

Gabriel Henrique Martinez Saraiva

**Implementação e Adaptação de um Sistema de
Arquivos Distribuídos Adaptável e Flexível para
Dispositivos Móveis**

São José do Rio Preto

2015

Gabriel Henrique Martinez Saraiva

Implementação e Adaptação de um Sistema de Arquivos Distribuídos Adaptável e Flexível para Dispositivos Móveis

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP)

Instituto de Biociências, Letras e Ciências Exatas (IBILCE)

Bacharelado em Ciência da Computação

Profa. Dra. Renata Spolon Lobato

São José do Rio Preto

2015

Gabriel Henrique Martinez Saraiva

Implementação e Adaptação de um Sistema de Arquivos Distribuídos Adaptável e Flexível para Dispositivos Móveis

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Prof^a. Dr^a. Renata Spolon Lobato

Gabriel Henrique Martinez Saraiva

Banca Examinadora:

Profa. Dra. Adriana Barbosa Santos

Prof. Dr. Aleardo Manacero Jr.

São José do Rio Preto

2015

Dedico esse trabalho, principalmente a minha mãe, Cássia, a mim mesmo, e a todos que algum dia, de alguma forma colaboraram para que tornasse realidade.

Agradecimentos

Agradeço a Deus, pela chance de escrever cada página desse livro que chamo de vida;

À minha mãe, Cassia, pelo amor, carinho, paciência, pelo exemplo, por acreditar em mim e sempre estar ao meu lado como meu anjo da guarda; Ao meu pai, Edevaldo, pelo amor e carinho, por sempre ter me desafiado e motivado a não ter medo de tentar;

Ao meu irmão, companheiro e melhor amigo, Douglas, por sempre estar ao meu lado, nas bagunças, nas obrigações e no estudo; A minha amiga, companheira e irmã, Thais, pela incrível paciência de me aturar quando pequeno; Aos meus amados sobrinhos, Ingrid e João Pedro, que são como meus irmãos, atentados e muito amados, por sempre serem meus companheiros de arte; À minha pequena monstrix, amada e linda irmãzinha, Anne, por me lembrar de como é bom ser criança;

Ao meu segundo pai, Cido, que sempre esteve presente, principalmente quando mais precisei; À Tati, pela amizade, carinho e por sempre me incentivar nos estudos;

À minha querida e amada Vó, Tunica, que sempre cuidou de mim com carinho e amor; Ao meu Vô, Vô Zé, que também cuidou muito de mim sempre com amor e carinho;

A minhas tias, Magda, Márcia e Jú, pelo carinho e por cuidarem sempre de mim com amor e carinho; Aos meus tios Branco, Victor, Waldemar e Lú pelo carinho e amor; Aos meus primos Deivid e Raul, pela amizade e por estarem juntos nas bagunças;

Ao Alexandre, meu cunhado, pela amizade e pela sinuca; A Neusa e a Ana que cuidaram muito de mim quando pequeno;

A minha amiga e namorada, Jéssica de Arruda, pelo amor, pela amizade, e por me aturar durante os momentos mais difíceis da graduação; A minha amiga, Natália Faria, que sempre me colocou no caminho certo;

Ao Gildo, por me mostrar que eu era capaz;

Aos **Putos™** : Rafael, Enéas, Alysson, Édipo, Victor e Santos, pela amizade verdadeira e duradoura, daquelas que a gente conta nos dedos;

Aos amigos que ajudaram um desconhecido no início de sua jornada: Matheus Della Croce (Napa), Diogo Tavares (Sertanejo), Pedro (Física), Lúcio (DM), Carlos (Lost), Marcos (Marquinho), Paulo (Kapial) e Victória Régia (capa bixo).

Aos amigos que moraram comigo e foram minha família: Tainara Alves (Tata), Alex Honorato, Lucas dos Santos (Preto), Igor Buttarello (Sabiá), Ronan (Maguilão), Tarcísio (Turco), Maine (Joaninha), James Miranda, Bruna Paula, Moisés (Mois), Marcélia (Marcela), Jéssica e Alisson, Julie (Paraguaia), Matheus Gonçalves, Rogério (Sheldon).

Ao Mâriô Cãmãrá Nêtó (☺), pela amizade sincera e por me ensinar a beber (e esse sorrisinho marooto??? kkk).

Aos bons amigos que conheci ao longo desses 5 anos, pela amizade e bons momentos sem os quais isso não seria suportável: Brait (Dunha), Denison Menezes, Daniel Angelotti (Pesaado), Leandro Barbosa (46! PosComp), Willian (Amarelo), Heitor (Suba), Gustavo Teixeira, Thiago (Japa), Guilherme (Turquinho), Danilo Costa (Gordinho Sexy), Vinícius (Preto), Vinícius (Galhardi), Gabriel Covello (Coca-Cola), Leonardo Santos (Dotinha), Cassio Forte (Parabéns!), Rafael Stabile (Pai), Porfirio Reis, Arthur Bressan (Sheldon) Arthur (Fatal), João Magri (Letra A), Victor H. Cândido (VHC) Brunno (Gordinho), Braulio; Ao Lucas Cazarote (Tesouro), Fernanda (Ferzinha do GSPD), Renan Alboy (Delicinha do GSDP), Lucas (Tatu), Aialy (Bixete do 1º ano!), Erika (Palmitão), Tinha (Tinha), Ricardo (Bozo), Fernando (Kawai), Marco Túlio e muitos outros que contribuíram de algum modo com suas amizades para essa conquista.

Aos meus professores, todos os que um dia me ensinaram algo, em especial a Professora Renata e ao Professor Aleardo, que além de professores foram grandes amigos e orientadores, os quais tive um imenso prazer em poder conhecer e trabalhar junto durante minha graduação. Meus sinceros agradecimentos.

A Unesp e ao IBILCE, e a todos os Unespianos que formam essa Universidade maravilhosa. *"Loouuucoooo.... Louco louco louco... Eu sou da UNESP!"*

Ao meu notebook MUB-342 por chegar até aqui, e resistir bravamente nesses 5 anos...

A todos esses, os quais isso jamais seria possível ou suportável. Obrigado por fazer valer, cada segundo, cada gota de suor.

E por fim, a mim mesmo, afinal de contas... quantas vezes você disse que não aguentava mais, mas continuou seguindo em frente?

*"E quantas vezes você disse que não aguentava mais,
mas continuou seguindo em frente?"*

-autor desconhecido

"You have a brain. Use it!"

-Gabriel Saraiva

Resumo

Sistemas de arquivos distribuídos são de grande importância pois fornecem uma das bases para a implementação de outros sistemas, principalmente os que utilizam computação em nuvem. Nesse trabalho é apresentado um estudo sobre sistemas distribuídos, sistemas de arquivos convencionais e distribuídos, chamada de procedimento remoto e dispositivos móveis. Esse estudo serve como fundamentação teórica para o desenvolvimento de um módulo cliente de um sistema de arquivo distribuído flexível e adaptável para dispositivos móveis que utilizem Android. Após a descrição do trabalho executado são apresentados testes de validação e desempenho, seguido de conclusões e sugestões para trabalhos futuros.

Palavras-chave: Android, chamada de procedimentos remotos, sistemas distribuídos, sistemas de arquivos distribuídos, dispositivos móveis.

Abstract

Distributed file systems have great importance as they provide one of the basis for the implementation of other systems, mainly those that use cloud computing. In this work is presented a study on distributed systems, conventional and distributed file systems, remote procedure call and mobile devices. This study serves as theoretical foundation to the development of a client module of a flexible and adaptable distributed file system for mobile devices that run Android. After the description of the executed work are presented performance and validation tests, followed by conclusions and suggestions for future works.

Keywords: Android, remote procedure call, distributed systems, distributed file systems, mobile devices.

Résumé

Systèmes des fichiers distribués sont important, car ils fournissent une base pour le déployer autres systèmes, surtout les que font le cloud computing. Ce travail présente une étude sur les systèmes distribués, systèmes de fichiers conventionnels et distribués, remote procedure call et appareils mobiles. Cette étude sert comme la motivation théorique pour développement du module client du système de fichier distribué flexible et adaptable par appareils mobiles. Après la description du travail a effectué sont exhibait tests de validation et performance, suivre des conclusions et suggestions pour travaux avenir.

MOTt-CLÉ : Android, remote procedure call, systèmes distribués, système des fichiers distribués, appareils mobiles.

Lista de ilustrações

Figura 1 – Arquitetura do NFS (COULOURIS; DOLLIMORE; KINDBERG, 2005)	22
Figura 2 – Arquitetura do AFS (COULOURIS; DOLLIMORE; KINDBERG, 2005)	24
Figura 3 – Arquitetura original do FlexA (FERNANDES, 2012).	26
Figura 4 – Arquitetura atual do FlexA (NETO, 2013).	26
Figura 5 – Fluxo de dados no FlexA original (FERNANDES, 2012).	27
Figura 6 – Fluxo de dados no FlexA atual, elaborado a partir de (NETO, 2013).	27
Figura 7 – Diagrama sobre a criação das chaves de acesso e criptografia do FlexA (NETO, 2013).	28
Figura 8 – Diagrama que exhibe a comunicação via XML-RPC. (CISCO, 2010) (Adaptado)	31
Figura 9 – Exemplo de mensagem XML utilizada na chamada de procedimentos remotos com XML-RPC.	32
Figura 10 – Camadas de software que compõe o sistema Android (Open Handset Alliance, 2014).	33
Figura 11 – Módulos do sistema atual e suas dependências. Criado com base no código fonte do projeto de Neto (2013).	35
Figura 12 – Classes e pacotes (arquivos) que estruturam a versão atual do FlexA	36
Figura 13 – Diagrama de Casos de uso com os requisitos funcionais do FlexA.	37
Figura 14 – Interface de comunicação dos servidores do FlexA.	38
Figura 15 – Protocolo de requisição dos metadados de um usuário com base na sua chave pública.	39
Figura 16 – Requisição do <i>salt</i> de um arquivo.	39
Figura 17 – Cadastro do arquivo nos servidores e negociação da porta de envio do arquivo	40
Figura 18 – Transmissão das porções dos arquivos aos servidores, feito via socket.	40
Figura 19 – Requisição da lista dos arquivos em um diretório	40
Figura 20 – Já com a <i>verify key</i> , é solicitado uma lista dos servidores que possuem as partes do arquivo	41
Figura 21 – Módulo cliente envia informação para os servidores, negociando o recebimento das partes	41
Figura 22 – Servidores enviando porções do arquivo para o cliente	41
Figura 23 – Protocolo que define a remoção das partes de um arquivo.	42
Figura 24 – Protocolo para mover um arquivo.	42
Figura 25 – Protocolo para o compartilhamento de arquivos	42
Figura 26 – Protocolo para o listar os arquivos compartilhados com um usuário.	43
Figura 27 – Remove o acesso do usuário <i>B</i> a um arquivo compartilhado pelo usuário <i>C</i>	43
Figura 28 – Servidor envia seu <i>serverID</i> para cliente.	43
Figura 29 – Servidor envia para cliente métricas sobre seu estado atual.	44

Figura 30 – Diagrama de classes mostrando a classe <i>Share</i> , responsável por armazenar os metadados sobre o compartilhamento dos arquivos.	44
Figura 31 – Hierarquia de pacotes que compõe o módulo Cliente para Android	45
Figura 32 – Classe <i>Client</i> , responsável por encapsular servidores e manipulação dos arquivos.	46
Figura 33 – Interface de comunicação com os servidores XML-RPC.	47
Figura 34 – Exemplo de interface gráfica listando os arquivos do usuário.	47
Figura 35 – Exemplo de duas tarefas assíncronas, uma feita via XML-RPC (<i>listFiles</i>) e outra via <i>socket</i> (<i>scanServers</i>).	48
Figura 36 – Seções das classes de criptografia e acesso a arquivos.	49
Figura 37 – Módulo de configurações, fornece uma melhor usabilidade do sistema. . . .	49
Figura 38 – Teste de integridade dos arquivos com Android. Nenhum arquivo foi danificado durante os testes.	53
Figura 39 – Teste de compatibilidade entre os dois módulos clientes do FlexA. Nenhum arquivo foi danificado durante os testes mostrando que os sistemas estão compatíveis.	53
Figura 40 – Teste de desempenho de criptografia com clientes diferentes do FlexA . . .	54
Figura 41 – Teste de desempenho de transmissão de arquivos com clientes diferentes do FlexA	55

Lista de tabelas

Tabela 1 – Notação utilizada para definição de protocolos de comunicação (ANDERSON, 2008).	38
Tabela 2 – Tempos de criptografia para os dispositivos utilizados.	54
Tabela 3 – Tempos de transmissão dos arquivos utilizados para testes através de rede <i>wi-fi</i>	54

Lista de abreviaturas e siglas

ACL	-	<i>Access Control List</i>
AFS	-	<i>Andrew File System</i>
EXT2	-	<i>second extended filesystem</i>
EXT3	-	<i>third extended filesystem</i>
EXT4	-	<i>fourth extended filesystem</i>
FAT	-	<i>File Allocation Table</i>
GSPD	-	Grupo de Sistemas Paralelos e Distribuídos
MD5	-	<i>Message-Digest algorithm 5</i>
NFS	-	<i>Network File System</i>
NTFS	-	<i>New Technology File System</i>
RK	-	<i>Read Key</i>
RPC	-	<i>Remote Procedure Call</i>
RSA	-	Rivest, Adi Shamir e Leonard Adleman
SAD	-	Sistemas de Arquivos Distribuídos
SA	-	Sistemas de Arquivos
SD	-	Sistema Distribuído
SHA	-	<i>Secure Hash Algorithm</i>
UML	-	<i>Unified Modeling Language</i>
VK	-	<i>Verify Key</i>
WK	-	<i>Write Key</i>
XGH	-	<i>eXtreme Go Horse</i>
XML	-	<i>eXtensible Markup Language</i>

Sumário

1	INTRODUÇÃO	16
1.1	Motivação	16
1.2	Objetivos	16
1.3	Organização do texto	17
2	FUNDAMENTAÇÃO TEÓRICA	18
2.1	Sistemas Distribuídos	18
2.1.1	Heterogeneidade	18
2.1.2	Abertura	18
2.1.3	Segurança	18
2.1.4	Escalabilidade	19
2.1.5	Tratamento de falhas	19
2.1.6	Transparência	20
2.2	Sistema de Arquivos	20
2.3	Sistemas de Arquivos Distribuídos	21
2.3.1	<i>Network File System</i> (NFS)	22
2.3.2	<i>Andrew File System</i> (AFS)	23
2.3.3	<i>Flexible and Adaptable distributed file system</i> (FlexA)	25
2.3.4	Segurança	26
2.3.5	Tolerância a falhas e Adaptabilidade	29
2.3.6	Escalabilidade	29
2.3.7	Flexibilidade	30
2.3.8	Abertura	30
2.4	Chamada de Procedimentos Remotos (RPC)	30
2.4.1	XML-RPC	31
2.5	Dispositivos Móveis	32
2.5.1	Dispositivos Móveis	32
2.5.2	Computação Móvel	32
2.6	Android	32
2.7	Considerações finais	33
3	DESCRIÇÃO E DESENVOLVIMENTO DO PROJETO	34
3.1	Restrições da versão atual do FlexA	34
3.2	Documentação do projeto	35
3.2.1	Módulos do Sistema	35
3.2.2	Diagrama de classes do sistema existente	36

3.2.3	Diagrama de Casos de Uso	37
3.2.4	Interface de Comunicação Cliente-Servidor	37
3.2.5	Protocolos de Comunicação Cliente-Servidor	38
3.2.6	Protocolos de manipulação de arquivos	38
3.2.7	Protocolos de obtenção de metadados dos servidores	43
3.3	Adaptação do servidor em Python	44
3.4	Módulo Cliente para Android	45
3.4.1	Organização	45
3.4.2	Modularização e Flexibilidade	46
3.4.3	Encapsulamento da complexidade do FlexA	46
3.4.4	Encapsulamento da comunicação com os servidores que utilizam XML-RPC	46
3.4.5	Interface Gráfica do Usuário no Android	47
3.5	Considerações finais	50
4	TESTES E RESULTADOS	51
4.1	Ambiente de Testes	51
4.2	Integridade dos Dados	52
4.3	Compatibilidade do sistema	52
4.4	Desempenho de criptografia	52
4.5	Desempenho de transmissão dos dados	54
5	CONCLUSÃO	56
5.1	Dificuldades	56
5.2	Trabalhos futuros	57
	Referências	58

1 Introdução

A rápida evolução da computação, devido a concorrência entre os fabricantes e sua demanda perene, possibilitou o barateamento e o aumento na capacidade dos sistemas computacionais. Resultado de grandes avanços tecnológicos, dos microprocessadores aliado aos sistemas de conexão e redes, a computação se tornou onipresente em muitos campos de nossa sociedade, como destacado por (TANENBAUM; STEEN, 2007).

