UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DIEGO KELLER DA SILVA

Desenvolvimento de uma API para o Gerenciamento de Arquivos Distribuídos

Prof. André L. Martinoto
Orientador

Prof.ª Dr.ª Maria F. W. P. Lima
Co-orientador

Caxias do Sul, Dezembro de 2009

"Dedico este trabalho em especial a minha tão amada família, e a todas as pessoas que há muito tempo tem me apoiado e me incentivado a vencer."

AGRADECIMENTOS

Ao Prof. André Luis Martinoto pelo comprometimento e apoio na realização deste trabalho.

A Prof.ª Drª. Maria de Fátima Webber do Prado Lima pela atenção dedicada.

A todos os professores da Universidade de Caxias do Sul pelo conhecimento e pela troca de experiências proporcionada.

A todos as pessoas que auxiliaram no desenvolvimento deste trabalho.

E principalmente a DEUS, pelo seu amor e pela oportunidade que nos é dada todos os dias de demonstrar nosso amor por Ele e pelos nossos irmãos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
IndexVideo	Projeto de Indexação de Vídeo
UCS	Universidade de Caxias do Sul
IP	*Internet Protocol*
RPC	*Remote Procedure Call*
RMI	*Remote Method Invocation*
DFS	*Distributed File System*
GFS	*Google File System*
NFS	*Network File System*
SSL	*Secure Sockets Layer*
VFS	*Virtual File System*
NIS	*Network Information Service*
HDFS	*Hadoop Distributed File System*
FTP	*File Transfer Protocol*
PDF	*Portable Document Format*
XML	*Extensible Markup Language*
IDC	*International Data Corporation*
MIT	*Massachusetts Institute of Technology*
JDK	*Java Development Kit*
JRE	*Java Runtime Enviroment*
DNS	*Domain Name Service*
MAC	*Media Access Control*

JDBC *Java Database Connectivity*

LISTA DE FIGURAS

RESUMO

O surgimento da *Internet* e de novos dispositivos como câmeras digitais tem contribuído para um aumento crescente na quantidade de informação que é criada e compartilhada entre pessoas e instituições. Esse grande volume de informações deve ser adequadamente armazenado e gerenciado por sistemas de arquivos eficientes.

Na grande maioria dos casos esse armazenamento é feito utilizando-se um único servidor central, porém a velocidade de crescimento dos dados e a quantidade de usuários fazem surgir a necessidade da criação de sistemas distribuídos, mais especificamente, de sistemas de arquivos distribuídos, que sejam capazes de acompanhar esse crescimento e manter a eficiência do sistema.

Diversos sistemas de arquivos distribuídos já foram implementados, no entanto, a maioria deles está focado em agrupar arquivos distribuídos em diversos servidores e apresentá-los aos usuários em uma visão única e centralizada, e não em realizar a distribuição dos arquivos entre os servidores.

Diferentemente desses sistemas já existentes, o objetivo deste trabalho é desenvolver um sistema de arquivos distribuído que permita, além do agrupamento de arquivos, que esses sejam distribuídos de forma balanceada entre os servidores do sistema, abstraindo sua localização e fornecendo uma solução simples e eficiente para o armazenamento de grandes quantidades de arquivos.

Palavras-chave: API, Sistemas de Arquivos Distribuídos, Distribuição e Armazenamento de Arquivos.

ABSTRACT

The upcoming of the Internet and new devices like digital cameras contributed to increase the amount of information that is created and shared between people and institutions. This large volume of information must be properly stored and managed by efficient file systems.

In most cases the storage is done using a single central server, but the growing rate of data and number of users is increasing the need for distributed systems, more specifically distributed file systems, which are capable to handle this increase and keep the performance of the system.

Several distributed file systems have been implemented, however, mostly focused on grouping files scattered on different servers and presenting them to users in a single, centralized view, but not to distribute files between servers.

In opposite to existing systems, the goal of this work is to develop a distributed file system that allows, besides of grouping files, balancing the files between the servers of the system, disregarding their location and providing a simple and efficient solution for the storage of a large quantity of files.

Keywords: API, Distributed File Systems, File Distribution and Storage.

1 INTRODUÇÃO

A Tecnologia da Informação tem se tornado cada vez mais popular. As vantagens oferecidas pelos recursos computacionais para capturar, gerar, armazenar e recuperar dados, contribuíram para um grande crescimento de sua utilização. Atualmente, grande parte de tudo que é escrito, lido, gravado e reproduzido é armazenado em sistemas de informação computacionais.

Uma consequência da popularização da tecnologia e do crescimento de sua utilização, é o crescimento da quantidade de dados que é criada e armazenada. Conforme um estudo realizado pelo IDC (*International Data Corporation*), esse crescimento tem se mostrado de ordem exponencial (CHUTE et al., 2008).

Um dos fatores que influenciou esse crescimento foi o surgimento e a popularização de novos dispositivos como câmeras digitais, telefones celulares multimídia, *music players*, TV Digital, etc. Além disso, todo esse crescimento foi impulsionado pela utilização da *Internet*, que permitiu que qualquer indivíduo ou organização se tornasse um produtor de conteúdo em potencial, facilitando a replicação e distribuição de dados através do mundo (CHUTE et al., 2008; CARVALHO et al., 2006).

Um outro fator que contribui para o crescimento da quantidade de dados é a utilização de arquivos multimídia. Esse tipo de arquivo, chamado de informação narrativa, ainda é pouco explorado se comparado com as informações descritivas (informação textual), mas seu crescimento tem se mostrado notável. Diversos tipos de aplicação tem feito uso desse recurso, como exemplo pode-se citar o entretenimento digital, aplicações médicas, bibliotecas digitais (CARVALHO et al., 2006) e em especial, o ramo da educação, que tem feito uso da multimídia como uma ferramenta no processo de ensino e aprendizagem, conforme descrito em (COSCARELLI, 1998).

Esse grande conjunto de dados precisa ser devidamente armazenado e disponibilizado aos usuários, de forma simples e eficiente. Para isso, a alternativa mais comum é a utilização de um servidor de arquivos centralizado, que armazena todos os arquivos de um grupo de usuários, como por exemplo de uma instituição ou organização. No entanto, nessa abordagem centralizada o maior desafio se encontra

na necessidade do armazenamento de grandes quantidades de dados, que crescem continuamente e exigem flexibilidade e escalabilidade do sistema de armazenamento.

Dependendo do número de usuários de um sistema, um único servidor pode vir a ser incapaz de armazenar toda a informação produzida. Sendo assim, o sistema deve permitir que a sua capacidade seja facilmente aumentada, o que é difícil quando se utiliza um servidor centralizado. Além disso, existem outros fatores que dificultam a utilização dessa abordagem, como por exemplo a sobrecarga do servidor, a indisponibilidade de todos os arquivos caso o servidor apresente algum problema, etc (TANENBAUM, 1995).

Uma alternativa ao modelo centralizado, é a utilização de um modelo distribuído, onde os arquivos não ficam armazenados em um único servidor, mas sim em um conjunto de servidores. Esse modelo apresenta uma série de melhorias em relação ao modelo anterior, como por exemplo: melhor desempenho, pois as requisições dos usuários são processadas por mais de um servidor; maior disponibilidade, pois o não funcionamento de um servidor não afeta o restante do sistema; facilidade de manutenção do sistema, podendo aumentar a capacidade de armazenamento sem que seja necessário tornar o sistema indisponível; possibilidade de adicionar tantos servidores quanto o necessário para realizar o armazenamento dos arquivos (TANENBAUM, 1995).

O modelo distribuído tem sido amplamente utilizado, constituindo uma área já consolidada em computação, denominada Sistemas de Arquivos Distribuídos (TANENBAUM; STEEN, 2007). Uma série de sistemas desse tipo já foram desenvolvidos, sendo que cada um deles foca em um cenário específico de aplicação, porém todos eles apresentam um objetivo em comum: realizar o armazenamento de arquivos distribuídos de forma escalável e transparente.

No entanto, ao se fazer uma análise de alguns desses sistemas e dos cenários onde podem ser aplicados, pode-se observar que, apesar de atingirem o objetivo a que se propõem, tais sistemas não constituem uma solução transparente para o armazenamento de arquivos, ou seja, a grande maioria das ferramentas existentes tem um grande foco em concentrar arquivos distribuídos e apresentá-los de forma integrada e centralizada ao usuário. Esses sistemas partem do princípio de que os arquivos já estão armazenados em diversos servidores, e que o objetivo a ser atingido com a sua implantação é apenas apresentar os arquivos de forma centralizada, ou seja, apenas concentrar os arquivos distribuídos.

Há ainda um pequeno número de ferramentas cujo objetivo é fornecer um sistema de armazenamento de arquivos, que realmente gerencie a distribuição dos arquivos, balanceando o espaço em disco disponível em cada servidor. Algumas dessas ferramentas são o GFS e o Amazon S3, que são melhor descritos na Seção 3.2, porém esses sistemas não apresentam o mesmo cenário de utilização que deseja-se apli-

car este trabalho, possuindo particularidades que os distanciam do objetivo que se deseja atingir.

1.1 Objetivo

O presente trabalho tem como principal objetivo desenvolver um sistema de arquivos distribuído que seja transparente ao usuário, escalável, e que principalmente, seja uma ferramenta para o armazenamento de grandes quantidades de arquivos, abstraindo do usuário qualquer indício sobre a sua localização e realizando uma distribuição uniforme dos mesmos de acordo com a capacidade de cada servidor.

Diferentemente dos sistemas de arquivos distribuídos já existentes, o presente trabalho terá como principal objetivo gerenciar a distribuição de grandes quantidades de arquivos em diferentes servidores, ou seja, não estará focado em agrupar os arquivos em uma visão unificada para apresentá-los aos usuários, mas sim em fornecer uma ferramenta que seja capaz de distribuir os arquivos entre os servidores que compõem o sistema, permitindo que posteriormente sejam recuperados, sem que para isso seja necessário conhecer a sua localização.

Apesar de serem extremamente importantes, questões específicas desse sistema como segurança, replicação e tolerância a falhas serão descritas em alguns momentos deste trabalho, mas não constituirão um objetivo a ser atingido. A decisão da não implementação dessas funcionalidades tem como objetivo delimitar e simplificar o desenvolvimento do presente trabalho, possibilitando que sejam objetos de estudos e aprimoramentos em um momento posterior.

O sistema distribuído desenvolvido no decorrer deste trabalho deve apresentar-se ao usuário como uma API (*Application Program Interface*) que fornecerá as principais funções de um sistema de arquivos convencional: adicionar, atualizar, excluir e obter arquivos. O usuário direto dessa API será um desenvolvedor que tenha a necessidade de armazenar grandes quantidades de arquivos através de sua aplicação, sem que, para isso, tenha que investir tempo no desenvolvimento de uma ferramenta de gerenciamento e armazenamento de arquivos.

1.2 Motivação

A principal motivação para o desenvolvimento deste trabalho foi a necessidade de armazenamento de grandes volumes de dados, a qual não é completamente atendida pelos sistemas distribuídos atuais. A verificação de que as ferramentas atuais não são capazes de suprir completamente as necessidades deste trabalho fez surgir a necessidade de propor um novo sistema de arquivos, que atendesse aos objetivos especificados.

Além disso, outro fator que contribuiu para a motivação deste trabalho foi o relatório resultante do seminário "Grandes Desafios da Pesquisa em Computação no Brasil –2006 –2016", realizado pela Sociedade Brasileira de Computação em São Paulo, nos dias 8 e 9 de maio de 2006 (CARVALHO et al., 2006). Nesse relatório, o primeiro grande desafio descrito é a "Gestão da Informação em grandes volumes de dados multimídia distribuídos".

Conforme é descrito nesse relatório, um grande desafio envolve questões associadas a problemas centrais que não podem ser resolvidos por pesquisas que objetivam resultados de curto-prazo. O presente trabalho não é um grande projeto de pesquisa, nem possui recursos intelectuais suficientes para desenvolver uma pesquisa que fosse capaz de propor soluções inovadoras, nem é seu objetivo. No entanto, a descrição desse desafio e o assunto que ele aborda foram considerados de grande interesse e afinidade, contribuindo assim para a motivação do desenvolvimento de um trabalho relacionado a esse assunto.

Um outro fator que motivou o desenvolvimento deste trabalho foi a necessidade apresentada pelo Projeto IndexVideo que está sendo desenvolvido no Centro de Ciências Exatas e Tecnologia da Universidade de Caxias do Sul, sob a coordenação da Profª Drª Maria de Fátima Webber do Prado Lima. Esse projeto tem como objetivo o desenvolvimento de uma ferramenta que permita a anotação de vídeos e a pesquisa dos mesmos através das anotações já feitas. Uma anotação de um vídeo nada mais é do que um conjunto de comentários sobre os trechos de um vídeo que descrevem o assunto ou os temas relacionados ao trecho de vídeo em questão. Através dessas anotações, usuários podem realizar consultas por palavras chaves e visualizar o trecho do vídeo correspondente ao assunto pesquisado.

A ferramenta desenvolvida para o Projeto IndexVideo foi implementada utilizando a linguagem JAVA, e foi implementada sob o modelo de arquitetura cliente/servidor, onde o servidor é responsável por armazenar os vídeos e as anotações, enquanto que o cliente fornece uma interface gráfica que permite o *upload* dos vídeos e das suas anotações, além de permitir também a realização de pesquisas e a exibição dos vídeos encontrados (PRADO LIMA; SILVA; CARBONERA, 2009).

Após a implementação das primeiras versões da ferramenta e a realização de testes, os idealizadores do projeto observaram que a utilização de um único servidor central para o armazenamento dos vídeos poderia não ser a melhor alternativa devido à grande capacidade de armazenamento que esses arquivos necessitam. Um único servidor poderia vir a ser incapaz de armazenar todos os vídeos necessários, e também dificultaria o crescimento do sistema. Uma outra consequência da utilização de um servidor centralizado é que, com o aumento do número de usuários realizando pesquisas e solicitando vídeos, o desempenho do sistema poderia vir a ser prejudicado consideravelmente.

Dadas essas condições, uma alternativa seria a utilização de vários servidores para o armazenamento dos vídeos, construindo assim um sistema de arquivos distribuído. Essa nova arquitetura forneceria a escalabilidade desejada, permitindo que novos servidores fossem adicionados ao sistema, tanto para um aumento da capacidade de armazenamento bem como para melhorar o desempenho. Dessa forma optou-se pela utilização do Projeto IndexVideo como um estudo de caso para a validação da API que será desenvolvida neste trabalho.

1.3 Estrutura do Texto

O presente trabalho está organizado da seguinte forma: no Capítulo 2 é feita uma revisão sobre sistemas distribuídos, descrevendo suas principais vantagens e os desafios que estão atrelados ao seu desenvolvimento. Nesse capítulo é feita também uma categorização dos tipos de sistemas distribuídos de acordo com o objetivo a que se destinam.

No Capítulo 3 é feito um estudo mais aprofundado sobre os sistemas de arquivos distribuídos, que são o tipo de sistema distribuído que este trabalho se encaixa. Nesse capítulo são descritas as principais questões de projeto envolvidas no desenvolvimento de sistemas distribuídos desse tipo. Em seguida é feito um estudo dos sistemas de arquivos distribuídos já existentes e que tem sido mais utilizados pela indústria e estudados no meio acadêmico, descrevendo os cenários de utilização aos quais se aplicam e algumas das decisões de projeto que têm algum relacionamento com o presente trabalho.

No Capítulo 4, o objetivo deste trabalho é descrito em mais detalhes, onde são listados o objetivo principal e os objetivos secundários que deseja-se atingir com o seu desenvolvimento. Nesse capítulo também são descritas as premissas que serão assumidas sobre o ambiente de utilização do sistema, esclarecendo e exemplificando o seu cenário de aplicação. Após isso, é feita a proposta de uma arquitetura para o desenvolvimento de um sistema de arquivos distribuído que atenda os objetivos especificados. Nesse momento, as questões de projeto abordadas no Capítulo 3 são retomadas, a fim de descrever as decisões que foram tomadas para o desenvolvimento do sistema que está sendo proposto.

No Capítulo 5 são descritos os detalhes relacionados à arquitetura e à implementação do sistema que foi proposto. Inicialmente é apresentada uma visão geral da arquitetura do sistema, onde os componentes de *software* envolvidos são identificados e a camada que permite a comunicação entre esses componentes é descrita. Em seguida é feita uma descrição mais detalhada sobre cada um dos componentes de *software* do sistema, onde são apresentados os objetivos específicos de cada componente e como cada um deles foi implementado. Por fim, são descritas as API's já

existentes que foram utilizadas para a implementação desse trabalho.

No Capítulo 6 são apresentadas as principais conclusões obtidas através da pesquisa realizada nesse trabalho. Em seguida, observações e conclusões relacionadas à implementação do sistema proposto serão descritas, avaliando o processo de desenvolvimento, o resultado final obtido e a utilização do Projeto IndexVideo como estudo de caso. Além disso, são citadas algumas sugestões de trabalhos futuros que podem ser utilizadas para dar continuidade ao desenvolvimento desse trabalho.

2 SISTEMAS DISTRIBUÍDOS

Analisando a história dos sistemas de computação observa-se que entre 1945 e 1985 o cenário tecnológico era muito diferente do atual. Naquela época, apenas grandes empresas e organizações tinham acesso a computadores, uma vez que esses eram grandes e caros. Além disso, a falta de uma forma de conectá-los acabava por determinar que eles trabalhassem de forma independente.

Atualmente tem-se um cenário completamente diferente, uma vez que os computadores tornaram-se mais rápidos e baratos. Aliado a isso, a invenção de redes de computadores que permitem a comunicação cada vez mais eficiente entre esses computadores espalhados por todo o mundo, possibilita que esses troquem dados entre si a uma velocidade que pode variar de 64 Kbits/s até *gigabits* por segundo (TANENBAUM; STEEN, 2007).

Essa mudança de paradigma abriu espaço para que novas arquiteturas de sistemas fossem criadas. A facilidade de conexão entre computadores e o baixo custo dos mesmos constitui-se como um fator motivador para a construção de sistemas distribuídos. De acordo com (TANENBAUM; STEEN, 2007), um sistema distribuído constitui-se de "um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente".

A partir dessa definição observam-se dois aspectos importantes a serem considerados no desenvolvimento de uma aplicação para um sistema distribuído. O primeiro deles diz respeito à autonomia dos computadores que compõem o sistema, os seja, os computadores de um sistema distribuído são independentes e autônomos entre si. O segundo deles é que os usuários do sistema distribuído devem ter a impressão de que estão utilizando um sistema único, não conhecendo o fato da distribuição, isto é, o sistema deve apresentar-se ao usuário de forma transparente, ocultando a distribuição dos computadores.

2.1 Vantagens dos Sistemas Distribuídos

Em resumo, um sistema distribuído, propõe que computadores independentes colaboram entre si, de forma organizada e transparente ao usuário, a fim de atingir um objetivo específico. Esse novo paradigma trouxe vantagens aos usuários de sistemas de computação. De acordo com (TANENBAUM; STEEN, 2007) e (SILBERSCHATZ; GALVIN; GAGNE, 2000), algumas dessas vantagens sobre os sistemas centralizados são:

Compartilhamento de Recursos: os exemplos mais comuns de compartilhamento de recursos são impressoras, computadores, capacidade de armazenamento, dados, etc. Esse compartilhamento apresenta como principal vantagem questões econômicas. O compartilhamento de recursos de alto custo entre os usuários, permite que todos tenham acesso a esse recurso sem que seja necessário um recurso exclusivo para cada usuário. Um exemplo disso pode ser o recurso de armazenamento, onde é mais barato compartilhar um dispositivo de armazenamento (como um disco rígido por exemplo) entre vários usuários do que fornecer um disco rígido para cada usuário.

Velocidade de Computação: a utilização de um sistema distribuído possibilita que a computação possa ser particionada entre os computadores do sistema, ou seja, cada parte de uma tarefa pode ser processada paralelamente por um computador independente. Esse processamento de forma paralela e distribuída aumenta a velocidade de computação do sistema, permitindo que mais tarefas sejam executadas dentro de um mesmo período de tempo. Além disso, caso um computador do sistema distribuído esteja sobrecarregado, essas tarefas podem ser redistribuídas a outros computadores, evitando assim a criação de gargalos que possam limitar a capacidade do sistema e, com isso, permitindo que um número maior de usuários possa ser atendido.

Confiabilidade: se o sistema distribuído for projetado de forma que todos os computadores executem tarefas autônomas, ou seja, não realizem tarefas específicas e críticas ao funcionamento do sistema, a indisponibilidade de um computador não irá afetar os demais. Isso permite que as requisições dos usuários continuem sendo atendidas, agregando assim uma maior confiabilidade.

Escalabilidade: a capacidade que um sistema tem de crescer, seja adicionando usuários ou recursos, caracteriza sua escalabilidade. Em sistemas distribuídos, a independência dos computadores permite que o sistema possa ser facilmente estendido, adicionando novos computadores e serviços conforme a necessidade de crescimento.

2.2 Desafios na criação de um Sistema Distribuído

Apesar das vantagens mencionadas, o fato dos computadores estarem distribuídos e conectados entre si, traz também alguns desafios a serem vencidos. Alguns deles são citados por (COULOURIS; DOLLIMORE; KINDBERG, 2007), e são resumidamente descritos a seguir:

Heterogeneidade: a heterogeneidade pode se apresentar sob aspectos como redes, protocolos, sistemas operacionais, *hardware*, linguagem de programação, etc. Em um sistema distribuído, onde os computadores trocam mensagens e executam tarefas de forma integrada, a heterogeneidade representa um desafio, devido às diferenças de implementação entre os protocolos de comunicação, topologias de rede e variações nas representações de tipos de dados entre diferentes sistemas operacionais e linguagens de programação.

Sistemas Abertos: um sistema computacional é considerado aberto quando é possível estendê-lo e reimplementá-lo de várias maneiras. Para isso é necessário que suas principais interfaces sejam publicadas, permitindo assim que o sistema seja construído a partir de diferentes fornecedores de *hardware* e *software*. Obviamente a compatibilidade de cada componente com o padrão publicado deve ser cuidadosamente testada e verificada a fim de garantir o correto funcionamento do sistema. Desta forma, tornar um sistema aberto, apesar de suas vantagens e benefícios, exige tempo e esforço.

Segurança: para garantir segurança a um sistema de computação, três tipos de proteção devem ser fornecidas: proteção contra a exposição dos dados a pessoas não autorizadas (confidencialidade); proteção contra a alteração ou danos causados aos dados do sistema (integridade); proteção contra a interferência nos meios de acesso aos recursos (disponibilidade). Todas essas questões de segurança são importantes, porém não são o foco deste trabalho.

Escalabilidade: essa corresponde à capacidade de adicionar mais usuários e recursos a um sistema (TANENBAUM; STEEN, 2007). Um sistema é dito mais escalável quanto maior for a facilidade de incrementar o número de recursos e usuários, sem que haja uma perda de desempenho perceptível pelo usuário. Sob esse aspecto, o desafio de tornar um sistema escalável consiste em duas questões: a primeira delas é projetá-lo de forma que a adição de novos computadores seja facilitada, sem que, para isso, seja necessário tornar o sistema indisponível; e a segunda é garantir um desempenho satisfatório do sistema quando o número de computadores e usuários aumentar.

Tratamento de falhas: sistemas de computadores podem apresentar falhas de *hardware* e *software*, como por exemplo, problemas de conexão, indisponibilidade de serviços ou servidores, erros de comunicação, etc. As falhas em sistemas distribuídos são chamadas de parciais, uma vez que alguns componentes ou computadores falham, enquanto outros continuam funcionando. Para tratar esse problema de falhas, pode-se utilizar diferentes técnicas que, de acordo com a situação, permitem ao sistema detectar, mascarar, tolerar ou recuperar-se dessas falhas (COULOURIS; DOLLIMORE; KINDBERG, 2007).

Transparência: de acordo com (COULOURIS; DOLLIMORE; KINDBERG, 2007), "a transparência de um sistema é definida como sendo a ocultação, para um usuário final ou programador de aplicativos, da separação dos componentes em um sistema distribuído de modo que o sistema seja percebido como um todo, em vez de uma coleção de componentes independentes". Tal transparência permite que um programador de aplicativos, por exemplo, detenha sua atenção sobre o projeto de seu aplicativo em particular, sem que seja necessário levar em consideração a distribuição dos componentes de um sistema.

2.3 Tipos de Sistemas Distribuídos

Quanto ao objetivo a que se destinam, existem diferentes tipos de sistemas distribuídos, segundo (TANENBAUM; STEEN, 2007), são eles:

Sistemas de Computação Distribuídos: seu objetivo é realizar o processamento de tarefas de computação de alto desempenho. Esses sistemas podem ser organizados na forma de *cluster* (BUYYA, 1999) (o sistema é composto por um conjunto de computadores semelhantes e estão conectados por uma rede local de alta velocidade) ou em forma de *grids* (FOSTER; KESSELMAN; TUECK, 2001; FOSTER; , EDS) computacionais (o sistema é composto por computadores heterogêneos em relação a *hardware*, *software* e tecnologia de rede).

Sistemas de Informação Distribuídos: são sistemas destinados a permitir o compartilhamento de informações entre processos e a comunicação entre eles, como por exemplo sistemas de bancos de dados, chamadas de procedimento remoto (*Remote Procedure Calls - RPC*) e invocação de métodos remotos (*Remote Method Invocations - RMI*). Um importante tipo de sistema de informação distribuída é o sistema de arquivos distribuído. Por ser o objetivo deste trabalho, esse tipo de sistema distribuído será melhor descrito no Capítulo 3.

Sistemas Distribuídos Pervasivos: sistemas onde os equipamentos são caracterizados pelo pequeno tamanho, alimentação por bateria, mobilidade e por

possuírem uma única conexão sem fio. Tais características não se aplicam a todos os dispositivos, e nem devem ser consideradas restritivas. Em sistemas pervasivos, os dispositivos estão espalhados no ambiente e, devido a isso, sistemas desse tipo são inerentemente distribuídos.

2.4 Considerações Finais

Nesse capítulo descreveu-se os fatores que motivaram o surgimento de sistemas distribuídos e algumas de suas características. Também foram citadas as suas principais vantagens sobre os sistemas centralizados e alguns desafios atrelados ao desenvolvimento de sistemas utilizando esse paradigma. Além disso, foram descritos os tipos de sistemas distribuídos de acordo com o seu objetivo. Para maiores informações sobre esse assunto, sugere-se (TANENBAUM; STEEN, 2007) e (COULOURIS; DOLLIMORE; KINDBERG, 2007).

No próximo capítulo, será feita uma descrição mais detalhada sobre os sistemas de informação distribuídos, mais especificamente sobre os sistemas de arquivos distribuídos, que caracteriza o tipo de sistema sobre o qual o presente trabalho está sendo desenvolvido.

