, é verificado se o caminho relativo está presente na lista dos arquivos do computador. Caso encontre um registro correspondente, irá verificar se o *hash* do arquivo de ambos é igual.

O caminho relativo de um arquivo ou diretório é o caminho que inicia-se no diretório selecionado pelo usuário. Se o caminho relativo e o *hash* estiverem presentes no servidor e no cliente, o aplicativo irá verificar qual arquivo possui a data de modificação mais recente, com uma tolerância de 5 segundos para mais e para menos. O arquivo mais recente será mantido no objeto de sufixo `sujoArqLista`, pois este objeto conterá os arquivos a serem sincronizados em ambos os sentidos. Os arquivos que não deverão ser copiados em nenhum sentido são adicionados num objeto de sufixo `limpoArqLista` e removidos da lista de sufixo `sujoArqLista`.

O casamento dos diretórios é feito de forma semelhante conforme apresentado no quadro 12, mas somente o caminho relativo é verificado, já que não é possível fazer o *hash* de um diretório e a data de modificação é um dado irrelevante para o problema a ser solucionado, pois somente é necessário verificar se o diretório está presente no cliente e no servidor, já que os arquivos que podem estar dentro deste diretório já foram devidamente verificados.

```
      listaDois = celularSujoDirLista.getListaDois();
      for(String caminhoOrigem : listaDois.keySet()) {
          String hashOrigem = (String) listaDois.get(caminhoOrigem);
          if(this.computadorSujoDirLista.contemHash(hashOrigem)) {
             listaExclusaoAmbos.add(caminhoOrigem);
          }
       }
       if(!listaExclusaoAmbos.isEmpty()) {
          for(String caminhoExclusao : listaExclusaoAmbos) {
             hashExclusao = (String) listaDois.get(caminhoExclusao);
             arquivoExclusao =
this.celularSujoArqLista.pegarPeloCaminho(caminhoExclusao);

this.celularSujoDirLista.removeRegistroCaminho(caminhoExclusao);

this.computadorSujoDirLista.removeRegistroCaminho(caminhoExclusao);
             this.celularLimpoDirLista.Add(

caminhoExclusao, hashExclusao, arquivoExclusao);
             this.computadorLimpoDirLista.Add(

caminhoExclusao, hashExclusao, arquivoExclusao);
          }
          listaExclusaoAmbos.clear();
          hashExclusao = "";
          arquivoExclusao = null;
       }
```

Quadro 12 – Código que efetua o casamento dos diretórios cliente e servidor

A classe `Dicionario` possui seis atributos, dos quais podemos destacar três, conforme apresentado no quadro 13.

```
private TreeMap<String, Arquivo> listaUm = new TreeMap<String,
Arquivo>();
private TreeMap<String, String> listaDois = new TreeMap<String,
String>();
private TreeMap<String, ArrayList<String>> listaTres = new
TreeMap<String, ArrayList<String>>();
```

Quadro 13 – Principais atributos da classe `Dicionario`

O atributo `listaUm` é um mapa formado pela chave do tipo `String` onde é armazenado o caminho relativo de um arquivo ou diretório, e o valor a ser relacionado com esta chave é um objeto do tipo `Arquivo` que guarda as informações relativas a um arquivo ou diretório.

O atributo `listaDois` é um mapa onde a chave é o caminho relativo de um arquivo, e o valor relacionado à essa chave é o *hash* deste arquivo.

O atributo `listaTres` é um mapa que é o *hash* de um arquivo ou diretório e o valor associado a esta chave é uma lista de caminhos relativos, pois podem existir arquivos com o mesmo *hash* em diferentes caminhos no sistema.

O quadro 11 apresenta como é feito o relacionamento destes atributos.

```
public void Add(String caminho, Arquivo arquivo) {
      listaUm.put(caminho, arquivo);
   }

   public void Add(String caminho, String hash) {
      listaDois.put(caminho, hash);
   }

   public void Add(String caminho, String hash, Arquivo
arquivo) {
      listaUm.put(caminho, arquivo);
      listaDois.put(caminho, hash);
      associaListaTres(hash, caminho);
   }

   private void associaListaTres(String hash, String caminho) {
      if(listaTres.containsKey(hash)) {
         ArrayList<String> temp;
         temp = listaTres.get(hash);
         temp.add(caminho);
         listaTres.put(hash, temp);
      } else {
         ArrayList<String> temp = new ArrayList<String>();
         temp.add(caminho);
         listaTres.put(hash, temp);
      }
   }
```

Quadro 14 – Relacionamento dos mapas

3.3.1.1 Componente *File Browser*

Segundo Netbeans (2011), o *File Browser* é um componente personalizado que fornece uma interface de usuário para trabalhar com o sistema de arquivos do dispositivo. Ele fornece a funcionalidade básica para navegar pelo conteúdo dos dispositivos de memória de armazenamento.