A evolução desses sistemas proporcionou a solução de problemas maiores e mais complexos, que comumente não são possíveis de serem solucionados com dispositivos isolados. Para a solução desses problemas são utilizados sistemas distribuídos, onde sistemas computacionais diversos cooperam, se comunicando via redes de computadores e internet para atingir um objetivo comum (COULOURIS; DOLLIMORE; KINDBERG, 2005).

Dentro da computação distribuída, como é necessário armazenar esses dados e fazer o compartilhamento deles, existem os sistemas de arquivos distribuídos, que são responsáveis por fornecer acesso aos dados de forma constante e confiável. Nesse trabalho será tratado o FlexA (*Flexible and Adaptable Distributed File System*), desenvolvido no GSPD (Grupo de Sistemas Paralelos e Distribuídos) da UNESP.

1.1 Motivação

Dentre os fatores que motivam esse trabalho estão a necessidade de tornar o FlexA um sistema mais fácil de ser expandido no futuro, através de interfaces que tornem o projeto mais aberto e flexível com relação ao desenvolvimento e arquitetura. Também, aproveitando a grande popularização de dispositivos de computação móvel como, por exemplo, *tablets* e *smartphones*, que com a presença e uso constante desses dispositivos o FlexA poderia ser utilizado pelo usuário de forma prática e rápida.

1.2 Objetivos

Os objetivos gerais desse trabalho são adaptar as interfaces de comunicação do FlexA, de modo a remodelar interfaces de comunicação entre cliente e servidor, e implementar uma versão do módulo cliente do sistema para dispositivos móveis que utilizem Android.

Os objetivos específicos são a implementação das operações desconectadas que são responsáveis por gerenciar o FlexA enquanto estiver desconectado e sincronizá-lo ao conectar novamente com os servidores; implementação de um módulo de controle de falhas de conexão, e mecanismos mínimos de segurança dos dados e da comunicação com os servidores, utilização

de criptografia e chaves de identificação. Como esses dispositivos possuem uma capacidade de armazenamento menor, existe a necessidade de um melhor gerenciamento dos arquivos residentes na memória cache do sistema de arquivo distribuído (SAD).

1.3 Organização do texto

No trabalho estão, além dessa introdução, no capítulo 2 a fundamentação teórica sobre a qual o trabalho foi desenvolvido, com o estudo de sistemas distribuídos, sistemas de arquivos distribuídos, chamada de procedimentos remotos e também sobre Android e dispositivos móveis. No capítulo 3 é descrito o desenvolvimento do projeto e são mostrados os diversos mecanismos utilizados e implementados. O capítulo 4 apresenta os cenários de testes utilizados, e os testes realizados para validação do projeto. Por fim, no capítulo 5 são feitos comentários sobre a execução do projeto desenvolvido e dos testes realizados, e são apresentadas as dificuldades e novos caminhos para projetos futuros.

2 Fundamentação Teórica

Neste capítulo são apresentados os conceitos sobre sistemas distribuídos, sistemas de arquivos, sistemas de arquivos distribuídos (com foco na comunicação entre cliente e servidor), tecnologias utilizadas como chamadas de procedimento remotos e uma breve descrição sobre dispositivos móveis, apresentando suas restrições.

2.1 Sistemas Distribuídos

Segundo Tanenbaum e Steen (2007), um Sistema Distribuído (SD) é um conjunto de computadores independentes que se apresenta aos usuários como um sistema único e coerente, ocultando detalhes da implementação que não são úteis aos usuários. Ainda sim um sistema distribuído pode ser definido de acordo com Coulouris, Dollimore e Kindberg (2005) como um sistema onde os componentes computacionais, tanto os de *hardware* como os de *software*, se comunicam e coordenam suas ações através de mensagens transmitidas por uma ou mais redes de computadores, independente da distância entre seus componentes. Essas características, agregam diversos desafios na implementação de sistemas distribuídos. De acordo com Coulouris, Dollimore e Kindberg (2005) esses desafios são:

2.1.1 Heterogeneidade

Como esses sistemas podem ser compostos por *hardwares* e arquiteturas, sistemas operacionais, *softwares* e configurações diversas é de grande importância que todo o sistema seja baseado em protocolos compatíveis que possam gerenciar essas diferenças e fornecer ao usuário um ambiente único e coeso.

2.1.2 Abertura

Um sistema é dito aberto quando suas interfaces de comunicação ou de desenvolvimento são acessíveis e programas podem ser feitos ou alterados por terceiros para que sejam compatíveis com esse sistema. Os grandes desafios para criar sistemas abertos são: fornecer principais interfaces publicamente, mecanismos de comunicação uniforme para acesso a recursos compartilhados e por fim, realizar os testes necessários para que esse sistema se comporte como esperado e seja compatível com o padrão publicado.

2.1.3 Segurança

Um dos grandes desafios de sistemas distribuídos é manter a segurança do próprio sistema de modo a manter a disponibilidade do serviço a seus clientes, confidencialidade dos

dados que ele armazena ou processa e não menos importante a integridade do sistema e dos dados que estão no SD. Esse tópico é de grande importância na maioria dos SD disponíveis pois afeta principalmente a disponibilidade do serviço aos usuários e a confiança que o usuário tem sobre o serviço (COULOURIS; DOLLIMORE; KINDBERG, 2005).

2.1.4 Escalabilidade

Um sistema é dito escalável se seu desempenho cresce proporcionalmente aos recursos que são inserido no SD, sem perda de qualidade ou desempenho do serviço. Essa característica é muito importante para que o sistema continue operando em cenários que exijam mais recursos do que o cenário inicial para o qual o foi sistema implantado. Embora pareça um desafio fácil de solucionar, como mostrado por Coulouris, Dollimore e Kindberg (2005), as soluções desses desafios não são sempre triviais, como por exemplo gerenciar a perda de desempenho do sistema ao aumentar o número de servidores e clientes em uma rede, ou manter a latência dentro de um patamar aceitável com o crescimento do número de servidores e a distância entre eles.

2.1.5 Tratamento de falhas

Como o sistema é distribuído, falhas tendem a não comprometer a totalidade do sistema, afetando apenas alguns pontos por vez enquanto outros continuam operando. Mas isso torna essas falhas complexas de serem tratadas, para que o sistema volte a operar como esperado (COULOURIS; DOLLIMORE; KINDBERG, 2005). Para tratar essas falhas são utilizadas as seguintes técnicas:

1. Detecção de falha: Erros de transmissão por exemplo podem ser detectados e facilmente corrigidos através de mecanismos próprios e algoritmos bem conhecidos como por exemplo os algoritmos de *hash Message-Digest algorithm 5* (MD5) e o *Secure Hash Algorithm* (SHA);
2. Mascaramento de falha: Essa técnica consiste em aumentar a resistência do sistema as falhas, como por exemplo replicar informações entre diversos servidores e solicitar novamente a transmissão das mensagens que estejam danificadas;
3. Tolerância a falhas: Nem sempre é possível construir sistemas a prova de falhas, sendo muito mais simples e barato projetá-los para tolerar algumas falhas até certo ponto e depois disso informar o usuário sobre a falha ocorrida. Um exemplo clássico citado por Coulouris, Dollimore e Kindberg (2005) é o do navegador *web* que, após um certo número de tentativas sem sucesso de carregar uma página *web*, desiste de tentar carregar a página e informa o erro ao usuário;
4. Redundância: Outra forma de tornar o sistema mais tolerante a falhas é acrescentar redundância ao sistema, para que caso uma parte falhe, outras partes detectem essa falha

e assumam as tarefas do módulo que falhou sem prejudicar o funcionamento do sistema como um todo. Exemplos são servidores replicados e múltiplas rotas para um destino (COULOURIS; DOLLIMORE; KINDBERG, 2005);

5. Recuperação de falhas: Mesmo que não seja possível evitar falhas, é importante que o sistema seja capaz de se recuperar delas. Um exemplo disso é o uso de *logs* nos sistemas de arquivos locais e de servidores que fazem sincronismo com seus pares após alguma falha que cause a falta de sincronismo do sistema (COULOURIS; DOLLIMORE; KINDBERG, 2005);

2.1.6 Transparência

Dentre diversos recursos para fornecer um sistema coerente ao usuário e esconder dele a complexidade do sistema, estão os conceitos de transparência. Como apresentado por Tanenbaum e Steen (2007), é possível listar os conceitos de transparência que o projeto atual utiliza de forma mais intensiva, como por exemplo:

- Transparência de acesso: toda a complexidade no acesso dos dados é escondida do usuário;
- Transparência de localização: oculta do usuário a localização dos recursos distribuídos;
- Transparência de migração: é responsável por omitir do usuário detalhes referente a migração dos dados para outras localidades físicas e lógicas;
- Transparência de replicação: ocultam do usuário os mecanismos de replicação dos dados;
- Transparência de falhas: esconde do usuário e tenta solucionar os problemas que ocorrem no sistema, de modo a evitar com que o usuário seja prejudicado toda vez que alguma falha aconteça no sistema, como visto anteriormente, detectando, mascarando, tolerando, usando módulos redundantes ou até mesmo recuperando o sistema quando uma falha ocorre.

Embora todas essas camadas de transparência sejam desejadas nem sempre é possível fornecer transparência total ao usuário por questões práticas, como usabilidade, complexidade na implementação e a degradação do desempenho que elas geram.

2.2 Sistema de Arquivos

Sistemas de arquivos (SA) foram criados para oferecer ao programador uma interface padronizada de acesso aos arquivos, independente de como e onde estão armazenados (COULOURIS; DOLLIMORE; KINDBERG, 2005) dentro dos dispositivos locais. Além dos dados (arquivos propriamente ditos), sistemas de arquivos devem armazenar também os metadados,

que são informações sobre os dados (arquivos). Esses metadados comumente são os seguintes (COULOURIS; DOLLIMORE; KINDBERG, 2005) (adaptado):

- Nome do arquivo;
- Dono do arquivo;
- Grupo do arquivo;
- Horário de criação;
- Horário de acesso;
- Horário de modificação;
- Horário de alteração de atributo;
- Contagem de referências;
- Tipo do arquivo (diretório, arquivo, *link*);
- Lista de permissões de acesso.

Além de gerenciar como os arquivos são armazenados e acessados, cabe ao sistema de arquivos a responsabilidade de fazer o controle de acesso dos arquivos que ele armazena, restringindo o acesso conforme as regras definidas pelos usuários ou sistema. Essas regras normalmente podem ser expressas pelos acessos de leitura, escrita e execução dos arquivos.

Em um sistema de arquivos convencional todo acesso aos arquivos é controlado pelo *kernel* do sistema operacional. O tratamento do acesso é feito pelo próprio núcleo do sistema. Em sistemas distribuídos esse tratamento é feito de forma diferente. Isso será abordado na sessão de Sistemas de Arquivos Distribuídos na sessão 2.3.

Exemplos de sistemas de arquivos são: os *extended file systems* (EXT2, EXT3, EXT4), XFS, ReiserFS, *Journaling FileSystem* (JFS) para sistemas *Unix-like* e não menos importantes os *File Allocation Table* (FAT16, FAT32, FAT32EX), e o *New Technology File System* (NTFS) para sistemas da Microsoft.

2.3 Sistemas de Arquivos Distribuídos

Sistemas de Arquivos Distribuídos (SAD) são sistemas que oferecem compartilhamento desses arquivos através de redes, fornecendo desempenho, integridade, confidencialidade e disponibilidade equivalentes ou superiores aos sistemas de arquivos tradicionais (COULOURIS; DOLLIMORE; KINDBERG, 2005), permitindo que programas armazenem e acessem os arquivos remotos como se fossem locais, fornecendo aos usuários liberdade para acessar seus

arquivos independente do computador que utilizem, desde que dentro de uma rede local ou que tenha acesso ao sistema de arquivos distribuído utilizado.

Um SAD básico é descrito por Coulouris, Dollimore e Kindberg (2005), como um sistema em que o objetivo é simplesmente simular o funcionamento de um sistema de arquivos tradicional para que programas clientes executem em computadores remotos. Esses não fazem o uso de sistema de réplicas, nem suportam sistemas multimídia que exigem grandes fluxos de dados e a capacidade de fazer a entrega desses dados com uma temporização bem limitada.

Assim, para projetar um sistema de arquivos distribuído de alto desempenho capaz de armazenar grandes quantidades de dados e que suporte escrita e leitura em larga escala, é necessário resolver diversos problemas, como balanceamento de carga, confiabilidade, disponibilidade, segurança, e concorrência de acesso aos dados (COULOURIS; DOLLIMORE; KINDBERG, 2005).

A seguir são elencados alguns sistemas de arquivos distribuídos que são referências clássicas e foram a base do desenvolvimento desse trabalho.

2.3.1 Network File System (NFS)

O *Network File System* (NFS) foi projetado pela *Sun Microsystems* para funcionar em suas estações de trabalho. Atualmente é compatível com quase todos os sistemas operacionais de grande uso. Foi lançado como o primeiro SAD destinado a ser um produto final e ter suas interfaces e protocolo abertos ao público. Sua grande compatibilidade somada a premissa de fornecer acesso transparente aos arquivos remotos através do uso das chamadas ao *Virtual File System* (VFS) e do uso de *Remote Procedure Call* (RPC) foram os grandes fatores que contribuíram para seu sucesso e sua popularização (COULOURIS; DOLLIMORE; KINDBERG, 2005). Sua arquitetura é apresentada na Figura 1.

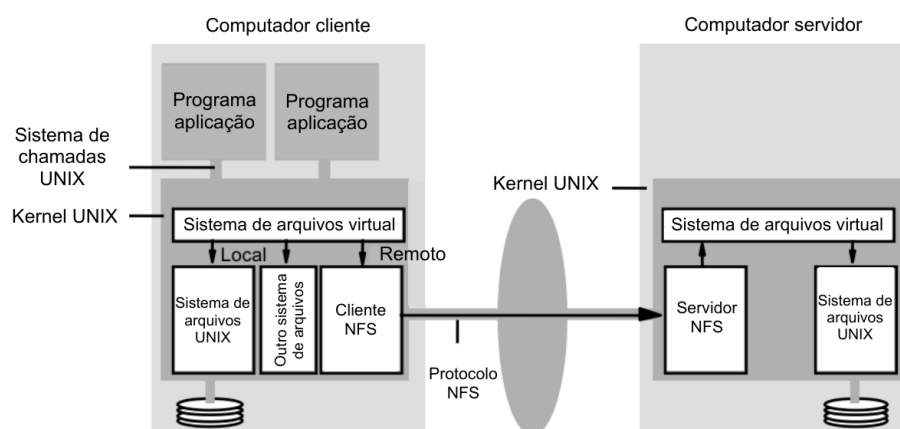


Figura 1 – Arquitetura do NFS (COULOURIS; DOLLIMORE; KINDBERG, 2005)

Seu projeto independente do estado do sistema (*stateless*) basicamente inviabiliza a utilização de mecanismos para recuperação de falhas (COULOURIS; DOLLIMORE; KINDBERG,

2005).