3 SISTEMAS DE ARQUIVOS DISTRIBUÍDOS

Inicialmente os sistemas de arquivos foram projetados para ambientes centralizados, porém, o desenvolvimento desses sistemas, juntamente com as características de desempenho e confiabilidade das redes locais, permitiram a criação de sistemas de arquivos distribuídos (*Distributed File System - DFS*). Esses, permitem o compartilhamento de informações através de arquivos e recursos de *hardware* com armazenamento persistente em toda uma intranet (COULOURIS; DOLLIMORE; KINDBERG, 2007). Segundo (TANENBAUM; STEEN, 2007), o compartilhamento de dados é fundamental para sistemas distribuídos, em função disso, os sistemas de arquivos distribuídos são a base para aplicações distribuídas, pois "permitem que vários processos compartilhem dados por longos períodos, de modo seguro e confiável".

De acordo com (SILBERSCHATZ; GALVIN; GAGNE, 2000), um DFS pode ser implementado como uma parte de um sistema operacional distribuído ou como uma camada de *software*, que pode ser usada para gerenciar a comunicação entre sistemas operacionais e sistemas de arquivos convencionais. Este último tipo representa o objetivo deste trabalho, ou seja, a construção de uma camada de *software* que fornecerá um serviço de armazenamento distribuído para outras aplicações, utilizando para isso o sistema de arquivos nativo do sistema operacional.

3.1 Questões de Projeto de Sistemas de Arquivos Distribuídos

Os requisitos de um sistema de arquivos definem como este deve estar organizado, ou seja, sua arquitetura e os módulos que a compõem (TANENBAUM; STEEN, 2007; SILBERSCHATZ; GALVIN; GAGNE, 2000). Em função disso, as principais questões de projeto de sistemas de arquivos distribuídos serão descritas a seguir, para que no Capítulo 4 sejam usadas como base para a proposta de um novo sistema de arquivos distribuído.

3.1.1 Arquitetura

Com relação a arquitetura, o modelo mais tradicional para o desenvolvimento de sistemas de arquivos distribuídos é o modelo **cliente-servidor**. Segundo (SILBERSCHATZ; GALVIN; GAGNE, 2000), este modelo é composto por três componentes: o serviço, que é um *software* que executa em uma ou mais máquinas e fornece algum tipo de função para clientes desconhecidos; o servidor, que é um *software* de serviço que executa em uma máquina específica; e o cliente, que é um processo que pode invocar um serviço usando um conjunto de operações que forma sua interface cliente.

Segundo essa terminologia, "um sistema de arquivos fornece serviços de arquivos a clientes". Para o servidor, o principal componente de hardware a ser controlado é o conjunto de dispositivos de armazenamento, geralmente, discos magnéticos, que serão utilizados para o armazenamento dos dados. A interface do cliente é formada por um conjunto de operações de arquivo, como por exemplo criar, excluir ou ler um arquivo. Para o cliente, a multiplicidade e dispersão dos servidores de armazenamento devem ser transparentes.

Segundo (TANENBAUM; STEEN, 2007), este modelo de arquitetura pode implementar o acesso a arquivos remotos de duas maneiras. Uma delas, chamada **modelo de acesso remoto**, sugere que o arquivo deve ser armazenado e gerenciado no servidor. Ao cliente, é oferecida apenas uma interface que contém várias operações de arquivos, semelhante a de um sistema de arquivos local convencional. Dessa forma, o cliente possui acesso transparente ao sistema de arquivos, ficando alheio da sua localização e, principalmente, o arquivo permanece sempre no servidor. A segunda alternativa é o **modelo de carga/atualização**, onde o arquivo é transferido para o cliente, e após a sua utilização, ele é enviado novamente ao servidor. Esses dois modelos de acesso a arquivos estão representados na Figura 3.1.

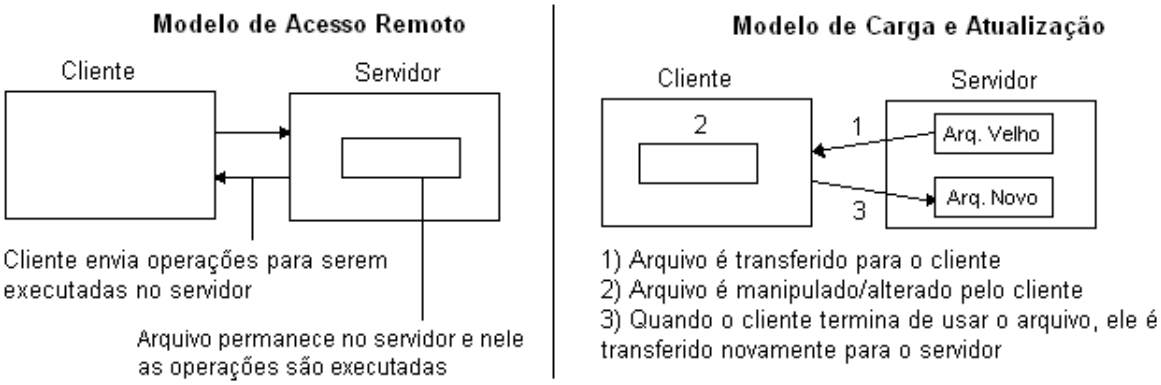

Figura 3.1: Modelos de acesso a arquivo remoto

De acordo com (TANENBAUM; STEEN, 2007), o modelo cliente/servidor pode ser estendido a um *cluster* de servidores, formando assim um **sistema de arquivos distribuído baseado em *cluster***. Sistemas desse tipo tiram proveito da

possibilidade de execução de aplicações paralelas (muito comum em *clusters*) para desmembrar o arquivo em partes e assim distribuí-lo em vários servidores de forma simultânea. Essa abordagem também é interessante nas operações de leitura de arquivos, pois permite que as diversas partes de um arquivo sejam lidas paralelamente. Apesar de suas vantagens, essa técnica não pode ser utilizada em todos os tipos de arquivos, sendo mais indicada para arquivos que tenham uma estrutura regular, como por exemplo uma matriz (densa). Um exemplo de sistema baseado em *cluster* é o GFS (TANENBAUM; STEEN, 2007), que é melhor descrito na Seção 3.2.4.

Outra arquitetura que pode ser utilizada para a criação de sistemas de arquivos distribuídos é a arquitetura simétrica, também conhecida como ponto-a-ponto (*peer-to-peer*). Sistemas baseados nessa arquitetura são compostos por um grande número de pontos (*hosts*) espalhados pela *Internet*, onde não há distinção de servidores e clientes, pois todos os pontos executam as mesmas tarefas. Em sistemas desse tipo, todos os computadores atuam como servidores e clientes simultaneamente, podendo fornecer e consumir serviços de todos os outros computadores. Uma consequência da utilização desse modelo de arquitetura é que não há nenhum sistema administrativo centralizado (TANENBAUM; STEEN, 2007).

3.1.2 Estado de processo

Segundo (TANENBAUM; STEEN, 2007), o aspecto mais importante a ser considerado sobre os processos de um sistema distribuído é se eles devem manter estado ou não. Um processo com estado mantém uma conexão com os clientes que atende, e também mantém informações sobre tais clientes. Já um processo que não mantém estado não armazena informações sobre os clientes que atende, e não mantém nenhuma espécie de conexão com tais clientes.

Processos sem estado são mais simples de serem implementados, visto que não exigem nenhuma fase de recuperação de estado, caso ocorra alguma falha com o servidor. Em contrapartida, por não armazenarem informações sobre seus clientes e não manterem uma conexão ativa, tornam-se mais limitados, pois dificultam a implementação de processos como autenticação e consistência da *cache*. Em função disso, manter um sistema de arquivos distribuído totalmente sem estado pode ser muito difícil ou até mesmo inviável. A evolução natural de um sistema de arquivos distribuído supõe a utilização de processos com estado. Embora apresentem uma complexidade maior de implementação, esses processos fornecem maior flexibilidade de implementação e facilitam o desenvolvimento de alguns processos, como por exemplo, manter a consistência da *cache*.

3.1.3 Comunicação

A comunicação é um aspecto importante a ser levado em consideração no projeto de um sistema distribuído, e, consequentemente em um DFS. Em função da distribuição dos computadores e da existência de diferentes processos, a comunicação entre esses computadores e processos é um ponto crucial na definição da arquitetura. É através da comunicação que os recursos do sistema distribuído irão interagir para a concretização de seu objetivo.

Para a implementação da comunicação pode-se adotar diversas técnicas e estratégias. Em (SILBERSCHATZ; GALVIN; GAGNE, 2000) tem-se uma descrição resumida das principais técnicas de comunicação entre processos, as quais serão listadas a seguir. Para uma referência completa de técnicas para comunicação de processos sugere-se (TANENBAUM; STEEN, 2007).

Uma técnica de **baixo nível** amplamente utilizada para realizar a comunicação de processos são os **soquetes** (QUINN; SHUTE, 1996). Esses podem ser definidos como as extremidades de um canal de comunicação. Um par de processos, remotos ou não, utilizam esse canal de comunicação para realizar as trocas de mensagens entre si. Cada extremidade é identificada por um endereço IP concatenado de um número de porta. Essa técnica de comunicação baseia-se no modelo cliente/servidor, onde o servidor aguarda por conexões de clientes em um número de porta conhecido por ambas as partes. Conhecendo o endereço IP do servidor e o número da porta, um cliente pode se conectar e iniciar a troca de mensagens com o servidor (SILBERSCHATZ; GALVIN; GAGNE, 2000).

Duas técnicas de comunicação entre processos de **alto nível** muito utilizadas são **RPC** (*Remote Procedure Call*) (SRINIVASAN et al., 1995) e **RMI** (*Remote Method Invocation*) (FARLEY, 1998). O RPC consiste em permitir que um processo execute um procedimento ou função de um outro processo, sendo que este pode estar sendo executado na mesma máquina ou em qualquer outra máquina conectada à mesma rede. Já o RMI permite que um processo invoque um método de um objeto remoto. Essas duas técnicas são semelhantes, porém apresentam algumas diferenças significativas entre si. Uma delas é que o RPC suporta apenas programação procedural, enquanto que RMI, por basear-se em objetos, permite a invocação de métodos em objetos remotos. Outra diferença é que em RPC os parâmetros são estruturas de dados comuns, ao passo que em RMI os parâmetros podem ser objetos (SILBERSCHATZ; GALVIN; GAGNE, 2000).

Internamente, RPC e RMI fazem uso de soquetes para implementar a troca de mensagem entre os processos. Porém, a utilização dessas técnicas tem vantagem sobre o uso de soquetes diretamente, pois abstraem o gerenciamento do canal de comunicação, permitindo que a implementação da aplicação se detenha em seu objetivo principal. Apesar disso, há casos em que a utilização de soquetes de forma

direta é mais indicada, principalmente quando se deseja um maior desempenho. Em função de sua simplicidade, os soquetes permitem uma comunicação mais rápida, pois não há um processamento adicional para gerenciar o canal de comunicação, empacotar e desempacotar as mensagens (SILBERSCHATZ; GALVIN; GAGNE, 2000).

3.1.4 Nomeação

De acordo com (SILBERSCHATZ; GALVIN; GAGNE, 2000), a nomeação é um "mapeamento entre objetos lógicos e físicos". Em um sistema de arquivos local, cada arquivo (objeto lógico) possui um nome único, que é composto do caminho de diretórios, nome do arquivo e extensão. Esse conjunto de informações permite que cada arquivo seja identificado de forma única no sistema. Nas camadas de nível mais baixo do sistema de arquivos, esse nome de alto nível é traduzido na localização física do arquivo (objeto físico), ou seja, no endereço físico do arquivo no disco rígido.

Em um sistema de arquivos distribuídos, o conceito de nomeação é o mesmo, porém adiciona-se mais um nível de abstração, que é o local da rede onde o arquivo está armazenado. De acordo com o nível de abstração, pode-se ter o conceito de replicação, isto é, o mesmo arquivo está localizado em mais de um local da rede (está replicado). Essa redundância permite que, mesmo que um servidor esteja indisponível, seja possível acessar um arquivo que esteja nele armazenado, pois existem outras cópias do mesmo arquivo em outros servidores (SILBERSCHATZ; GALVIN; GAGNE, 2000).

Em sistemas de arquivos, dois conceitos importantes relacionados a nomeação devem ser destacados. O primeiro deles é a **transparência de posição**, isto é, o nome do arquivo não fornece qualquer indício de sua localização. E o segundo é a **independência de posição**, ou seja, o nome do arquivo não precisa ser alterado se a sua localização física for alterada. Esses dois conceitos podem ser observados em sistemas de arquivos locais, como por exemplo: supondo a existência de um arquivo chamado "c:\dados.txt". O nome do arquivo não indica qualquer informação sobre sua localização física, isto é, cilindro, trilha ou setor do disco rígido; e sua localização poderia ser alterada (através de um desfragmentador de disco por exemplo) sem que fosse necessário alterar seu nome (SILBERSCHATZ; GALVIN; GAGNE, 2000).

Esses dois conceitos também são desejáveis em sistemas de arquivos distribuídos, e sua concretização depende do esquema de nomeação que é adotado. No sistema Ibis (TICHY; RUAN, 1984), por exemplo, o nome de um arquivo é composto pelo nome do *host* que está armazenado, concatenado com o nome do arquivo local. Essa abordagem não permite transparência nem independência de posição (SILBERSCHATZ; GALVIN; GAGNE, 2000). Já no sistema NFS (*Network File System*), a abordagem é completamente diferente. Nele, um cliente pode montar subdiretórios

de um servidor remoto em seu próprio sistema de arquivos local, dessa forma, os usuários do sistema não compartilham um espaço comum de nomes, pois o nome de um arquivo depende de como cada cliente organiza seu próprio espaço de nomes local (TANENBAUM; STEEN, 2007).

Para evitar esse problema, pode-se optar por uma outra abordagem: a utilização de um espaço global de nomes que cobre todos os arquivos do sistema, ou seja, o servidor central do sistema, ou todos os servidores, possuem uma tabela ou alguma outra estrutura de armazenamento na qual todos os arquivos do sistema estão registrados, juntamente com sua localização. Essa abordagem cria uma nova visão dos arquivos para o usuário, permitindo que ele os visualize de forma unificada. Além disso, essa visão dos arquivos é a mesma para todos os usuários do sistema.

Neste caso, a estrutura do sistema de arquivos seria semelhante à de um sistema de arquivos local, onde todos os usuários têm a mesma visão da hierarquia dos arquivos (SILBERSCHATZ; GALVIN; GAGNE, 2000). Essa abordagem apresenta os dois benefícios citados, pois não há nenhuma relação entre o nome lógico do arquivo e sua localização física; e também permite que os arquivos sejam movidos para outro lugar sem que seja necessário alterar o seu nome (TANENBAUM; STEEN, 2007). Um exemplo de sistema que implementa esse esquema de nomeação é o DFS (*Distributed File System*) da Microsoft, que é descrito em mais detalhes na Seção 3.2.3. Em função de seus benefícios, e de acordo com os objetivos deste trabalho, essa será a abordagem adotada para a implementação do esquema de nomeação neste trabalho, e será melhor descrito na Seção 3.2.3

3.1.5 Sincronização de Processos

Em um sistema de arquivos (que não seja somente leitura), pode-se executar pelo menos duas operações básicas sobre os arquivos: leitura e escrita. O processo de sincronização consiste em gerenciar essas operações de leitura e escrita de forma a impedir possíveis problemas de inconsistência que possam ser ocasionados pelo acesso concorrente de diferentes usuários sobre o mesmo arquivo, ou até mesmo do mesmo usuário sobre o mesmo arquivo (TANENBAUM; STEEN, 2007).

Para solucionar esse problema, torna-se necessário definir de forma clara a semântica das operações de leitura e escrita que são realizadas sobre um arquivo durante um acesso concorrente. Diferentes soluções foram propostas, e de acordo com (TANENBAUM; STEEN, 2007), as mais importantes são as que estão descritas a seguir.

A **semântica Unix** sugere uma ordenação de tempo absoluto sobre as operações de leitura e escrita, de modo que sempre o valor mais atual é retornado. Seguindo essa abordagem, caso o sistema receba uma operação de leitura imediatamente após uma operação de escrita, será retornado o valor mais atualizado, obtido após a exe-

cução da operação de escrita. Em sistemas de processador único, essa abordagem é eficiente, porém em sistemas distribuídos, seria necessário que todas as operações de leitura e escrita fossem direcionadas a um único servidor, o que degradaria o desempenho do sistema. Além disso, atrasos de rede podem fazer com que um comando de leitura, que foi executado após um comando de escrita, seja entregue antes ao servidor, tendo como resultado da leitura o valor antigo.

Para solucionar o problema da abordagem anterior, pode-se utilizar o esquema de sessões. Essa abordagem é conhecida como **semântica de sessão**. Segundo essa abordagem, antes de realizar operações de leitura e escrita, o cliente deve executar uma operação de abertura do arquivo, indicando assim ao servidor que pretende trabalhar com o arquivo. Após isso, o cliente pode enviar comandos de leitura e/ou escrita, e quando concluir, deve enviar um comando de fechamento do arquivo, sinalizando assim ao servidor que já terminou de utilizar o arquivo. Nesse caso, as alterações do arquivo só serão efetivadas após a operação de fechamento do arquivo. Caso outro cliente queira ler o arquivo durante um processo de atualização, receberá os dados antigos, sem as alterações enviadas pelo outro cliente. Essa solução é incapaz de tratar de forma adequada uma situação em que dois clientes desejam alterar o mesmo arquivo simultaneamente. Inevitavelmente, apenas um dos clientes terá suas alterações efetivadas.

Outra abordagem possível, conhecida como **arquivos imutáveis**, sugere que existam apenas duas operações possíveis sobre arquivos: leitura e criação. Segundo essa perspectiva, um arquivo nunca podem ser parcialmente alterado, apenas recriado. Sendo assim, embora seja impossível atualizar um arquivo, ainda é possível substituí-lo atomicamente por outro. Da mesma forma que na abordagem anterior, caso dois usuários tentem recriar simultaneamente o mesmo arquivo, apenas um deles obterá sucesso.

A quarta abordagem possível é a utilização de **transações** (RAMEZ; NAVATHE, 2005). De acordo com essa abordagem, antes de executar operações sobre um arquivo, o processo deverá executar uma primitiva que indica o início de uma transação, após isso, deverá enviar as operações que deseja executar, e concluir com a execução de uma primitiva que indica o término da transação. A utilização dessa semântica garante atomicidade, consistência, isolamento e durabilidade, no entanto, da mesma forma que as anteriores, não garante que dois usuários poderão atualizar o mesmo arquivo, ou a mesma região do arquivo, ao mesmo tempo.

3.1.6 Cache

De acordo com (SILBERSCHATZ; GALVIN; GAGNE, 2000), pode-se definir *cache* como a técnica de manter os dados mais utilizados por um processo em um meio de acesso mais rápido do que o padrão do sistema. A utilização dessa técnica

tem como principal objetivo garantir um melhor desempenho do sistema. Em sistemas de arquivos convencionais, a *cache* é mantida em memória, e é utilizada para minimizar as operações de E/S de disco. Já em um sistema de arquivos distribuído, a *cache* é mantida geralmente no disco local, com o objetivo de minimizar o tráfego dos arquivos pela rede.

O espaço destinado à *cache* é limitado e varia de acordo com o sistema, podendo geralmente ser configurado. A escolha de quais arquivos devem ser armazenados na *cache* é uma decisão de projeto, sendo que algumas das alternativas podem ser: manter os arquivos mais frequentemente acessados, ou então, os arquivos mais recentemente acessados. Devido ao tamanho limitado da *cache*, independente do critério que seja escolhido, em algum momento, arquivos terão que ser descartados para dar lugar a outros.

A técnica de utilização de *cache* é amplamente utilizada em sistemas de computação, e proporciona uma melhora significativa no desempenho do sistema. Porém apresenta alguns desafios a serem tratados, como por exemplo a garantia de consistência entre os dados que estão na *cache* e os dados que estão no servidor. Para tratar esse problema, faz-se necessário definir a política de atualização da *cache*. Tal política tem efeito crucial no desempenho e confiabilidade do sistema.

De acordo com (SILBERSCHATZ; GALVIN; GAGNE, 2000), diferentes técnicas podem ser utilizadas, dependendo do tipo e objetivo do sistema. A mais simples delas seria a de **gravação simultânea**, onde as alterações são imediatamente propagadas ao servidor tão logo tenham sido atualizadas na *cache*. Essa abordagem, apesar de garantir a confiabilidade dos dados, pode prejudicar o desempenho do sistema, pois torna-se necessário que o cliente aguarde até que todas as alterações sejam enviadas ao servidor.

Uma outra política é a de **gravação adiada**, onde os dados são gravados na *cache* e posteriormente enviados ao servidor. Essa abordagem possui diversas variações, e a principal diferença entre elas é a definição de quando os dados alterados devem ser transferidos para o servidor. Algumas dessas variações são: varrer a *cache* periodicamente em busca de arquivos alterados; enviar os arquivos alterados quando estes estiverem sendo retirados da *cache*; ou ainda, quando o cliente fechar o arquivo.

3.1.7 Replicação de Dados

Em alguns sistemas de arquivos distribuídos, onde a disponibilidade do sistema dos arquivos é a prioridade, pode-se implementar ainda um mecanismo de **replicação**. Esse mecanismo sugere que um mesmo arquivo, ou partes dele, estejam gravadas em mais de um local da rede. Isto significa que o arquivo será copiado para mais de um servidor do sistema, ou seja, será replicado. Essa técnica garante

uma maior disponibilidade tanto do sistema, como dos arquivos, pois caso um servidor esteja indisponível por algum problema, o arquivo ainda poderá ser acessado, bastando para isso consultar outro servidor que tenha uma réplica do arquivo. Além disso, o uso de replicação permite que clientes trabalhem sobre diferentes réplicas do arquivo, aumentando assim o desempenho do sistema.

Apesar da vantagem da disponibilidade, o uso de replicação pode acarretar em alguns problemas. Quando uma operação de escrita ou atualização é realizada sobre um arquivo, todas as réplicas têm de ser atualizadas. Isso implica em um aumento do tráfego de rede, para propagar as alterações para os outros servidores. Acrescentando-se a isso a necessidade de sincronização de operações concorrentes, provoca-se um aumento de comunicação, e, consequentemente, queda no desempenho.

3.1.8 Tolerância a falhas e Segurança

Tem-se ainda duas questões de projeto igualmente importantes às demais citadas: tolerância a falhas e segurança. Apesar de sua importância em sistemas de arquivos distribuídos, esses dois temas serão abordados de forma superficial durante o desenvolvimento deste trabalho, pois não constituem o objetivo principal a ser atingido. Sugere-se que essas duas questões, juntamente com outros assuntos citados do decorrer deste trabalho, sirvam de motivação para o desenvolvimento de trabalhos futuros sobre esse projeto.

De acordo com (TANENBAUM; STEEN, 2007), pode-se descrever **tolerância a falhas** como a capacidade que o sistema tem de continuar funcionando, até certo ponto, mesmo na presença de falhas, ou seja, o sistema tolera as falhas e continua atendendo os usuários, dentro dos limites possíveis. Um sistema apresenta falhas quando não está funcionando de acordo com o objetivo que foi projetado, seja por um erro interno, ou por entidades terceiras que integram o sistema. Um forma de implementar tolerância a falhas em sistemas de arquivos distribuídos é através da utilização de replicação, cujo conceito já foi descrito anteriormente.

A respeito da **segurança**, pode-se afirmar que é um fator de grande importância em qualquer tipo de sistema de computação distribuída. Através de sua implementação, o sistema torna-se confiável (revelação de informações apenas às partes autorizadas) e íntegro (permite alterações apenas com autorização). Em sistemas de arquivos distribuídos, pode-se implementar segurança utilizando mecanismos de autenticação e controle de acesso. A autenticação irá garantir que apenas usuários que têm acesso poderão utilizar o sistema; já o controle de acesso irá garantir que os usuários do sistema poderão alterar apenas o conteúdo sobre o qual tiverem permissão. Outros mecanismos adicionais podem ser usados para complementar o esquema de segurança do sistema, um deles é a utilização de SSL (*Secure Soc-*

kets Layer), que é um canal de comunicação seguro (criptografado) sobre conexões TCP/IP. Para maiores informações sobre tolerância a falhas e segurança, sugere-se (TANENBAUM; STEEN, 2007).

3.2 Estudo de ferramentas já existentes

Nesta seção descreve-se os sistemas de arquivos distribuídos já existentes, e que têm sido mais utilizados em ambientes empresariais e estudados no meio acadêmico. Serão descritas as principais decisões de projeto de cada um desses sistemas, suas semelhanças e diferenças com este projeto, e porque nenhum deles se mostrou totalmente adequado para a utilização no Projeto IndexVideo.

3.2.1 NFS - Network File System

O NFS (*Network File System*) foi desenvolvido pela Sun Microsystems e é utilizado em grande parte por sistemas baseados em Unix e Linux, apesar de ser independente de plataforma. Para a análise desse sistema, optou-se pelo NFSv3 (CALLAGHAN et al., 1995), que é a terceira versão do NFS e é amplamente utilizada. Há também uma versão mais recente, o NFSv4 (SHEPLER et al., 2003), que apresenta algumas diferenças de projeto se comparado com o NFSv3 (TANENBAUM; STEEN, 2007). As diferenças mais relevantes também serão aqui destacadas.

Uma questão de projeto importante do NFS, e que precisa ser descrita antes de iniciar-se a análise desse sistema, é que esse faz uso de um sistema de arquivos virtual, chamado VFS (*Virtual File System*), cujo principal objetivo, segundo (COULOURIS; DOLLIMORE; KINDBERG, 2007), é permitir que usuários acessem de forma transparente diferentes tipos de sistemas de arquivos, sejam estes locais ou remotos. O VFS serve como uma camada de abstração, e permite a integração de sistemas de arquivos distintos. As requisições para leitura ou escrita de um arquivo são passadas para o VFS, e este repassa cada requisição para o módulo apropriado (o sistema de arquivos UNIX, o módulo de cliente NFS ou o módulo de serviço de outro sistema de arquivos). De acordo com (TANENBAUM; STEEN, 2007), o VFS já é um padrão estabelecido e implementado em praticamente todos os sistemas operacionais.

O NFS foi desenvolvido sob o modelo de **arquitetura cliente/servidor**, utilizando o conceito de **acesso remoto a arquivos**, ou seja, o cliente NFS manipula arquivos que estão localizados em servidores remotos. Para isso, ele implementa as operações do sistema de arquivos como chamadas **RPC** para o servidor, o qual é responsável por manipular as requisições que chegam do cliente e transformar em operações comuns do VFS. Ao chegar à camada VFS, essas operações serão convertidas nas operações específicas de cada sistema de arquivos. O principal objetivo

da utilização desse sistema de arquivos virtual é permitir que o cliente trate arquivos locais e remotos da mesma forma, além de permitir a integração de diferentes sistemas de arquivos (TANENBAUM; STEEN, 2007). A Figura 3.2, baseada no modelo descrito por (COULOURIS; DOLLIMORE; KINDBERG, 2007), ilustra o funcionamento do NFS e a integração do VFS com outros sistemas de arquivos.