O componente *File Browser* é distribuído juntamente com a aplicação Netbeans e foi utilizado na aplicação cliente para que seja possível selecionar um diretório de uma maneira gráfica. Entretanto, o componente, que é disponibilizado através de uma classe Java, precisou ser alterado para que quando o usuário selecionar um diretório na interface seja possível retornar este caminho para a tela principal do aplicativo cliente. Para isto o trecho de código apresentado no quadro 12 foi adicionado a classe `FileBrowser`.

```
public static final Command selecionaDiretorio = new Command("Seleciona
Pasta", Command.SCREEN, 2);

private String selectedPath;

public void setaDiretorio(List curr) {
     currFile = curr.getString(curr.getSelectedIndex());
     selectedURL = "file:///" + currDirName + currFile;
     selectedPath = currDirName + currFile;
   }

public String getSelectedPath() {
   return selectedPath;
}
```

Quadro 15 – Código inserido na classe `FileBrowser`

3.3.2 Operacionalidade da implementação

Esta seção apresenta a funcionalidade e a operacionalidade da ferramenta, através de um estudo de caso, aonde será sincronizada uma massa com vinte e seis arquivos dispostos em sete diretórios em três subníveis de pastas.

Para que seja possível iniciar o processo de sincronização, o usuário deverá executar o aplicativo servidor em algum computador com acesso à internet. A tela apresentada é demonstrada na figura 9.

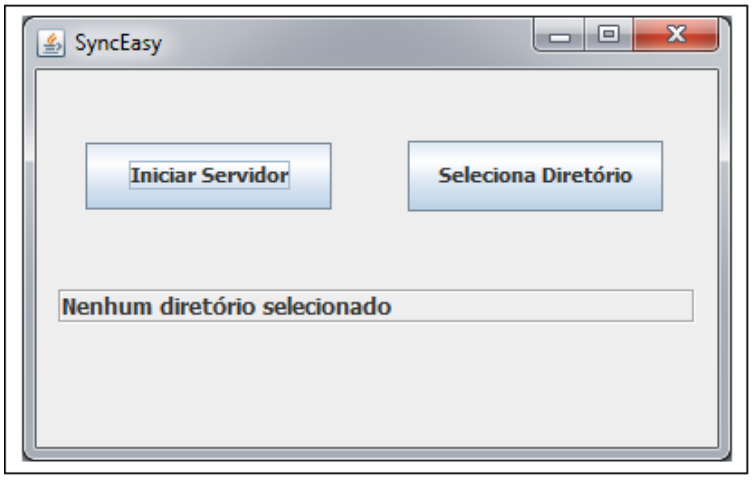

Figura 11 – Tela inicial do servidor

O usuário deverá então selecionar um diretório clicando no botão "Seleciona Diretório". Ao clicar no botão, o aplicativo irá apresentar a tela de seleção de diretório, conforme demonstrado na figura 12.

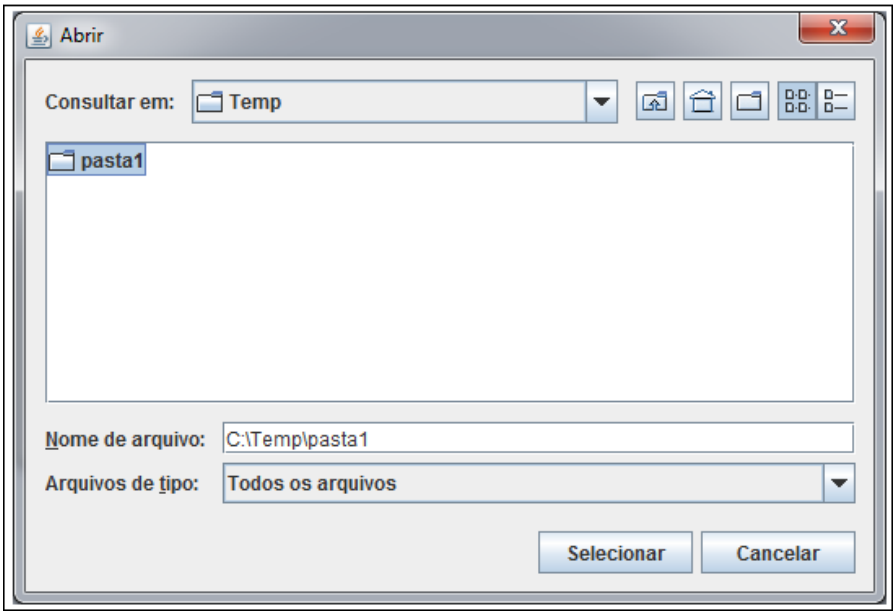

Figura 12 – Tela de seleção de diretório

Após selecionada a pasta que deverá ser sincronizada, o usuário deverá clicar no botão "Iniciar Servidor". Para identificar que o servidor está iniciado, este botão permanecerá desabilitado como demonstra a figura 11. O aplicativo ficará então aguardando por uma conexão vinda de um cliente.