O uso de memória *cache* no servidor tenta antecipar o que o cliente irá solicitar e já manter esses dados na memória para que, quando o cliente solicitá-los, possam ser entregues sem o gargalo de acesso ao disco. O uso de *cache* na escrita se baseia em manter o arquivo na memória por um tempo sem que o cliente o altere e só então gravar o arquivo no disco. Ainda sim existe a operação *sync* que grava as informações pendentes no disco a cada 30 segundos. Quanto a *cache* dos arquivos no cliente, é feita através da datas de modificação dos arquivos. Caso haja discrepâncias entre cliente e servidor, o cliente é atualizado (COULOURIS; DOLLIMORE; KINDBERG, 2005).

Para questões de segurança o NFS utiliza dois mecanismos externos. Para a transmissão dos dados, autenticação e comunicação do cliente com o servidor é utilizado o protocolo Kerberos v5 junto com RPCSEC_GSS. Para proteger os dados nos clientes é utilizado a lista de controle de acesso (ACL) que é responsável por definir as permissões para acesso aos arquivos. (TANENBAUM; STEEN, 2007).

Quanto a desempenho, o NFS fornece uma solução satisfatória até mesmo para sistemas com grande demanda, ao utilizar diversos servidores. Mas essa solução é limitada pelo fato do sistema não fornecer suporte a replicação de arquivos para leitura e escrita (apenas a replicação para leitura é suportada). Ainda sim, mesmo que bastante difundido e utilizado o NFS carece de mecanismos de migração de dados e réplicas que estão presentes em outros SAD mais avançados (COULOURIS; DOLLIMORE; KINDBERG, 2005).

2.3.2 Andrew File System (AFS)

Da mesma forma que o NFS visto na sessão anterior, o *Andrew File System* (AFS) além de ser compatível com o NFS também fornece acesso de forma transparente aos arquivos, embora utilize uma abordagem um pouco diferente para atingir esse objetivo. Tendo como objetivo principal a escalabilidade do sistema seus mecanismos de cache no cliente são bem agressivos e nas versões mais atuais o sistema passou a trabalhar armazenando o estado dos clientes com o uso de *callbacks* para diminuir a comunicação entre cliente e servidor e manter baixo o uso dos processadores nos servidores como apresentado por Coulouris, Dollimore e Kindberg (2005).

O AFS tem seu projeto baseado na suposição sobre o tamanho médio e máximo dos arquivos em discos nos sistemas *UNIX* de ambientes acadêmicos e outros. De acordo com Coulouris, Dollimore e Kindberg (2005) as observações mais importantes sobre a implementação do AFS são:

- Os arquivos costumam ser pequenos (normalmente com menos de 10KB).
- Operações de leitura são muito mais frequentes que as de escrita (6 vezes mais recorrentes).
- O acesso aos arquivos é feito de forma sequencial na maioria das vezes.

- A maioria dos arquivos é lida e escrita por apenas um usuário, e mesmo quando compartilhado a maioria das vezes apenas um único usuário é responsável por escrever no arquivo.
- Os arquivos tendem a ser referenciados repetidas vezes em momentos próximos.

Com base nessas premissas o AFS opera da seguinte forma:

- Servir arquivos inteiros.
- Cache de arquivos inteiros nos clientes.
- Uso de cache persistente nos clientes (armazenados em disco).
- Uso de um kernel modificado nos clientes para interceptar as chamadas de acesso aos arquivos.

Com base nos estudos feitos e nas características citadas acima é fácil perceber que existe um grupo de arquivos que é totalmente fora desse escopo: arquivos de bancos de dados. Dessa forma, o projeto do AFS declara explicitamente que o objetivo do sistema não é atender a essa categoria de arquivos.

A arquitetura do AFS utiliza servidores e clientes. O *software* utilizado nos servidores é chamado de Vice, enquanto o que executa nas estações clientes é o Venus, como pode ser visto na Figura 2 .

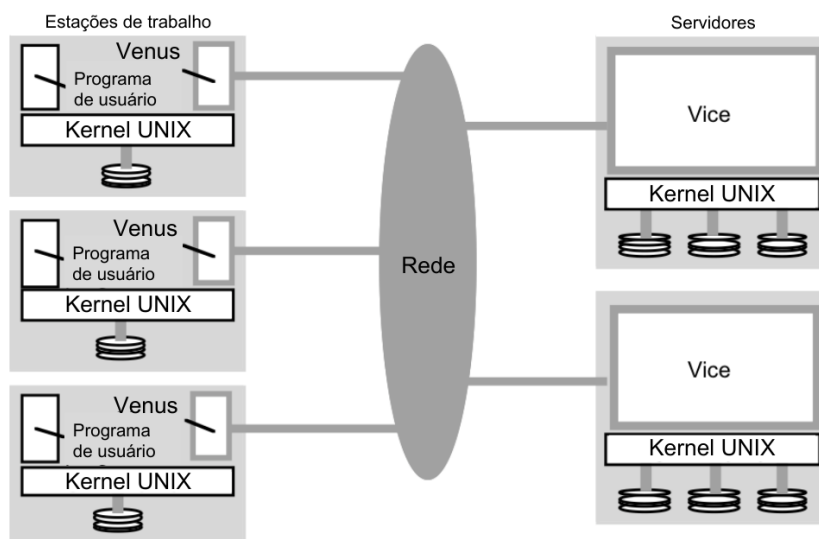


Figura 2 – Arquitetura do AFS (COULOURIS; DOLLIMORE; KINDBERG, 2005)

O gerenciamento de cache no cliente é tratado pelo processo Venus, que ao receber uma operação open verifica se o arquivo solicitado já existe na cache e se existe alguma versão mais

nova desse arquivo nos servidores. Caso o arquivo da cache esteja defasado a nova versão é solicitada aos servidores e é fornecida ao programa que solicitou o arquivo. Quando o arquivo é fechado através da operação *close* o processo Venus (cliente) verifica se o arquivo foi alterado e, caso afirmativo, envia a nova versão do arquivo ao servidor. É importante ainda ressaltar que o AFS não dispõe de nenhum mecanismo para tratar atualização concorrente de um mesmo arquivo, de forma que se vários clientes enviarem versões diferentes do mesmo arquivo ao servidor, todas exceto a última a ser processada pelo servidor (Vice) serão perdidas sem nenhum aviso ou erro, restando apenas a última requisição processada pelo servidor, que será a versão do arquivo que será salva e estará disponível (COULOURIS; DOLLIMORE; KINDBERG, 2005).

Quanto aos metadados vale citar que os banco de dados ficam replicados integralmente em todos os servidores (Vice) (COULOURIS; DOLLIMORE; KINDBERG, 2005), de forma que cada servidor sabe exatamente onde se encontram os arquivos.

Semelhante ao NFS, o AFS só suporta réplica de arquivos para leitura, direcionando todas as escritas nesse arquivo para apenas um servidor, que então faz o sincronismo posteriormente e tem que ser feita de forma explícita (COULOURIS; DOLLIMORE; KINDBERG, 2005).

As características citadas acima fornecem ao AFS um bom desempenho para cargas relativamente grandes sem sobrecarregar demasiadamente os servidores, o que fornece grande taxa de escalabilidade, sendo em alguns casos até 60% mais eficiente no uso de CPU que o NFS com a mesma carga de trabalho (COULOURIS; DOLLIMORE; KINDBERG, 2005).

2.3.3 *Flexible and Adaptable distributed file system* (FlexA)

O *Flexible and Adaptable distributed file system* (FlexA) (FERNANDES, 2012), desenvolvido pelo Grupo de Sistemas Paralelos e Distribuídos (GSPD), que é um sistema de arquivo distribuído não básico, objeto de estudo desse trabalho, tem foco na utilização de recursos computacionais das estações clientes para diminuir a carga de processamento dos servidores, além de fornecer um sistema de segurança descentralizado e o uso de mecanismo de tolerância a falhas, aproveitando características de diversos SADs, focando em fornecer um SAD com grande escalabilidade (FERNANDES, 2012).

O FlexA mudou muito desde sua versão inicial, desenvolvida em 2012 (NETO, 2013). A versão que será tratada nesse texto é a última versão que ainda está em desenvolvimento, baseada na versão original. Essa versão é o resultado da cooperação de diversos integrantes do GSPD (NETO, 2013), durante a evolução do sistema nesses dois anos desde sua versão inicial. A arquitetura do sistema original pode ser vista na Figura 3 .

A arquitetura atual do projeto, se baseia muito no que foi proposto por Fernandes (2012) é apresentada na Figura 4.

Além disso a distribuição dos arquivos também foi alterada, para que o sistema apresente um desempenho melhor do lado cliente, a replicação das porções dos arquivos na fração de 2/3

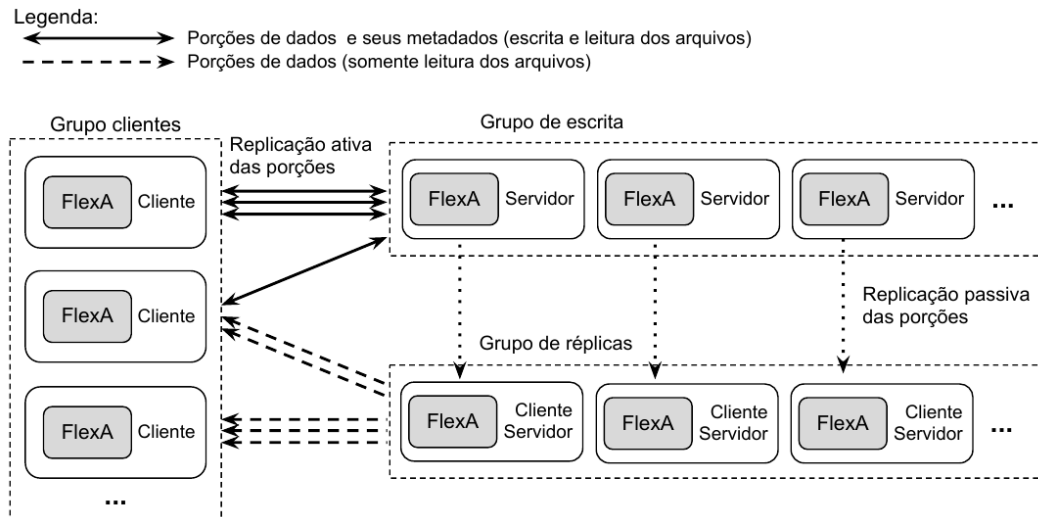


Figura 3 – Arquitetura original do FlexA (FERNANDES, 2012).

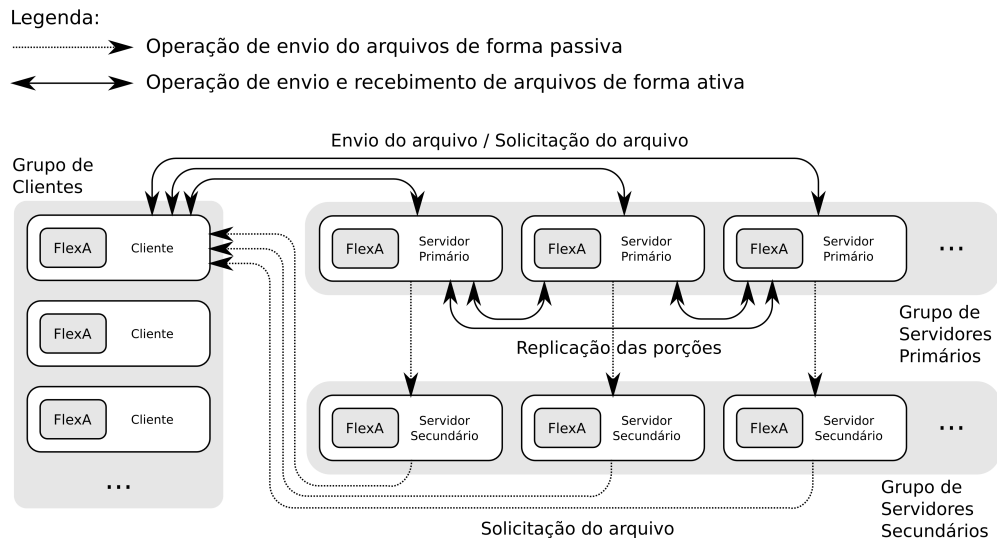


Figura 4 – Arquitetura atual do FlexA (NETO, 2013).

em cada servidor primário fica agora na responsabilidade dos servidores primários. o diagrama apresentado na figura 5 mostra como era a movimentação dos dados no FlexA original, e a figura 6 mostra a nova versão (NETO, 2013).

Para atingir os objetivos descritos acima o FlexA possui diversos mecanismos que serão elencados nas sessões a seguir.

2.3.4 Segurança

A segurança do sistema é baseada no controle de acesso aos arquivos e também no controle de escrita sobre o mesmo (FERNANDES, 2012).

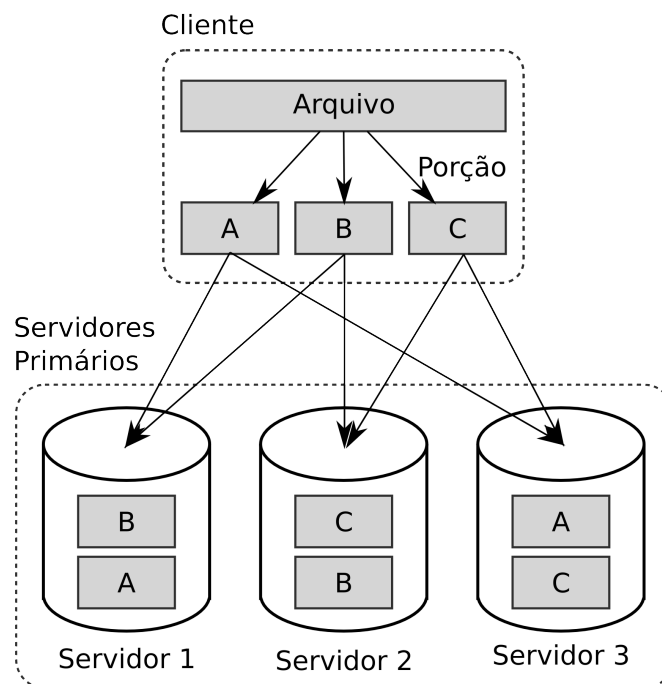


Figura 5 – Fluxo de dados no FlexA original (FERNANDES, 2012).

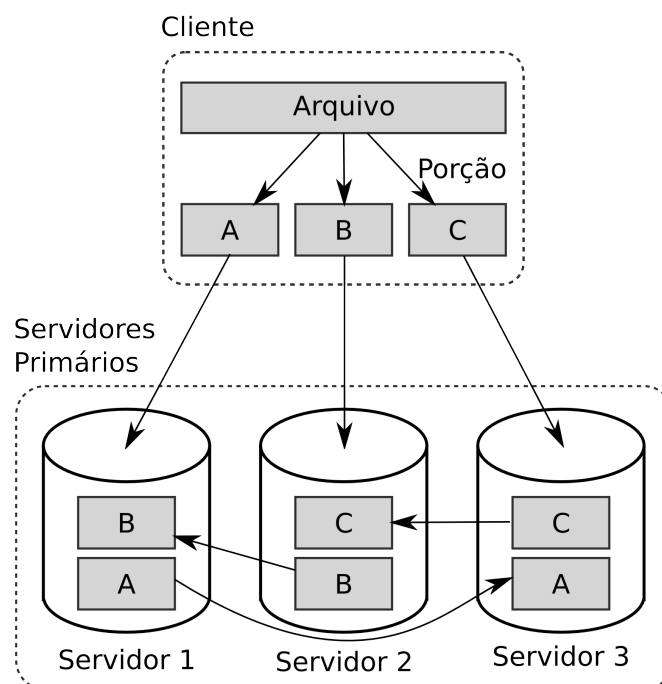


Figura 6 – Fluxo de dados no FlexA atual, elaborado a partir de (NETO, 2013).