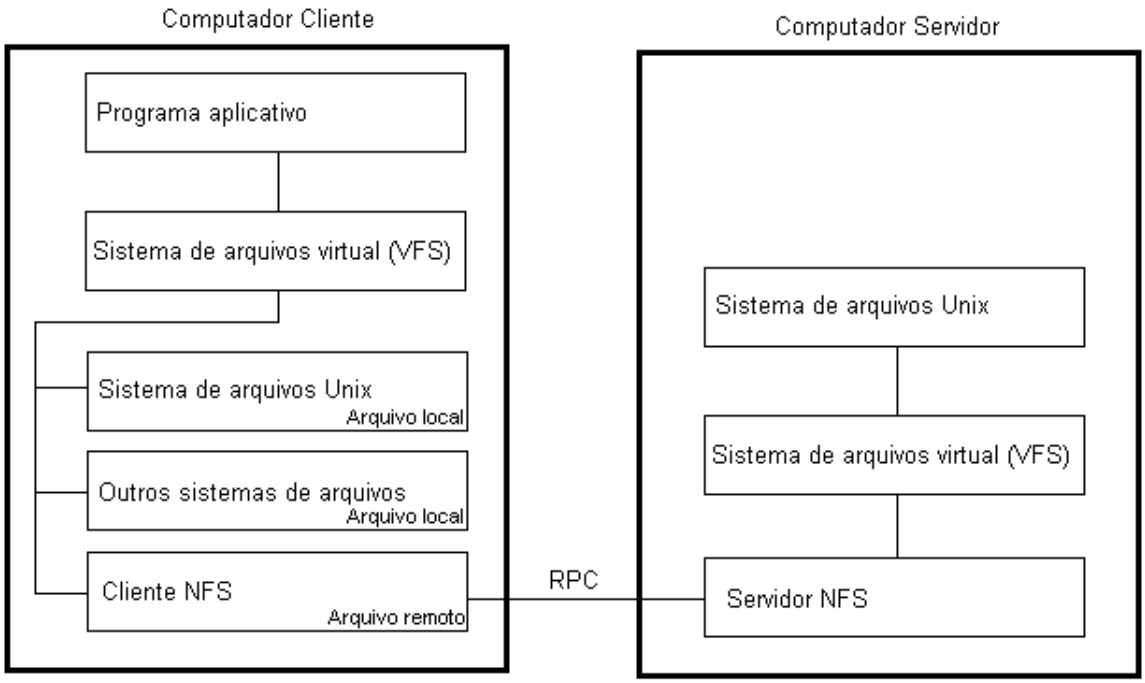

Figura 3.2: Ilustração da arquitetura do NFS e funcionamento do VFS

Com relação a **processos**, no NFSv3, a abordagem utilizada é a sem estados. Essa abordagem permite uma implementação simples e não exige processos de recuperação de estado, ou seja, caso o servidor venha a apresentar alguma falha, basta continuar do ponto em que havia parado. Porém, no NFSv4, essa abordagem foi abandonada e substituída pela utilização de processos com estado. Um dos fatores que motivou essa mudança de estratégia foi a necessidade de utilizar o NFS em redes de longa distância, onde a implementação da *cache* é fundamental para um bom desempenho. Conforme visto anteriormente, o uso da *cache* requer a implementação de mecanismos de consistência, que são mais facilmente implementados quando se mantém informações sobre os clientes (processos com estado) (TANENBAUM; STEEN, 2007).

Tratando-se de **nomeação**, o NFS permite que clientes montem partes de sistemas de arquivos remotos em seu próprio sistema de arquivos local. Para isso, o servidor NFS disponibiliza ao cliente o diretório a ser exportado e todas as suas entradas, permitindo que o cliente NFS monte esse diretório localmente. Um cliente NFS pode montar diferentes diretórios provenientes de diversos servidores NFS, do

mesmo modo que um servidor NFS pode exportar o mesmo diretório para diversos clientes. Dessa forma, cada cliente organiza seu sistema de arquivos local da forma que desejar. A principal implicação disso é que os clientes não compartilham um espaço de nomes comum, ou seja, cada cliente pode apresentar uma hierarquia de arquivos diferente (TANENBAUM; STEEN, 2007).

Para a **sincronização** de acessos concorrentes ao mesmo arquivo, o NFSv4 utiliza um esquema de travamento de arquivo. Esse mecanismo permite que diferentes clientes tenha acesso de leitura ao mesmo arquivo, porém, quando um cliente deseja alterar um arquivo, ou parte dele, deve solicitar ao servidor NFS o travamento da faixa de *bytes* que deseja atualizar. Nesse momento, os demais clientes podem apenas efetuar operações de leitura sobre a faixa de *bytes* que está sendo atualizada. Nenhum outro cliente poderá atualizar o arquivo, nem mesmo solicitar o travamento de uma faixa de *bytes* que se sobreponha à faixa já travada. Após a atualização do arquivo, o cliente que solicitou a trava deve também solicitar a sua liberação (TANENBAUM; STEEN, 2007).

A utilização de ***cache*** pelo cliente no NFSv3 não foi especificada no protocolo. Em função disso, diversas políticas de *cache* foram implementadas, embora nenhuma delas tenha garantido consistência. No NFSv4, foram feitas algumas melhorias, porém, em essência, a consistência da *cache* ainda depende da implementação. Em função disso, não será detalhado o funcionamento da política da *cache* para esse sistema, mas de uma forma geral, o NFS permite que clientes mantenham uma *cache* em memória dos arquivos recentemente lidos, que pode ser estendida para uma *cache* em disco (TANENBAUM; STEEN, 2007). Para verificar se os dados em *cache* são válidos, o NFS utiliza o mecanismo de *timestamp* (COULOURIS; DOLLIMORE; KINDBERG, 2007).

De acordo com (COULOURIS; DOLLIMORE; KINDBERG, 2007), a **replicação** no NFS é utilizada apenas em meios de armazenamento de somente leitura, não havendo suporte nativo para a replicação de arquivos com atualização. No entanto, pode-se fazer uso do NIS (*Network Information Service*) (STERN; EISLER; LABIAGA, 2001), que fornece um repositório compartilhado de informações de sistema, que é utilizado para gerenciar a distribuição de atualizações e acesso a arquivos replicados com base em um modelo de replicação mestre-escravo simples, com possibilidade de replicação parcial ou total do banco de dados em cada site.

Apesar do NFSv4 manter processos com estado, as informações mantidas sobre os clientes são mínimas (TANENBAUM; STEEN, 2007), e para efeitos de **tratamento de falhas**, ele se assemelha ao NFSv3. Segundo (COULOURIS; DOLLIMORE; KINDBERG, 2007), mesmo que um servidor apresente uma falha, e seja necessário aguardar até que o serviço seja restabelecido, a característica sem estado do protocolo NFS permite que os clientes continuem do ponto em que o serviço foi

interrompido, sem conhecimento da falha.

Como descrito anteriormente, o NFS baseia toda sua comunicação entre cliente e servidor através de RPC. Em função disso, a garantia de **segurança** em NFS se resume a estabelecer chamadas de RPC seguras. Isso é conseguido através do uso de autenticação e controle de autorização sobre os arquivos. Em sistemas NFS, a autenticação é realizada separadamente através do Kerberos (NEUMAN et al., 2005), que é um sistema de autenticação padrão desenvolvido pelo MIT. Já o controle de autorização sobre os arquivos é realizado pelo próprio servidor NFS (COULOURIS; DOLLIMORE; KINDBERG, 2007; TANENBAUM; STEEN, 2007). Para maiores informações sobre a garantia de RPC seguras, sugere-se (TANENBAUM; STEEN, 2007).

Considerando-se as características do NFS e as necessidades deste trabalho e do Projeto IndexVideo, conclui-se que a sua utilização não supriria a principal necessidade do projeto, que é abstrair o armazenamento e a localização dos arquivos. Caso o NFS fosse agregado ao Projeto IndexVideo, ainda assim, o cliente (no caso o desenvolvedor da ferramenta) teria que, indiretamente, decidir em qual servidor o arquivo deve ser armazenado, pois o NFS funciona simplesmente como um agrupador de arquivos, e não um distribuidor. Além disso, a inexistência de um espaço global de nomes, e a possibilidade de cada cliente montar uma estrutura de diretórios remotos diferente, dificultaria a localização de um arquivo.

3.2.2 AFS - Andrew File System

Antes de iniciar a descrição desse sistema de arquivos, torna-se interessante citar algumas características sobre o seu cenário de utilização. De acordo com (COULOURIS; DOLLIMORE; KINDBERG, 2007), este sistema foi projetado tendo como principal objetivo a escalabilidade. Além disso, todo o seu projeto foi baseado em algumas premissas sobre o ambiente onde seria utilizado, dentre as quais, pode-se destacar:

- A maior parte dos arquivos raramente é atualizado, e quando é atualizado, geralmente é pelo mesmo usuário. Isso significa que um arquivo pode ficar muito tempo na *cache*.
- A maioria dos arquivos é pequeno.
- O tamanho da *cache* pode ser grande o suficiente para comportar o conjunto de arquivos mais utilizado por um usuário.
- A maioria dos arquivos é lida e escrita por apenas um usuário.
- As operações de leitura são mais comuns que as de escrita.

- O acesso sequencial é muito mais frequente que o acesso aleatório.

- Arquivos referenciados recentemente têm grande probabilidade de serem referenciados novamente em um futuro próximo.

O AFS se assemelha em alguns aspectos com o NFS, dentre eles, pode-se destacar que ele também fornece acesso transparente a arquivos remotos compartilhados para programas UNIX que executam em estações de trabalho. O acesso aos arquivos se dá através das primitivas de arquivo UNIX normais. Além disso, é compatível com o NFS, pois o sistema de arquivos do servidor é baseado em NFS. Em função disso, os arquivos de um servidor AFS podem ser acessados de forma remota através do NFS (COULOURIS; DOLLIMORE; KINDBERG, 2007).

Da mesma forma que o NFS, o AFS é baseado em um modelo de **arquitetura cliente/servidor**, implementada com base em dois componentes de *software* que existem como processos UNIX em nível de usuário. O processo que executa em cada servidor é denominado *Vice*, e o cliente é chamado *Venus*. A comunicação entre cliente e servidor é feita utilizando **RPC**, e a **nomeação** funciona da mesma forma que o NFS, onde cada cliente monta seu próprio espaço de nomes, que pode conter arquivos locais e remotos (COULOURIS; DOLLIMORE; KINDBERG, 2007).

De acordo com (COULOURIS; DOLLIMORE; KINDBERG, 2007), com relação a **sincronização de processos**, o AFS deixa a cargo do cliente a implementação do controle de concorrência, caso esse seja realmente necessário. Caso contrário, o AFS se limita a efetivar a última operação de escrita realizada sobre o arquivo e descarta as anteriores, sem gerar nenhum erro.

Com relação a utilização de ***cache***, o AFS possui uma característica particular, que é a *cache* de arquivos inteiros (apenas na versão 2, a versão 3 faz uma segmentação do arquivo em arquivos menores de 64 *Kbytes*), permitindo que o conteúdo inteiro de diretórios e arquivos seja transferido para o cliente. Essa abordagem caracteriza um **modelo de carga/atualização** para o acesso dos arquivos remotos, e implica na utilização de grandes tamanhos de *cache*. Para manter a consistência da *cache*, é utilizado um mecanismo chamado "promessa de *callback*", cujo funcionamento básico consiste em notificar os clientes que possuem um arquivo em *cache* quando este for atualizado por algum cliente no servidor.

A **replicação** é permitida apenas no formato somente leitura, podendo haver diversas cópias, porém apenas uma única réplica que permite leitura e escrita, e para a qual todas as operações de atualizações são direcionadas. Isso proporciona **tolerância a falhas**, mas apenas em arquivos somente leitura (COULOURIS; DOLLIMORE; KINDBERG, 2007). A **segurança** é garantida através de mecanismos de autenticação e listas de acesso, que são melhor descritos em (HOWARD, 1988).

Conforme descrito anteriormente, o AFS possui diversas semelhanças com o NSF, dentre elas o esquema de nomeação e armazenamento de arquivos. Em função dessas características em comum, a utilização do AFS pelo IndexVideo seria dificultada devido aos mesmos motivos que o NFS não é a solução mais adequada. Além disso, esses dois sistemas de arquivos distribuídos não fornecem uma solução que facilite e abstraia o armazenamento de arquivos de forma distribuída como este trabalho se propõe a fazer.

3.2.3 DFS - Distributed File System

Outro sistema de arquivos distribuído existente é o DFS (*Distributed File System*) da Microsoft, que é composto de duas tecnologias: *DFS Namespaces e DFS Replication*. A primeira delas possibilita que pastas compartilhadas em diferentes servidores sejam agrupadas e apresentadas aos usuários como uma árvore virtual de diretórios, chamada de *namespace*. (Microsoft Corporation, 2003, 2005). A Figura 3.3 ilustra a utilização do DFS Namespace.

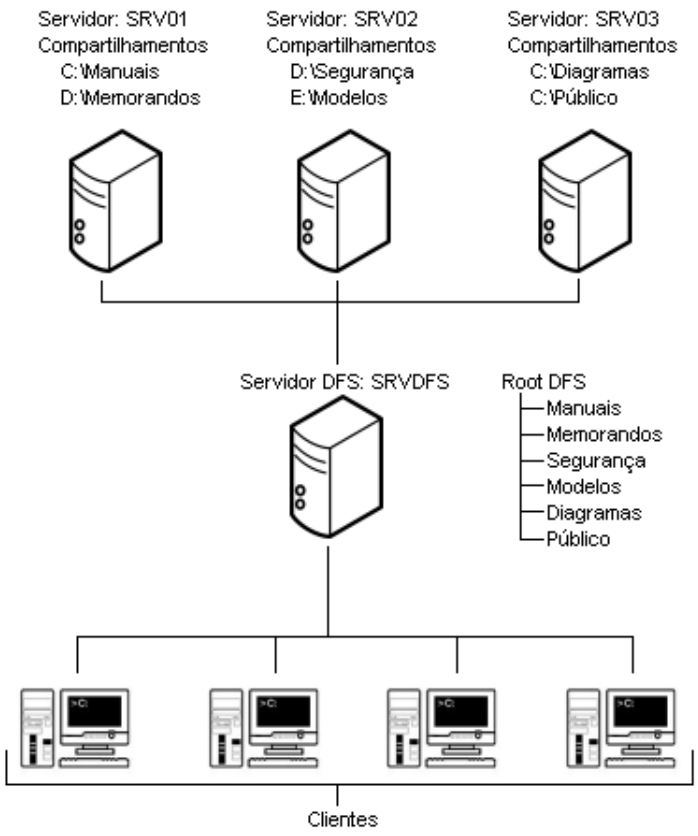

Figura 3.3: Exemplo de utilização de um servidor DFS

A segunda tecnologia, chamada de *DFS Replication*, é um mecanismo de replicação baseado em estado que permite o agendamento de replicação e controle de largura de banda. Esse controle é conseguido através de um protocolo de compressão

que detecta as alterações realizadas nos arquivos e replica apenas as modificações. Essas duas tecnologias podem ser utilizadas em conjunto para a implementação de soluções nos seguintes cenários (Microsoft Corporation, 2005):

Coleta de Dados: através do mecanismo de replicação, coletar os dados de diferentes filiais de uma empresa e armazená-los na matriz ou em um *datacenter*.

Distribuição de Dados: publicar documentos e programas para usuários de uma organização, fornecendo mecanismos para que um usuário acesse a réplica do servidor que tem o menor custo de conexão.

Compartilhar arquivos entre filiais: permitir que usuários de uma filial acessem arquivos localizados em outras filiais ou até mesmo na matriz de uma empresa.

Torna-se importante salientar que nenhum dos possíveis cenários para a utilização do DFS se assemelha com a necessidade do Projeto IndexVideo, que consiste em um sistema de armazenamento e recuperação de arquivos que seja transparente ao usuário. Embora o DFS seja transparente para o **acesso** aos arquivos, o **armazenamento**, da mesma forma que o NSF e o AFS, só será feito de forma distribuída se o usuário distribuir seus arquivos entre os diretórios mapeados. Caso contrário, todos os arquivo serão armazenados em um único servidor.

3.2.4 GFS - Google File System

O GFS (*Google File System*) também é um sistema de arquivos distribuído baseado no modelo de **arquitetura cliente/servidor**. Sua criação foi motivada pelas características específicas de armazenamento que os projetos desenvolvidos no Google apresentam. Os engenheiros dessa empresa, não encontraram nenhuma ferramenta que atendesse suas necessidades e optaram por construir um sistema de arquivos distribuído próprio que atendesse de forma eficiente suas necessidades de armazenamento. Atualmente, o GFS é amplamente utilizado no Google como sua plataforma de armazenamento para geração e processamento de grandes conjuntos de dados utilizados por seus serviços e projetos de pesquisa e desenvolvimento (GHEMAWAT; GOBIOFF; LEUNG, 2003).

Segundo (GHEMAWAT; GOBIOFF; LEUNG, 2003), a arquitetura do GFS foi dirigida pelas observações das cargas de seus sistemas e suas necessidades de armazenamento, dentre as quais pode-se citar a manipulação de arquivos muito grandes, da ordem de mais de um *gigabyte*, e que são mais comumente alterados por operações de acréscimo de dados ao invés de operações de sobrescrita ou atualização. Uma vez criado, o arquivo é somente lido, e frequentemente, de forma sequencial.

O GFS também foi desenvolvido presumindo-se que o *hardware* utilizado para armazenar os dados é barato e que frequentemente apresenta falhas, ou seja, a falha é o normal, e não a exceção. Em função disso, o sistema deve continuamente monitorar, detectar, tolerar e recuperar-se de falhas. Para atingir esse objetivo, a utilização de réplicas foi incorporada ao projeto do sistema (GHEMAWAT; GOBIOFF; LEUNG, 2003).

Em função de seu objetivo específico, que difere deste trabalho, e de seu cenário de utilização não compartilhar das mesmas características do Projeto IndexVideo, o GFS não será profundamente detalhado em todos os seus aspectos neste trabalho, para isso, sugere-se (GHEMAWAT; GOBIOFF; LEUNG, 2003). Porém torna-se interessante descrever o sistema de nomeação desse sistema de arquivos, visto que este apresenta algumas semelhanças com o projeto do sistema que deseja-se desenvolver neste trabalho.

De acordo com (TANENBAUM; STEEN, 2007), um *cluster* GFS é constituído por um único servidor mestre e vários servidores de porção. O servidor mestre GFS mantém em essência um espaço de nomes de arquivos e um mapeamento desses nomes de arquivos para os servidores de porção. O cliente GFS, ao fazer uma requisição de um arquivo, consulta o servidor mestre, e este por sua vez, consulta sua tabela de nomes de arquivos e identifica qual o servidor de porção que armazena o arquivo, retornando essa informação para o cliente GFS. A partir desse momento, o cliente entra em contato diretamente com o servidor de porção para realizar a transferência do arquivo, não sendo mais necessária a comunicação com o mestre GFS. Além disso, periodicamente o mestre GFS contata seus servidores de porção e obtém a lista de arquivos que cada um armazena, atualizando assim seu espaço de nomes de arquivos. A Figura 3.4 ilustra de forma simplificada o funcionamento dessa arquitetura.

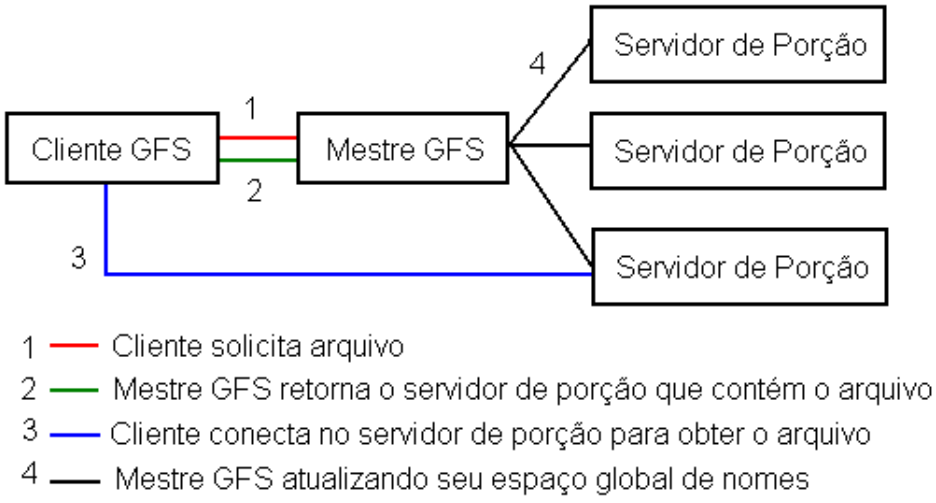

Figura 3.4: Funcionamento simplificado do GFS

Essa decisão de projeto permite que um único mestre GFS gerencie centenas de servidores de porção. Apesar de ser o centro do sistema, o mestre GFS não representa um gargalo, pois o processamento e a transferência dos arquivos são realizados pelos servidores de porção. Além disso, o espaço de nomes de arquivos mantido pelo mestre GFS é implementado como uma tabela simples de único nível, mantida em memória, o que garante um maior desempenho do sistema, pois reduz as operações de E/S do servidor (TANENBAUM; STEEN, 2007).

Torna-se importante salientar que o GFS é utilizado internamente por diversos serviços do Google, e que não está disponível para ser utilizado em projetos que não sejam dessa empresa. Porém tem-se uma outra ferramenta que implementa um sistema de arquivos distribuído que segue os mesmos padrões do GFS, o *Hadoop Distributed File System* (HDFS) da Apache (GHEMAWAT; GOBIOFF; LEUNG, 2003). No entanto, como este sistema de arquivos também não compartilha do mesmo cenário de utilização do Projeto IndexVideo, e é muito semelhante ao GFS, não será descrito neste trabalho. Para maiores informações sobre este sistema, sugere-se (FOUNDATION, 2009), e para mais detalhes sobre a arquitetura do HDFS, sugere-se (GHEMAWAT; GOBIOFF; LEUNG, 2003).

3.2.5 Amazon S3 - Amazon Simple Storage Service

A última ferramenta que foi avaliada neste trabalho foi o Amazon S3, que é um serviço de armazenamento de dados comercial disponibilizado pela Amazon. Ele permite que arquivos de 1 *byte* até 5 *gigabytes* sejam escritos, lidos e apagados através de uma interface de *web services*. Os arquivos ficam armazenados nos servidores da Amazon nos Estados Unidos ou na Europa, à escolha do cliente. O preço do serviço é calculado mensalmente com base na quantidade de arquivos que é transferida e armazenada durante o mês, variando de acordo com o local de armazenamento escolhido (AMAZON, 2009). Para o armazenamento de arquivos nos Estados Unidos por exemplo, o preço é de $ 0,15 por *gigabyte* por mês. Para uma referência completa dos valores do serviço, sugere-se (AMAZON, 2009);

Dentre as garantias oferecidas pelo serviço, pode-se citar: escalabilidade, tanto em termos de capacidade de armazenamento como em número de usuários e requisições; disponibilidade de 99.99% do serviço; tolerância a falhas e rapidez de resposta, para suportar aplicações que exigem alto desempenho. Além disso, o objetivo do serviço é ser simples, abstraindo do usuário toda a dificuldade por trás de um sistema de arquivos distribuído (AMAZON, 2009).

O Amazon S3 é sem dúvida uma excelente solução transparente para o armazenamento de dados através da *Internet*. Porém alguns fatores devem ser levados em consideração antes de sua utilização. Por se tratar de um serviço comercial, não há garantias sobre a continuidade do serviço, torna-se impossível personalizá-

lo para que atenda a necessidades específicas de um determinado projeto, não há controle sobre os dados, perde-se flexibilidade, etc. Além disso, destaca-se que, por estar na *Internet*, é mais lento do que se estivesse em uma rede local. Devido à impossibilidade de adaptações e possível perda de desempenho por estar na *Internet*, descartou-se a utilização do Amazon S3.

3.3 Considerações Finais

Neste capítulo investigou-se as principais decisões de projeto que devem ser tomadas durante a construção de um sistema de arquivos distribuído. Além disso, descreveu-se as principais abordagens que podem ser adotadas para cada uma dessas decisões de projeto. Após isso, realizou-se uma análise de alguns sistemas de arquivos distribuídos e serviços de armazenamento de dados disponíveis. A partir dessa análise observou-se que nenhuma dessas ferramentas de sistemas de arquivos distribuídos atende completamente às necessidades deste trabalho. Dessa forma, decidiu-se pelo desenvolvimento de um novo sistema de arquivos distribuído, o qual será descrito no próximo capítulo.

4 PROPOSTA DE UM SISTEMA DE ARQUIVOS DISTRIBUÍDO

No capítulo anterior foi feita uma análise das principais decisões de projeto de um sistema de arquivos distribuído, em seguida, algumas ferramentas existente foram avaliadas, e concluiu-se que tais ferramentas não apresentavam o mesmo objetivo ou não atendiam completamente às necessidades deste trabalho. Dadas essas condições, decidiu-se propor um novo sistema de arquivos distribuído que fosse capaz de atender as necessidades descritas no objetivo deste trabalho, e que possa ser utilizado pelo Projeto IndexVideo, no intuito de se fazer um estudo de caso concreto.

O presente capítulo estará focado em descrever mais detalhes a respeito do objetivo desse sistema, as premissas que serão assumidas sobre seu ambiente de utilização, exemplos de cenários de sua utilização e, principalmente, a descrição de uma proposta de arquitetura para a sua implementação. Para isso, todas as questões de projeto descritas no Capítulo 3 serão aqui abordadas, descrevendo as decisões que foram tomadas sobre cada uma delas.

4.1 Objetivo do sistema

Após a realização da análise dos sistemas de arquivos já existentes, pôde-se observar uma diferença substancial entre esses e o objetivo do trabalho que está sendo desenvolvido. Enquanto os primeiros têm seu foco voltado para o agrupamentos dos arquivos, o presente trabalho tem seu foco não apenas no agrupamento dos arquivos, mas principalmente na sua distribuição. A Figura 4.1 ilustra essa diferenciação dos objetivos.