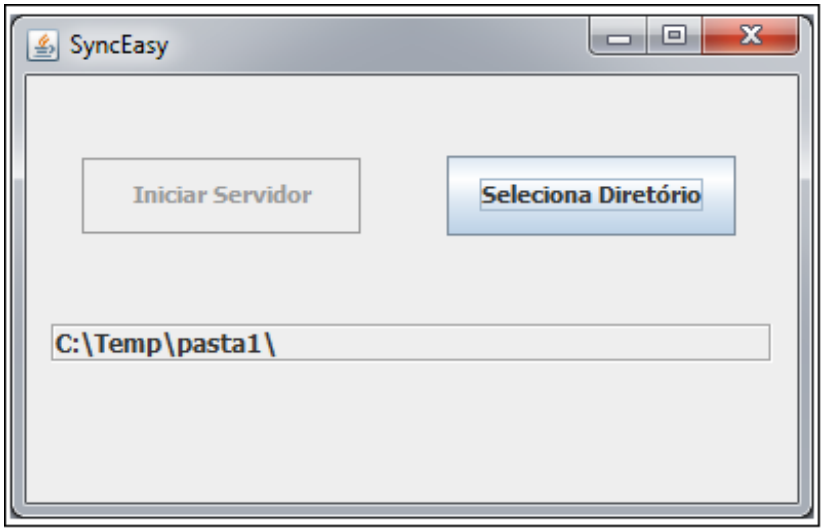

Figura 13 – Servidor iniciado

Quando o usuário iniciar o aplicativo cliente em um celular, a tela inicial é apresentada, conforme demonstra a figura 12.

Figura 14 – Tela inicial do aplicativo cliente

Nesta tela, o usuário deverá selecionar a opção “Menu” e em seguida “Seleciona Local”. Na figura 15 esta tela é apresentada. O aplicativo então irá apresentar ao usuário uma tela onde o usuário poderá navegar no sistema de arquivos do celular, podendo assim selecionar facilmente um diretório.

Figura 15 – Opção seleciona local

Na figura 16 é demonstrado como o usuário deve selecionar um diretório. Após posicionar o cursor no diretório desejado, o usuário deve selecionar a opção “Seleciona Pasta”.

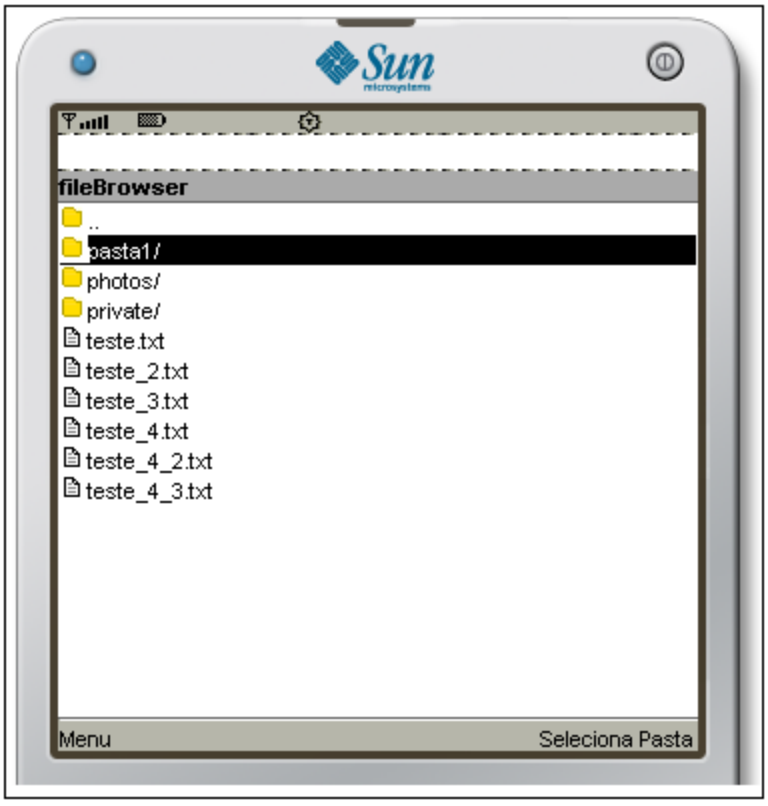

Figura 16 – Seleção de diretório

Para que esta seleção seja acessível à tela inicial do programa, o usuário deverá selecionar a opção “Menu” e “Seleciona Local”, conforme a figura 17.