- **Controle de Acesso:** é implementado utilizando um trio de chaves para cada arquivo. As chaves que compõe esse trio são:
 - *Verify Key (VK)*: fornece acesso ao arquivo, servindo como um identificador único do

arquivo no sistema.

- *Read Key* (RK): é a chave utilizada para cifrar o arquivo. Sem essa chave, mesmo que seja conhecida a VK não é possível ler o conteúdo do arquivo.
- *Write Key* (WK): fornece acesso de escrita no arquivo dentro dos servidores.

Das três chaves citadas acima a única que não é enviada aos servidores é a chave usada na criptografia do arquivo, a *Read Key*. Dessa forma mesmo que o servidor seja comprometido não é possível obter o conteúdo original dos arquivos que ele armazena (NETO, 2013).

Essas chaves são criadas com base em uma chave privada RSA (SHAMIR, 1979) especificada pelo usuário, junto com um valor único para cada arquivo chamado de *salt* que é fornecido pelo servidor. Uma vez com a RSA e o *salt* o processo feito para obtenção do trio de chaves é o apresentado na Figura 7.

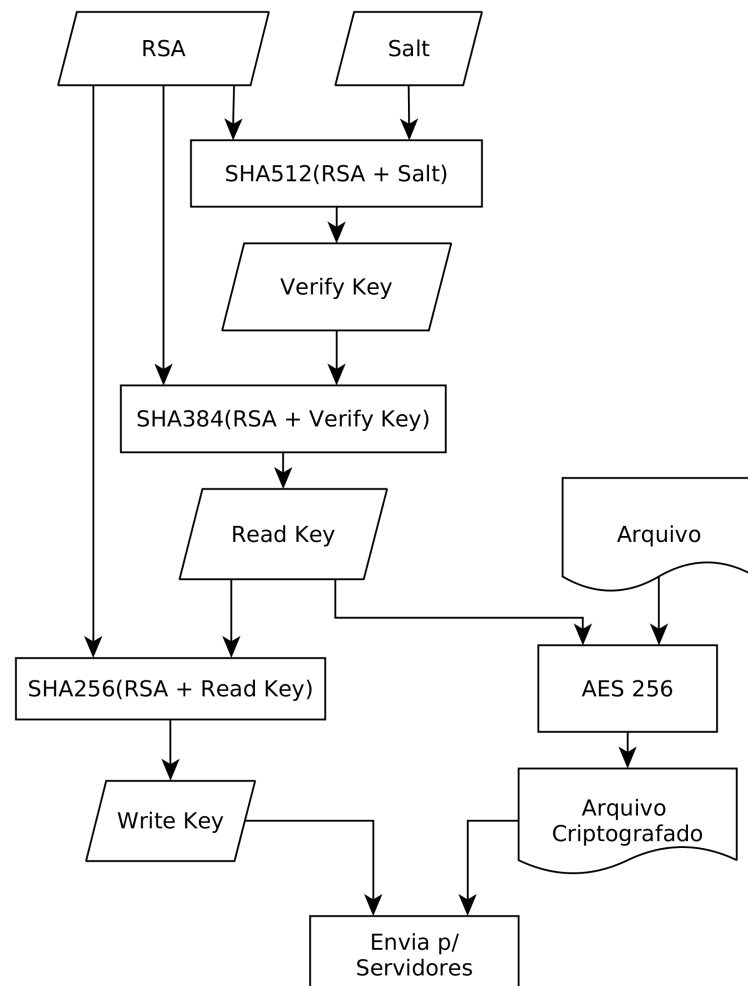


Figura 7 – Diagrama sobre a criação das chaves de acesso e criptografia do FlexA (NETO, 2013).

Como mostrado na Figura 7 a chave de identificação do arquivo (VK) é gerada utilizando o *hash* SHA512 da RSA do usuário concatenada com o do *salt* retornado pelo servidor.

Após isso a chave de criptografia (RK) é gerada utilizando a o hash SHA386 da RSA do usuário concatenada com o do identificador do arquivo (*Verify Key*). Por fim, é gerada chave de escrita (WK), utilizando a RSA do usuário concatenada com a chave de leitura.

Todo o processo criptográfico e de geração de chaves é feito de forma transparente, sem que o usuário tenha que fazer isso de forma manual, notando apenas o tempo que é necessário para fazer a criptografia / descriptografia do arquivo.

- **Integridade:** O mecanismo de integridade do arquivo é implementado utilizando a chave WK mostrada anteriormente. A permissão de escrita (chave WK) é solicitada sempre que um cliente deseja fazer a operação de escrita (atualização) de um arquivo nos servidores. Dessa forma, o cliente deve fornecer a WK correspondente ao arquivo que deseja atualizar, então, o servidor compra a WK fornecida com a armazenada e caso sejam idênticas o servidor faz a substituição do arquivo antigo pelo novo.

Mecanismos para evitar ataques de repetição e outros semelhantes ainda precisam ser implementadas no FlexA, pois a transmissão das chaves de acesso e escrita podem ser facilmente capturadas e reutilizadas com a implementação atual.

2.3.5 Tolerância a falhas e Adaptabilidade

Os mecanismos utilizados pelo FlexA para executar esse objetivo são o uso de réplicas das porções dos arquivos, que são enviados para diversos servidores e lá são replicados com o grupo de servidores secundários. Também é utilizada a possibilidade de um servidor secundário assumir além da sua função a de um servidor primário caso seja necessário por falha de algum primário ou sobrecarga do grupo de servidores primários (NETO, 2013).

Sempre que um servidor primário ou secundário falha, é executado um processo que faz o sincronismo do servidor com os outros servidores ativos para que esse assuma o estado atual do sistema e volte o mais breve possível a colaborar com o atendimento dos clientes (FERNANDES, 2012).

2.3.6 Escalabilidade

Os principais fatores que auxiliam na escalabilidade do sistema são:

- **Criptografia e descriptografia dos arquivos nos clientes:** Evitam o consumo de UCP nos servidores, deixando-os mais disponíveis para atender a novas requisições.
- **Divisão dos arquivos em diversas porções nos clientes:** Diminuem o uso de armazenamento e uso dos discos dos servidores.

- União das diversas porções que compõe um arquivo nos clientes: Também influenciam no uso de disco dos servidores, liberando os discos para atenderem outras requisições mais rapidamente.
- Leitura e escrita das porções dos arquivos de diferentes servidores, evitam sobrecarregar o uso de banda de apenas um servidor e aumentam o uso da banda no cliente.
- Utilização agressiva de cache nos clientes: Essa técnica evita acessos recorrentes aos servidores, principalmente para arquivos que são pouco atualizado ou são utilizados por apenas um usuário.
- Uso de sistemas de replicação para servidores secundários: Auxiliam nas operações de leitura dos arquivos pelos clientes.

2.3.7 Flexibilidade

Como boa parte da carga de processamento e uso do disco no FlexA é transferida aos clientes, o sistema pode operar com *hardware* de baixo custo sem grandes problemas. Isso ainda fornece ao FlexA a possibilidade de que caso seja necessário clientes com mais recursos disponíveis podem passar a fazer parte do grupo de servidores, auxiliando no atendimento a requisições de outros clientes. Essa característica faz com que o sistema aproveite muito recurso que estaria ocioso em sua rede, principalmente em momentos de uso intenso dos servidores (FERNANDES, 2012).

Além disso, o sistema é projetado para que seja possível fazer a troca dos mecanismos de segurança, regras de gerenciamento da cache nos clientes, métricas de divisão das porções e diversos outros elementos do sistema de forma simples, principalmente devido a escolha do Python como linguagem de programação, facilita o acesso ao código fonte do FlexA (FERNANDES, 2012).

2.3.8 Abertura

A abertura do FlexA é dada basicamente pelo fato de o sistema ter o código fonte aberto e disponível (FERNANDES, 2012), sendo necessário aprimorar a documentação existente.

2.4 Chamada de Procedimentos Remotos (RPC)

Para realizar comunicação entre cliente e servidores é possível utilizar diversas técnicas diferentes. Nesse trabalho a Chamada de Procedimento Remoto ou *Remote Procedure Call* (RPC) é uma ferramenta muito utilizada pois é a partir dela que a maior parte das comunicações entre cliente e servidor acontecem. A escolha desse paradigma foi feita pela simplicidade do projeto e

desenvolvimento, visando reduzir de forma drástica a complexidade no desenvolvimento dos módulos de comunicação do projeto, se comparado ao uso de *sockets*.

O RPC utiliza um paradigma de comunicação de alto nível, ocultando do desenvolvedor quase todo o processo de estabelecimento de conexão, transmissão dos dados, conversão dos dados e bloqueio do cliente ao realizar requisições (BIRRELL; NELSON, 1984).

Para a utilização do RPC na implementação de um serviço é necessário um processo chamado de servidor RPC (Servidor) que possui as funções disponíveis que serão solicitadas por um cliente através de uma requisição RPC. Assim cabe ao Servidor registrar todas as funções que o cliente pode requisitar. Após registrar as operações o Servidor inicia a escuta por requisições.

Em outro *host*, um processo chamado Cliente faz uma Requisição RPC ao Servidor. O Servidor executa a função solicitada, com os parâmetros enviados e retorna o resultado ao cliente.

2.4.1 XML-RPC

No mercado existem diversas soluções para o uso de RPC. O XML-RPC é uma implementação de RPC que utiliza TCP/IP junto com HTTP e XML para a troca de mensagens entre servidor e cliente (CISCO, 2010). O uso de um padrão como o XML-RPC é também justificado pela representação dos dados que é mantida através de diversas arquiteturas (HUCKA et al., 2001)

A versão atual do FlexA utiliza o XML-RPC para todas as comunicações entre cliente/servidor exceto para a transmissão de arquivos (NETO, 2013). A transmissão dos arquivos é feita via *socket* para evitar o processo de *marshalling* dos arquivos que tende a ser demasiadamente custoso em questões de uso de UCP e demorado, dessa forma reduzindo o tempo de entrega dos arquivos ao cliente.

Um diagrama bem simples sobre o funcionamento desse paradigma descrito acima é exibido na Figura 8.

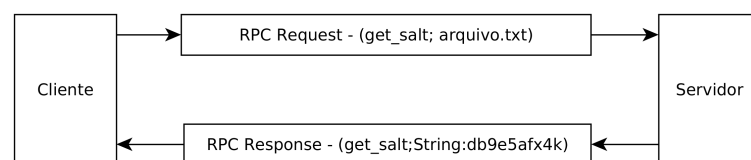


Figura 8 – Diagrama que exhibe a comunicação via XML-RPC. (CISCO, 2010) (Adaptado)

Como pode ser visto na figura 8, o cliente faz a requisição de uma função remota, passando os parâmetros. Toda a comunicação é feita com o uso de XML. O servidor recebe a requisição, processa e retorna o resultado também em XML. Um outro exemplo mostra como é feita a comunicação de uma requisição com XML-RPC como pode ser visto na Figura 9. O processo de resposta da requisição segue o mesmo princípio.


```
1 <methodCall>
2   <methodName>get_salt</methodName>
3   <params>
4     <param>
5       <value>
6         <string>arquivo.txt</string>
7       </value>
8     </param>
9   </params>
10 </methodCall>
```

Figura 9 – Exemplo de mensagem XML utilizada na chamada de procedimentos remotos com XML-RPC.

2.5 Dispositivos Móveis

Quanto a classificação de dispositivos móveis, é muito difícil fornecer uma definição formal que seja adequada. Assim, essa definição pode ser feita em duas partes, como feita por Zheng e Ni (2010).

2.5.1 Dispositivos Móveis

De acordo com (ZHENG; NI, 2010), dispositivos móveis são dispositivos portáteis como *laptops*, *Personal Digital Assistants* (PDA), *tablets*, *smart phones*, *handhelds*, *MP3 Players*, consoles de jogos portáteis entre outros. Variando de dispositivos com pouquíssima autonomia, baixa ou nula conectividade e pouca capacidade computacional até dispositivos de última geração com autonomia relativamente alta (algumas dezenas de horas de uso constante), potência computacional muitas vezes superior a computadores de mesa e *notebooks* e grande conectividade utilizando diversas tecnologias diferentes.

2.5.2 Computação Móvel

Dentro do escopo desse trabalho mais importante que a definição de dispositivos móveis é a definição computação móvel, que é apresentada por (ZHENG; NI, 2010) como sendo um conjunto de dispositivos móveis que fornecem ao usuário um sistema computacional que pode operar em dois modos: conectado e desconectado. Quando desconectado de redes de dados esses dispositivos devem funcionar de forma dessincronizada com sua fonte de dados, e quando conectados novamente a redes que fornecem acesso a transmissão de dados devem fazer o sincronismo dos dados através de operações de *upload/download*.

2.6 Android

Na maioria das vezes é necessário um sistema operacional (SO) que gerencie um dispositivo móvel, esse sistema operacional pode ser o Android (Open Handset Alliance, 2014)

ou outro sistema operacional comum ou preferencialmente próprio para dispositivos desse tipo. Existem atualmente no mercado dezenas de SOs diferentes para dispositivos móveis, mas os que mais se destacam frente aos usuários e grandes fabricantes são Android (*Open Handheld Alliance*) e iOS (Apple) (International Data Corporation, 2014).

Mais do que um simples sistema operacional, o Android é composto por diversas camadas de software (*Android Software Stack*) que fornecem ao sistema suporte a uma grande variedade de dispositivos e flexibilidade no seu uso (Open Handset Alliance, 2014). Essas camadas são exibidas na Figura 10.

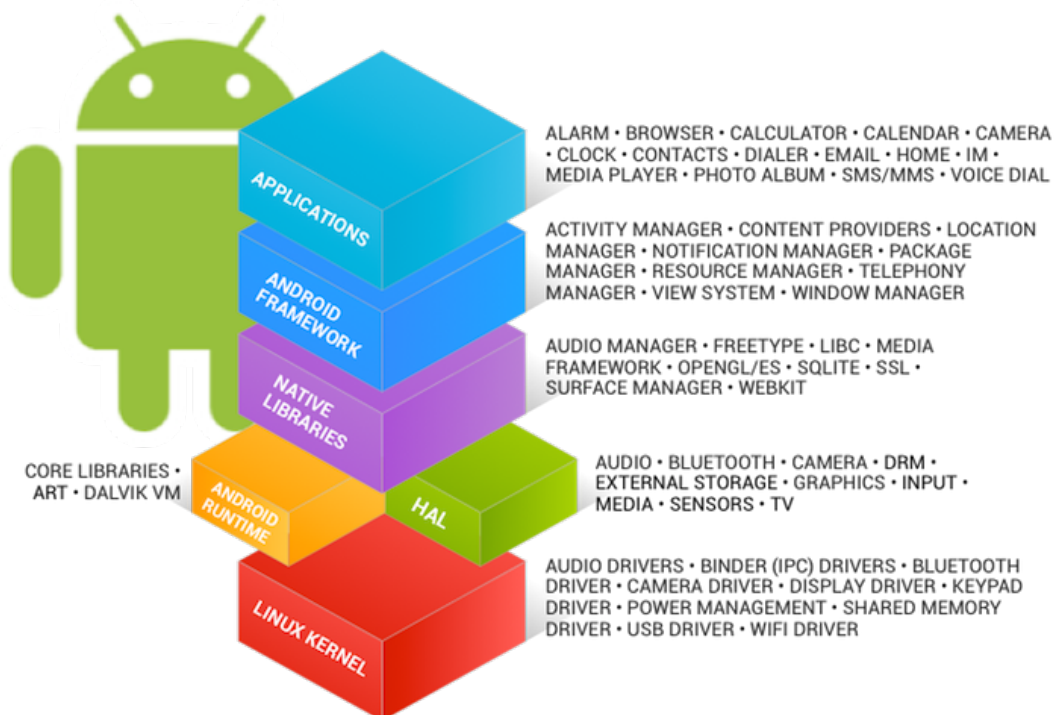


Figura 10 – Camadas de software que compõe o sistema Android (Open Handset Alliance, 2014).