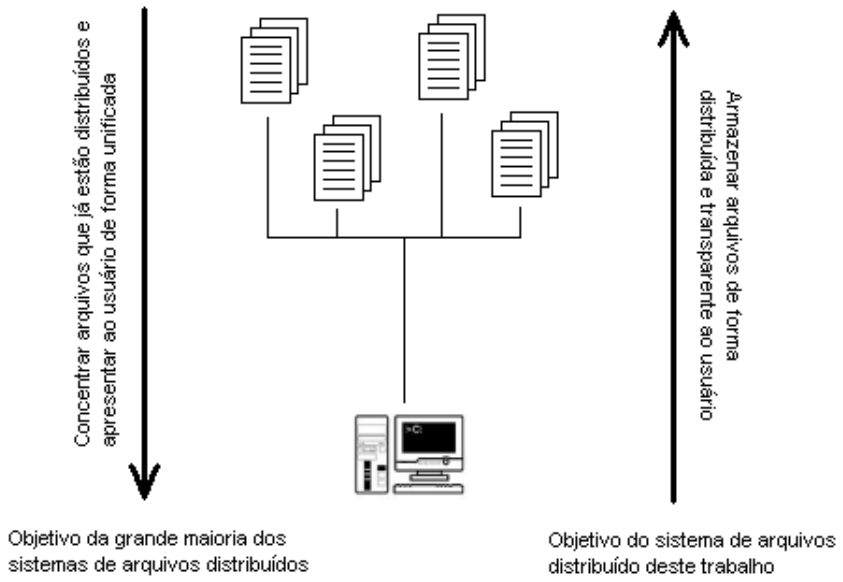

Figura 4.1: Diferença de objetivo entre os sistemas já existentes e o presente trabalho.

Além desse objetivo principal de armazenar arquivos de forma distribuída e transparente, o desenvolvimento deste trabalho também possui objetivos secundários no que diz respeito à forma como o objetivo principal será atingido. Um desses objetivos secundários é que o sistema deve ser escalável, ou seja, deve permitir que novos servidores sejam adicionados ao sistema a qualquer momento, seja para aumentar a capacidade de armazenamento ou para suportar um número maior de usuários. Além disso, o processo de inclusão de novos servidores deve ser simples e rápido.

Em função de seu objetivo principal, o balanceamento de carga dos servidores também constitui um objetivo secundário a ser atingido. O sistema deverá realizar a distribuição dos arquivos de forma a minimizar a sobrecarga de um servidor em específico. Além disso, o espaço de armazenamento que cada servidor dispõe também deve ser gerenciado, a fim de evitar que os arquivos fiquem concentrados em um pequeno grupo de servidores.

Outro objetivo secundário no desenvolvimento deste trabalho é permitir que o sistema possa armazenar arquivos de qualquer tipo e de diferentes tamanhos, não restringindo sua utilização a um conjunto limitado de tipos de arquivo. Especificamente com relação ao tamanho dos arquivos que o sistema irá armazenar, será dada prioridade ao suporte para arquivos de tamanho pequeno e médio.

Como já citado na introdução, a replicação, a tolerância a falhas e a segurança não constituem um objetivo deste trabalho nesse momento. A justificativa para essa decisão é a delimitação do problema e dos assuntos que serão abordados durante o desenvolvimento do sistema. No entanto, tais assuntos poderão ser retomados na ocasião do desenvolvimento de trabalhos futuros sobre este trabalho.

Como objetivo final, o sistema de arquivos distribuído deste trabalho deverá se apresentar ao usuário (desenvolvedor) como uma API, através da qual as operações sobre os arquivos serão realizadas. Essa API deverá oferecer ao usuário as principais operações de um sistema de arquivos convencional, criação, leitura, escrita e exclusão. A razão para escolha de uma API para o acesso ao sistema é permitir que esta seja inserida dentro de aplicações já existentes, ou novas aplicações, fornecendo um meio flexível e transparente para o armazenamento de grandes quantidades de arquivos.

4.2 Pressupostos sobre o ambiente

Após a definição dos objetivos deste trabalho, torna-se importante descrever alguns pressupostos que serão assumidos sobre o ambiente onde esse sistema será utilizado. A descrição desses pressupostos é importante porque auxilia a detalhar o ambiente onde o sistema será utilizado, fornecendo informações relevantes que serão utilizadas no momento da tomada de decisões sobre as questões de projeto.

O primeiro pressuposto a ser assumido é que tanto o sistema de arquivos distribuído como os clientes que farão uso dele estarão em uma rede local. A partir dessa premissa, a transferência de um arquivo do servidor para o cliente, ou o contrário, deve ser relativamente rápida.

Essa primeira premissa se baseia no segundo pressuposto, no qual assume-se que a grande maioria dos arquivos do sistema são pequenos ou médios. Para que se tenha uma base do tamanho médio dos arquivos que esse sistema irá armazenar, pode-se observar os estudos realizados por (GILL et al., 2007) e (CHENG; DALE; LIU, 2007). Dentre os diversos assuntos abordados nesses estudos, um deles é o tamanho médio dos arquivos do *website* YouTube, cujo objetivo é armazenar e exibir vídeos públicos aos seus usuários através da *Internet*. Torna-se interessante a avaliação dos resultados encontrados nesses estudos, visto que, assim como no Projeto IndexVideo, os arquivos gerenciados pelo YouTube também são vídeos.

Os resultados encontrados sobre a caracterização do tamanho dos arquivos foi semelhante nos dois estudos. Em (CHENG; DALE; LIU, 2007), o cálculo do tamanho médio dos arquivos do YouTube chegou à 8.4 MB, sendo que 98.8 % dos arquivos analisados possuíam um tamanho inferior à 30 MB. Esses valores têm uma grande relação com os dados descritos em (GILL et al., 2007), onde afirma-se que 90 % dos arquivos analisados possuíam um tamanho inferior à 21.9 MB.

O cenário de aplicação do presente trabalho será baseado nesses dois estudos realizados, assumindo que o tamanho médio dos arquivos esteja entre 8 MB e 10 MB, e que a grande maioria dos arquivos possua um tamanho inferior a 30 MB. Nada impede que o sistema seja utilizado para o armazenamento de arquivos com

tamanho superior a esse, porém não será o seu foco, e consequentemente, não haverá garantia de um bom desempenho para o armazenamento de arquivos com essas características.

Uma terceira premissa que será assumida sobre o ambiente, e que se beneficia das premissas anteriores, é que o tamanho da *cache* pode ser grande o suficiente para armazenar a maioria dos arquivos que um usuário utiliza com mais frequência. Além disso, assume-se também que depois de acessar um arquivo do sistema, há uma grande probabilidade de que o usuário volte a utilizar o mesmo arquivo em um futuro próximo. Essas duas premissas são comuns ao AFS também (HOWARD, 1988), e esse sistema comprovou que a sua utilização em conjunto permite que a utilização da rede seja reduzida significativamente, visto que a grande maioria dos arquivos que um usuário utiliza com mais frequência já está disponível em seu disco rígido, não sendo necessário que o arquivo seja transferido novamente do servidor para o cliente.

A quarta e última premissa que será assumida sobre o ambiente onde esse sistema de arquivos será utilizado é que a grande maioria das operações realizadas sobre o sistema são as de inclusão de novos arquivos e consulta de arquivos já existentes. As operações de atualização de arquivos serão completamente suportadas pelo sistema, porém assume-se que estas serão em menor número. Além disso, considera-se pouco provável que dois usuários atualizem o mesmo arquivo simultaneamente.

4.3 Cenário de utilização

Após a descrição desses pressupostos sobre o ambiente e os arquivos, torna-se interessante descrever alguns cenários que poderiam vir a ser utilizados para a aplicação dessa API. Nessa seção serão descritos dois cenários fictícios, porém extremamente comuns e possíveis, onde o sistema de arquivos descrito neste trabalho poderia ser facilmente aplicado. Além disso, será detalhado um pouco mais o cenário de utilização desse sistema sobre o Projeto IndexVideo.

4.3.1 Cenário Biblioteca Virtual

Um primeiro exemplo de cenário seria o de uma universidade que deseja digitalizar todos os livros e publicações de sua biblioteca (desde que nenhuma questão legal seja infringida) e permitir que os alunos de seu campus possam acessar todo esse conteúdo digital através da rede da universidade, fornecendo assim um meio simples e prático para que os alunos possam realizar pesquisas acadêmicas e diminuindo a necessidade de comprar diversos exemplares de uma mesma obra.

Nesse cenário, cada livro ou publicação poderia ser armazenado como um arquivo PDF (*Portable Document Format*). As operações mais comuns seriam a inclusão

de novos arquivos e a consulta de arquivos já existentes. As operações de atualização de arquivos seriam praticamente inexistentes. Considerando-se que um aluno geralmente consulta os mesmos livros sobre um mesmo assunto enquanto cursa uma disciplina, haveria uma grande economia na utilização da rede, visto que os livros mais consultados pelo aluno estariam armazenados na *cache*.

4.3.2 Discoteca Virtual

Um outro cenário de utilização do sistema de arquivos que está sendo descrito neste trabalho seria o de uma rádio que possui diversas filiais espalhadas em diferentes cidades, que estão conectadas por um *link* dedicado, e possuem uma boa velocidade de conexão entre si. Essa empresa deseja disponibilizar a todas as suas unidades o acesso ao seu acervo digital, que contém todas as músicas que são tocadas em todas as rádios de sua rede.

Essa empresa deseja permitir que as diversas unidades adicionem novas músicas à sua discoteca virtual. Além disso, os programadores musicais, que são responsáveis por criar a lista de músicas que será tocada em cada programa da rádio, devem ter a possibilidade de consultar todo o acervo e solicitar as músicas que desejam.

Considerando que uma rádio, na grande maioria das vezes, toca um mesmo conjunto de músicas, que geralmente são as que estão fazendo mais sucesso, também haveria uma grande economia na utilização da rede, pois o servidor de cada unidade seria capaz de armazenar em *cache* praticamente todas as músicas que a unidade precisa para um dia de programação. Da mesma forma que no cenário anterior, as operações mais comuns são as de inclusão de arquivos e solicitação de arquivos. Salienta-se ainda que esse cenário, assim como o anterior, necessita de um sistema escalável, que seja capaz de absorver o crescimento do número de arquivos. Necessidade esta, que é um dos objetivos do sistema de arquivos distribuído deste trabalho.

4.3.3 IndexVideo

Além de descrever alguns exemplos de cenários onde o sistema de arquivos distribuído deste trabalho poderia vir a ser utilizado, torna-se interessante descrever também algumas características sobre o cenário do Projeto IndexVideo, que é o estudo de caso deste trabalho.

Conforme descrito na motivação deste trabalho, o objetivo do Projeto IndexVideo é o desenvolvimento de uma ferramenta através da qual os usuários possam fazer anotações sobre o trecho de um vídeo e, posteriormente possam efetuar buscas pelos vídeos através dessas anotações. Conforme descrito em (PRADO LIMA; SILVA; CARBONERA, 2009), o cenário de utilização dessa ferramenta é o contexto educacional, mais especificamente, o ensino de assuntos relacionados a saúde e ao

curso de Medicina da UCS.

Nesse contexto, o cenário típico de utilização do IndexVideo seriam os alunos e professores da área médica que, estando dentro da rede da Universidade, podem efetuar pesquisas de vídeos relacionados ao seu objeto de estudo, buscando dessa forma novos meios que os auxiliem a compreender o funcionamento do corpo humano.

Uma característica importante desse cenário é que os arquivos de vídeo podem vir a ter tamanhos superiores a 30 MB. Nesse caso, para que se obtenha um melhor aproveitamento do sistema de arquivos distribuído descrito neste trabalho, seria interessante particionar arquivos grandes em vários arquivos menores. Dependendo do estudo detalhado que será realizado sobre o Projeto IndexVideo na continuação deste trabalho, é possível que essa divisão do arquivo seja incorporada à implementação da API.

Torna-se importante detalhar um pouco mais a arquitetura do IndexVideo, descrevendo como esse funciona atualmente. O projeto é constituído de dois componentes que funcionam de forma integrada; o **servidor**, que é responsável por armazenar os arquivos de vídeo e as anotações de cada vídeo, que são gravadas em um arquivo XML; e o **cliente**, que fornece uma interface gráfica ao usuário para realizar a anotação, pesquisa e exibição de vídeos. A arquitetura do Projeto IndexVideo é ilustrada na Figura 4.2.

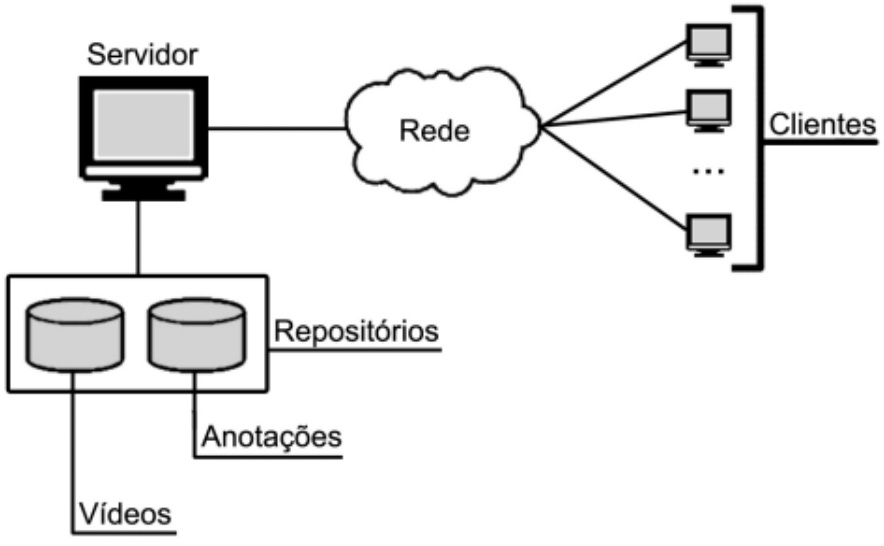

Figura 4.2: Arquitetura do Projeto IndexVideo.

Torna-se importante salientar que o objetivo da aplicação desse sistema de arquivos distribuído sobre o IndexVideo é o de armazenar apenas os arquivos de vídeo, e não os arquivos XML que contém as anotações dos vídeos. Dessa forma, a pesquisa dos vídeos através das anotações ainda é responsabilidade do servidor do IndexVideo. A Figura 4.3 ilustra a integração do Projeto IndexVideo com o presente

trabalho.

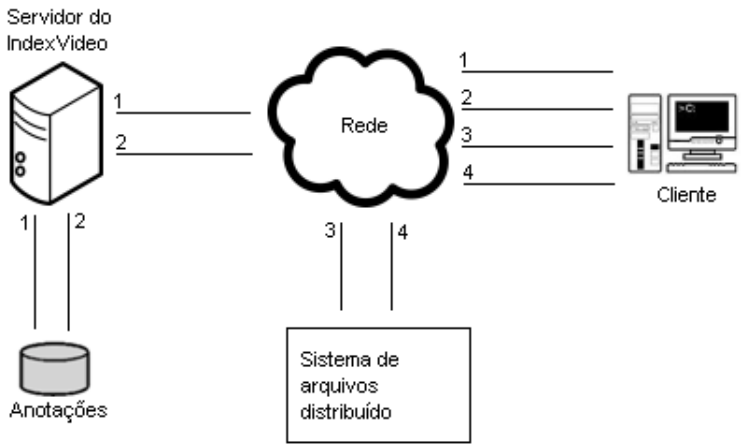

Figura 4.3: Integração do IndexVideo com o sistema de arquivos distribuído descrito neste trabalho.

4.4 Proposta de arquitetura

Após a descrição do cenário de utilização desse sistema, das premissas assumidas com relação ao seu ambiente de aplicação e das características dos arquivos, passa-se agora para a descrição de uma proposta de arquitetura que seja capaz de atingir os objetivos propostos neste trabalho e que possa gerenciar e armazenar de forma eficiente os arquivos com as características descritas. Nessa seção, as questões de projeto de sistema de arquivos distribuídos que já foram listadas no Capítulo 3 serão retomadas, e a decisão tomada sobre cada questão será brevemente descrita.

4.4.1 Arquitetura

O sistema de arquivos proposto neste trabalho será desenvolvido sob o modelo de **arquitetura cliente/servidor**. Sendo assim, torna-se relevante identificar e descrever os componentes dessa arquitetura. O **serviço** oferecido por esse sistema será o armazenamento e recuperação de arquivos de forma distribuída e transparente. O **servidor** será responsável por executar o *software* que fornecerá esse serviço aos clientes. E o **cliente** será a entidade que fará uso desse serviço, adicionando e consultado arquivos.

Após a escolha desse modelo de arquitetura, passe-se agora para uma descrição mais ampla da organização de cada uma dessas partes, cliente e servidor. Da mesma forma que no GFS, a parte servidora será constituída por um conjunto variável

de servidores dedicados responsáveis por realizar o armazenamento dos arquivos, denominados **servidores de armazenamento**. Esse conjunto de servidores de armazenamento será gerenciado por um único servidor, denominado **servidor de controle**.

Os servidores de armazenamento serão responsáveis exclusivamente pelo armazenamento dos arquivos, não conhecendo os demais servidores de armazenamento que compõem o sistema, apenas o servidor de controle. Já o servidor de controle terá conhecimento de todos os servidores de armazenamento do sistema. Ele será o ponto central do sistema, constituindo o ponto de entrada que o cliente irá utilizar para conectar-se ao sistema. Também será responsável por efetuar a distribuição dos arquivos entre os servidores de armazenamento.

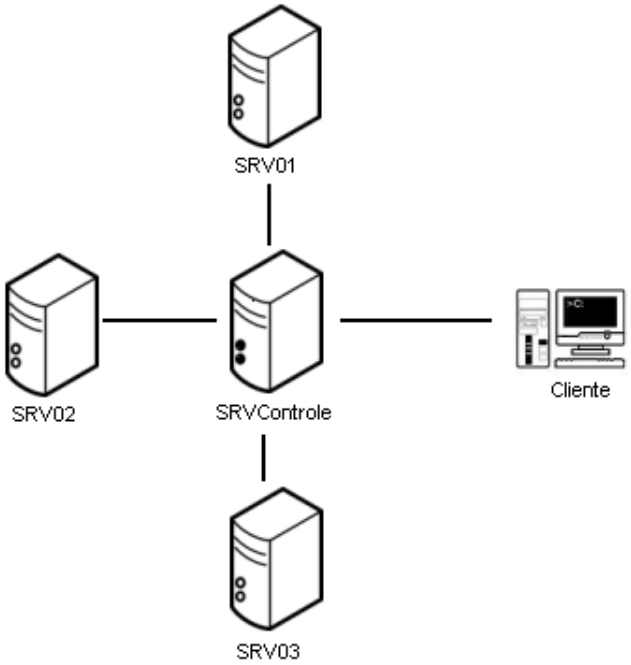

Figura 4.4: Ilustração da arquitetura do sistema

O cliente será representado por uma API, que fornecerá um conjunto de funções ao usuário, neste caso um desenvolvedor, que permitirão que arquivos sejam adicionados, recuperados, atualizados, e excluídos do sistema. A API irá realizar a comunicação com o servidor de controle e abstrair o fato da distribuição para o usuário. Este, terá conhecimento apenas do conjunto de funções que a API disponibilizará, as quais não deverão fornecer qualquer indício da distribuição do sistema e dos servidores que o compõem.

O acesso aos arquivos remotos será feito através do **modelo de carga e atualização**, descrito na Seção 3.1.1. Dessa forma, quando um cliente desejar ler ou atualizar um arquivo, este deverá ser completamente transferido para a *cache* do cliente, para que então possa ser utilizado. Após realizar as alterações desejadas

no arquivo, o cliente deverá enviá-lo novamente ao servidor, utilizando para isso as funções da API. Internamente, para a realização da transferência dos arquivos entre cliente e servidor, será utilizado o protocolo FTP (POSTEL; REYNOLDS, 1985).

4.4.2 Nomeação

Para a implementação da nomeação, optou-se pela utilização de um sistema de arquivos multinível ou hierárquico (TANENBAUM, 2003), no qual existem diretórios hierárquicos para a organização dos arquivos. Apesar de sua implementação ser mais complexa se comparada com a proposta de sistema de arquivos de único nível, ainda assim sua utilização se torna interessante, visto que essa alternativa proporciona uma solução mais elegante para a organização dos arquivos, além de permitir que exista mais de um arquivo com o mesmo nome no sistema, desde que estes não estejam no mesmo diretório.

Associada a essa decisão de um sistema de arquivos hierárquico, optou-se pela criação de um espaço global de nomes, abordagem essa que é descrita na Seção 3.1.4. Dessa forma, todos os caminhos (diretórios) e nomes de arquivos serão armazenados no servidor de controle através de uma lista ou tabela. Sendo assim, quando um arquivo for adicionado ao sistema, deverá ser informado em qual diretório o arquivo será armazenado e também o nome do arquivo. Essas informações de localização (diretório) e nome do arquivo poderão ser usadas posteriormente para solicitar o arquivo.

Salienta-se que o usuário da API não terá nenhum conhecimento sobre a localização física do arquivo nos servidores de armazenamento, nem mesmo da existência de tais servidores. Será conhecido apenas o caminho lógico (estrutura de diretórios) e o nome do arquivo. Essa abordagem fornecerá transparência e independência de posição sobre o arquivo, uma vez que esse caminho e nome de arquivo não fornecem qualquer indício sobre sua localização. Além disso, o arquivo pode facilmente ser movido para outro servidor, sem que seja necessário alterar seu nome ou estrutura de diretórios onde está logicamente armazenado, sendo necessário apenas informar o servidor de controle sobre a nova localização do arquivo.

4.4.3 Cache

O cliente da API fará uso de uma *cache* local, onde os arquivos mais recentemente solicitados ficarão armazenados. Ao solicitar um arquivo para a API, esta, irá verificar antes se o arquivo está na *cache*, e se é a versão mais atual. Se for este o caso, o cliente fará o uso do arquivo local, sem que seja necessário copiá-lo do servidor. No entanto, se o arquivo não estiver na *cache*, ou se não estiver atualizado, a API se encarregará de obter a versão mais atual do arquivo e armazená-lo na *cache*.

O sistema utilizará um esquema de versão para garantir que o cliente possua

sempre a versão mais atualizada do arquivo. Mais especificamente, cada arquivo terá uma versão associada a ele, que será representada por um número inteiro. Toda vez que um arquivo for atualizado em um servidor de armazenamento, o servidor de controle será informado, e a versão do arquivo será incrementada.

Uma característica importante da *cache* nesse sistema é que ela deverá ter um tamanho grande o suficiente para armazenar os arquivos mais utilizados pelo usuário. Quanto maior for o tamanho da *cache* no cliente, melhor será o desempenho do sistema. A vantagem da utilização dessa abordagem é que ela aumenta as chances de que o arquivo solicitado pelo usuário esteja na *cache*, diminuindo assim o tráfego de rede. Optou-se pela utilização dessa abordagem uma vez que essa já é utilizada com sucesso em outros sistemas, como por exemplo o AFS.

4.4.4 Sincronização de processos

Para realizar a sincronização de processos, a abordagem escolhida foi a de **arquivos imutáveis**, onde arquivos podem apenas ser recriados, e não atualizados parcialmente. Dessa forma, vários clientes poderão ter uma cópia do mesmo arquivo em sua *cache* local, porém, uma vez que um cliente inicie o processo de atualização de um arquivo, enviando-o completamente ao servidor, o arquivo ficará travado até que a transferência seja concluída. Assim, nesse momento não serão permitidas operações de leitura ou atualização.

Embora essa abordagem possua a desvantagem de não permitir que dois usuários atualizem o mesmo arquivo simultaneamente, ela se torna atrativa, uma vez que uma das premissas sobre o ambiente de aplicação deste trabalho é que operações de atualização concorrentes sobre o mesmo arquivo dificilmente serão executadas.

4.4.5 Estado de processos e Comunicação

Em função da utilização da *cache* local nos clientes, e conforme as questões de projeto que foram estudadas, optou-se pela utilização de conexões TCP/IP persistentes entre cliente e servidor, ou seja, **com estado**. Isso permitirá um melhor gerenciamento dos arquivos em *cache*, além de fornecer maior flexibilidade para a implementação de novas funcionalidades que possam vir a ser adicionadas ao sistema.

Dessa forma, enquanto a aplicação que utiliza a API estiver executando, será mantida uma conexão com o servidor de controle. Já quando for necessário realizar a transferência de arquivos entre o cliente e um dos servidores de armazenamento, uma conexão FTP (POSTEL; REYNOLDS, 1985) será feita, e, tão logo o arquivo tenha sido transferido, essa conexão será encerrada.

A conexão do servidor de controle com os servidores de armazenamento e com os clientes que utilizarão a API será feita através de **soquetes**. Conforme descrito

na Seção 3.1.3, apesar de exigir um trabalho maior de implementação, em função de sua simplicidade, essa abordagem proporciona um maior desempenho ao sistema quando comparada com a utilização de RMI ou RPC, uma vez que o uso de soquetes não exige nenhum processamento extra para realizar o gerenciamento do canal de comunicação.

4.4.6 Replicação, Tolerância a Falhas e Segurança

O principal objetivo para a implementação de replicação em um sistema, além do aumento de desempenho proporcionado, é a garantia de tolerância a falhas. Em função de existirem várias cópias do mesmo arquivo espalhadas entre os servidores, mesmo que um dos servidores fique indisponível, o sistema ainda poderá fornecer o arquivo ao cliente, pois fará uso de uma das cópias que está em um servidor que esteja disponível.

Como o objetivo deste trabalho não é garantir a tolerância a falhas, o recurso de replicação não será implementado. Mesmo assim, torna-se necessário definir como o sistema irá manipular as falhas que possam vir a ocorrer. Neste sentido, a decisão de projeto foi que, toda e qualquer falha será lançada em forma de exceção para a aplicação que está utilizando a API, permitindo assim que essa decida o que deverá ser feito, podendo optar por tentar novamente ou mostrar uma mensagem adequada ao usuário da aplicação.

Com relação a segurança, como esta também não constitui um objetivo a ser atingido neste trabalho, não serão implementados recursos de autenticação nem de controle de acesso a arquivos. Um outro motivo para esta decisão é que, por se apresentar ao usuário como uma API, é possível que a aplicação que venha a fazer uso desse sistema já tenha implementado algum mecanismo de segurança ou de controle de acesso aos arquivos. Sendo assim, ficará a critério da aplicação que utilizará a API a garantia de segurança.

4.5 Considerações Finais

Neste capítulo foi definido o objetivo do sistema que está sendo proposto e também foram descritos os pressupostos que serão assumidos sobre o ambiente onde o sistema será utilizado. Além disso, foram exemplificados possíveis cenários de aplicação desse sistema. Em seguida foi descrita uma proposta de arquitetura para esse sistema, as questões de projeto de sistema de arquivos distribuídas que foram analisadas no Capítulo 3 foram retomadas a fim de definir as decisões de projeto do sistema que está sendo proposto.

No capítulo seguinte será descrita a implementação do sistema proposto, onde serão apresentados detalhes da arquitetura do sistema, os componentes de *software*

envolvidos e as ferramentas utilizadas para a implementação do sistema.