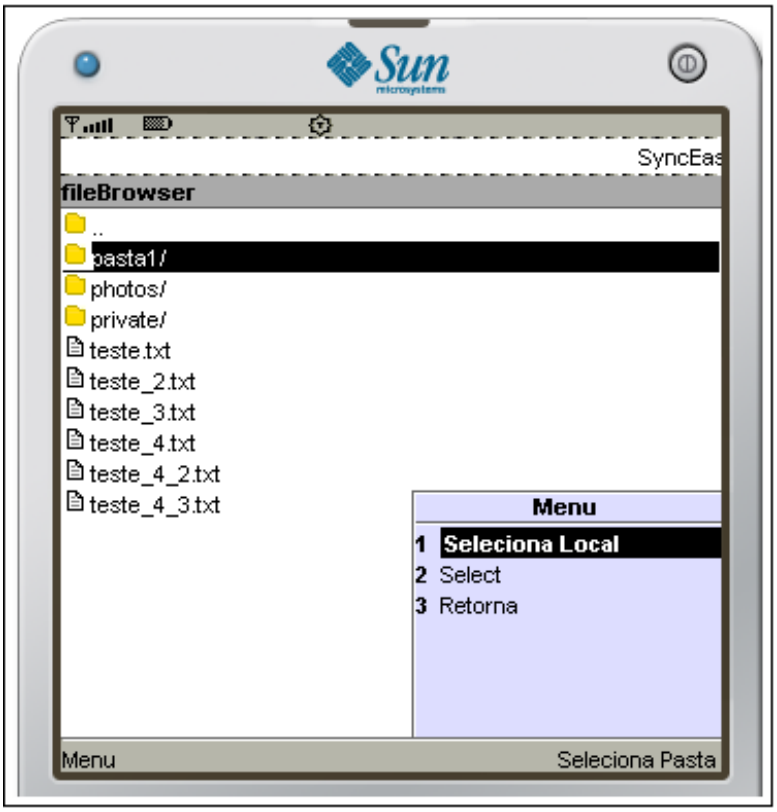

Figura 17 – Grava o diretório selecionado

Após selecionar o diretório o usuário deve retornar a tela inicial, selecionando a opção “Menu” e “Retorna”, como demonstrado na figura 16.

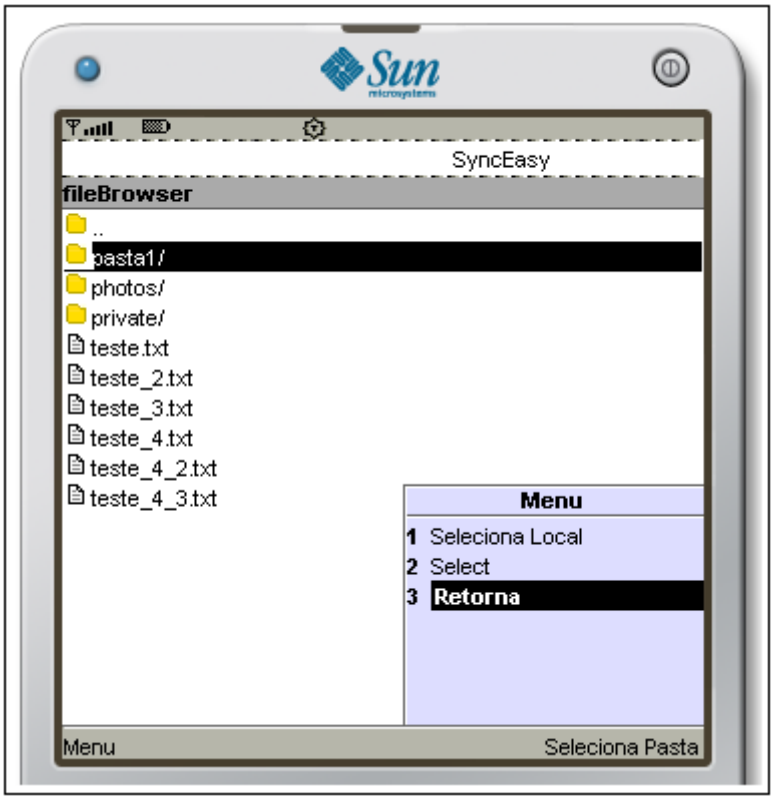

Figura 18 – Retorno à tela principal

Na tela inicial, para que a busca e indexação dos arquivos seja feita, o usuário deve selecionar a opção “Efetua Busca”. Após a busca ter sido efetuada, o aplicativo irá criar um arquivo texto denominado “lista.bea” contendo os metadados referente aos arquivos encontrados e irá gravá-lo no diretório selecionado pelo usuário.