2.7 Considerações finais

Com os principais recursos, tecnologias e os principais pontos explicados por meio de exemplos e menções aos pontos mais significativos de cada tópico, a fundamentação teórica necessária para o trabalho está completa. No capítulo seguinte será abordado o desenvolvimento do trabalho proposto que faz uso dos termos e tecnologias apresentados nesse capítulo.

3 Descrição e desenvolvimento do projeto

Neste capítulo são apresentados o desenvolvimento da proposta do projeto e também sua implementação no FlexA. Inicialmente serão abordados as restrições encontradas no projeto atual e a motivação para corrigi-las. Então será apresentada a documentação desenvolvida para que o projeto possa ser viável a longo prazo, então serão detalhadas as adaptações realizadas no módulo servidor para que esse passe a ser compatível com a nova especificação. Por fim, com o novo servidor pronto, será apresentada a implementação do cliente para Android.

3.1 Restrições da versão atual do FlexA

A versão atual do FlexA, ainda em um estágio inicial de desenvolvimento, carece de um modelo de programação bem definido, sua documentação precisa ser aprofundada e não menos importante de um protocolo de comunicação precisa ser bem explicitado.

Os principais pontos abordados nesse projeto são:

- Documentação inicial do projeto, como um diagrama de classes, casos de uso e também um diagrama com os módulos do sistema e como eles se integram.
- Formalização do protocolo de comunicação.
- Adaptação incremental do módulo servidor para que ele passe a ser compatível com a documentação gerada.
- Implementação do módulo cliente para Android.

A motivação para essas melhorias é que elas fornecerão uma base sólida para que no futuro possam ser incorporados no projeto atual os resultados de trabalhos já realizados em versões anteriores do FlexA, que são listados pelos títulos:

- Detecção de Falhas de Comunicação e Balanceamento de Carga no FlexA (SEGURA, 2013);
- Metodologia para Recuperação de Falhas e Garantia de Disponibilidade no FlexA (OKADA, 2013);
- Implementação e avaliação de desempenho de algoritmo de criptografia em GPU para o FlexA (BARBOSA, 2013);
- Sincronização, consistência e falhas no FlexA (OLIVEIRA, 2013);

- Disponibilidade em um sistema de arquivos distribuído flexível e adaptável (CARVALHO, 2014);

Além de tornar mais fácil a incorporação dos resultados obtidos com os trabalhos citados, a nova documentação deve tornar o sistema fácil de se manter e evoluir e permitir que diversas equipes colaborem com o trabalho de forma simultânea.

3.2 Documentação do projeto

Um estudo a fundo do sistema foi feito, utilizando a documentação que existia e a análise do código-fonte, para entender o funcionamento e comportamento do sistema e também como era definida a estrutura atual do projeto.

3.2.1 Módulos do Sistema

Para representar o sistema de forma mais abstrata foi criado um diagrama dos módulos do sistema atual, e suas dependências conforme pode ser visto na Figura 11.

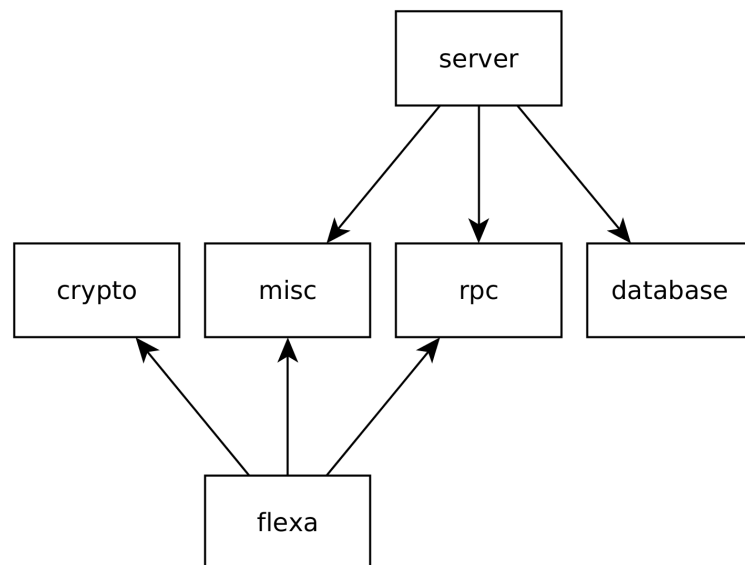


Figura 11 – Módulos do sistema atual e suas dependências. Criado com base no código fonte do projeto de Neto (2013).

Esse diagrama foi gerado a partir do estudo do código-fonte da versão atual do projeto. Apenas por esse diagrama já é possível notar que o módulo Servidor não utiliza os serviços de criptografia, e que apenas o servidor tem acesso aos bancos de dados de metadados, o que ajuda a evidenciar a segurança dos dados do cliente que já chegam aos servidores criptografados.

3.2.2 Diagrama de classes do sistema existente

Para que fosse possível analisar a estrutura mais interna dos módulos foi criado o diagrama de classes da *Unified Language Model* (UML) (IBM, 2004). Esse diagrama é apresentado na figura 12. Embora seja um diagrama de classes, alguns abusos de notação tiveram que ser utilizados devido a lacuna existente entre a representação dos diagramas de classes e a linguagem Python em que o FlexA é desenvolvido. Esses abusos são a representação do código que não pertence a nenhuma classe, mas estão dentro de módulos, e também o uso de pacotes para representar arquivos.

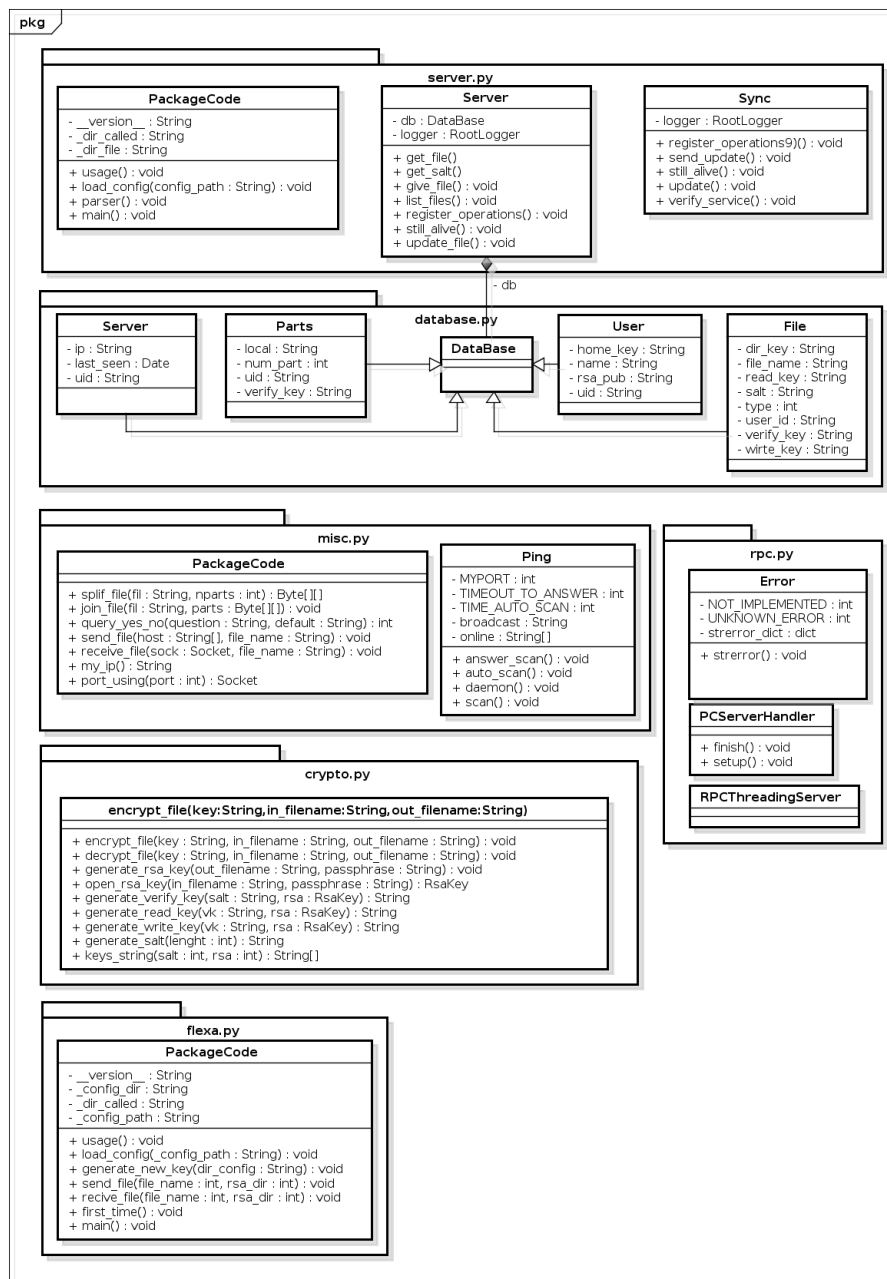


Figura 12 – Classes e pacotes (arquivos) que estruturam a versão atual do FlexA

Para que o FlexA seja mais aberto e flexível é necessário realizar adaptações no código

afim de fazer uma melhor separação das funções e dos módulos.

3.2.3 Diagrama de Casos de Uso

Para que fosse possível elaborar a adaptação do projeto, num primeiro momento foi feito o levantamento dos requisitos do FlexA, junto com os objetivos de melhoria do código existente. Dessa forma foi elaborado um diagrama UML de casos de uso de acordo com (Visual Paradigm, 2011). O diagrama com os casos de uso é apresentado na Figura 13.

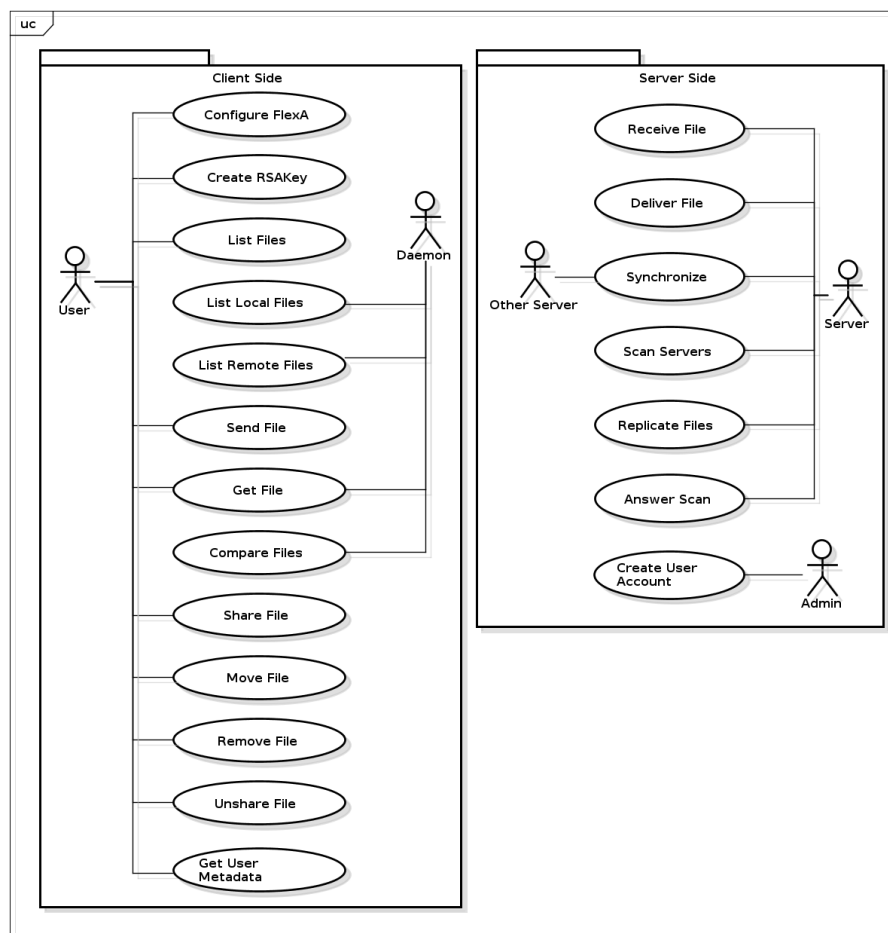


Figura 13 – Diagrama de Casos de uso com os requisitos funcionais do FlexA.

Dos casos de uso apresentados na figura 13, serão implementados e utilizados nesse trabalho apenas os referentes ao módulo cliente devido ao escopo do projeto. Os restantes foram definidos juntos para preparar os trabalhos futuros.

3.2.4 Interface de Comunicação Cliente-Servidor

Já com os objetivos do trabalho definidos, e uma documentação do que existia que permitisse entender o projeto, foi então formulada a interface que servirá de alicerce desse trabalho que será responsável por padronizar e fornecer as comunicações entre o cliente e o

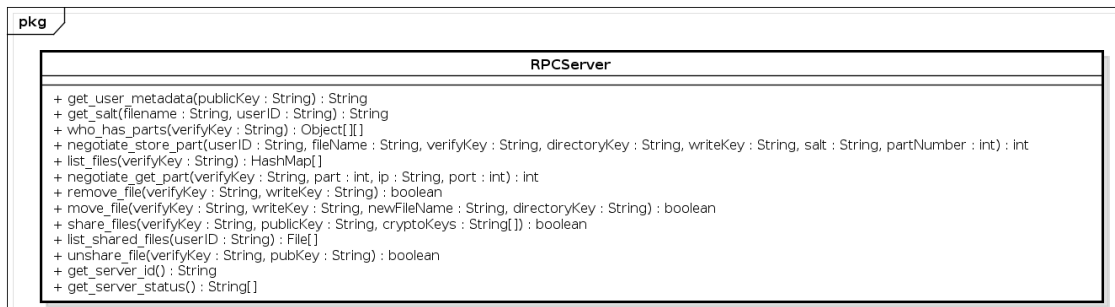


Figura 14 – Interface de comunicação dos servidores do FlexA.

servidor. Essa interface, que é apresentada na figura 14, foi desenvolvida em reunião com os outros desenvolvedores do FlexA para que fossem atendidas todas as necessidades do trabalho atual e também do FlexA como um todo no futuro.

Essa interface é de grande importância para o projeto, pois é ela que servirá como referência para o desenvolvimento do módulo cliente e também do módulo servidor, uma vez que define os métodos que deverão ser invocados durante a comunicação entre os clientes e os servidores, utilizando XML-RPC e independentemente da linguagem utilizada.

3.2.5 Protocolos de Comunicação Cliente-Servidor

Com as funcionalidades que são esperadas do sistema e a interface de comunicação definida, foi feita a análise das comunicações entre cliente e servidor. Formalizou-se então os protocolos de comunicação, utilizando a notação para protocolos de comunicação apresentada em (ANDERSON, 2008) que é mostrada na tabela 1.

Tabela 1: Notação utilizada para definição de protocolos de comunicação (ANDERSON, 2008).