5 IMPLEMENTAÇÃO DO SISTEMA DE ARQUIVOS DISTRIBUÍDO

Até o presente momento descreveu-se os conhecimentos relacionados a sistemas distribuídos, mais especificamente, os sistemas de arquivos distribuídos, ressaltando as vantagens, ferramentas disponíveis e técnicas relacionadas à sua implementação. Além disso, no capítulo anterior foi realizada uma proposta de um novo sistema de arquivos distribuído que fosse capaz de atender as necessidades elencadas na introdução deste trabalho, visto que nenhum dos sistemas de arquivos analisados atendia completamente tais necessidades.

Apesar de importantes e necessárias para o desenvolvimento desse trabalho, as definições do capítulo anterior não são capazes de fornecer informações detalhadas e completas sobre a arquitetura e implementação desse sistema. Esse nível de detalhamento torna-se importante, pois fornece uma visão mais clara sobre o funcionamento do sistema, e permite que trabalhos futuros sejam desenvolvidos, tendo como base o trabalho atual.

Esse capítulo tem o objetivo de descrever a arquitetura do sistema que foi proposto, fornecer informações relevantes sobre seu projeto e implementação, descrever os componentes de *software* que constituem o sistema, sua organização em camadas, e como os componentes interagem entre si.

5.1 Visão Geral do Sistema

O sistema de arquivos que foi proposto e implementado é composto de diversos componentes de *software*. No contexto desse trabalho, por componente de *software* entende-se um aplicativo ou programa que executa em uma máquina e interage com outros aplicativos a fim de atingir um objetivo. Nesse caso, o objetivo é o armazenamento de arquivos de forma distribuída através da utilização de uma API como meio de manipulação dos arquivos.

Conforme descrito no Capítulo 4, o sistema é composto de um servidor de controle, diversos servidores de armazenamento e uma API para a manipulação dos

arquivos. De acordo ainda com a proposta de arquitetura, foi utilizado o protocolo FTP para a transferência dos arquivos entre os servidor de armazenamento e o cliente que esteja utilizando a API. Sendo assim, em cada servidor de armazenamento é necessário que esteja instalado um servidor de FTP. Já entre o servidor de controle e a API, a comunicação é feita através de soquetes utilizando uma camada de comunicação, a qual será melhor descrita na Seção 5.2. A Figura 5.1 ilustra os componentes de *software* envolvidos no sistema e a comunicação entre eles.

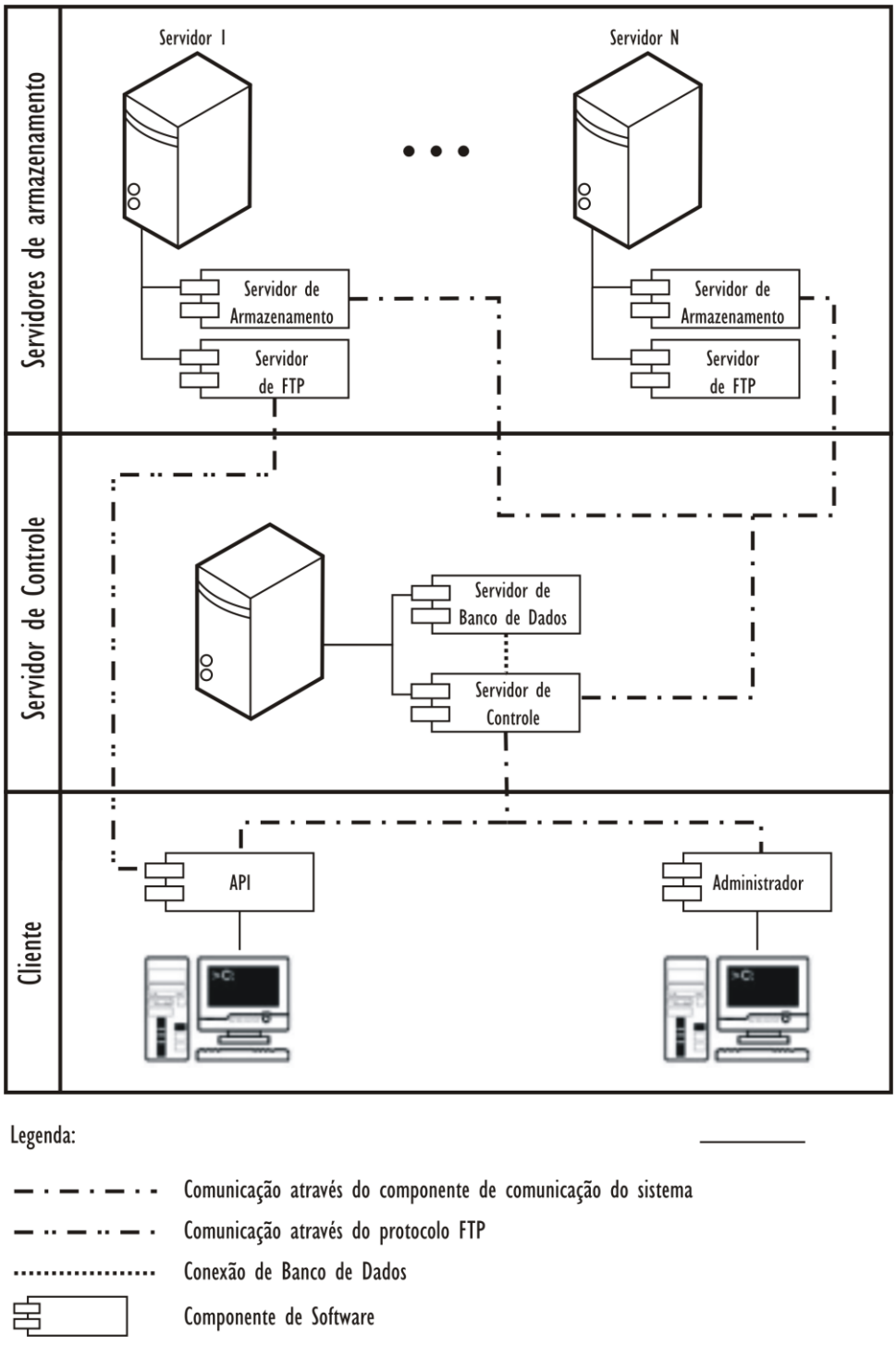

Figura 5.1: Visão geral do sistema.

5.2 Camada de Comunicação

Para permitir a comunicação entre os componentes de *software* que foram desenvolvidos, implementou-se uma camada de **Comunicação**, que é comum a todos os componentes do sistema, podendo ser utilizada entre:

- A API e o Servidor de Controle
- O Servidor de Controle e o Servidor de Armazenamento
- O Servidor de Controle e o Administrador (será melhor descrito na seção 5.3.4)

A camada de **Comunicação** encapsula toda a comunicação entre dois componentes, permitindo a conexão, troca de mensagens e desconexão. É responsabilidade dessa camada fornecer métodos para o envio e recebimentos de todas as mensagens do sistema. Através dela os componentes do sistema criam mensagem em formato de objeto, os quais são serializados em formato XML e enviados através de soquetes. A Figura 5.2 ilustra esse processo através de uma exemplo de comunicação entre a API e o Servidor de Controle.

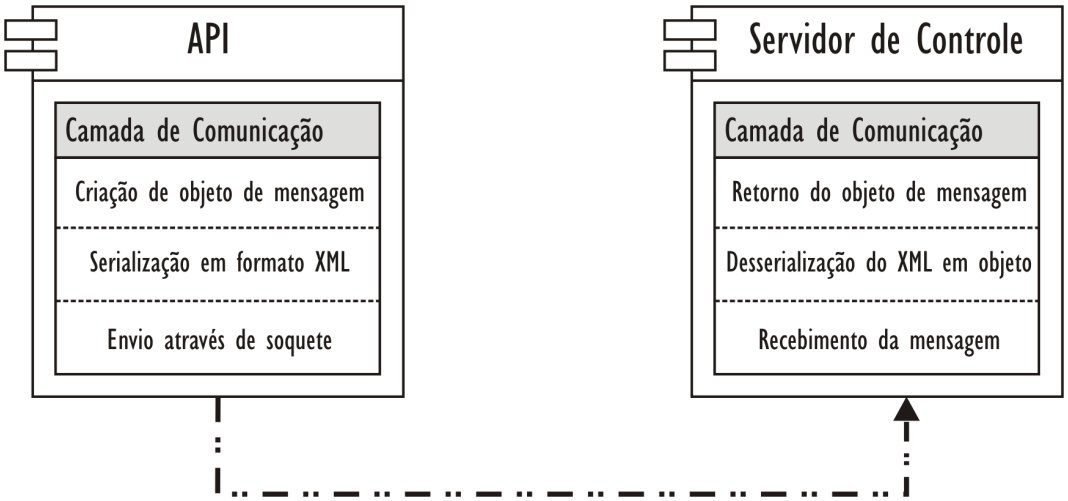

Figura 5.2: Exemplo de comunicação entre API e Servidor de Controle através da camada de Comunicação.

A criação desse camada de comunicação está de acordo com a decisão de projeto que foi tomada durante a proposta da arquitetura, descrita na Seção 4.4.5, a qual sugere a utilização de soquetes, devido a sua simplicidade e maior desempenho em comparação com RMI ou RPC. No entanto, acredita-se que, caso fosse adotada a utilização de RMI para realizar a comunicação entre os componentes, o desenvolvimento do sistema teria sido consideravelmente simplificado, sem que houvesse um grande comprometimento de performance.

5.3 Componentes do Sistema

Ao projetar e implementar um sistema, torna-se interessante dividi-lo em módulos e camadas, agrupando funções semelhantes e proporcionando uma organização que torne o sistema flexível, fácil de ser compreendido e mantido. Nesta seção serão descritos os componentes de *software* envolvidos na implementação do sistema de arquivos distribuído proposto, que são basicamente os mesmos que foram citados no Capítulo 4: o **servidor de controle**, o **servidor de armazenamento** e a **API**. Além dos componentes propostos, também foi desenvolvido um outro, que apesar de ser opcional, facilita a utilização do sistema desenvolvido. Esse outro componente, chamado **Administrador**, trata-se de uma interface gráfica que permite a administração do sistema e visualização da carga dos servidores de armazenamento.

Esses componentes de *software* serão descritos nas próximas seções, sendo que para cada um desses componentes será feita uma descrição de sua arquitetura. Além disso, serão apresentadas informações relevantes sobre sua implementação.

5.3.1 Componente Servidor de Controle

O servidor de controle é o ponto central do sistema de arquivos distribuído desenvolvido, pois esse concentra a informação sobre todos os servidores de armazenamento e todos os arquivos do sistema. De uma forma geral, o servidor de controle pode ser considerado como um grande catálogo de informações, que é utilizado pela **API** e pelo **Administrador** para obter informações sobre os arquivos e os servidores de armazenamento. No entanto, mesmo sendo responsável por gerenciar toda essa informação, ele não representa um gargalo de processamento, visto que todo o armazenamento e transferência dos arquivos é feito diretamente entre a API e os servidores de armazenamento.

Em um nível mais detalhado, as funções do servidor de controle são:

- Manter informações sobre todos os arquivos do sistema, como por exemplo: nome, localização e tamanho. Além disso, esse deve fornecer funcionalidades que permitam incluir, atualizar e excluir informações sobre os arquivos do sistema.

- Manter informações sobre todos os servidores de armazenamento, como o espaço reservado, o espaço utilizado e o espaço disponível em disco.

- Manter informações sobre as transferências que estão sendo realizadas entre os servidores de armazenamento e um cliente (sistema que esteja utilizando a API).

- Fornecer à API o melhor servidor de armazenamento disponível no momento

em que esta solicitar o armazenamento de um arquivo (o método para a escolha do melhor servidor de armazenamento será descrito mais adiante nessa seção).

- Fornecer informações ao componente **Administrador**, para que este possa listar os servidores de armazenamento, e também funcionalidades que permitam a inclusão e atualização dos dados de um servidor de armazenamento.

Para que o servidor de controle fosse capaz de executar todas essas funções, o projeto de sua arquitetura teve de ser elaborado de forma a proporcionar camadas ou componentes que permitissem:

- Representar os arquivos e os servidores de armazenamento através de entidades.

- Efetuar validações sobre as operações realizadas com os arquivos.

- Persistir as informações sobre os arquivos e servidores de armazenamento, ou seja, armazená-las em um meio persistente de modo que seja possível recuperá-las posteriormente.

- Fornecer um canal de comunicação através do qual a **API** e o **Administrador** poderiam receber e enviar informações.

Essas necessidades nortearam o projeto da arquitetura do servidor de controle, e serviram de base para a divisão da sua implementação em camadas. A organização de sua arquitetura é ilustrada pela Figura 5.3.

Figura 5.3: Arquitetura simplificada do Servidor de Controle

A camada de **Modelo** é responsável por representar as entidades do sistema (arquivos e servidores de armazenamento), sendo que cada modelo possui um conjunto de atributos que são utilizados para representar uma entidade. Esse conjunto de atributos pode ser obtido e alterado através dos métodos de acesso, também conhecidos como *get* e *set*. Essas classes de modelo são utilizadas como um meio de transporte de informação entre camadas, e são utilizadas em conjunto pelas camadas de persistência e negócio.

A camada de **Negócio** é responsável por aplicar as regras de funcionamento do sistema. Nela está localizada toda a lógica de funcionamento do sistema de arquivos distribuído. No entanto, esta camada, assim como todo o servidor controle, está limitada a gerenciar apenas as informações sobre as entidades, ou seja, ela interage apenas com os modelos, alterando seus atributos através do métodos de acesso. A interação com as entidades reais, como os arquivos e os servidores de armazenamento, acontece na API, que será explicada na Seção 5.3.3.

Em resumo, a camada de negócio trabalha apenas com os modelos, alterando seus atributos e persistindo essa informação através da camada de **Persistência**, a qual é responsável por fornecer um meio de salvar e recuperar as informações sobre as entidades. Essa camada de persistência foi projetada e implementada de modo que é possível alterar o meio de armazenamento das informações sem que

seja necessário reescrever todo o servidor de controle. Para isso foram utilizadas interfaces que definem um conjunto de métodos que devem ser implementados, a fim de que um novo meio de armazenamento possa ser utilizado. Para o desenvolvimento desse trabalho, foi implementada a persistência em dois bancos de dados relacionais, o *MySQL* (SUN MICROSYSTEMS, 2009a) e o *Microsoft SQL Server* (CORPORATION, 2009). Para a alteração entre um meio de armazenamento e outro, basta alterar uma linha do arquivo de configuração do servidor de controle. Também é possível utilizar outros bancos de dados e meios de armazenamento, no entanto faz-se necessária a implementação da interface que é definida pela camada de **Persistência**.

As camadas que foram descritas (modelo, negócio e persistência) são conhecidas apenas pelo servidor de controle. Um cliente que se conecta ao servidor de controle interage apenas com as camadas de **Comunicação**, a qual já foi descrita na seção 5.2 e a camada de **Serviço** que é responsável pela criação do serviço TCP, ou seja, um soquete em uma porta TCP/IP que ficará aguardando por conexões de clientes. Ao receber uma conexão de um cliente, a camada de serviço disponibiliza uma *thread* que será responsável por atender esse cliente. Essa *thread* irá receber as requisições do cliente através da camada de comunicação, processar as mensagens, executando a operação adequada através da camada de negócio, e então responder ao cliente através da camada de comunicação novamente.

Uma questão importante sobre a implementação do servidor de controle é o critério utilizado para decidir qual o melhor servidor de armazenamento para receber um arquivo que irá ser adicionado ao sistema através da API. Para isso, o servidor de controle mantém uma lista das transferências que estão acontecendo entre todos os clientes conectados (API's) e os servidores de armazenamento. O servidor de armazenamento que estiver realizando menos transferências no momento em que a API solicitou a inclusão do arquivo, será escolhido para receber o arquivo. Caso exista mais de um servidor com o mesmo número de transferências em andamento, será utilizado como critério de desempate o espaço livre dos servidores de armazenamento.

Para que o servidor de controle seja capaz de manter essa lista de transferências em andamento, a API deve informá-lo sempre que iniciar e terminar uma transferência com um servidor de armazenamento, seja ela para envio ou recebimento do arquivo.

5.3.2 Componente Servidor de Armazenamento

O servidor de armazenamento é responsável pelo armazenamento dos arquivos do sistema. Ele é o componente de *software* mais simples do sistema, pois é constituído apenas de um servidor de FTP e um serviço TCP/IP, o qual utiliza a mesma camada

de comunicação já citada. Esse serviço aguarda por conexões em uma porta TCP que é configurável, e fornece informações sobre a utilização de seu disco rígido para o servidor de controle.

Uma representação simplificada de sua arquitetura pode ser observada na Figura 5.4, onde são representados o servidor de FTP e o serviço, juntamente com a camada de comunicação.

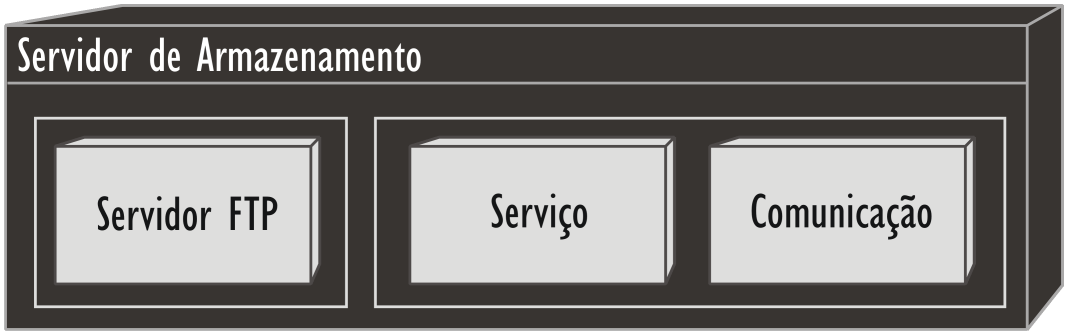

Figura 5.4: Arquitetura simplificada do Servidor de Armazenamento

As camadas de **Comunicação** e **Serviço** têm a mesma função das descritas no servidor de controle, exercendo as mesmas responsabilidades e fornecendo as mesmas funcionalidades já descritas.

O servidor de FTP será utilizado pela API para realizar as transferências de arquivos, e o serviço TCP/IP irá receber conexões do servidor de controle e fornecer informações sobre a utilização de seu espaço em disco. Torna-se importante salientar que, apesar de executarem no mesmo computador, não existe relação entre o servidor de FTP e o serviço TCP/IP. Além disso, o servidor FTP é uma entidade externa e não foi implementado no sistema. O objetivo disso é permitir que qualquer servidor de FTP possa ser usado para o armazenamento dos arquivos, fornecendo assim flexibilidade e liberdade de escolha.

5.3.3 Componente API (Application Program Interface)

A API é o componente de *software* através do qual o usuário (desenvolvedor) interage com o sistema. É através dela que o usuário poderá manipular os arquivos do sistema, realizando as operações de inclusão, atualização, requisição e exclusão de arquivos. Para a utilização da API, o usuário deverá executar uma sequência de passos que consiste em: instanciar a API; efetuar a configuração da API, informando o caminho do arquivo de configuração e conectar. Após isso, é possível manipular todos os arquivos do sistema, realizando tantas operações quantas forem necessárias, e, ao final, efetuar a desconexão.

A Figura 5.5 apresenta um exemplo de código escrito em Java que utiliza a API para adicionar um arquivo ao sistema. Nas linhas 1 a 6 do exemplo de código apre-

sentado, é feita a declaração e instanciação da API. Na linha 9 é feita a configuração da API através do arquivo de configuração (para maiores informações sobre a configuração da API sugere-se o Anexo 7.3). Em seguida é feita a conexão da API com o servidor de controle (linha 12). Após essa conexão já é possível manipular os arquivos. Nesse exemplo, é adicionado um novo arquivo ao sistema. Para isso, é instanciado um objeto da classe `File` (nativo do Java - `java.io.File`), que irá representar o arquivo local que se deseja adicionar os sistema (linha 15) e em seguida o arquivo é adicionado através da API (linha 18). Para a adição do arquivo, são informados três parâmetros: o arquivo que se deseja adicionar (instância da classe `java.io.File`), o caminho do arquivo na estrutura de diretório do sistema (nesse exemplo `/docs/`) e o nome do arquivo dentro do sistema distribuído (nesse exemplo `Teste.txt`). Esse caminho e nome podem ser utilizados posteriormente para solicitar o arquivo através da API. Depois de efetuar a manipulação dos arquivos através da API, é feita a desconexão (linha 21).

```
1  // Declara a API
2  KS2API api = null;
3  try {
4
5      // Instancia a API
6      api = new KS2API();
7
8      // Configura
9      api.configure("E:\\TCC\\API\\ks2api.config");
10
11     // Conecta
12     api.connect();
13
14     // Cria o arquivo que será adicionado
15     File file = new File("E:\\TCC\\Temp\\Teste.txt");
16
17     // Adiciona o arquivo
18     api.addFile(file, "/docs/", "Teste.txt");
19
20     // Desconecta
21     api.disconnect();
22
23 } catch (Exception e) {
24     e.printStackTrace();
25     api.disconnect();
26 }
```

Figura 5.5: Trecho de código em Java que exemplifica a utilização da API para adicionar um arquivo ao sistema.

Para facilitar a utilização da API, todos os métodos foram documentados utilizando o padrão Javadoc. Além disso, foi gerada a documentação completa da API em formato HTML, a qual encontra-se no Anexo 7.4 deste trabalho.

Quanto ao projeto da **arquitetura** da API, este se mostrou simples com relação

ao número de pacotes, porém complexo quanto ao tratamento de erros e garantia de consistência entre as informações armazenadas no servidor de controle, os arquivos armazenados no servidores de armazenamento e os arquivos armazenados na *cache* local. A Figura 5.6 ilustra de forma simplificada essa arquitetura.

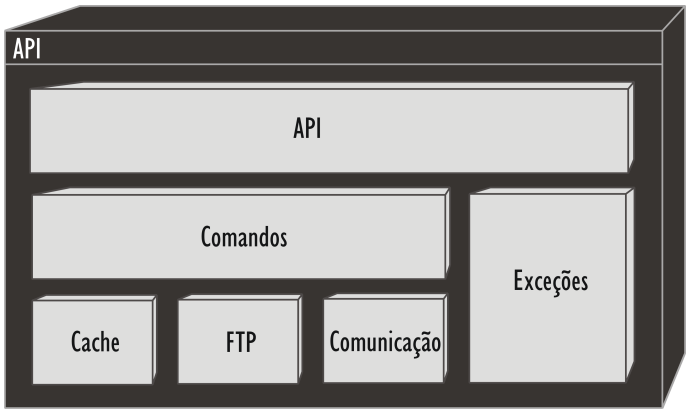

Figura 5.6: Arquitetura simplificada da API

Da mesma forma que no servidor de controle e de armazenamento, a camada de **Comunicação** é responsável por realizar a troca de mensagens com o servidor de controle. A camada de **FTP** se responsabiliza por enviar e receber os arquivos dos servidores de armazenamento. Já a camada de ***cache*** é responsável por gerenciar os arquivos que estão armazenados na *cache* local.

A camada de **Comandos** fornece uma estrutura que permite encapsular cada operação, seja ela de comunicação, FTP ou *cache*, em um comando, de forma que seja possível desfazer esse comando caso ocorra algum erro. Os possíveis erros que podem ocorrer durante a utilização da API estão mapeados em exceções específicas, localizadas na camada de **Exceção**.

A maioria das exceções geradas pela API são do tipo *Checked Exceptions*, ou seja, o usuário da API é obrigado a implementar o tratamento da exceção. Porém, algumas exceções lançadas pela API são do tipo *Unchecked Exception*, que não obrigam o desenvolvedor a realizar o tratamento da exceção, pois representam erros inesperados no sistema, geralmente provenientes da utilização incorreta do sistema, ou devido a erros que estão fora do controle do usuário. Algumas das ocasiões em que a API lança esse tipo de exceção são:

- O desenvolvedor tentou realizar uma operação sobre um arquivo sem ter efetuado configuração ou a conexão da API.
- No servidor de controle, caso esse não consiga realizar a conexão com o banco de dados.
- No servidor de armazenamento, caso esse não consiga informar ao servidor de controle quanto espaço livre possui em seu disco rígido.

Em todos esses casos, não há nenhuma ação que o desenvolvedor possa executar para restaurar o sistema do erro, ou seja, não há nada que possa ser implementado de forma a corrigir esse erro. Assim, não sendo necessário obrigar o desenvolvedor a efetuar o tratamento da exceção. No entanto, todos as exceções, sejam elas *checked* ou *unchecked*, são armazenadas em um arquivo de *log* da API, que pode ser consultado pelo desenvolvedor a fim de obter mais detalhes sobre um erro que possa ter ocorrido durante a utilização da API. Para maiores informações sobre *checked* e *unchecked exceptions* sugere-se (SUN MICROSYSTEMS, 2009b).

Todas as camadas que foram citadas são utilizadas e controladas pela camada da **API**. É através dela que o desenvolvedor interage com o sistema, fazendo uso dos métodos que ela disponibiliza para manipular os arquivos.

5.3.4 Componente Administrador

Esse componente de *software* não havia sido considerado na fase de proposta do sistema, porém, no decorrer do desenvolvimento do trabalho algumas questões vieram a tona, dentre elas:

- Como o usuário da API irá adicionar e configurar os servidores de armazenamento do sistema?

- Não seria interessante fornecer ao usuário da API uma ferramenta através da qual fosse possível verificar a carga do sistema, o espaço já utilizado e o espaço livre?

Em função dessas questões, optou-se pela implementação de novo componente de *software*, o **Administrador**, que consiste de uma interface gráfica que permite ao usuário do sistema conectar a um servidor de controle e obter informações sobre a utilização dos servidores de armazenamento, e até mesmo adicionar mais servidores ao sistema. Através do Administrador, é possível consultar os servidores de armazenamento, espaço utilizado, espaço disponível, gerenciar o espaço reservado, os dados de conexão ao servidor de FTP (usuário e senha), adicionar servidores de armazenamento, etc. A Figura 5.7 ilustra a interface do Administrador, onde é exibida a tela que lista os servidores de armazenamento. Maiores informações sobre os recursos disponíveis no Administrador podem ser encontradas no Anexo 7.2, onde está localizado o manual de utilização dessa ferramenta.