Depois a opção de sincronização, deve ser informado o endereço IP do servidor, e selecionar a opção de sincronização desejada, para que seja possível iniciar a sincronização. Para efeito de teste, foi informado o IP 127.0.0.1 e selecionada a opção de sincronização “Igualar diretórios em Computador e Celular”.

A sincronização pode ser iniciada através da opção “Menu” e “Sincroniza” como pode ser observado na figura 19.

Figura 19 – Inicia a sincronização

O aplicativo cliente irá então se conectar ao servidor e enviar a lista gerada no processo de busca. O servidor ao receber a lista do cliente, irá efetuar a busca no diretório selecionado e criar a lista "listaServidor.bea" com as informações dos arquivos encontrados na busca. Um exemplo da lista gerada na busca, contendo os metadados dos arquivos encontrados pode ser observada no quadro 16.

A primeira linha é referente a um diretório a ser copiado. A segunda linha refere-se a um arquivo chamado `arquivo1.txt`. Cada informação referente ao arquivo ou diretório é separada por ponto-e-vírgula. O primeiro parâmetro é o nome do arquivo ou diretório. O segundo parâmetro é o tamanho do arquivo. Como a primeira linha refere-se a um diretório, este parâmetro é carregado com o valor zero, já que um diretório não possui um tamanho, o tamanho atribuído a um diretório é a soma dos arquivos presentes nele e nas suas subpastas, e para o problema a ser solucionado neste trabalho, não é necessário saber o tamanho total de uma diretório, precisa-se apenas saber se o destino e a origem possuem o mesmo diretório criado, identificando-o pelo nome e caminho relativo. O terceiro parâmetro é o caminho completo desde a raiz do sistema até o diretório que será criado. O quarto parâmetro é a data de ultima modificação do arquivo ou diretório. Na linguagem Java, o valor da ultima modificação de um arquivo é retornado em milissegundos, contando-se a partir do dia

primeiro de janeiro de 1970. Ou seja, quanto maior o resultado retornado, mais recente é o arquivo. O quinto parâmetro é a extensão do arquivo. Como um diretório não possui extensão, é informada a extensão `dir`. O próximo parâmetro é o *hash* do arquivo ou diretório. Como não é possível fazer *hash* de um diretório, é informado o nome do diretório para que seja possível fazer o casamento das listas baseado no *hash* presente nos metadados. O sétimo parâmetro é o caminho relativo, retirando-se o caminho selecionado pelo usuário. No exemplo citado, o caminho selecionado pelo usuário na interface é `C:\Temp\pasta1\`.

```
subnivel\;0;C:\Temp\pasta1\subnivel;1308870669012;dir;subnivel\;\subnivel\;

arquivo1.txt;10035;C:\Temp\pasta1\subnivel\arquivo1.txt;1307060475785;txt;0
5918675593577c848f68e550f795ea2;\subnivel-2\arquivo1.txt;
```

Quadro 16 – Exemplo de lista com metadados

Baseado na opção que o usuário selecionou no aplicativo cliente, o servidor irá efetuar o casamento das duas listas para identificar os conflitos e quais arquivos devem ser copiados em ambos os sentidos. Na opção "Igualar os diretórios em Computador e Celular", os diretórios em cliente e servidor serão igualados. Em todas as opções de sincronização os arquivos iguais, mas com data de modificação diferente, será mantido o arquivo mais recente.

Ao final da sincronização, o aplicativo servidor irá gravar no seu diretório selecionado um arquivo texto chamado "resultado.txt" contendo o caminho dos arquivos copiados, conforme pode ser visto no quadro 17.

```
Copiado para CELULAR
\algo_3.txt
\subnivel\segundo\
\subnivel\segundo\bernardo.txt
\subnivel\segundo-2\denovo.txt

Copiado para COMPUTADOR
\subnivel\oie.txt
\subnivel\outro\mais um\123456.txt
```

Quadro 17 – Arquivo gerado ao fim da sincronização

3.4 RESULTADOS E DISCUSSÃO

A proposta inicial deste trabalho foi criar um aplicativo baseado na arquitetura cliente/servidor para sincronizar arquivos e diretórios entre um computador e um dispositivo móvel baseado na plataforma Symbian. No decorrer dos testes, encontrou-se um problema de

conexão via *socket* nesta plataforma, o que não tornou possível a sua execução na mesma. No entanto, o presente trabalho teve seu desenvolvimento totalmente voltado à plataforma J2ME, para que seja executado em qualquer dispositivo móvel com suporte a aplicativos MIDP utilizado o perfil CLDC.