C	módulo cliente
S	módulo servidor
$privKey_X$	chave privada X
$pubKey_X$	chave pública X correspondente a $privKey_X$
$C \rightarrow S : dado$	C envia $dado$ para S
$C \rightarrow S : dado_{pubKey_X}$	C envia $dado$ criptografado com a chave pública X para S

Formalizadas as definições que serão utilizadas a seguir, são apresentados os protocolos para as comunicações. Esses protocolos foram elaborados com base no código fonte (NETO, 2013) e na interface de comunicação apresentada, já inserindo as adaptações para fornecer maior modularização, compatibilidade e segurança ao sistema.

3.2.6 Protocolos de manipulação de arquivos

A primeira formalização é o protocolo para solicitação dos metadados do usuário, que é feita quando o FlexA é iniciado pela primeira vez ou caso seja necessário obter o *userID* do

usuário. Isso é feito com base em sua chave pública previamente cadastrada no sistema por um administrador. O protocolo é apresentado na figura 15. Esse protocolo é implementado pela função *get_user_metadata*, que é apresentada na figura 14.

$$\begin{aligned} C &\rightarrow S : pubKey_C \\ S &\rightarrow C : \{userID, homeKey\}_{pubKey_C} \end{aligned}$$

Figura 15 – Protocolo de requisição dos metadados de um usuário com base na sua chave pública.

Com os metadados do usuário (*userID* e *homeKey* que é o identificador de sua pasta raiz) o usuário pode começar a utilizar o sistema enviando e recebendo arquivos.

Para que o usuário possa enviar um arquivo para o sistema é necessário verificar se existe um arquivo com o mesmo nome no mesmo endereço especificado através do *salt*. Caso o arquivo não exista ainda, é feito o cadastro e solicitado o *salt* referente ao arquivo. Na figura 16 é formalizado o protocolo de requisição do *salt* de um arquivo. É importante ressaltar que *fileName* é composto pelo nome completo do arquivo com o diretório em que o arquivo se encontra. Esse endereço é relativo ao diretório mapeado do FlexA para o usuário. Esse protocolo é implementado pela função *get_salt*, que é apresentada na figura 14.

$$\begin{aligned} C &\rightarrow S : fileName, userID \\ S &\rightarrow C : salt \end{aligned}$$

Figura 16 – Requisição do *salt* de um arquivo.

Caso o *salt* retornado pelo servidor seja igual 0 é assumido que esse arquivo ainda não existe no servidor, e então o próprio cliente gera um *salt* para esse arquivo.

Já com o *salt*, o módulo cliente gera o trio de chaves para o arquivo (VK, RK e WK) de acordo com a figura 7, criptografa o arquivo utilizando a RK, faz a divisão do arquivo em *N* porções de acordo com o tamanho do arquivo, (*N* e o tamanho do arquivo são configurações definidas pelo usuário) caso necessário e envia cada porção para um servidor. Primeiro o módulo cliente faz o cadastro do arquivo no servidor e então envia as porções. Na figura 17 é descrito o protocolo utilizado no cadastro do arquivo, onde *N* é o número de porções em que o arquivo foi dividido, *directoryKey* é o identificador do diretório que o arquivo está e *fileType* é o tipo do arquivo (diretório ou arquivo comum). Esse protocolo é implementado pela função *negociate_store_part*, que é apresentada na figura 14.

Após o cadastro do arquivo e a negociação da porta de transmissão do arquivo, é feita a transmissão do arquivo em uma nova conexão com o servidor na porta *port*.

Com a negociação pronta basta enviar para o servidor a porção correspondente. Essa comunicação é definida conforme o protocolo mostrado na figura 18. Esse protocolo é implementado através de *sockets*, sem o uso do XML-RPC, por questões de desempenho.

Para i de 1 até N :
 $C \rightarrow S_i : userID, fileName, verifyKey, directoryKey, writeKey, salt, partNumber_i$
 $S_i \rightarrow C : port$

Figura 17 – Cadastro do arquivo nos servidores e negociação da porta de envio do arquivo

Para i de 1 até N :
 $C \rightarrow S_i : filePart_i$

Figura 18 – Transmissão das porções dos arquivos aos servidores, feito via socket.

Com os protocolos já definidos é possível enviar um arquivo para o servidor. Mas ainda não é possível recuperá-lo. A seguir serão tratados os protocolos envolvidos na recuperação dos arquivos enviados aos servidores.

Para recuperar um arquivo, é de grande importância a capacidade de listar os arquivos de um diretório. Para essa ação é utilizado o protocolo da figura 19. Esse protocolo é implementado pela função *list_files*, que é apresentada na figura 14.

$C \rightarrow S : directoryKey$
 $S \rightarrow C : metaFile_0, metaFile_1, metaFile_2, \dots, metaFile_n$

Figura 19 – Requisição da lista dos arquivos em um diretório

Ao requisitar ao servidor os arquivos do diretório referenciado por *directoryKey*, o servidor manda pacotes de informação *metaFile* referente aos arquivos. Cada pacote desse é composto por:

- nome do arquivo
- tamanho do arquivo
- dono do arquivo
- data de criação
- data de modificação

Essas características do arquivo são enviadas junto com o nome do arquivo para evitar acessos desnecessários ao servidor para recuperar cada uma dessas informações posteriormente. Isso influencia no tempo de resposta do sistema.

Com a lista dos arquivos, o usuário pode requisitar um arquivo específico ao servidor. Ao requisitar o arquivo, o módulo cliente faz a solicitação do *salt* referente ao arquivo pelo atributo

nome, com essa informação gera o trio de chaves do arquivo. Com o identificador do arquivo (*verify key*), o cliente solicita a um servidor uma lista com quais servidores possuem as porções do arquivo. Essa comunicação é formalizada na figura 20. Esse protocolo é implementado pela função *who_has_parts*, que é apresentada na figura 14.

$$\begin{array}{l} C \rightarrow S : VK, userID \\ S \rightarrow C : (S_0, 1), (S_0, 2), (S_1, 1), (S_1, 3), (S_2, 2), (S_2, 3), \dots \end{array}$$

Figura 20 – Já com a *verify key*, é solicitado uma lista dos servidores que possuem as partes do arquivo

Ao fazer a solicitação de quais servidores possuem as partes do arquivo desejado, o cliente recebe uma lista de registros de quais servidores possuem qual parte do arquivo, no seguinte formato: (*SERVIDOR, PARTEDOARQUIVO*). Esse é o formato utilizado na figura 20.

Com a lista dos servidores que possuem as partes do seu arquivo o cliente pode finalmente requisitar as partes do arquivo. Esse processo é feito de acordo com o protocolo apresentado pela figura 21 e é implementado pela função *negociate_get_parts*, que é apresentada na figura 14.

$$\begin{array}{l} \text{Para } i \text{ de } 1 \text{ até } N: \\ C \rightarrow S_i : verifyKey, parti, ip, port \end{array}$$

Figura 21 – Módulo cliente envia informação para os servidores, negociando o recebimento das partes

Esse protocolo diz que o cliente é quem irá abrir a conexão para receber o arquivo, e o servidor irá se conectar com o cliente utilizando o *ip* e *port* para isso. Após conectado com o cliente o servidor enviará o arquivo para o cliente baseado pelo protocolo apresentado pela figura 22. Uma vez que esse protocolo também não é feito via XML-RPC não é definida uma função específica.

$$\begin{array}{l} \text{Para } i \text{ de } 1 \text{ até } N: \\ S_i \rightarrow C : filePart_i \end{array}$$

Figura 22 – Servidores enviando porções do arquivo para o cliente

Uma vez que o cliente tenha todas as partes do arquivo a junção dessas partes é feita e o arquivo é descriptografado utilizando a chave de criptografia *read key* que apenas o cliente possui.

A operação de atualização de um arquivo nos servidores é equivalente a remover o arquivo atual e cadastrar um novo arquivo. Como o protocolo de envio de arquivos já foi apresentado nas figuras 17 e 18, são feitas a seguir a definição do protocolo de remoção de um

arquivo dos servidores. Esse protocolo é apresentado na figura 23 e é implementado pela função *remove_file* da figura 14.

Para i de 1 até N: C → S_i : verifyKey, writeKey

Figura 23 – Protocolo que define a remoção das partes de um arquivo.

Como definido pelo protocolo de remoção de arquivos, o cliente deve fazer a busca dos servidores que possuem as partes do arquivo desejado, utilizando o protocolo já especificado na figura 20. Então o cliente envia ao servidor o identificador do arquivo que deseja excluir e a chave de escrita no arquivo. O cliente deve fazer isso para todos os servidores. Essa operação é feita no cliente para manter a filosofia do FlexA de trazer a complexidade de processamento e comunicação para o cliente.

Além de enviar, listar, excluir e atualizar os arquivos remotos, uma operação que também é de grande importância é a de mover arquivos para outro diretório. No FlexA essa operação é feita totalmente no módulo servidor de forma que o cliente não necessita enviar novamente o arquivo ao move-lo de diretório. Essa operação tem o protocolo de comunicação especificado na figura 24, e é implementado pela função *move_file* da figura 14.

C → S : verifyKey, writeKey, destinationFileName, directoryKey

Figura 24 – Protocolo para mover um arquivo.

Outro recurso muito importante do FlexA é o compartilhamento de arquivos, que é implementado através do compartilhamento das chaves de criptografia. Para que seja possível fazer o compartilhamento e armazenar o estado deles o servidor cadastra esses dados criptografados, utilizando a chave pública do usuário que recebe a autorização para acessar o arquivo compartilhado. O protocolo de comunicação para a execução desse procedimento é apresentado na figura 25, e é implementado pela função *share_file* da figura 14.

C → S : verifyKey, pubKey_B, {accessKeys}_{pubKey_B}

Figura 25 – Protocolo para o compartilhamento de arquivos

Como mostrado na figura 25, o cliente envia as chaves de acesso que desejar ao usuário *B*, cifrando-as com a chave pública de *B*.

Para que seja possível executar o procedimento de compartilhamento de arquivos é necessário obter a chave pública de um usuário. Esse procedimento é feito através do compartilhamento da chave pública entre os próprios usuários, sem o uso do sistema para isso.

Quando o usuário *B* deseja requisitar o arquivo que foi compartilhado com ele, o mesmo protocolo de recebimento de arquivos é utilizado, apenas omitindo a parte de solicitação do *salt*, pois já possui as chaves de acesso ao arquivo. Para saber quais arquivos o usuário *B* tem acesso ele deve listar os arquivos compartilhados cadastrados com sua chave pública, utilizando o protocolo exibido na figura 26, que é implementado pela função *list_shared_files* da figura 14.

$C \rightarrow S : userID$

Figura 26 – Protocolo para o listar os arquivos compartilhados com um usuário.

Por fim, após compartilhar um arquivo, caso seja necessário remover o compartilhamento, é necessário refazer a criptografia do arquivo, removê-lo do sistema e cadastrá-lo novamente, já que as chaves de acesso foram fornecidas a outro usuário no compartilhamento. Para remover o acesso a um arquivo compartilhado é utilizado o protocolo apresentado na figura 27 que é implementado pela função *unshare_file* da figura 14.

$C \rightarrow S : verifyKey, pubKey_B$

Figura 27 – Remove o acesso do usuário *B* a um arquivo compartilhado pelo usuário *C*

Vale ressaltar que embora seja custoso computacionalmente ter que recriptografar os arquivos que foram compartilhados, o FlexA possui bons mecanismos para fazer essa operação, que foram propostos (FERNANDES, 2012), estudados e implementados por (BARBOSA, 2013)

Esses são todos os protocolos que coordenam a comunicação entre o módulo cliente e o servidor (e vice e versa) para a manipulação de arquivos. Além desses protocolos ainda existem os que são utilizados para a obtenção dos metadados dos servidores, que são apresentados a seguir.

3.2.7 Protocolos de obtenção de metadados dos servidores

Para que o cliente possa se comunicar com um servidor ele deve conhecê-lo, isso inclui saber seu *serverID*, e o estado atual dos recursos do servidor.

O protocolo que rege a comunicação para a obtenção do ID do servidor é apresentado na figura 28 e é implementado pela função *get_server_id* da figura 14.

$S \rightarrow C : serverID$

Figura 28 – Servidor envia seu *serverID* para cliente.

Além do *serverID* é importante que o servidor tenha um meio de passar informações sobre o estado atual de seus recursos para o cliente. Esse protocolo é de grande importância, pois

permite que futuramente possam ser incorporados os mecanismos de balanceamento de carga propostos e implementados por (SEGURA, 2013). O protocolo para a obtenção do estado do servidor é apresentado na figura 29 e é implementado pela função *get_server_status* da figura 14.

$$S \rightarrow C : status_1, status_2, status_3, \dots, status_n$$

Figura 29 – Servidor envia para cliente métricas sobre seu estado atual.

Com todos esses protocolos definidos e formalizados o sistema possui documentação suficiente para passar para a fase de implementação da versão Android do módulo cliente.

3.3 Adaptação do servidor em Python

Com relação ao módulo servidor, a parte envolvida nesse trabalho é a de comunicação com o cliente. Dessa forma foi necessário implementar as funções definidas na figura 14, utilizando os protocolos definidos na seção anterior.

As novas funções foram agregadas ao servidor, de forma que ele ainda mantenha a compatibilidade com o módulo cliente em Python existente. Quando o módulo cliente for remodelado para trabalhar igual ao novo cliente desenvolvido, será necessário apenas remover as funções de comunicação antigas e sem uso do servidor.

Outra alteração que se fez necessária no módulo servidor foi a criação de uma nova entidade para armazenar os metadados do compartilhamento dos arquivos entre os usuários. Isso foi feito com a criação de uma nova classe para o mapeamento objeto-relacional utilizado ou *Object-Relational Mapping* (ORM) com SQLAlchemy e SQLite. Essa nova classe é apresentada na figura 30. Essas alterações foram feitas de acordo com o que é recomendado em SQLAlchemy (2014).

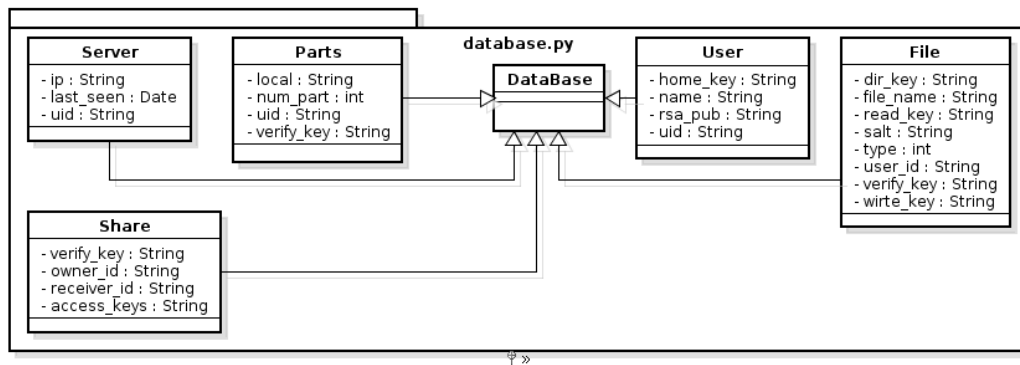


Figura 30 – Diagrama de classes mostrando a classe *Share*, responsável por armazenar os metadados sobre o compartilhamento dos arquivos.

3.4 Módulo Cliente para Android

Um módulo cliente implementado em outra arquitetura é de grande interesse como caso de teste de que as alterações propostas tornam o FlexA capaz de executar em sistemas diferentes e também testar sua nova interface de comunicação, e ainda fornecer ao usuário outra alternativa de uso do sistema.

3.4.1 Organização

Uma das principais contribuições do cliente para Android ao projeto original é a modularização e organização do projeto, separando as funcionalidades e implementação de recursos em pacotes bem definidos com o uso de interfaces de desenvolvimento que forneçam flexibilidade ao projeto.