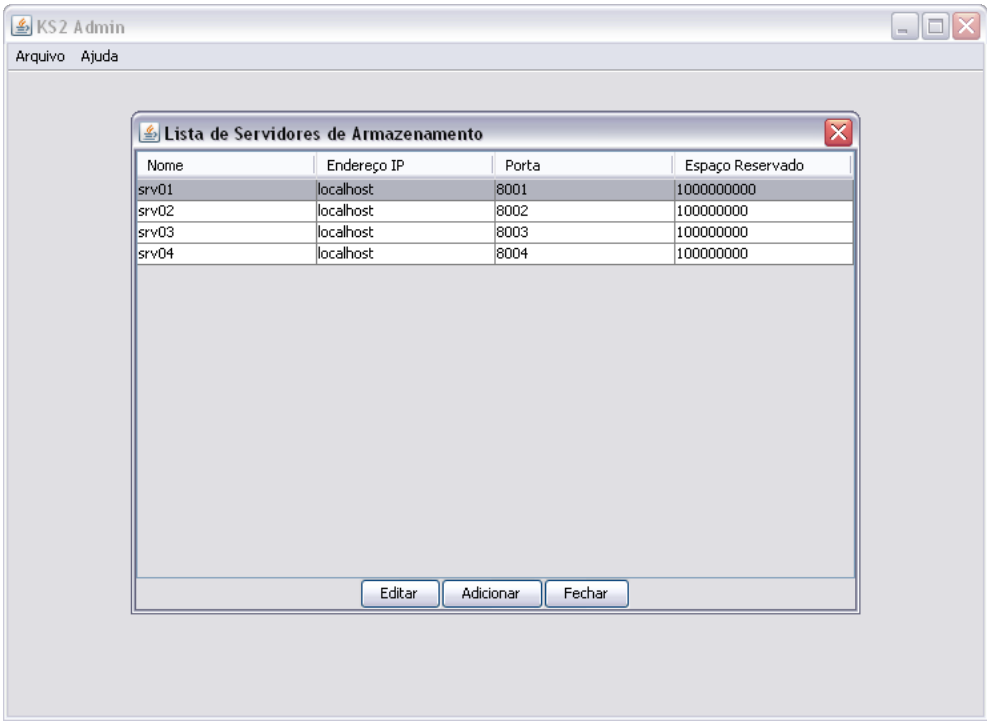

Figura 5.7: Tela do Administrador onde é mostrada a listagem dos servidores de armazenamento.

Esse componente de *software* é opcional, sendo assim, sua instalação não é obrigatória para o funcionamento do sistema. Além disso, ele pode ser instalado em qualquer computador que tenha acesso ao servidor de controle, não necessariamente no mesmo computador onde o servidor de controle está executando, pois a comunicação entre o Administrador e o Servidor é feita usando a camada de comunicação já descrita.

5.4 Funcionamento do Sistema

Até o momento, foi feita uma apresentação geral do sistema, da camada de comunicação utilizada para a troca de mensagens entre os componentes de *software* e alguns detalhes da implementação desses componentes. No entanto, ainda não foi descrito nenhum detalhe sobre como esses componentes interagem para o efetuar o armazenamento dos arquivos. Esse é o objetivo dessa seção, onde será apresentada de forma resumida, as funcionalidades que a API fornece e a sequência de passos que é executada em cada uma delas para que os arquivos possam ser gerenciados no sistema.

Adicionar arquivo: quando o usuário da API solicita que um arquivo seja adicionado ao sistema, o primeiro passo da API é informar ao servidor de controle o tamanho do arquivo, o caminho e o nome que será utilizado para a recuperação do arquivo posteriormente. O servidor de controle, ao receber essas

informações, as adiciona em seu banco de dados, marcando o arquivo como inválido, uma vez que o arquivo propriamente dito ainda não foi transferido para o servidor de armazenamento. Após o servidor de controle verifica qual servidor de armazenamento possui uma maior disponibilidade para receber o arquivo, retornando para a API os dados de conexão ao servidor de FTP correspondente. Através desses dados, a API efetua uma conexão com o servidor de FTP, envia o arquivo e posteriormente efetua a desconexão. Em seguida, a API informa ao servidor de controle o fim da transferência do arquivo. Por fim, o servidor de controle marca o arquivo como válido, e esse já pode ser solicitado por outro cliente.

Atualizar arquivo: de acordo com a proposta de arquitetura descrita na Seção 4.4.1, implementou-se o modelo de carga e atualização, sendo assim, antes de atualizar um arquivo, o usuário (desenvolvedor) deverá solicitá-lo para que este seja transferido para a *cache* local. Após isso, o processo de atualização de um arquivo é o mesmo de inclusão, diferindo apenas pelo fato de que nesse caso, o servidor de armazenamento do arquivo já está definido. Da mesma forma que na inclusão, antes de iniciar a atualização do arquivo no servidor de armazenamento, o servidor de controle bloqueia o arquivo, impedindo que outros clientes o solicitem. Além disso, através das informações sobre o arquivo que a API recebeu do servidor de controle, ela irá verificar se a versão do arquivo que está na cache *local* é a mais atual, e caso não seja, a atualização será cancelada e uma exceção específica será lançada. Ao realizar a atualização do arquivo, a sua versão anterior que estava no servidor de armazenamento é sobrescrita, não havendo assim um histórico das versões do arquivo.

Solicitar arquivo: ao solicitar um arquivo do sistema através da API, esta irá enviar ao servidor de controle uma mensagem solicitando a localização do arquivo. O servidor de controle, ao receber essa mensagem, responde com os dados de conexão ao servidor de FTP que contém o arquivo. A API então informa ao servidor de controle o início da transferência do arquivo, e em seguida, efetua uma conexão com o servidor de FTP e inicia a transferência do arquivo para a máquina local, armazenando-o dentro do diretório que foi configurado como *cache*. Ao final da transferência do arquivo, a API desconecta do servidor de FTP e informa ao servidor de controle sobre o término da transferência.

Excluir arquivo: ao receber uma solicitação para excluir um arquivo do sistema, a API solicita o bloqueio do mesmo no servidor de controle, dessa forma, nenhum cliente pode efetuar qualquer operação sobre este arquivo. Em seguida a API efetua uma conexão com o servidor de FTP de forma a excluir o arquivo.

Após a exclusão, a API fecha a conexão com o servidor de FTP e solicita ao servidor de controle que exclua as informações relacionadas ao arquivo do banco de dados. Por fim, a API ainda verifica se o arquivo está armazenado na *cache* local, e se for o caso, efetua sua exclusão.

5.5 API's Utilizadas

O fato da tecnologia Java ser livre e possuir uma grande comunidade de desenvolvedores ao redor do mundo, contribuiu para o surgimento de diversas ferramentas, *frameworks* e API's, que propõem-se a facilitar a execução de algumas atividades simples e muitas vezes repetitivas. Um exemplo disso é a transferência de arquivos através do protocolo FTP. Qualquer desenvolvedor Java poderia utilizar as classes que a linguagem fornece e desenvolvedor um programa que conecta à um servidor de FTP e efetua a transferência de arquivos. No entanto, essa é uma atividade cujo o desenvolvimento ocupa um tempo considerável.

Em função disso, ao projetar e desenvolver um sistema, torna-se interessante a utilização de algumas ferramentas já existentes, e que facilitem a construção do sistema, permitindo que o desenvolvedor dedique seu tempo e atenção às questões específicas do sistema que está sendo desenvolvido, sem que seja necessário se preocupar com detalhes específicos de algo que não é o objetivo principal de seu trabalho.

Esse também o objetivo do desenvolvimento desse trabalho, fornecer uma ferramenta para o armazenamento de arquivos de forma distribuída, de forma que o desenvolvedor não precise se preocupar em como os arquivos são distribuídos ou armazenados.

Tendo em vista as vantagens da utilização de ferramentas já existentes, esse trabalho fez uso de quatro API's para facilitar a execução das seguintes atividades:

- Geração de *logs* do sistema (*Log4j*)
- Transferência de arquivos via FTP (*Jakarta Commons Net*)
- Obter informações sobre a utilização do espaço em disco (*Hyperic Sigar*)
- Serializar objetos em formato XML (*XStream*)

As próximas seções são dedicadas a descrever as ferramentas que foram utilizadas, realizando uma breve descrição de cada uma delas e citando em que ocasiões essas foram utilizadas para a implementação do sistema de arquivos distribuído.

5.5.1 Log4j

Seja durante o desenvolvimento, teste ou utilização de uma aplicação, erros sempre podem ocorrer. Esses erros podem ocorrer por incoerências na lógica do desen-

volvimento do programa, ou devido a problemas de acesso a recursos externos ao sistema, como por exemplo um arquivo ou uma conexão com um servidor.

Durante o desenvolvimento de um sistema, ferramentas de *debug* de código (execução do programa passo-a-passo) geralmente estão disponíveis para auxiliar desenvolvedores a identificar a causa de um erro. No entanto, há casos em que tais ferramentas não estão disponíveis, como por exemplo em alguns tipos de aplicações distribuídas. Além disso, durante a utilização de um programa em um ambiente de produção, esses recursos de *debug* não podem ser utilizados.

Uma alternativa à utilização de ferramentas de *debug* é a utilização de *logs*, ou seja, o desenvolvedor pode inserir trechos de código em seu programa que armazenam em meio persistente informações relevantes acerca da execução do programa de forma que essas possam ser verificadas posteriormente. Dessa forma, caso ocorra algum erro na aplicação, essas informações podem ser consultadas e analisadas a fim de identificar a causa do problema.

Essa abordagem torna-se interessante, visto que não necessita de uma ferramenta adicional para a realização de *debug* do código. Além disso, os *logs* podem ser gerados e consultados tanto em desenvolvimento, teste, ou utilização do sistema em um ambiente de produção.

O desenvolvedor de uma aplicação poderia implementar uma API ou camada que fosse responsável por gerar esses *logs* em um arquivo. No entanto já existem ferramentas estáveis que se propõem a simplificar a tarefa de geração de *logs*. O próprio JDK (a partir do 1.4) já inclui o pacote (`java.util.logging`) que permite a geração de *logs* de forma simples e nativa. Outra opção consiste na utilização do *Log4j* da *Apache Group*.

O *Log4j* é uma ferramenta que auxilia o desenvolvedor na geração de *logs* em diferentes formatos de saída, dentre eles pode-se citar: arquivos; a console do Java; objetos nativos do Java que representam fluxos de saída, como por exemplo `java.io.OutputStream` e `java.io.Writer`; dentre outros. Essa é uma ferramenta otimizada, flexível e segura, além de contar com uma grande comunidade de desenvolvedores ao redor do mundo, o que lhe proporcionou uma grande vantagem competitiva em relação à ferramenta de *log* nativa do JDK. Além disso, o *Log4j* possui inúmeras funcionalidades que não estão presentes na ferramenta de *log* da JDK (FOUNDATION, 2007).

O *Log4j* trabalha com cinco diferentes níveis de *log*, que em nível hierárquico são: *debug*, *info*, *warn*, *error* e *fatal*. O desenvolvedor, ao solicitar que seja gravada uma mensagem de *log*, deve informar à qual nível essa mensagem pertence. Posteriormente, através do arquivo de configuração do *Log4j*, o desenvolvedor pode informar qual o nível de mensagens de *log* ele deseja que sejam exibidas. Esses níveis de *log* são hierárquicos, ou seja, se for configurado que sejam exibidas as mensagens

de nível *warn*, então as mensagens de *error* e *fatal* também serão exibidas, e as mensagens que estão abaixo na hierarquia (*debug* e *info*) serão ignoradas.

Para o desenvolvimento desse trabalho, optou-se pela utilização do *Log4j* 1.2 (última versão estável), em função de suas vantagens e flexibilidade em relação a API nativa de *log* do JDK. Essa ferramenta foi utilizada em todos os componentes de *software* do sistema: servidor de controle, de armazenamento, API e Administrador. O usuário da API tem acesso ao arquivo de configuração do *Log4j* e pode alterá-lo da maneira que julgar mais apropriada, ou ainda, poderá utilizar a configuração padrão, onde serão exibidas apenas as mensagens de nível *info*, *error* e *fatal*.

5.5.2 Jakarta Commons Net

A *Jakarta Commons Net* é uma biblioteca desenvolvida pela *Apache Software Foundation* que implementa o lado cliente de diversos protocolos básicos da Internet. Seu principal objetivo é fornecer uma ferramenta capaz de realizar as operações básicas sobre os protocolos suportados. Ela oferece suporte a diversos protocolos, como por exemplo: *FTP/FTPS*, *NNTP*, *SMTP*, *POP3*, *Telnet*, *TFTP*, *Finger*, *Whois*, *rexec/rcmd/rlogin*, *Time (rdate)* e *Daytime*, *Echo*, *Discard*, *NTP/SNTP*. Dentre esses protocolos, utilizou-se o FTP para transferência de arquivos (FOUNDATION, 2008)

Para o desenvolvimento desse trabalho utilizou-se a versão 2.0 dessa biblioteca, e sua aplicação restringiu-se apenas à API. É através da *Jakarta Commons Net* que a API envia e obtém os arquivos dos servidores de FTP que estão executando nos servidores de armazenamento.

5.5.3 Hyperic Sigar

O *Hyperic Sigar* é uma API que permite obter-se informações do computador utilizado independente de plataforma. Dentre essas informações pode-se citar:

- Memória *RAM* utilizada e livre do sistema, memória *swap* utilizada e livre do sistema, percentual de utilização do processador, etc.

- Unidades de disco rígido montadas, com o tipo do sistema de arquivos, tamanho da unidade, espaço utilizado e livre.

- Interfaces de rede, endereço *IP*, máscara de rede, endereço *MAC*, nome do computador, endereços de DNS.

Atualmente as plataformas suportadas são: Linux, FreeBSD, Windows, Solaris, AIX, HP-UX e MacOSX. Além disso, são suportadas diversas versões e arquiteturas das plataformas citadas (HYPERIC, 2009).

Para a implementação desse trabalho, utilizou-se a última versão estável da API (1.6.3), e sua aplicação foi necessária no servidor de armazenamento e na API. Em ambos os casos, a única informação de sistema obtida foi a utilização do disco rígido, mais precisamente, a quantidade de espaço livre em disco.

No servidor de controle, essa informação foi utilizada em duas ocasiões. A primeira delas na configuração do espaço reservado em disco para o sistema, onde há uma validação que impede que seja reservado um espaço em disco superior ao espaço livre. E a segunda aplicação tem por objetivo impedir que seja enviado ao servidor de armazenamento um arquivo com um tamanho superior ao espaço livre em disco.

Da mesma forma que no servidor de armazenamento, na API também foi obtida a quantidade de espaço livre em disco, a fim de impedir que o usuário configure um espaço reservado para a *cache* que seja superior ao espaço livre.

5.5.4 XStream

A XStream é uma biblioteca simples e com objetivo de serializar objetos em XML e posteriormente restaurar os objetos através do XML gerado. Ela é de simples utilização, não exige mapeamentos entre classes e XML, é rápida e possui um baixo consumo de memória (XSTREAM, 2008).

Para o desenvolvimento desse trabalho, essa biblioteca foi utilizada em duas ocasiões. Na primeira delas para a serialização e desserialização das mensagens da camada de comunicação, para que essas pudessem ser transmitidas como texto através dos soquetes de comunicação. Além disso, devido à grande popularidade do formato XML e ao seu padrão aberto, a sua utilização na camada de comunicação permitirá que aplicações de terceiros possam interagir com o sistema realizando trocas de mensagens usando esse padrão. A outra ocasião de utilização dessa biblioteca foi a API, onde é gerado um arquivo XML contendo informações sobre os arquivos que estão armazenados na *cache* local.

5.6 Estudo de Caso: Projeto IndexVideo

Após a implementação de qualquer sistema de computador, torna-se importante realizar testes com o objetivo de avaliar se esse atende as necessidades para as quais foi projetado. Uma das formas de realizar esses testes é a utilização de um cenário concreto de aplicação do sistema que foi desenvolvido, de forma que seja possível avaliar se houveram melhorias com a sua utilização.

Um possível cenário de aplicação do sistema que foi desenvolvido nesse trabalho é o Projeto IndexVideo, que foi melhor descrito na Seção 4.3.3. A ferramenta desenvolvida por esse projeto possui um servidor que é responsável por armazenar os vídeos e as anotações (comentários) dos mesmos; e um cliente que permite a inclusão

dos vídeos, juntamente com suas anotações, a busca de vídeos através das anotações e a exibição dos vídeos.

A utilização do Projeto IndexVideo como estudo de caso nesse trabalho teve como objetivo substituir o armazenamento dos arquivos de vídeo do servidor pela utilização do sistema de armazenamento que foi desenvolvido. Para que isso fosse possível, foi necessário alterar o código fonte do cliente do IndexVideo para que este fizesse uso da API desenvolvida de forma a realizar o armazenamento e a solicitação dos arquivos de vídeo. Dessa forma, o servidor do IndexVideo seria utilizado apenas para o armazenamento das anotações e realização de buscas de vídeos através das anotações.

5.6.1 Alteração do cliente do IndexVideo

Antes da alteração do cliente do IndexVideo foi necessário compreender-se como o sistema foi implementado e como é feita a comunicação com o servidor do IndexVideo. Tanto o cliente como o servidor estão organizados em camadas (pacotes) de acordo com sua finalidade. Dentre as camadas do cliente há uma delas chamada de **Comunicação**, a qual é responsável por realizar a comunicação com o servidor do IndexVideo. Essa comunicação também e feita através de soquetes.

Além da comunicação com o servidor de controle para troca de informações sobre as anotações dos vídeos, a camada de comunicação também é responsável por gerenciar a transferência dos arquivos de vídeo entre o servidor e o cliente. Isso é feito através de duas *threads* chamadas *ThreadEnviaArquivo* e *ThreadRecebeArquivo*, as quais são responsáveis por enviar e receber os arquivos de vídeo respectivamente. A Figura 5.8 ilustra essa estrutura do cliente do IndexVideo.

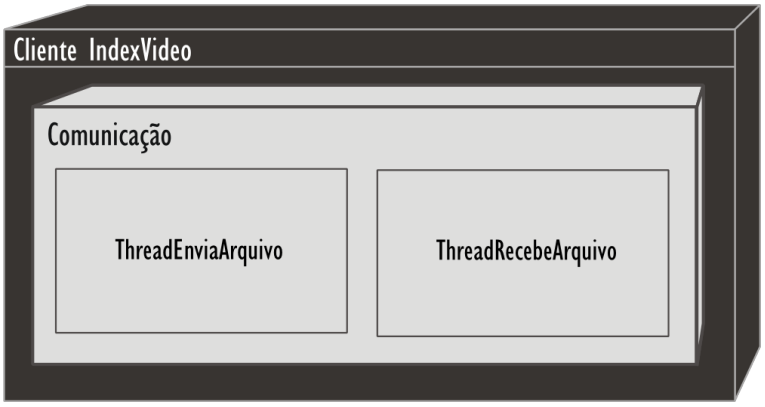

Figura 5.8: Arquitetura simplificada do cliente do IndexVideo.

Para permitir que o cliente do IndexVideo relizasse o gerenciamento dos arquivos de vídeo através da API, foi criada uma classe chamada `KS2APISingleton` dentro do cliente, a qual encapsula a instanciação e configuração da API. Em seguida, as duas *threads* responsáveis pelo envio e recebimento dos arquivos foram alteradas, de

forma a incluir chamadas à API para conectar, enviar/receber o arquivo e por fim desconectar.

Uma questão importante sobre a integração do IndexVideo com o sistema desenvolvido é que o originalmente IndexVideo não permitia a utilização de diretórios para o armazenamento dos vídeos no servidor, ou seja, o usuário apenas adiciona os vídeos ao servidor, sem que seja necessário informar em qual diretório deseja armazená-los. Sendo assim, ao integrar o sistema desenvolvido com o IndexVideo, optou-se por armazenar todos os arquivos de vídeos na raiz, ou seja, sempre que um vídeo é adicionado ao sistema, ele é armazenado no **diretório lógico** representado por `"/"`.

Torna-se importante salientar também que o cliente do IndexVideo já fazia uso de uma *cache* local para o armazenamento dos vídeos. Dessa forma, quando o usuário solicitava um vídeo do servidor, este deveria ser completamente transferido para o cliente antes que começasse a ser exibido. Após a integração com a API, esse procedimento continuou da mesma forma, sendo que a única alteração necessária foi a alteração do arquivo de configuração da API (mais detalhes na Seção 7.3) para que a *cache* local do sistema que foi desenvolvido usasse o mesmo diretório que já era utilizado pelo IndexVideo.

Uma melhoria interessante que pode ser feita no IndexVideo é que ao invés de utilizar uma *cache* local para o armazenamento dos vídeos, o cliente poderia utilizar um fluxo de dados (um *stream*) para exibir o vídeo. Dessa forma, o cliente não precisaria transferir todo o arquivo para a *cache* local antes de iniciar a exibição do vídeo.

Além dessas alterações na camada de comunicação, também foi necessária uma alteração na camada de interface com o usuário. Conforme pôde-se observar na Figura 5.9, a área em destaque que está circulada é utilizada pelo cliente do IndexVideo para exibir a lista dos vídeos que estão armazenados no servidor. Antes da integração com o sistema desenvolvido, essa lista de arquivos era obtida enviando uma solicitação ao servidor do IndexVideo, o qual montava uma lista com todos os arquivos localizados no diretório de vídeos e então a enviava ao cliente. Este por sua vez, atualizava a interface com a lista de arquivos enviada pelo servidor.

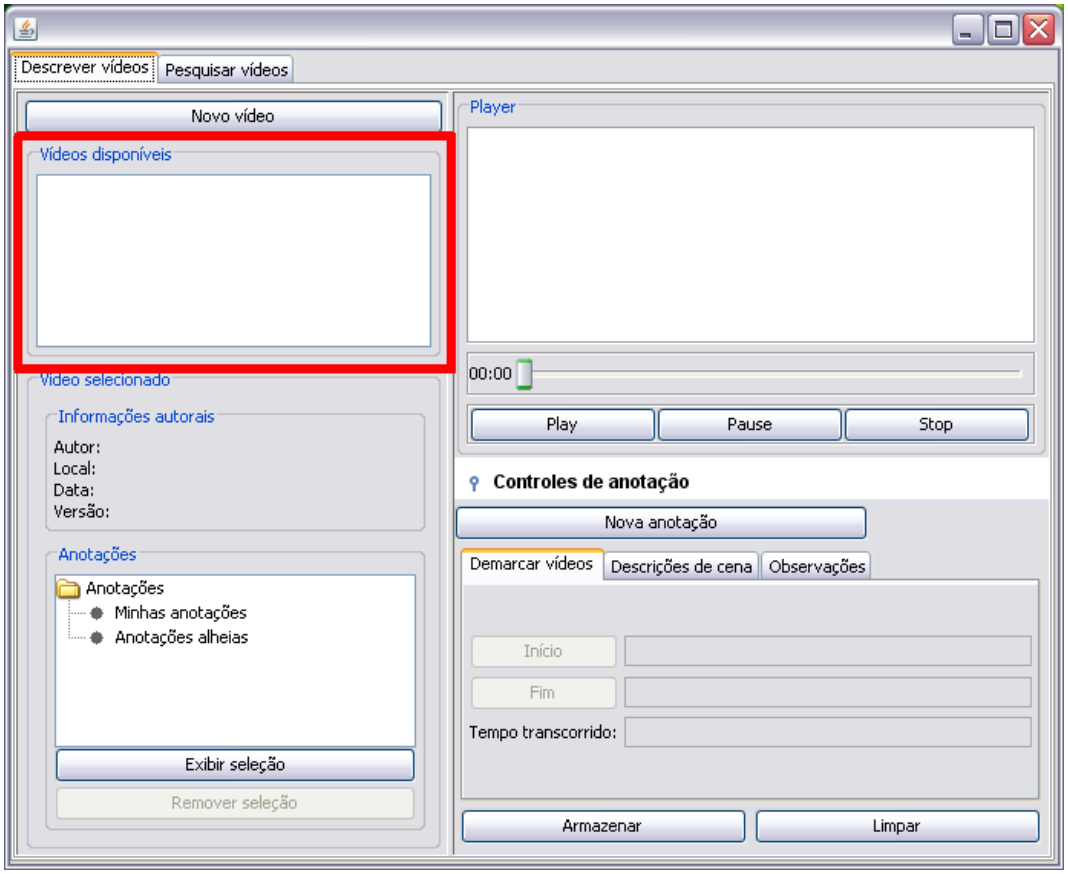

Figura 5.9: Interface do cliente do IndexVideo.

Para a realização da integração com o sistema desenvolvido, o cliente do IndexVideo foi alterado para que, ao invés de solicitar ao servidor do IndexVideo a lista dos vídeos armazenados no servidor, o cliente fizesse uma chamada a API para que esta retornasse a lista dos arquivos localizados no diretório lógico identificado por `"/"`. A partir dessa lista de arquivos retornada pela API, o cliente procede da mesma forma para atualizar a interface com o usuário, exibindo a lista de vídeos disponíveis.

Por fim, foi necessária uma alteração da tela de inclusão de vídeos, onde foi necessário desabilitar a opção de cancelar o envio do vídeo, visto que a API não oferece suporte ao cancelamento de uma operação que já esteja em execução.

5.6.2 Avaliação da integração do IndexVideo com a API

Após a realização da integração do IndexVideo com o sistema que foi desenvolvido, foram feitas algumas avaliações, tanto do processo de alteração do cliente do IndexVideo como de performance, onde foi comparado o tempo de armazenamento e solicitação de arquivos usando a implementação original do IndexVideo e após a integração do sistema que foi desenvolvido.

Com relação à alteração do cliente do IndexVideo, esta foi relativamente simples, uma vez que toda troca de arquivos entre cliente e servidor estavam localizadas em

apenas dois arquivos. Além disso, o fato do IndexVideo já utilizar o conceito de *cache* local também contribuiu para que essa integração ocorresse de forma natural e simplificada.

Outro fator importante que pôde ser observado com a utilização da API no cliente do IndexVideo foi que o código fonte do cliente tornou-se mais claro e simples, visto que a API substituiu uma considerável quantidade de linhas de código que antes eram necessárias para realizar a comunicação com o servidor do IndexVideo e a transferência dos arquivos. Isso permite que o usuário (desenvolvedor) concentre sua atenção na implementação de questões específicas da aplicação, deixando a cargo da API e do sistema de arquivos distribuído o armazenamento e recuperação dos arquivos.

Após a realização dessas alterações no IndexVideo, foi configurado um ambiente de testes com as seguintes características para a realização de um teste de performance: um único computador foi configurado para executar o servidor e o cliente do IndexVideo, um servidor de controle e quatro servidores de armazenamento. O teste baseou-se em cronometrar o tempo necessário para a inclusão de um vídeo de 70 MB através do cliente do IndexVideo, antes e após a sua alteração para utilização da API. Para cada um dos cenários, o teste foi realizado dez vezes e o tempo de cada operação foi cronometrado utilizando o objeto `Datetime` nativo do Java para obter o tempo antes do início e após o término da transferência do arquivo. Em seguida foi feita uma média com os valores obtidos e pôde ser observado que o envio e requisição de arquivos tornou-se em média 30% mais rápido após a integração com a API. Isso se deve principalmente à substituição da transferência do arquivo por soquetes, que antes era feita manualmente, pela utilização do protocolo FTP, usado internamente pela API para a transferência dos arquivos entre o cliente e o servidor de armazenamento.