A falta de uma implementação oficial para fazer *hash* de arquivo em J2ME também precisou ser contornado adicionando-se ao trabalho uma biblioteca não oficial, a qual não é otimizada para fazer *hash* de arquivo, limitando o tamanho do arquivo a ser sincronizado.

O trabalho desenvolvido apresenta como vantagem a utilização da estrutura cliente/servidor, onde o usuário é quem gerencia ambas as aplicações, sem que seus arquivos fiquem armazenados em um local fora do domínio do mesmo.

Uma preocupação durante o desenvolvimento foi manter a qualidade do código, visando otimizar as rotinas, de forma que executassem de forma eficiente. A escolha do método de sincronização baseado em dicionário de chave dupla se mostrou adequado, o que pode ser verificado na seção anterior.

No quadro 18 é feita uma comparação com os trabalhos correlatos a fim de destacar as principais características dos produtos.

	Plataforma	Código aberto
SyncEasy	J2ME	Sim
ActiveSync	Windows Mobile	Não
Dropbox	iOS, Android, BlackBerry	Não

Quadro 18 – Comparação com trabalhos correlatos

4 CONCLUSÕES

Os resultados esperados foram alcançados. O aplicativo desenvolvido é capaz de sincronizar diversos arquivos entre um computador e um dispositivo móvel. Não foram encontradas soluções similares que façam a sincronização via internet com um servidor do próprio usuário, e o presente trabalho foi desenvolvido para preencher esta lacuna.

O algoritmo desenvolvido para a sincronização dos arquivos entre cliente e servidor, se mostrou eficiente e se mostrou muito conciso nos resultados encontrados, sempre sincronizando corretamente os arquivos.

As limitações da linguagem J2ME também trouxeram uma barreira adicional que não era esperada, pois foi necessário, quase que em todo o desenvolvimento, buscar soluções alternativas para os problemas encontrados na linguagem, tal como a falta de um método para fazer *hash* de arquivo e a falta da classe *ArrayList*.

A rotina de *hash* de arquivo não está disponível na linguagem J2ME, e foi encontrada somente uma solução para este problema, ainda que não fosse por completa, pois a mesma não possui um método específico para *hash* de arquivo, somente de *array* de *bytes*.

A dificuldade de entendimento do funcionamento de uma conexão *socket* também gerou uma complexidade não esperada. No entanto os objetivos propostos foram alcançados, disponibilizando um aplicativo de sincronização baseado na plataforma cliente/servidor, sendo que o aplicativo cliente foi desenvolvido na plataforma J2ME que permite que o aplicativo seja executado em uma grande diversidade de dispositivos.

A ferramenta disponibilizada é capaz de sincronizar qualquer tipo de arquivo entre um dispositivo móvel com suporte a J2ME e um computador, via internet de forma confiável.

4.1 EXTENSÕES

Para trabalhos futuros, são apresentadas as seguintes sugestões:

a) implementar uma rotina de *hash* (MD5) de arquivo em J2ME;
b) adicionar algum nível de segurança, por login e senha ou encriptação dos dados trafegados na rede;

c) solucionar o problema de envio de arquivo via *socket* na plataforma Nokia/Symbian Série S60;
d) disponibilizar mais opções de sincronização;
e) disponibilizar mais opções de configuração para o servidor e o cliente;
f) implementar um serviço que fique monitorando o diretório selecionado para identificar mudanças nos arquivos;
g) acrescentar a visualização dos arquivos a serem sincronizados no celular.

REFERÊNCIAS BIBLIOGRÁFICAS

ALECRIM, Emerson. **Sistema de arquivos fat16 e fat32**. [S. l.], 2011. Disponível em: <http://www.infowester.com/fat.php>. Acesso em: 24 abr. 2011.

CALDEIRA, Hyalen M. Java 2 Micro Edition: do Pager ao PDA, com J2ME. **Java Magazine**, Grajaú, Ano 1, n. 1, p. 32-35, abr. 2002.

CARVALHO, Roberto P. **Sistema de arquivos paralelos**: alternativas para a redução do gargalo no acesso ao sistema de arquivos. 2005. 131 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

DEBONE, José E. Z. **Introdução aos diagramas UML**. [S. l.], 2004. Disponível em: <http://www.voxxel.com.br/pages/introdiauml.html>. Acesso em: 24 abr. 2011.

DROPBOX. **Dropbox** – online backup, file sync and sharign made easy. [S.l.], 2010. Disponível em: <https://www.dropbox.com/anywhere>. Acesso em: 08 set. 2010.