A modularização do trabalho desenvolvido é mostrada através da árvore de pacotes. Dentro de cada pacote permanecem apenas implementações de tarefas do mesmo tipo. Essa árvore é apresentada na figura 31.

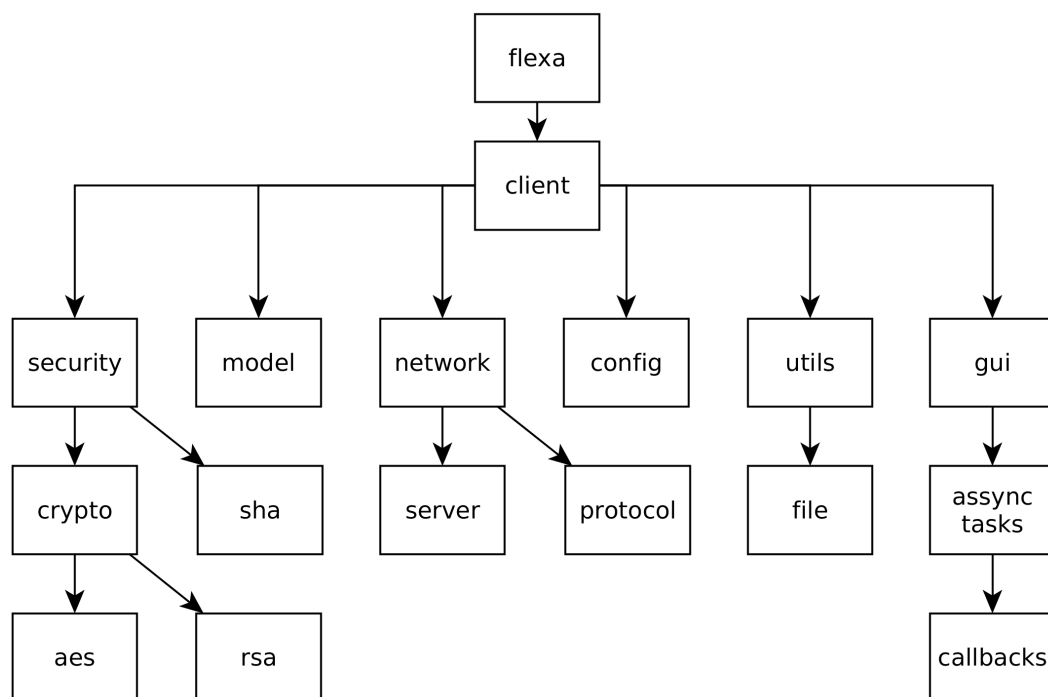


Figura 31 – Hierarquia de pacotes que compõe o módulo Cliente para Android

Da forma como o sistema está estruturado, quando no futuro for necessário alterar o módulo cliente ou até mesmo implementar o módulo servidor para Android ou em Java, essa tarefa será bem simples de ser feita pois toda a estrutura do sistema estará pronta, e assim o módulo servidor poderá importar apenas os pacotes que necessita, sem que para isso, seja necessário carregar junto funções e bibliotecas desnecessárias.

3.4.2 Modularização e Flexibilidade

A modularização do projeto ocorre não apenas pelo uso de pacotes e separação do código em classes bem definidas. Para que o projeto possa se manter a longo prazo flexível, são utilizadas diversas interfaces para garantir que caso seja necessário realizar alterações em mecanismos de comunicação e na interface gráfica do usuário ou *Graphical User Interface* (GUI), essas alterações sejam simples de serem implementadas. Os principais pontos são listados a seguir.

3.4.3 Encapsulamento da complexidade do FlexA

Embora o XML-RPC forneça um alto grau de encapsulamento da comunicação, ainda é necessário obter uma forma de simplificar o acesso as funções fornecidas pelos servidores XML-RPC e toda a sequência ações que devem ser feitas. Esse encapsulamento é obtido utilizando a classe *Client*, que é mostrada na figura 32. Dessa forma, caso aconteça qualquer alteração nos protocolos ou nos parâmetros o resto do cliente não precisa ser alterado, ou irá precisar de apenas pequenos ajustes.

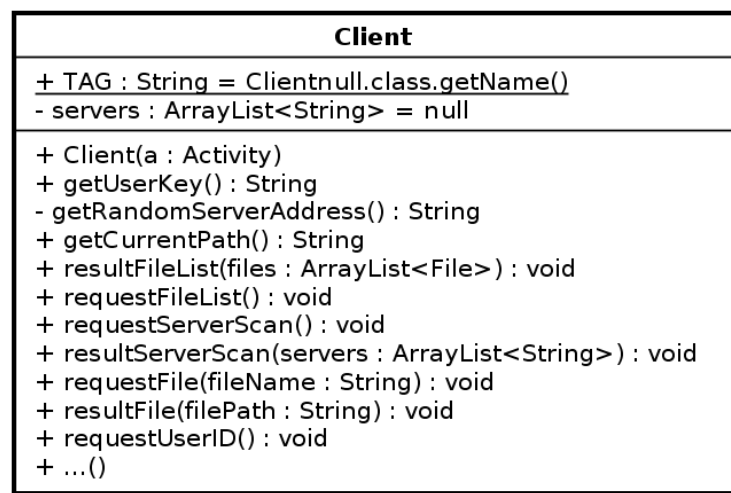


Figura 32 – Classe *Client*, responsável por encapsular servidores e manipulação dos arquivos.

3.4.4 Encapsulamento da comunicação com os servidores que utilizam XML-RPC

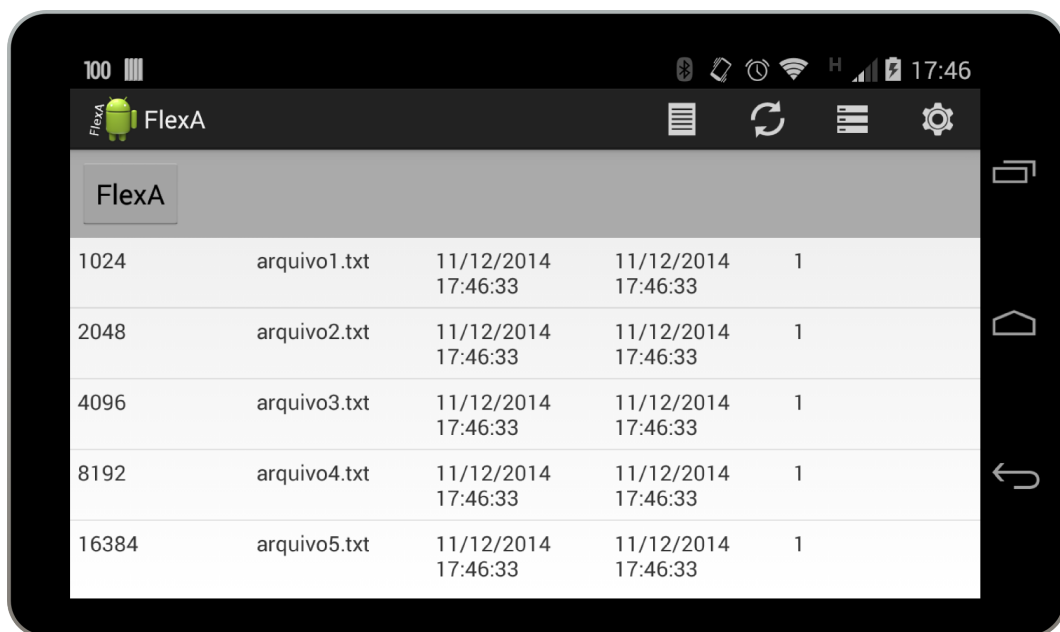
Ainda sim, para isolar o módulo cliente do cliente XML-RPC utilizado existe a classe *FlexaServer* que é apresentada na figura 33. Essa classe é a responsável por se conectar com os servidores RPC e implementar as operações do lado do cliente mostradas na seção anterior na figura 14.

FlexaServer
+ TAG : String = FlexaServer.null.class.toString()
+ FlexaServer(address : String, port : int)
+ listFiles(callback : ListFilesTaskCallback, directoryKey : String) : void
+ negotiateGetPart(verifyKey : String, part : int, ip : int, port : int) : int
+ getSalt(fileName : String, userKey : String) : String
+ getUserMetadata(public_key : String) : String
+ whoHasParts(verify_key : String) : Object[]
+ negotiateStorePart(user_id : String, file_name : String, verify_key : String, directory_key : String, write_key : String, salt : String, part_number : int) : int
+ negotiateGetPart(verify_key : String, part_number : int, ip : String, port : int) : int
+ removeFile(verify_key : String, write_key : String) : boolean
+ moveFile(verify_key : String, write_key : String, new_file_name : String, directory_key : String) : boolean
+ shareFiles(verify_key : String, public_key : String, access_keys : String[]) : boolean
+ listSharedFiles(user_id : String) : File[]
+ unshareFile(verify_key : String, public_key : String) : boolean
+ getServerId() : String
+ getServerStatus() : String[]

Figura 33 – Interface de comunicação com os servidores XML-RPC.

3.4.5 Interface Gráfica do Usuário no Android

Com as classes *FlexaServer* e *Client*, o resto do módulo cliente fica encapsulado e livre da comunicação com os servidores, devendo assim apenas cuidar da entrada e saída dos dados e da comunicação com o usuário do sistema através de uma interface gráfica. Uma das telas do sistema é exibida na figura 34.



The screenshot shows the FlexA application on an Android device. The status bar at the top displays the time as 17:46 and various system icons. The app's title bar shows 'FlexA' with an Android icon. Below the title bar, there is a list of files with the following columns: ID, filename, upload date, download date, and a status value.

ID	filename	upload date	download date	status
1024	arquivo1.txt	11/12/2014 17:46:33	11/12/2014 17:46:33	1
2048	arquivo2.txt	11/12/2014 17:46:33	11/12/2014 17:46:33	1
4096	arquivo3.txt	11/12/2014 17:46:33	11/12/2014 17:46:33	1
8192	arquivo4.txt	11/12/2014 17:46:33	11/12/2014 17:46:33	1
16384	arquivo5.txt	11/12/2014 17:46:33	11/12/2014 17:46:33	1

Figura 34 – Exemplo de interface gráfica listando os arquivos do usuário.

A utilização de uma GUI para Android, embora seja necessária para a comodidade do usuário final, é bem mais complexa que a implementação para interface de linha de comando ou *Command-line Interface* (CLI) que existe atualmente no cliente em Python. Para obter um resultado satisfatório no uso de uma GUI, é necessário tratar de forma assíncrona e também utilizar *callbacks* nas operações de entrada e saída, de modo a evitar o congelamento do aplicativo ao realizar essas tarefas (Android Open Source Project; GOOGLE, 2014). Dois exemplos dessas tarefas são mostrados na figura 35.

Junto com as tarefas assíncronas estão também como exemplo uma parte da classe

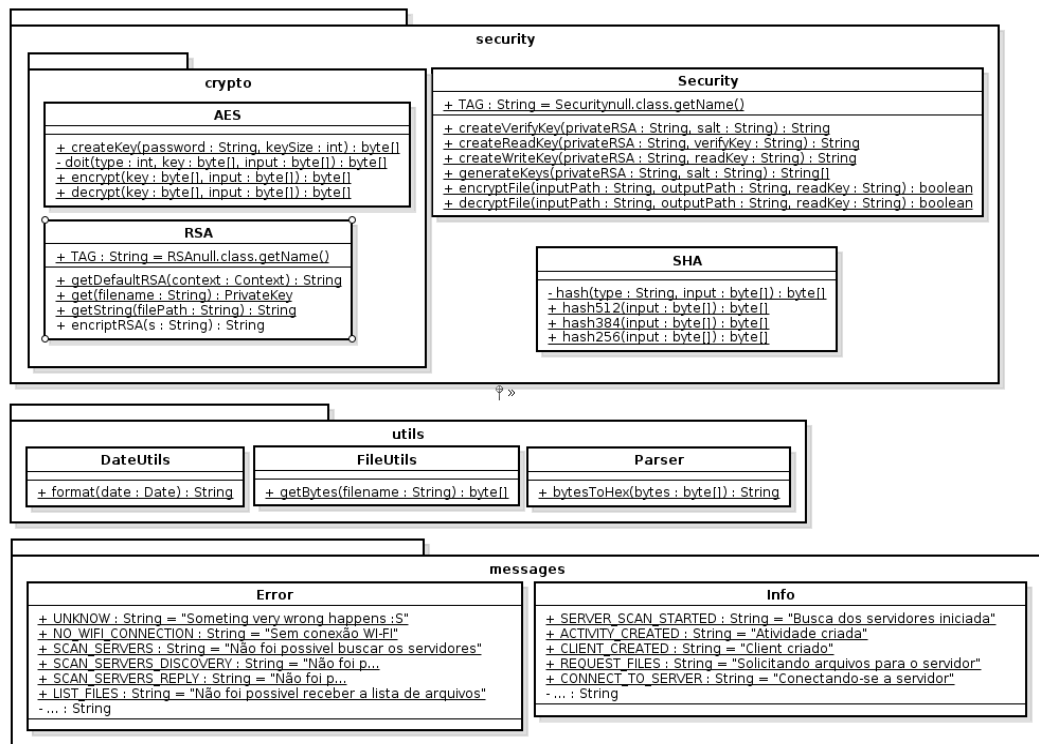


Figura 36 – Seções das classes de criptografia e acesso a arquivos.



Figura 37 – Módulo de configurações, fornece uma melhor usabilidade do sistema.

3.5 Considerações finais

Após a descrição do projeto proposto junto com seu desenvolvimento, são apresentados a seguir os cenários e testes realizados para validar o funcionamento do sistema bem como seu desempenho.

4 Testes e resultados

Nesse capítulo serão apresentados os ambientes de testes e os resultados obtidos.

4.1 Ambiente de Testes

O ambiente de testes utilizou os equipamentos disponíveis no laboratório do GSPD e um celular próprio.

Esses equipamentos são:

- *Cluster* heterogêneo com 16 nós:
 - 8 nós equipados Intel(R) Core(TM) i7, 16GB de RAM, HD de 500GB
 - 8 nós com Intel(R) Pentium(R) Dual, 4 GB de RAM, HD de 80GB
 - Conexão 10/100/1000
- *Tablet* Samsung GALAXY Note 10.1 - 2014 (referente ao processo FAPESP 2012/02926-5):
 - Android 4.3 (Jelly Bean)
 - 3 GB de memória RAM
 - 16GB de armazenamento interno (*flash*)
 - bateria de 8220 mAh
 - Chipset Qualcomm Snapdragon 800
 - CPU Quad-core 1.9 GHz Cortex-A15
 - GPU Adreno 330
- *Smart-phone* Motorola Moto X XT1058:
 - Android 4.4.4 (KitKat)
 - 2GB de memória RAM
 - 16GB de armazenamento interno (*flash*)
 - bateria de 2200 mAh
 - Chipset Qualcomm MSM8960DT Snapdragon S4 Pro
 - CPU Dual-core 1.7 GHz Krait 300

– GPU Adreno 320

- Roteador Wi-fi Asus 300 Mbps

Os arquivos utilizados para realizar os testes são fotografias de altíssima resolução obtidas em (European Space Agency, 2014). Sendo uma fotografia de 6.6MB, outra de 15M e a maior de 172MB.

Nos testes de desempenho e eficiência, para amenizar influências de outros processos e do sistema operacional, os testes foram realizados 10 vezes com intervalos aleatórios de 1 a 2 minutos. Os tempos utilizados nos gráficos são as médias dessas execuções.

4.2 Integridade dos Dados

Segundo Coulouris, Dollimore e Kindberg (2005), um dos principais deveres de um sistema de arquivos distribuídos é manter a integridade de seus arquivos. Assim foram feitos testes de integridade dos arquivos, para verificar se o processo de criptografia e transmissão dos arquivos não danificam os arquivos.