5.7 Considerações Finais

Neste capítulo foi apresentada da visão geral do sistema, onde foram citados os componentes de *software* que o compõem. Além disso, foi descrito a camada de comunicação utilizada para a realização de trocas de mensagens entre os componentes de *software*, e como ocorre a interação entre esses componentes. Em seguida, a arquitetura de cada componente do sistema foi descrita de forma mais detalhada, onde foram apresentadas as camadas de *software* de cada componente e como estas interagem entre si. Ao final, foram descritas as ferramentas já existentes que foram utilizadas para a implementação do sistema, onde também foi apontado o objetivo a que cada uma delas se propõem e qual o seu relacionamento com o sistema que foi implementado. Além disso, neste capítulo também foi descrito o estudo de caso

do IndexVideo, onde foi apresentado o processo de integração do IndexVideo com a API e os resultados que foram obtidos após essa integração.

6 CONCLUSÕES

Após a realização do presente trabalho de pesquisa e implementação, pode-se concluir que a utilização de sistemas de arquivos distribuídos tem se apresentado como uma alternativa interessante para o armazenamento de grandes quantidades de arquivos. Apesar dos desafios a serem superados na implementação de sistemas desse tipo, as vantagens dessa arquitetura justificam o esforço investido nos projetos de pesquisa e implementação que têm sido desenvolvidos.

Um fator importante que pôde ser percebido através da análise dos sistemas de arquivos distribuídos existentes, é que o objetivo da maioria deles é concentrar arquivos espalhados em uma rede local a fim de apresentá-los ao usuário de forma unificada e centralizada. Existe ainda uma pequena parcela dos sistemas de arquivos distribuídos que se propõem a representar uma solução de armazenamento e distribuição de arquivos, no entanto sua utilização é focada em cenários específicos e são geralmente proprietários.

Em função dessas características, o presente trabalho buscou propor e implementar um novo sistema de arquivos distribuído que fosse transparente, escalável e que permitisse o armazenamento de grandes quantidades de arquivos, sendo esses em sua maioria pequenos e médios. Para atingir esse objetivo, descreveu-se uma proposta de arquitetura e implementou-se um sistema de armazenamento de arquivos de forma distribuída que seguisse essa arquitetura e as demais definições propostas. Sendo que a interação entre o usuário e o sistema de arquivos é feita através de uma API que fornece os métodos necessários para efetuar a manipulação dos arquivos.

O processo de implementação desse novo sistema mostrou-se simples porém trabalhoso. Apesar de terem sido aplicadas tecnologias populares, o conceito de sistemas distribuídos representou um desafio a ser superado. Aliado a isso, a necessidade de desenvolvimento de mais de um componente de *software* exigiu um processo cuidadoso de projeto e arquitetura, onde buscou-se sempre a organização, modularização e clareza.

A utilização de soquetes para a comunicação entre os componentes de *software*, apesar de ter sido desenvolvida com sucesso, mostrou-se trabalhosa e demandou um

considerável tempo de desenvolvimento, o qual poderia ter sido minimizado com a utilização de RMI. Essa questão se mostrou ainda mais acentuada no momento em que os possíveis erros que pudessem vir a ocorrer no servidor de controle tiveram que ser repassados para a API em forma de exceção, de forma que o desenvolvedor pudesse realizar o seu tratamento adequado. Caso fosse utilizado RMI para realização da comunicação, essa passagem de exceções entre servidor e cliente teria ocorrido de forma nativa, tendo em visto o funcionamento dessa tecnologia.

Apesar disso, pode-se afirmar que houve um grande ganho com a utilização de soquetes. Em função de todas as mensagens serem transportadas em formato XML, e esse formato de representação de informações ser aberto, independente de linguagem de programação, simples e popular, permite-se que outros sistemas ou ferramentas possam interagir com o sistema que foi desenvolvido, sem que seja necessário conhecer a implementação do sistema, apenas o formato e os parâmetros das mensagens de requisição e resposta. Uma integração desse nível não seria possível com a utilização de RMI.

Com a aplicação da API desenvolvida no IndexVideo como um estudo de caso, pôde-se confirmar o que já havia sido observado durante a implementação do sistema: a utilização de ferramentas ou API's já existentes facilita o processo de desenvolvimento de aplicações, permitindo que o desenvolvedor mantenha seu foco em questões específicas do sistema que está sendo desenvolvido. Durante o desenvolvimento desse trabalho, diversas API's existentes foram utilizadas, as quais economizaram um tempo considerável de desenvolvimento, visto que abstraem e facilitam a realização de toda uma tarefa, seja ela a geração de *logs*, serialização de objetos em XML, ou o armazenamento de grandes quantidades de arquivos de forma distribuída.

Além disso, avalia-se que o IndexVideo teve um grande valor agregado à sua implementação com a utilização da API, tendo em vista que com pequenas alterações, agora esse sistema é capaz de armazenar seus vídeos de forma distribuída, o que lhe garante escalabilidade e flexibilidade. Também como consequência da utilização da API, o tempo de transferência de arquivos entre o cliente do IndexVideo e os servidores de armazenamento de arquivos tornou-se em média 30% menor, o que representa um tempo significativo para um usuário do sistema.

O tema de sistemas distribuídos fornece um grande nicho de pesquisa, muitas funcionalidades e melhorias ainda podem ser desenvolvidas e agregadas a esse sistema, em função disso, espera-se que trabalhos futuros sejam realizados e deem continuidade ao desenvolvimento dessa sistema. Além disso, as vantagens da utilização de sistemas de arquivos distribuídos sobre sistemas centralizados justificam todo esforço relacionado ao seu desenvolvimento. Por fim, apesar dos grandes desafios relacionados à implementação de sistemas de arquivos distribuídos, considera-se

que o presente trabalho foi capaz de atingir com sucesso todos os objetivos que foram estabelecidos.

6.1 Trabalhos Futuros

Após a conclusão desse trabalho, sugere-se que algumas questões que foram explicitamente desconsideradas, sejam retomadas e agregadas ao sistema. A primeira dessas questões é a garantia de segurança, que poderia ser conseguida através da implementação de mecanismos de autenticação e definição de permissões de acesso aos arquivos.

Além disso, a replicação também seria um recurso interessante a ser incorporado ao sistema, principalmente pela garantia de tolerância a falhas e aumento do desempenho que sua implementação proporciona. Aliada a replicação, pode-se considerar a implementação de um mecanismo de particionamento do arquivo, de forma que um arquivo grande seja divido em várias partes, e cada uma delas seja armazenada em um servidor diferente, o que adicionaria ao sistema a capacidade de manipular de forma mais adequada arquivos de tamanho grande.

Um outra funcionalidade que seria interessante de ser implementada pelo sistema, principalmente no caso específico do Projeto IndexVideo, seria a capacidade de fornecer um fluxo de dados, um *stream* do vídeo que deseja ser visualizado pelo cliente, dessa forma, evita-se que o arquivo de vídeo seja completamente transferido para o computador cliente, e também não é necessário esperar que a cópia do arquivo termine para que este possa começar a ser executado.

REFERÊNCIAS

AMAZON. **Amazon simple storage service**. Disponível em: <http://aws.amazon.com/s3/>. Acesso em: junho 2009.

BUYYA, R. **High Performance Cluster Computing**: Architecture and Systems. [S.l.]: Prentice Hall, 1999. v.1.

CALLAGHAN, B. et al. **Nfs version 3 protocol specification**. RFC 1813. Disponível em: <http://www.ietf.org/rfc/rfc1813.txt>. Acesso em: junho 2009.

CARVALHO, A. C. P. L. F. et al. **Grandes desafios da pesquisa em computação no brasil 2006 2016** : relatório sobre o seminário realizado em 8 e 9 de maio de 2006.

CHENG, X.; DALE, C.; LIU, J. **Understanding the characteristics of internet short video sharing**: youtube as a case study. 2007. Artigo — Simon Fraser University, Burnaby, BC, Canada. Disponível em: <http://arxiv.org/PS_cache/arxiv/pdf/0707/0707.3670v1.pdf>. Acesso em: junho 2009.

CHUTE, C. et al. **The Diverse and Exploding Digital Universe**.

CORPORATION, M. **Sql server developer center**. Disponível em: <http://msdn.microsoft.com/pt-br/sqlserver/default.aspx>. Acesso em: novembro 2009.

COSCARELLI, C. V. O uso da informática como instrumento de ensino-aprendizagem. **Presença Pedagógica**, v.4, n.20, p.37–45, Mar./ 1998.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas Distribuídos**: conceitos e projeto. 4.ed. Porto Alegre: Bookman, 2007. Tradução João Tortello.

FARLEY, J. **Java distributed computing**. 1.ed. Sebastopol, CA: O'Reilly & Associates, Inc., 1998.

FOSTER, I.; KESSELMAN, C.; TUECK, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. **The International Journal of High Performance Computing Applications**, v.15, n.3, p.220–222, 2001.

FOSTER, I. T.; (EDS), C. K. **The Grid**: Blueprint for a new Computing Infrastructure. [S.l.]: Morgan Kaufmann, 1999.

FOUNDATION, T. A. S. **Apache log4j**. Disponível em: <http://logging.apache.org/log4j/>. Acesso em: novembro 2009.

FOUNDATION, T. A. S. **Jakarta commons net**. Disponível em: <http://commons.apache.org/net/>. Acesso em: novembro 2009.

FOUNDATION, T. A. S. **Welcome to apache hadoop!** Disponível em: <http://hadoop.apache.org/>. Acesso em: junho 2009.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. **The google file system**. 2003. Artigo — Google, Mountain View, CA. Disponível em: <http://labs.google.com/papers/gfs-sosp2003.pdf>. Acesso em: junho 2009.

GILL, P.; ARLITT, M.; LI, Z.; MAHANTI, A. Youtube traffic characterization: a view from the edge. In: IMC '07: PROCEEDINGS OF THE 7TH ACM SIGCOMM CONFERENCE ON INTERNET MEASUREMENT, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.15–28.

HOWARD, J. H. **An overview of the andrew file system**. 1988. Artigo — Carnegie Mellon University, Pittsburgh, PA. Disponível em: <http://reports-archive.adm.cs.cmu.edu/anon/anon/home/ftp/itc/CMU-ITC-062.pdf>. Acesso em: junho 2009.

HYPERIC. **Hyperic sigar api**. Disponível em: <http://www.hyperic.com/products/sigar.html>. Acesso em: novembro 2009.

Microsoft Corporation. **What is dfs?** Disponível em: <http://technet.microsoft.com/en-us/library/cc779627(WS.10).aspx>. Acesso em: junho 2009.

Microsoft Corporation. **Overview of the distributed file system solution in microsoft windows server 2003 r2**. Disponível em: <http://www.microsoft.com/downloads/details.aspx?familyid=5e547c69-d224-4423-8eac-18d5883e7bc2&displaylang=en>. Acesso em: junho 2009.

NEUMAN, C. et al. **The kerberos network authentication service (v5)**. RFC 4120. Disponível em: <http://www.ietf.org/rfc/rfc4120.txt>. Acesso em: junho 2009.

POSTEL, J.; REYNOLDS, J. **File transfer protocol (ftp)**. RFC 959. Disponível em: <http://www.ietf.org/rfc/rfc959.txt>. Acesso em: junho 2009.

PRADO LIMA, M. F. W.; SILVA, J. L. T. d.; CARBONERA, J. L. **A semantic annotation tool for educational video retrieval**. In: 9th IFIP World Conference on Computers in Education (WCCE 2009), 2009, Bento Gonçalves. 9th IFIP World Conference on Computers in Education (WCCE 2009), 2009.

QUINN, B.; SHUTE, D. **Windows sockets network programming**. [S.l.]: Addison Wesley, 1996.

RAMEZ, E.; NAVATHE, S. B. **Sistemas de banco de dados**. 4.ed. São Paulo: Addison Wesley, 2005.

SHEPLER, S. et al. **Network file system (nfs) version 4 protocol**. RFC 3530. Disponível em: <http://www.ietf.org/rfc/rfc3530.txt>. Acesso em: junho 2009.

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Sistemas Operacionais**: conceitos e aplicações. Rio de Janeiro: Elsvier, 2000. Tradução Adriana Rieche.

SRINIVASAN, R. et al. **Rpc**: remote procedure call protocol specification version 2. RFC 1831. Disponível em: <http://www.ietf.org/rfc/rfc1831.txt>. Acesso em: junho 2009.

STERN, H.; EISLER, M.; LABIAGA, R. **Managing nfs and nis**. 2.ed. [S.l.]: O'Reilly, 2001. Disponível em: <http://oreilly.com/catalog/9781565925106/preview.html>. Acesso em: junho 2009.

SUN MICROSYSTEMS, I. **Mysql** :: the world's most popular open source database. Disponível em: <http://www.mysql.com/>. Acesso em: novembro 2009.

SUN MICROSYSTEMS, I. **Lesson**: exceptions. Disponível em: <http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html>. Acesso em: novembro 2009.

TANENBAUM, A. **Distributed operating systems**. New Jersey: Person Prentice Hall, 1995. ISBN 0-13-219908-4.

TANENBAUM, A. **Sistemas Operacionais Modernos**. 2.ed. São Paulo: Person Prentice Hall, 2003. Tradução Ronaldo A. L. Golçalves, Luís A. Consularo.

TANENBAUM, A.; STEEN, M. V. **Sistemas Distribuídos**: princípios e paradigmas. 2.ed. São Paulo: Person Prentice Hall, 2007. Tradução Arlete Simili Marques.

TICHY, W. F.; RUAN, Z. **Towards a distributed file system**. Disponível em: <http://www.cs.purdue.edu/research/technical_reports/1984/TR%2084-480.pdf>. Acesso em: junho 2009.

XSTREAM. **About xstream**. Disponível em: <http://xstream.codehaus.org/index.html>. Acesso em: novembro 2009.

7 ANEXOS

7.1 Manual de instalação do Sistema

Para que seja feita a instalação do sistema desenvolvido, inicialmente deve-se configurar o servidor de controle. Em seguida, devem ser configurados quantos servidores de armazenamento forem necessários. Além disso, novos servidores de armazenamento pode ser adicionados ao sistema a qualquer momento.

Para configurar o servidor de controle, um banco de dados deverá ser criado (*MySQL* ou *Microsoft SQL Server*), e os arquivos executáveis deverão ser copiados e configurados. Já para o servidor de armazenamento, um servidor de FTP deverá ser instalado e os arquivos executáveis deverão ser copiados e configurados. Todos os passos para a instalação e configuração do servidor de controle e dos servidores de armazenamento serão descrito na seção a seguir.

7.1.1 Instalação e Configuração do Servidor de Controle

7.1.1.1 Pré-requisitos

O servidor de controle possui alguns pré-requisitos de *software* para sua instalação. As instruções de instalação desses pré-requisitos não serão descritas nesse manual. Os pré-requisitos são:

- O Java JDK ou JRE 6 deve estar instalado na mesma máquina onde o servidor de controle será instalado.
- Servidor de banco de dados *MySQL* 5 ou *Microsoft SQL Server 2005*, podendo estar instalado na mesma máquina em uma máquina diferente do servidor de controle.

7.1.1.2 Criação do banco de dados para um servidor MySQL

Para instalar o sistema em um servidor de banco de dados *MySQL*, crie um banco de dados com o nome de sua preferência e em seguida execute o seguinte *script* sobre esse banco.

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;

/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;

--
-- Definition of table `storageserver`
--

DROP TABLE IF EXISTS `storageserver`;
CREATE TABLE `storageserver` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `name` varchar(50) NOT NULL,
  `host` varchar(50) NOT NULL,
  `servicePort` int(10) NOT NULL,
  `ftpUser` varchar(50) NOT NULL,
  `ftpPassword` varchar(50) NOT NULL,
  `ftpWorkingDir` varchar(255) DEFAULT NULL,
  `storedFilesPath` varchar(255) NOT NULL,
  `blockSize` int(10) NOT NULL,
  `reservedSpace` bigint(19) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=16 DEFAULT CHARSET=latin1;

--
-- Definition of table `file`
--

DROP TABLE IF EXISTS `file`;
CREATE TABLE `file` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `storageServerId` int(10) NOT NULL,
  `valid` tinyint(4) NOT NULL,
  `locked` tinyint(4) NOT NULL,
  `lockDate` datetime DEFAULT NULL,
  `path` varchar(255) NOT NULL,
  `name` varchar(255) NOT NULL,
  `size` int(10) NOT NULL,
  `version` int(10) NOT NULL,
  `uploadDate` datetime DEFAULT NULL,
  `updateDate` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_File_StorageServer` (`storageServerId`),
  CONSTRAINT `FK_File_StorageServer` FOREIGN KEY (`storageServerId`)
  REFERENCES `storageserver` (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=33 DEFAULT CHARSET=latin1;

--
-- Definition of table `transfer`
--

DROP TABLE IF EXISTS `transfer`;
CREATE TABLE `transfer` (
  `id` bigint(19) NOT NULL AUTO_INCREMENT,
  `fileId` int(10) NOT NULL,
  `type` char(1) NOT NULL,
  `beginDate` datetime NOT NULL,
  `endDate` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_Transfer_Transfer` (`fileId`),
  CONSTRAINT `FK_Transfer_Transfer` FOREIGN KEY (`fileId`)
  REFERENCES `file` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=35 DEFAULT CHARSET=latin1;

--
-- Definition of function `getOpenedTransfersForStorageServer`
--

DROP FUNCTION IF EXISTS `getOpenedTransfersForStorageServer`;

DELIMITER $$

/*!50003 SET @TEMP_SQL_MODE=@@SQL_MODE,
SQL_MODE='STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION' */ $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`getOpenedTransfersForStorageServer`(storageServerId INT) RETURNS int(11)
```

```
BEGIN

DECLARE openedTransfers INT;

SELECT COUNT(Transfer.id)
INTO openedTransfers
FROM Transfer
INNER JOIN File
        ON File.id = Transfer.fileId
WHERE File.storageServerId = storageServerId
        AND Transfer.endDate IS NULL;

RETURN openedTransfers;

END $$
/*!50003 SET SESSION SQL_MODE=@TEMP_SQL_MODE */  $$

DELIMITER ;

--
-- Definition of function `getUsedSpaceForStorageServer`
--

DROP FUNCTION IF EXISTS `getUsedSpaceForStorageServer`;

DELIMITER $$

/*!50003 SET @TEMP_SQL_MODE=@@SQL_MODE,
SQL_MODE='STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION' */ $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`getUsedSpaceForStorageServer`(storageServerId INT) RETURNS bigint(20)
BEGIN

DECLARE usedSpace INT;

SELECT  SUM(
  CEILING( `File`.`size` / CAST(`StorageServer`.`blockSize` AS DECIMAL(18,2)))
  * `StorageServer`.`blockSize`
)
INTO usedSpace
FROM `File`
INNER JOIN `StorageServer`
        ON `StorageServer`.`id` = `File`.`storageServerId`
WHERE `File`.`storageServerId` = storageServerId;

RETURN usedSpace;

END $$
/*!50003 SET SESSION SQL_MODE=@TEMP_SQL_MODE */  $$

DELIMITER ;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
```

7.1.1.3 Criação do banco de dados para um servidor Microsoft SQL Server

Para instalar o sistema em um servidor de banco de dados *Microsoft SQL Server*, crie um banco de dados com o nome de sua preferência e em seguida execute o seguinte *script* sobre esse banco.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[StorageServer]') AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[StorageServer](
  [id] [int] IDENTITY(1,1) NOT NULL,
  [name] [varchar](50) NOT NULL,
```

```
  [host] [varchar](50) NOT NULL,
  [servicePort] [int] NOT NULL,
  [ftpUser] [varchar](50) NOT NULL,
  [ftpPassword] [varchar](50) NOT NULL,
  [ftpWorkingDir] [varchar](255) NOT NULL,
  [storedFilesPath] [varchar](255) NOT NULL,
  [blockSize] [int] NOT NULL,
  [reservedSpace] [bigint] NOT NULL,
 CONSTRAINT [PK_ServidorArmazenamento] PRIMARY KEY CLUSTERED
(
  [id] ASC
)WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
END
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Transfer]') AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Transfer](
  [id] [bigint] IDENTITY(1,1) NOT NULL,
  [fileId] [int] NOT NULL,
  [type] [char](1) NOT NULL,
  [beginDate] [datetime] NOT NULL,
  [endDate] [datetime] NULL,
 CONSTRAINT [PK_Transfer] PRIMARY KEY CLUSTERED
(
  [id] ASC
)WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
END
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[File]') AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[File](
  [id] [int] IDENTITY(1,1) NOT NULL,
  [storageServerId] [int] NOT NULL,
  [valid] [bit] NOT NULL,
  [locked] [bit] NOT NULL,
  [lockDate] [datetime] NULL,
  [path] [varchar](255) NOT NULL,
  [name] [varchar](255) NOT NULL,
  [size] [int] NOT NULL,
  [version] [int] NOT NULL,
  [uploadDate] [datetime] NULL,
  [updateDate] [datetime] NULL,
 CONSTRAINT [PK_File] PRIMARY KEY CLUSTERED
(
  [id] ASC
)WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
END
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[getUsedSpaceForStorageServer]')
AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))
BEGIN
execute dbo.sp_executesql @statement = N'CREATE FUNCTION [dbo].[getUsedSpaceForStorageServer]
(
  @storageServerId int
)
RETURNS int
AS
BEGIN
  DECLARE @usedSpace BIGINT

  SELECT @usedSpace =
  (
```

```
      SELECT  SUM(
        CEILING( [File].[size] / CAST([StorageServer].[blockSize] AS FLOAT))
        * [StorageServer].[blockSize]
      )
      FROM [File]
      INNER JOIN [StorageServer]
        ON [StorageServer].[id] = [File].[storageServerId]
      WHERE [File].[storageServerId] = @storageServerId
  )

  RETURN @usedSpace

END
'
END

GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[getOpenedTransfersForStorageServer]')
AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))
BEGIN
execute dbo.sp_executesql @statement = N'CREATE FUNCTION [dbo].[getOpenedTransfersForStorageServer]
(
  @storageServerId int
)
RETURNS int
AS
BEGIN
  DECLARE @openedTransfers INT

  SELECT @openedTransfers =
    (
      SELECT COUNT([Transfer].[id])
      FROM [Transfer]
      INNER JOIN [File]
        ON [File].[id] = [Transfer].[fileId]
      WHERE [File].[storageServerId] = @storageServerId
        AND [Transfer].[endDate] IS NULL
    )

  RETURN @openedTransfers

END


'
END

GO
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_Transfer_Transfer]')
AND parent_object_id = OBJECT_ID(N'[dbo].[Transfer]'))
ALTER TABLE [dbo].[Transfer]  WITH CHECK ADD  CONSTRAINT [FK_Transfer_Transfer] FOREIGN KEY([fileId])
REFERENCES [dbo].[File] ([id])
ON UPDATE CASCADE
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[Transfer] CHECK CONSTRAINT [FK_Transfer_Transfer]
GO
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_File_StorageServer]')
AND parent_object_id = OBJECT_ID(N'[dbo].[File]'))
ALTER TABLE [dbo].[File]  WITH CHECK ADD  CONSTRAINT [FK_File_StorageServer] FOREIGN KEY([storageServerId])
REFERENCES [dbo].[StorageServer] ([id])
GO
ALTER TABLE [dbo].[File] CHECK CONSTRAINT [FK_File_StorageServer]
```

7.1.1.4 Instalação e Configuração do Servidor de Controle

A instalação do servidor de controle compreende o seguintes arquivos:

ControlServer.jar: pacote Java que contém o código compilado do servidor de

controle.

ControlServer.config: arquivo de propriedades para a configuração do servidor de controle.

ControlServer.log4j.config: arquivo de propriedades para a configuração do Log4j para os *logs* do servidor de controle.

Esses três arquivos devem ser copiados para o sistema de arquivos do computador onde o servidor irá executar. Não há uma localização específica para esses arquivos, dessa forma, fica a critério do usuário que está realizando a instalação o local onde devem ser copiados.

Após realizar a cópia dos arquivos, edite o arquivo `ControlServer.config`, para que seja possível realizar a configuração do servidor de controle. A listagem a seguir apresenta o arquivo de configuração padrão do servidor de controle:

```
1  # ========================================================================
2  # KS2 Control Server Configuration File
3  # ========================================================================
4
5  # ------------------------------------------------------------------------
6  # Control Server Configuration
7  # ------------------------------------------------------------------------
8
9  # Database Server Type
10 # mysql OR sqlserver
11 ControlServer.DatabaseServerType = mysql
12
13 # JDBC Driver
14 # For MS SQL Server
15 # ControlServer.JDBCDriver = com.microsoft.sqlserver.jdbc.SQLServerDriver
16 # For My SQL Server
17 ControlServer.JDBCDriver = com.mysql.jdbc.Driver
18
19 # Database server
20 ControlServer.DatabaseServer = 127.0.0.1
21
22 # Database port
23 # For MS SQL Server
24 # ControlServer.DatabasePort = 1433
25 # For My SQL Server
26 ControlServer.DatabasePort = 3306
27
28 # Database name
29 ControlServer.DatabaseName = ks2db
30
31 # Database user
32 ControlServer.DatabaseUser = ks2
33
34 # Database password
35 ControlServer.DatabasePassword = ks2
36
37 # Control Server Service Port
38 ControlServer.Port = 9001
```

```
39
40 # ------------------------------------------------------------------------
41 # Log4j Configuration
42 # ------------------------------------------------------------------------
43
44 # Log4j Configuration File Path
45 Log4j.ConfigFile = /opt/ks2/control_server/ControleServer.log4j.config
```

As opções do arquivo de configuração são:

ControlServer.DatabaseServerType: indica o tipo de servidor de banco de dados que será usado para o armazenamento das informações do arquivos. As opções disponíveis são **mysql** e **sqlserver**.

ControlServer.JDBCDriver: caminho do driver JDBC para a conexão com o banco de dados. Caso seja utilizado um servidor *Microsoft SQL Server*, descomente a linha 15 e comente a linha 17. Caso deseje utilizar um servidor *MySQL*, mantenha a configuração padrão.

ControlServer.DatabaseServer: endereço IP do computador onde o servidor de banco de dados está instalado.

ControlServer.DatabasePort: número da porta onde o serviço do servidor de banco de dados está executando. Caso deseje utilizar um servidor *MySQL*, mantenha a configuração padrão. Caso deseje utilizar um servidor *Microsoft SQL Server*, descomente a linha 24 e comente a linha 26. É importante salientar que os números de porta que estão no arquivo de configuração são os padrões sugeridos pela instalação do servidor de banco de dados. No entanto esses números podem ser alterados. Sendo assim, é importante certificar-se do número da porta em que o serviço está executando, de modo a fazer a correta configuração do servidor de controle.