FERREIRA, João E.; SOUZA, Marcelo C. de. **Integração assíncrona de instâncias de banco de dados em esquemas homogêneos**. São Paulo, 2009. Disponível em: <http://www.ime.usp.br/~mcsouza/jef_mcsouza.pdf>. Acesso em: 09 abr. 2011.

GUBIAN, Paolo; SAVOLDI, Antonio. Issues in Symbian S60 platform forensics. **Journal of Communication and Computer**. USA, Mar. 2009. Disponível em: <http://www.informatics.org.cn/doc/ucit200903/ucit20090303.pdf>. Acesso em: 11 maio 2011.

IBGE. **Banco de metadados**. [S.l.], 2008. Disponível em: <http://www.metadados.ibge.gov.br/>. Acesso em: 21 fev. 2011.

IMASTERS. **Metadados -** imasters. [S.l.], 2003. Disponível em: <http://imasters.com.br/artigo/1569/bi/metadados/>. Acesso em: 21 fev. 2011.

JIPPING, Michael J. **Smartphone operating system concepts with symbian OS.** [S.l.]: Wiley, 2007. 356p.

KIOSKEA. **Ambiente cliente/servidor**. [S.l.], 2009. Disponível em: <http://pt.kioskea.net/contents/cs/csintro.php3>. Acesso em: 26 abr. 2011.

MACINTA, Timothy W. **Fast MD5 implementation in java**. [S.l.], 2011. Disponível em: <http://www.twmacinta.com/myjava/fast_md5.php>. Acesso em: 30 jun. 2011.

MENEZES, Eduardo D. B. de. **Princípios de análise e projeto de sistemas com UML**. Rio de Janeiro: Campus, 2002. 286p.

MICROSOFT. **ActiveSync 4.5 -** português (Brasil). [S.l.], 2007. Disponível em: <http://www.microsoft.com/downloads/details.aspx?displaylang=pt-br&FamilyID=9e641c34-6f7f-404d-a04b-dc09f8141141>. Acesso em: 15 ago. 2010.

MORIMOTO, Carlos E. **NTFS**. [S.l.], 2007. Disponível em: <http://www.hardware.com.br/livros/hardware/ntfs.html>. Acesso em: 21 abr. 2011.

NETBEANS. **Visual mobile designer custom components:** creating a mobile device file browser. [S.l.], 2011. Disponível em: <http://netbeans.org/kb/docs/javame/filebrowser.html>. Acesso em: 12 maio 2011.

ORFALI, Robert et al. **Essential client/server survival guide**. [S.l.], 1994. 527p.

PRADO. **JEE –** um caminho prazeroso e interessante! [S.l.], 2009. Disponível em: <http://www.devmedia.com.br/articles/post-3747-JEE-um-caminho-prazeroso-e-interessante.html>. Acesso em: 12 mar. 2011.

SCHMITT, Alan et al. **Exploiting schemas in data synchronization**. [S.l.], 2005. Disponível em: <http://www.cis.upenn.edu/~bcpierce/papers/sync-dbpl.pdf>. Acesso em: 12 abr. 2011.

SOFTONIC. **ActiveSync 4.5**. [S.l.], 2008. Disponível em: <http://activesync.softonic.com.br/windowsmobile>. Acesso em: 08 set. 2010.

_____. **Dropbox 1.2.3**. [S.l.], 2010. Disponível em: <http://dropbox.softonic.com.br/celular>. Acesso em: 08 set. 2010.

SYNCSHARP. **Syncsharp –** plug & sync. [S.l.], 2010. Disponível em: <http://code.google.com/p/syncsharp/>. Acesso em: 21 fev. 2011.

SUEL, Torsten; MEMON, Nasir. **Algorithms for delta compression and remote file synchronization**. New York, v.1, p. 11-16, 2002. Disponível em: <http://cis.poly.edu/~suel/papers/delta.pdf>. Acesso em: 12 mar. 2011.

SUN MICROSYSTEMS. **Java ME technology.** [S.l.], [2010]. Disponível em: <http://java.sun.com/javame/technology/index.jsp>. Acesso em: 04 set. 2010.

_____. **Java EE FAQ**. [S.l.], 2009. Disponível em: <http://java.sun.com/javaee/overview/faq/javaee_faq.jsp>. Acesso em: 10 mar. 2011.

TANENBAUM, Andrew S.; WOODHULL, Albert S. **Sistemas operacionais**: projeto e implementação. 2. ed. Tradução Edson Furmankiewicz. Porto Alegre: Bookman, 2000.

VALENTE, Marco T. O. et al. Chamada remota de métodos na plataforma J2ME/CLDC. **J2ME**, Minas Gerais, v .12, p. 1-12, 2003. Disponível em: <http://compilers.cs.ucla.edu/fernando/publications/papers/5oWCSF.pdf>. Acesso em: 12 mar. 2011.