Inicialmente era calculado o *hash* MD5 dos arquivos, que então eram copiados via cabo de dados para os dispositivos móveis. Então, eram transferidos e recuperados do FlexA via *Wi-fi* e, então, eram novamente copiados para um computador via cabo de dados. Depois era calculado o *hash* MD5 dos arquivos para mostrar a integridade dos mesmos. Na figura 38 são mostrados os resultados.

Todos os testes mostraram resultados positivos quanto a integridade dos arquivos ao serem enviados e retornarem pelo FlexA utilizando o Android.

4.3 Compatibilidade do sistema

Como o módulo cliente implementado do FlexA deve ser compatível com o módulo existente, foram feitos testes semelhantes aos de integridade mostrado anteriormente, exceto que os arquivos eram inseridos no FlexA pelo cliente em Python e depois recuperados pelo cliente Android. Na figura 39 são apresentados os resultados desse teste.

4.4 Desempenho de criptografia

Esse teste mostra o tempo necessário para criptografar arquivos os arquivos de testes em cada um dos dispositivos móveis e também em um nó do primeiro grupo do *cluster* (computadores com processador Intel Core i7) para comparação. Os resultados são mostrados na figura 40 e os tempos são apresentados na tabela 2.

```

→ FlexA tree ..
..
├── FlexA
│   ├── Hubble - Andromeda.tif
│   ├── Hubble - Pilares da Criação.tif
│   └── Hubble - Sombreiro.tif
└── Original
    ├── HASH
    ├── Hubble - Andromeda.tif
    ├── Hubble - Pilares da Criação.tif
    └── Hubble - Sombreiro.tif

2 directories, 7 files
→ FlexA ls -lht
total 193M
-rw-r--r-- 1 gabriel users 172M Dez 28 09:59 Hubble - Sombreiro.tif
-rw-r--r-- 1 gabriel users 15M Dez 28 09:40 Hubble - Andromeda.tif
-rw-r--r-- 1 gabriel users 6,6M Dez 28 09:28 Hubble - Pilares da Criação.tif
→ FlexA md5sum -c ../Original/HASH
Hubble - Andromeda.tif: SUCESSO
Hubble - Pilares da Criação.tif: SUCESSO
Hubble - Sombreiro.tif: SUCESSO
→ FlexA

```

Figura 38 – Teste de integridade dos arquivos com Android. Nenhum arquivo foi danificado durante os testes.

```

→ Compatibilidade tree
.
├── Android
│   ├── HASH
│   ├── Hubble - Andromeda.tif
│   ├── Hubble - Pilares da Criação.tif
│   └── Hubble - Sombreiro.tif
└── Python
    ├── Hubble - Andromeda.tif
    ├── Hubble - Pilares da Criação.tif
    └── Hubble - Sombreiro.tif

2 directories, 7 files
→ Compatibilidade ls -lht Python
total 193M
-rw-r--r-- 1 gabriel users 172M Dez 28 10:45 Hubble - Sombreiro.tif
-rw-r--r-- 1 gabriel users 6,6M Dez 28 10:17 Hubble - Pilares da Criação.tif
-rw-r--r-- 1 gabriel users 15M Dez 28 10:13 Hubble - Andromeda.tif
→ Compatibilidade cd Python
→ Python md5sum -c ../Android/HASH
Hubble - Andromeda.tif: SUCESSO
Hubble - Pilares da Criação.tif: SUCESSO
Hubble - Sombreiro.tif: SUCESSO
→ Python

```

Figura 39 – Teste de compatibilidade entre os dois módulos clientes do FlexA. Nenhum arquivo foi danificado durante os testes mostrando que os sistemas estão compatíveis.

Como esperado, o computador do *cluster* foi capaz de criptografar os arquivos aproximadamente 3,4 vezes mais rápido que o *tablet* e 3,8 vezes mais rápido que o *smart-phone*.

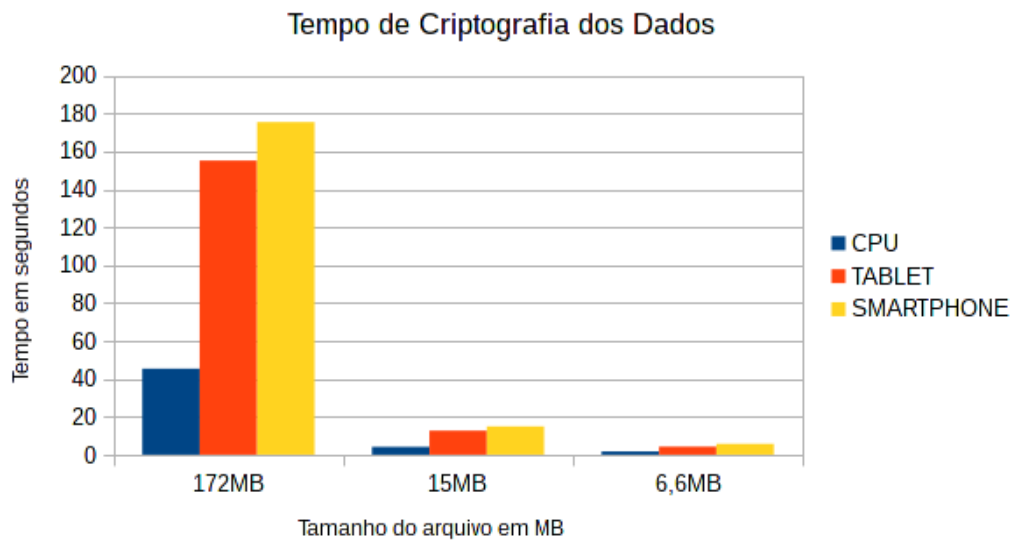


Figura 40 – Teste de desempenho de criptografia com clientes diferentes do FlexA

Tabela 2: Tempos de criptografia para os dispositivos utilizados.

Tamanho (MB)	Tempo por dispositivo (s)		
	CPU	TABLET	SMARTPHONE
172MB	45,36	155,13	175,44
15MB	4,12	12,7	14,89
6,6MB	1,7	4,2	5,7

4.5 Desempenho de transmissão dos dados

Esse teste mostra o tempo necessário para transmitir os arquivos via *Wi-fi* para o FlexA. Os resultados são mostrados na figura 41, os tempos são detalhados na tabela 3.

Tabela 3: Tempos de transmissão dos arquivos utilizados para testes através de rede *wi-fi*.

Tamanho do Arquivo (MB)	Tempo (s)
6,6	3,53
15	6,9
172	68,59

Como esperado, o particionamento do arquivo quase não influenciou na transmissão, uma vez que o gargalo nesse caso é a transmissão sem fio dos dados.

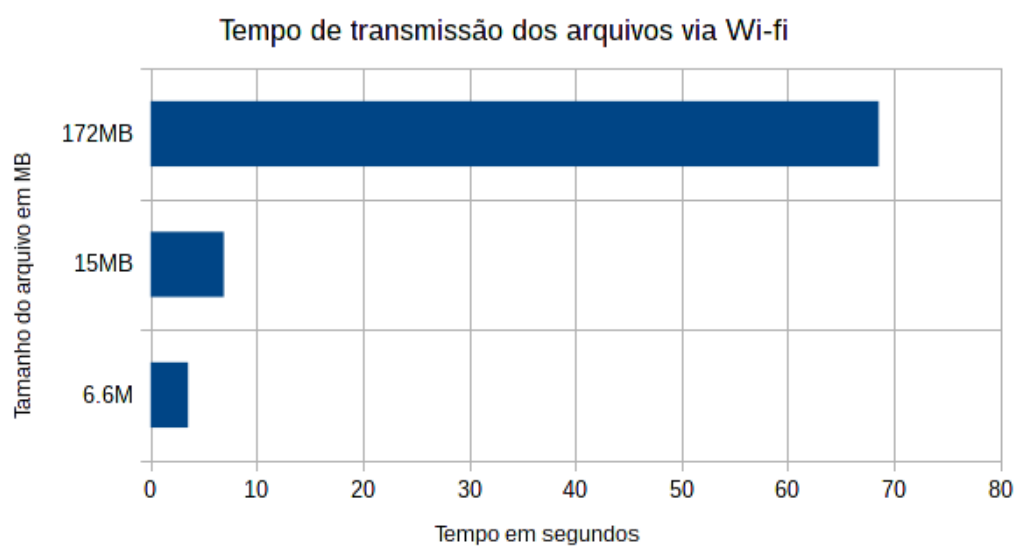


Figura 41 – Teste de desempenho de transmissão de arquivos com clientes diferentes do FlexA

5 Conclusão

A execução desse projeto mostrou-se benéfica ao FlexA como um todo, desde a formalização dos protocolos de comunicação, que tornou o sistema muito mais fácil de ser implementado e entendido, testado e no futuro expandido, até a implementação do cliente Android, que além de oferecer ao usuário outra alternativa mais simples para uso, ainda serviu para aprimorar a flexibilidade do projeto, ao torná-lo mais aberto e padronizado.

Com os testes executados, foi possível perceber principalmente que o sistema está se comportando de forma esperada e que embora o FlexA não seja um sistema de arquivos distribuídos voltado para dispositivos móveis, a segurança dos dados que o sistema oferece, faz com que o tempo total de envio e recebimento dos arquivos via *Wi-fi* seja aceitável. Quanto ao desempenho de criptografia e divisão dos arquivos, os dispositivos móveis são seriamente prejudicados por restrições de autonomia energética, aumentando significativamente o consumo elétrico para os clientes, o que é não é o ideal, mas inevitável dentro das premissas do projeto. Os testes de compatibilidade e integridade mostram que os módulos cliente em Python e em Android estão trabalhando de forma compatível e fornecem os resultados esperados, não danificando os arquivos que trafegam pelo sistema.

Dessa forma, é possível dizer que o projeto atingiu seu objetivo e o trabalho mostrou resultados positivos, principalmente no que se diz respeito a aumento da flexibilidade e abertura do FlexA.

5.1 Dificuldades

Os principais desafios para a execução do trabalho foram modelar interfaces concisas para o FlexA e adaptar o módulo servidor, devido a pouca experiência com a linguagem Python. Outro desafio encontrado foi o aprendizado do uso da base de dados SQLite com a ferramenta de ORM SQLAlchemy.

Deve-se ressaltar também o aprendizado de programação para Android, que apesar de ser feito utilizando linguagem Java, apresenta diversos desafios principalmente relacionados a construção de uma interface gráfica de forma modular e assíncrona.

Por fim, talvez o mais complexo foi a elaboração de interfaces de comunicação que fossem genéricas o suficiente, mas não muito relaxadas, o que fornece ao sistema uma boa flexibilidade.

5.2 Trabalhos futuros

Como sugestões para trabalhos futuros, sugere-se que os protocolos formalizados sejam adaptados para garantir ao sistema proteção contra ataques simples de repetição e injeção de pacotes, utilizando criptografia de toda a comunicação sensível e uso de *tokens* únicos nas comunicações.

É de grande interesse a utilização de *hash* para validação da integridade dos arquivos transmitidos de e para os servidores, e também a assinatura digital desses dados. Para a assinatura também é necessário a implementação de um mecanismo de troca de chaves criptográfica entre o servidor e o cliente. Fica sugerida a implementação do algoritmo Diffie-Hellman para este problema (MERKLE, 1978).

Por fim, também é sugerido o teste com ferramentas de compressão, para comprimir os dados antes de realizar a criptografia, o que pode acelerar o processo de criptografia e também a transmissão dos dados.

Referências

ANDERSON, R. *Security engineering*. [S.l.]: John Wiley & Sons, 2008.

Android Open Source Project; GOOGLE. *Processes and Threads - Android Developers*. 2014. <<http://developer.android.com/reference/android/os/AsyncTask.html>>. Acessado em 20/12/2014 as 14:15.

BARBOSA, L. M. *Implementação e avaliação de desempenho de algoritmo de criptografia em GPU para o FlexA*. Monografia (Graduação) — Instituto de Biociências Letras e Ciências Exatas, Universidade Estadual Paulista, 2013.

BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 2, n. 1, p. 39–59, 1984.

CARVALHO, L. R. de. *Disponibilidade em um sistema de arquivos distribuído flexível e adaptável*. Dissertação (Mestrado) — Instituto de Biociências Letras e Ciências Exatas, Universidade Estadual Paulista, 2014.

CISCO. *XML-RPC Getting Started*. 2010. <http://www.cisco.com/c/en/us/td/docs/security/physical_security/cnbm_mgr/1-x/API/XML_RPC_MGR/Gettingstarted.html>. Acessado em 20/10/2014 as 15:30.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed systems: concepts and design*. [S.l.]: Addison-Wesley, 2005.

European Space Agency. *Hubble Telescope*. 2014. <<http://spacetelescope.org/images/viewall/>>. Acessado em 28/12/2014 as 08:30.

FERNANDES, S. E. N. *Sistema de Arquivos Distribuído Flexível e Adaptável*. Dissertação (Mestrado) — Universidade Estadual Paulista, 2012.

HUCKA, M. et al. *A Comparison of Two Alternative Implementations of Message-Passing in the Systems Biology Workbench*. 2001. <<http://sbw.sourceforge.net/sbw/docs/messaging-comparison/html/messaging-comparison.html>>. Acessado em 20/12/2014 as 02:15.

IBM. *UML basics: The class diagram*. 2004. <<http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell>>. Acessado em 05/11/2014 as 10:30.

International Data Comporation. *Smartphone OS Market Share, Q3 2014*. 2014. <<http://www.idc.com/prodserv/smartphone-os-market-share.jsp>>. Acessado em 23/12/2014 as 23:40.

MERKLE, R. C. Secure communications over insecure channels. *Communications of the ACM*, ACM, v. 21, n. 4, p. 294–299, 1978.

NETO, M. C. *FlexA: Desenvolvimento de um Sistema de Arquivos Flexível e Adaptável*. Monografia (Relatório Técnico) — Instituto de Biociências Letras e Ciências Exatas, Universidade Estadual Paulista, 2013.

OKADA, T. K. *Metodologia para Recuperação de Falhas e Garantia de Disponibilidade no FlexA*. Monografia (Graduação) — Instituto de Biociências Letras e Ciências Exatas, Universidade Estadual Paulista, 2013.

OLIVEIRA, M. D. C. *Sincronização, consistência e falhas no FlexA*. Monografia (Graduação) — Instituto de Biociências Letras e Ciências Exatas, Universidade Estadual Paulista, 2013.

Open Handset Alliance. *Android Open Source Project*. 2014. <<http://web.archive.org/web/20141009024350/http://developer.android.com/legal.html>>. Acessado em 22/12/2014 as 10:00.

SEGURA, D. C. M. *Detecção de Falhas de Comunicação e Balanceamento de Carga no FlexA*. Monografia (Graduação) — Instituto de Biociências Letras e Ciências Exatas, Universidade Estadual Paulista, 2013.

SHAMIR, A. How to share a secret. *Communications of the ACM*, ACM, v. 22, n. 11, p. 612–613, 1979.

SQLAlchemy. *SQLAlchemy 0.9 Documentation*. 2014. <http://docs.sqlalchemy.org/en/rel_0_9/dialects/sqlite.html>. Acessado em 20/11/2014 as 15:40.

TANENBAUM, A. S.; STEEN, M. V. *Distributed systems: principles and paradigms*. [S.l.]: Pearson Prentice Hall, 2007.

Visual Paradigm. *Writing Effective Use Case*. 2011. <<http://www.visual-paradigm.com/tutorials/writingeffectiveusecase.jsp>>. Acessado em 01/11/2024 as 16:30.

ZHENG, P.; NI, L. *Smart phone and next generation mobile computing*. [S.l.]: Morgan Kaufmann, 2010.