ControlServer.DatabaseName: nome do banco de dados, o qual foi definido pelo usuário que fez a criação do banco.

ControlServer.DatabaseUser: usuário de banco de dados que tenha permissão de leitura e escrita no banco de dados do servidor de controle.

ControlServer.DatabasePassword: senha do usuário de banco de dados.

ControlServer.Port: número da porta onde o serviço do servidor de controle estará executando. O padrão é 9001, mas pode ser alterado para qualquer valor, desde que esse número já não esteja sendo utilizado por outra aplicação. Esse número de porta deverá informando no arquivo de configuração da API, conforme descrito na Seção 7.3.

Log4j.ConfigFile: caminho para o arquivo de configuração do *Log4j*.

Além da configuração do servidor de controle, opcionalmente, pode-se alterar a configuração dos *logs* gerados pelo servidor. Para isso edite o arquivo `ControlServer.log4j.config`. Para maiores informações sobre a configuração do *Log4j* sugere-se a documentação oficial do projeto que está disponível em (FOUNDATION, 2007).

Após a realização dessas configurações, já é possível iniciar o servidor de controle. Para isso, é necessário executar o arquivo `ControlServer.jar` através do *prompt* de comandos do sistema operacional, passando como parâmetro o caminho do arquivo de configuração do servidor de controle. Supondo que os três arquivos (`ControlServer.jar`, `ControlServer.config` e `ControlServer.log4j.config`) estejam dentro do mesmo diretório, por exemplo `E:\ks2`, o comando para executar o servidor a partir do *prompt* de comandos do sistema operacional seria:

```
java -jar E:\ks2\ControlServer.jar E:\ks2\ControlServer.config
```

7.1.2 Instalação e Configuração do Servidor de Armazenamento

7.1.2.1 Pré-requisitos

O servidor de armazenamento possui alguns pré-requisitos de *software* para sua instalação. As instruções de instalação desses pré-requisitos não serão descritas nesse manual. Os pré-requisitos são:

- O Java JDK ou JRE 6 deve estar instalado na mesma máquina onde o servidor de armazenamento será instalado.
- Um servidor de FTP deve estar instalado e configurado na mesma máquina onde o servidor de armazenamento irá executar. Além disso, já deve existir um usuário FTP criado, preferencialmente com senha.

7.1.2.2 Instalação e Configuração do Servidor de Armazenamento

A instalação do servidor de armazenamento compreende o seguintes arquivos:

StorageServer.jar: pacote Java que contém o código compilado do servidor de armazenamento.

StorageServer.config: arquivo de propriedades para a configuração do servidor de armazenamento.

StorageServer.log4j.config: arquivo de propriedades para a configuração do Log4j para os *logs* do servidor de controle.

Esses três arquivos devem ser copiados para o sistema de arquivos do computador onde o servidor irá executar. Não há uma localização específica para esses arquivos, dessa forma, fica a critério do usuário que está realizando a instalação o local onde devem ser copiados.

Após realizar a cópia dos arquivos, edite o arquivo `StorageServer.config`, para que seja possível realizar a configuração do servidor de armazenamento. A listagem a seguir apresenta o arquivo de configuração padrão do servidor de armazenamento:

```
1  # ==========================================================================
2  # KS2 Storage Server Configuration File
3  # ==========================================================================
4
5  # --------------------------------------------------------------------------
6  # Storage Server Configuration
7  # --------------------------------------------------------------------------
8
9  # Port
10 StorageServer.Port = 8001
11
12 # Full path to the directory where files will be stored
13 StorageServer.StoredFilesPath = /opt/ks2/storage_server/files
14
15 # --------------------------------------------------------------------------
16 # Log4j Configuration
17 # --------------------------------------------------------------------------
18
19 # Log4j Configuration File Path
20 Log4j.ConfigFile = /opt/ks2/storage_server/StorageServer.log4j.config
```

As opções do arquivo de configuração são:

StorageServer.Port: número da porta onde o serviço do servidor de armazenamento estará executando. O padrão é 8001, mas pode ser alterado para qualquer valor, desde que esse número já não esteja sendo utilizado por outra aplicação.

StorageServer.StoredFilesPath: caminho completo para o diretório onde os arquivos serão armazenados dentro do servidor de armazenamento. É importante salientar que o usuário de FTP deve ter permissão de leitura e escrita nesse diretório.

Log4j.ConfigFile: caminho para o arquivo de configuração do *Log4j*.

Além da configuração do servidor de armazenamento, opcionalmente, pode-se alterar a configuração dos *logs* gerados pelo servidor. Para isso edite o arquivo `StorageServer.log4j.config`. Para maiores informações sobre a configuração do *Log4j* sugere-se a documentação oficial do projeto que está disponível em (FOUNDATION, 2007).

Após a realização dessas configurações, já é possível iniciar o servidor de armazenamento. Para isso, é necessário executar o arquivo `StorageServer.jar` através do

prompt de comandos do sistema operacional, passando como parâmetro o caminho do arquivo de configuração do servidor de armazenamento. Supondo que os três arquivos (`StorageServer.jar`, `StorageServer.config` e `StorageServer.log4j-.config`) estejam dentro do mesmo diretório, por exemplo `E:\ks2`, o comando para executar o servidor a partir do *prompt* de comandos do sistema operacional seria:

```
java -jar E:\ks2\StorageServer.jar E:\ks2\StorageServer.config
```

Para finalizar a instalação do servidor de armazenamento, é necessário adicioná-lo ao servidor de controle, para que este saiba da existência do novo servidor e possa então passar a utilizá-lo para armazenar os arquivos. Para adicionar o servidor de armazenamento ao servidor de controle, deve-se utilizar o **Administrador**, consultando a Seção 7.2.4 do Anexo 7.2.

7.2 Manual de utilização do Administrador

O **Administrador** é uma ferramenta simples que permite o gerenciamento dos servidores de armazenamento e análise da carga do sistema, consultando o espaço reservado e disponível para cada servidor, além de permitir visualizar as transferências de arquivo que cada servidor de armazenamento está executando. Cada uma dessas funcionalidades será descrita nas seções a seguir.

7.2.1 Executando o Administrador

O Administrador foi desenvolvido usando a Linguagem Java e seus arquivos binários foram empacotados em um arquivo no padrão JAR. Sendo assim, para que seja possível executar o Administrador, é necessário que o computador tenha instalado o Java JDK ou JRE 6 ou superior. Após isso, o Administrador poderá ser executado pelo seguinte comando:

```
java -jar Admin.jar
```

Todos as operações realizadas no Administrador, bem como possíveis mensagens de erro, são armazenadas em um arquivo de *logs* através do Log4j. Opcionalmente, ao executar o Administrador, pode-se informar o caminho do arquivo de configuração do *Lo4j*. Se for esse o caso, o comando para executar o Administrador seria:

```
java -jar Admin.jar log4j.config
```

Após a execução de qualquer um dos comandos acima citados, o Administrador é iniciado, e a janela de conexão com o servidor de controle é apresentada.

7.2.2 Conectando a um Servidor de Controle

O Administrador pode ser ser usado para gerenciar qualquer servidor de controle. Para isso, deve-se informar o endereço IP e número da porta onde o servidor de controle está sendo executado. A Figura 7.1 ilustra a tela de conexão do Administrador.

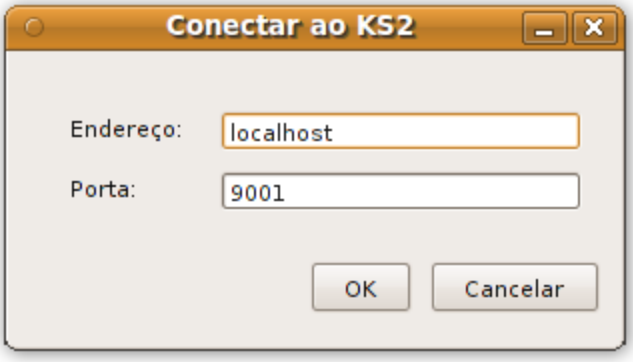

Figura 7.1: Tela de conexão ao servidor de controle.

Após preencher os campos com o endereço IP e porta do servidor de controle, basta clicar em "OK"e o Administrador será iniciado.

7.2.3 Listando os Servidores de Armazenamento

Através do Administrador é possível consultar os servidores de armazenamento que foram configurados para serem utilizado pelo sistema. Isso pode ser feito acessando o menu "Arquivo", "Lista de servidores de armazenamento". Ao selecionar essa opção, será exibida uma nova janela listando os servidores de armazenamento do sistema. Os servidores são apresentados em formato de tabela, onde cada linha representa um servidor. Nessa tela é possível consultar o nome do servidor (um nome simbólico, não o nome do computador), o seu endereço IP, a porta em que o serviço está executando e a quantidade de espaço reservado para esse servidor (em *bytes*). A Figura 7.2 ilustra a tela de listagem dos servidores de armazenamento.

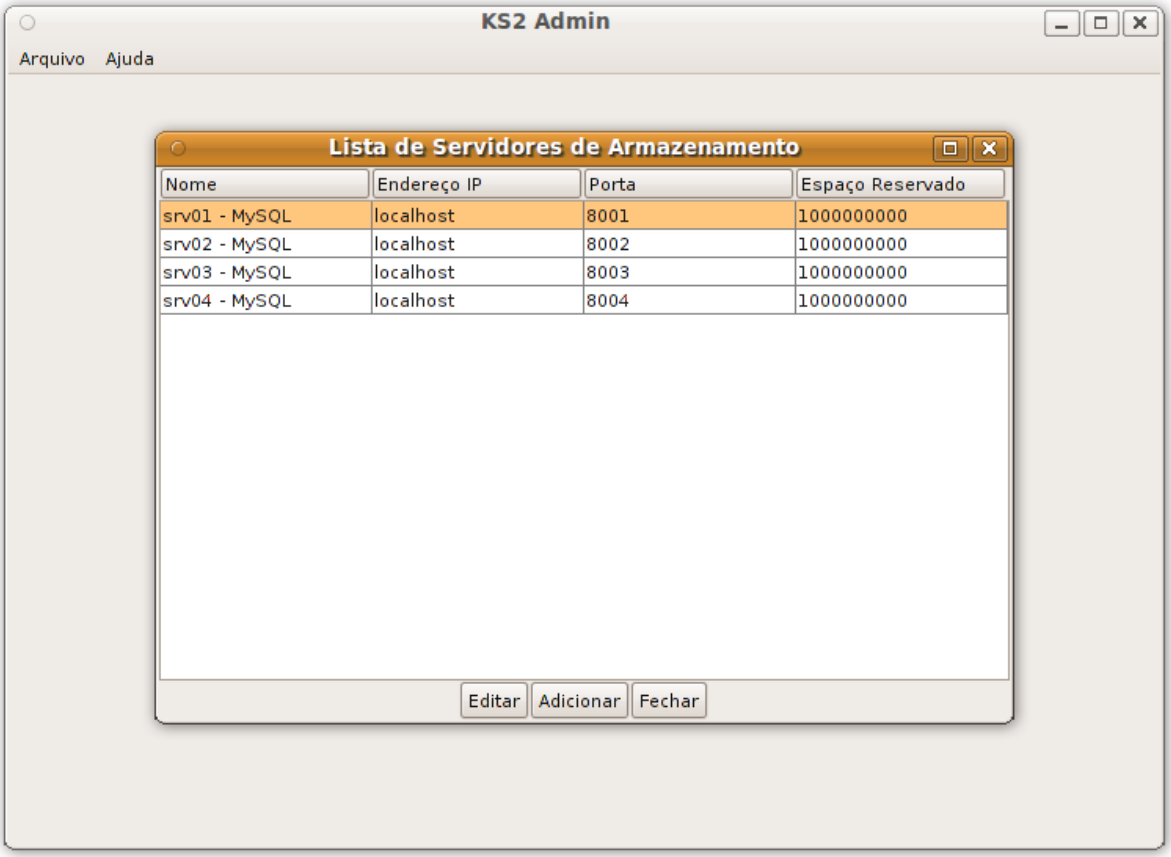

Figura 7.2: Lista dos servidores de armazenamento.

A partir dessa tela é possível editar um servidor existente ou adicionar um novo servidor. Essas funcionalidades são descritas na seção a seguir.

7.2.4 Adicionando ou editando um Servidor de Controle

Para incluir um novo servidor de armazenamento, pode-se acessar o menu "Arquivo", "Adicionar servidor de armazenamento", ou através do botão "Adicionar"localizado na tela de listagem dos servidores de armazenamento. Já para editar os dados de um servidor de armazenamento, deve acessar o menu "Arquivo", "Lista de servidores de armazenamento", selecionar o servidor que se deseja editar, e então clicar sobre o botão "Editar".

Tanto para a opção de inclusão de um novo servidor como de edição de um servidor existente, a tela e as validações são as mesmas. A Figura 7.3 ilustra a tela de edição de um servidor de armazenamento existente.

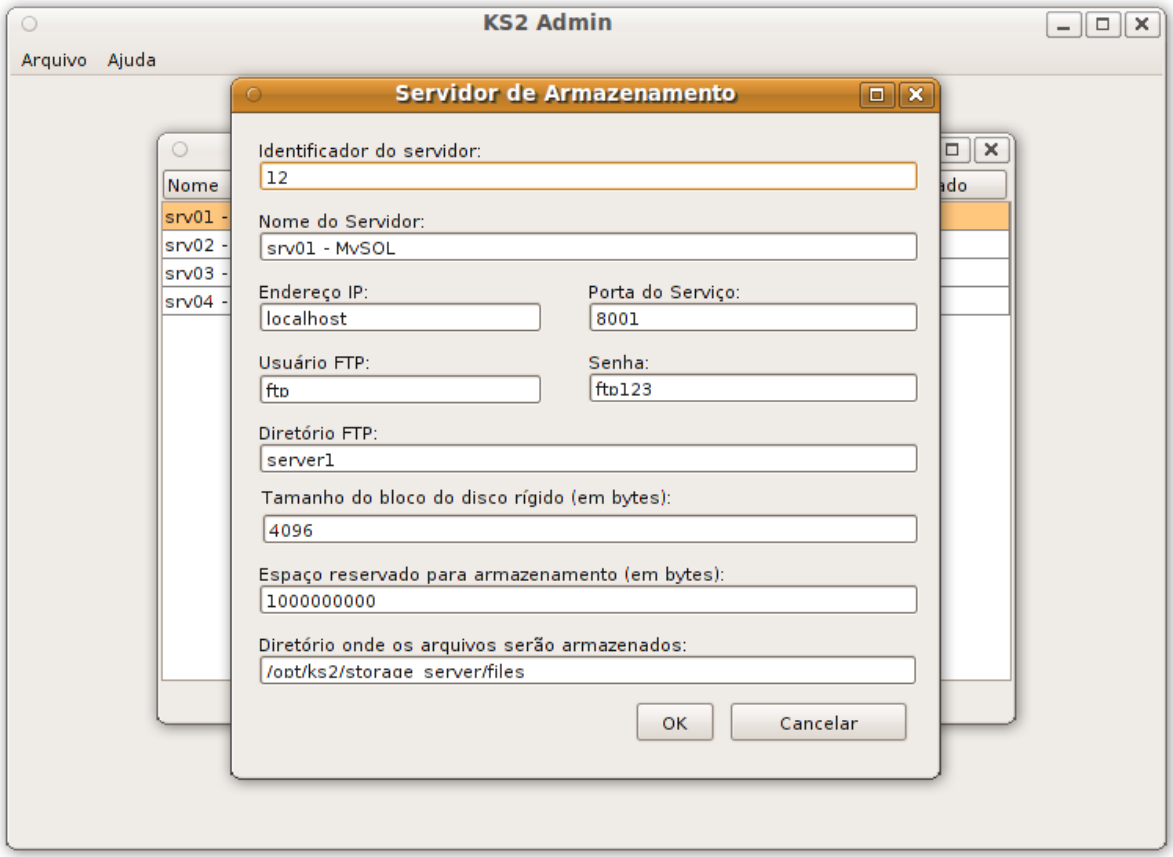

Figura 7.3: Edição de um servidor de armazenamento existente.

Os campos a serem informados para o servidor de armazenamento são os seguintes:

Identificador do servidor: esse campo não precisa ser preenchido, ele será gerado automaticamente quando o servidor de armazenamento for adicionado ao servidor de controle. Durante a edição de um servidor, esse campo não pode ser alterado.

Nome do servidor: um nome simbólico para ser usando na listagem dos servidores.

Endereço IP: O endereço IP ou nome do *host* onde o servidor de armazenamento está instalado.

Porta do serviço: número da porta onde o serviço do servidor de armazenamento está executando. Esse número é definido no arquivo de configuração do servidor de armazenamento, conforme descrito na Seção 7.1.2.2.

Usuário FTP: nome do usuário FTP que será usado para conectar ao servidor de armazenamento para realizar as transferências dos arquivos.

Senha: Senha do usuário informado no campo Usuário FTP.

Diretório FTP: quando o usuário FTP efetua o *login* no servidor de FTP, ele é direcionado para seu diretório de trabalho (diretório inicial ou diretório *home*), que foi definido quando o usuário foi adicionado ao servidor de FTP. Se nada for informado nesse campo, os arquivos serão transferidos para o diretório de trabalho do usuário. No entanto, se for informado um caminho de diretório válido, quando o usuário FTP efetuar o *login* no servidor de FTP, ele irá executar um comando para trocar seu diretório de trabalho para o caminho informado nesse campo.

Tamanho do bloco do disco rígido: discos rígidos são segmentados em blocos para realizar o armazenamento dos arquivos, sendo que o tamanho desses blocos pode variar de computador para computador. Aqui deve ser informado o tamanho do bloco (em *bytes*) do disco rígido onde o servidor de armazenamento irá armazenar os arquivos que para ele forem transferidos. Esse campo não pode ser editado depois que o servidor de armazenamento for salvo.

Espaço reservado para o armazenamento: nesse campo é informado (em *bytes*) a quantidade de espaço em disco que se deseja reservar para o esse servidor de armazenamento. Após a inclusão do servidor, esse valor pode ser aumentado, desde que o servidor de armazenamento possua espaço em disco, ou diminuído, desde que o servidor ainda não esteja utilizando esses espaço.

Diretório onde os arquivos serão armazenados: aqui deve-se informar o caminho completo onde os arquivos serão armazenados no servidor de armazenamento. É importante salientar que esse campo deve possuir exatamente o mesmo valor que foi definido no parâmetro "`StorageServer.StoredFilesPath`", dentro do arquivo de configuração do servidor de armazenamento. Além disso, também deve ser o mesmo diretório que o diretório de trabalho do usuário FTP

já definido. Esse campo não pode ser editado depois que o servidor de armazenamento for salvo.

Após o preenchimento desses campos, basta clicar em "OK"para que o servidor de armazenamento seja adicionado ao sistema. Nesse momento uma série de validações será efetuada, dentre elas:

- Todos os campos do formulário devem estar corretamente preenchidos.
- O servidor de FTP deve estar executando no endereço informado para o servidor de armazenamento, e o usuário e senha informados serão testados.
- Verifica se o serviço do servidor de armazenamento está executando na porta informada.
- Verifica se o servidor de armazenamento possui o diretório informado para o armazenamento dos arquivos.
- Verifica se o servidor de armazenamento possui espaço em disco livre para armazenar a quantidade de *bytes* que lhe foi reservada.

Caso alguma dessas validações não seja satisfeita, o servidor de armazenamento não será adicionado ao sistema. Caso contrário, o servidor é adicionado é já poderá ser usado pelo servidor de controle para o armazenamento de novos arquivos.

7.2.5 Consultando a carga do sistema

Através do Administrador também é possível consultar a carga do sistema. Isso pode ser feito acessando o menu "Arquivo", "Carga dos Servidores de Armazenamento". Ao acessar essa opção, é apresentada uma tela com a lista dos servidores de armazenamento na ordem em que esses foram escolhidos para a inclusão. Dessa forma, o primeiro servidor da lista está mais disponível e o último servidor da lista está menos disponível. A lógica para decidir qual o servidor de armazenamento está mais disponível já foi descrita na Seção 5.3.1, mas basicamente se baseia em escolher o servidor que estiver realizando menos transferências no momento, usando como critério de desempate o espaço disponível. A Figura 7.4 ilustra um exemplo dessa tela.

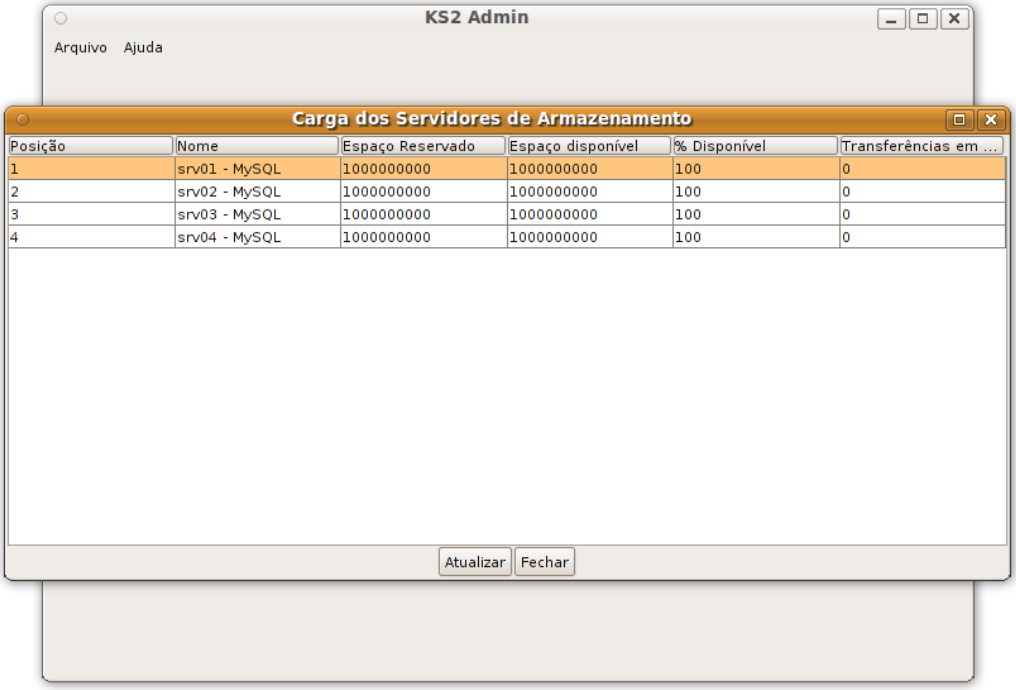

Figura 7.4: Carga dos servidores de armazenamento do sistema.

Da mesma forma que na tela de listagem dos servidores de armazenamento, aqui os servidores são apresentados em uma tabela onde cada linha representa um servidor de armazenamento, e as colunas apresentam as seguintes informações:

Posição: posição do servidor de armazenamento na lista. Sendo que quanto menor for a posição, mais próximo do topo o servidor irá aparecer, o que indica que ele está mais disponível para ser utilizado. Quanto maior a posição, mais abaixo o servidor irá aparecer, o que indica que este servidor está menos disponível para ser utilizado.

Nome: nome simbólico do servidor de armazenamento.

Espaço Reservado: quantidade de espaço em *bytes* que está reservada para o armazenamento de arquivos no servidor.

Espaço Disponível: quantidade de espaço em *bytes* que está disponível para o armazenamento de arquivos no servidor.

Transferências em aberto quantidade de transferências que estão sendo executadas no momento entre o servidor de armazenamento e os clientes (API).

As informações exibidas na tabela podem ser atualizadas através do botão "Atualizar", localizado na parte inferior da tela.

7.3 Manual de utilização da API

A API é constituída de três arquivos, são eles:

api.jar: pacote Java que contém o código compilado da API.

api.config: arquivo de configuração da API.

api.log4j.jar: arquivo de configuração do Log4j dos *logs* gerados pela API.

Para utilizar a API é necessário apenas editar seu arquivo de configuração, informando os dados do servidor de controle. A partir disso a API já pode ser utilizada por outra aplicação para o armazenamento de arquivos de forma distribuída. O arquivo de configuração da API é descrito pela listagem a seguir.

```
1  # ==========================================================================
2  # KS2API Configuration File
3  # ==========================================================================
4
5  # --------------------------------------------------------------------------
6  # Control Server Configuration
7  # --------------------------------------------------------------------------
8
9  # Address
10 ControlServer.Address = localhost
11
12 # Port
13 ControlServer.Port = 9001
14
15 # --------------------------------------------------------------------------
16 # Local Cache Configuration
17 # --------------------------------------------------------------------------
18
19 # Local cache path. Without a trailing separator
20 Cache.Path = /media/Multimidia/TCC/Cache
21
22 # Cache size in bytes
23 Cache.Size = 1000000000
24
25 # Drive block size in bytes
26 Cache.DriveBlockSize = 4096
27
28 # --------------------------------------------------------------------------
29 # Log4j Configuration
30 # --------------------------------------------------------------------------
31
32 # Log4j Configuration File Path
33 Log4j.ConfigFile = /media/Multimidia/TCC/API/log4j.config
```

As opções disponíveis no arquivo de configuração são:

ControlServer.Address: endereço IP ou nome do *host* do servidor de controle.

ControlServer.Port: endereço da porta onde o serviço do servidor de controle está sendo executado.

Cache.Path: caminho no sistema de arquivos local que será usado como *cache* para o armazenamento de arquivos.

Cache.Size: tamanho em *bytes* que deverá ser reservado para a *cache* local.

Cache.DriveBlockSize: da mesma forma que na configuração do servidor de armazenamento descrita na Seção 7.2.4, aqui deve ser informado o tamanho do bloco no disco rígido onde os arquivos da *cache* local serão armazenados.

Log4j.ConfigFile: caminho do arquivo de configuração do *Log4j*.

Além da configuração da API, opcionalmente também pode-se alterar a configuração do *Log4j* através do seu arquivo de configuração. Para maiores informações sobre a configuração do *Log4j* sugere-se a documentação oficial do projeto que está disponível em (FOUNDATION, 2007). Caso nenhuma alteração seja feita no arquivo de configuração do *Log4j*, as mensagens de *log* da API serão armazenadas em um arquivo chamado "`api.log`"no mesmo diretório do arquivo de configuração do *Log4j*.

Após a realização dessas configurações, a API já pode ser utilizada por outra a aplicação para o armazenamento de arquivos. Para maiores informações sobre como utilizar a API para o armazenamento de arquivos, sugere-se o Anexo 7.4 que contém a documentação da API no padrão *Javadoc*.

7.4 Documentação da API no padrão Javadoc

Nesta seção está disponível a documentação da classe de API que é utilizada diretamente pelo desenvolvedor para interagir com o sistema que foi desenvolvido. A documentação completa de todas as classes da API está disponível do CD que contém o trabalho final, juntamente com o código fonte.