APÊNDICE A – Código da classe Gerador para efetuar o casamento dos dicionários

O trecho de código apresentado no quadro 19 efetua o casamento dos metadados a partir das informações encontradas no cliente. O método carrega as informações dos metadados do cliente e do servidor em objetos da classe `Dicionario`. Estes objetos possuem na sua estrutura listas e métodos que facilitam a busca por arquivos iguais.

```
  for(Arquivo arquivo : listaCelular) {
    if(!arquivo.isEhDiretorio()) {
      this.celularSujoArqLista.Add(
      arquivo.getCaminhoRelativo(), arquivo.getHashArquivo(), arquivo);
    } else {
      this.celularSujoDirLista.Add(
      arquivo.getCaminhoRelativo(), arquivo.getHashArquivo(), arquivo);
    }
  }
      this.computadorSujoDirLista = new Dicionario();
      this.computadorSujoArqLista = new Dicionario();
      for(Arquivo arquivo : listaComputador) {
         if(!arquivo.isEhDiretorio()) {
            this.computadorSujoArqLista.Add(
        arquivo.getCaminhoRelativo(), arquivo.getHashArquivo(), arquivo);
         } else {
            this.computadorSujoDirLista.Add(
        arquivo.getCaminhoRelativo(), arquivo.getHashArquivo(), arquivo);
         }
       }
      Arquivo arquivoDestino = new Arquivo();
      Arquivo arquivoOrigem = new Arquivo();
      ArrayList<String> listaExclusaoCelular = new ArrayList<String>();
      ArrayList<String> listaExclusaoComputador = new ArrayList<String>();
      ArrayList<String> listaExclusaoAmbos = new ArrayList<String>();
      String hashExclusao;
      Arquivo arquivoExclusao;
    TreeMap<String,String> listaDois = celularSujoArqLista.getListaDois();
     for(String caminhoOrigem : listaDois.keySet()) {
         String hashOrigem = (String) listaDois.get(caminhoOrigem);
         if(this.computadorSujoArqLista.contemHash(hashOrigem)) {
            if(this.computadorSujoArqLista.contemCaminho(caminhoOrigem)) {
             arquivoDestino =
this.computadorSujoArqLista.pegarPeloCaminho(caminhoOrigem);
             Long dataDestino = arquivoDestino.getDataUltimaModificacao();
             arquivoOrigem =
this.celularSujoArqLista.pegarPeloCaminho(caminhoOrigem);
             Long dataOrigem = arquivoOrigem.getDataUltimaModificacao();
             Long resultado = dataDestino - dataOrigem;
              if(resultado > diferencaDataMais) {
                  listaExclusaoCelular.add(caminhoOrigem);
               } else {
                  if(resultado < diferencaDataMenos) {
                     listaExclusaoComputador.add(caminhoOrigem);
                  } else {
                     listaExclusaoAmbos.add(caminhoOrigem);
                  }
               }
```

```
            }
         }
      }
      if(!listaExclusaoAmbos.isEmpty()) {
         for(String caminhoExclusao : listaExclusaoAmbos) {
            hashExclusao = (String) listaDois.get(caminhoExclusao);
            arquivoExclusao =
this.celularSujoArqLista.pegarPeloCaminho(caminhoExclusao);

this.celularSujoArqLista.removeRegistroCaminho(caminhoExclusao);

this.computadorSujoArqLista.removeRegistroCaminho(caminhoExclusao);
            this.celularLimpoArqLista.Add(caminhoExclusao, hashExclusao,
arquivoExclusao);
            this.computadorLimpoArqLista.Add(caminhoExclusao,
hashExclusao, arquivoExclusao);
         }
         listaExclusaoAmbos.clear();
         hashExclusao = "";
         arquivoExclusao = null;
      }
       if(!listaExclusaoCelular.isEmpty()) {
         for(String caminhoExclusao : listaExclusaoCelular) {
           hashExclusao = (String) listaDois.get(caminhoExclusao);
          this.celularSujoArqLista.removeRegistroCaminho(caminhoExclusao);
         }
         listaExclusaoCelular.clear();
         hashExclusao = "";
      }
      if(!listaExclusaoComputador.isEmpty()) {
         for(String caminhoExclusao : listaExclusaoComputador) {
           hashExclusao = (String) listaDois.get(caminhoExclusao);
       this.computadorSujoArqLista.removeRegistroCaminho(caminhoExclusao);
         }
         listaExclusaoComputador.clear();
         hashExclusao = "";
      }
```

Quadro 19 - Casamento da lista do cliente com a do servidor