FUNDAÇÃO DE ENSINO EURÍPIDES SOARES DA ROCHA
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA - UNIVEM
PROGRAMA DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

IVAIR LIMA

AW-KERNEL: UM PROTÓTIPO PARA SIMULAÇÃO DE FACILIDADES DE DISTRIBUIÇÃO DE BYTECODE EM NÍVEL DE *KERNEL*

MARÍLIA
2008

IVAIR LIMA

AW-KERNEL: UM PROTÓTIPO PARA SIMULAÇÃO DE FACILIDADES DE DISTRIBUIÇÃO DE BYTECODE EM NÍVEL DE *KERNEL*

Dissertação apresentada ao Centro Universitário Eurípides de Marília – UNIVEM, mantido pela Fundação de Ensino Eurípides Soares da Rocha, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Arquitetura de Sistemas).

Orientadora:
Profª. Drª. Kalinka R. L. J. C. Branco.

MARÍLIA
2008

LIMA, Ivair

Aw-*Kernel*: Um Protótipo para Simulação de Facilidades de Distribuição de *bytecode* em nível de *Kernel*. Ivair Lima; orientadora: Kalinka Regina Lucas Jaquie Castelo Branco. Marília, SP: [s.n], 2008.

106f.

Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

1.Sistemas Distribuídos 2. Sistemas Operacionais 3. *Kernel* 4. Balanceamento de Cargas

CDD: 004.22

IVAIR LIMA

AW-KERNEL*: UM PROTÓTIPO PARA SIMULAÇÃO DE FACILIDADES DE DISTRIBUIÇÃO DE *BYTECODE* EM NÍVEL DE *KERNEL

Banca examinadora da dissertação apresentada ao Programa de Mestrado da UNIVEM,/F.E.E.S.R., para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Arquitetura de Sistemas).

Resultado: _____

ORIENTADORA: Prof[a]. Dr[a]. Kalinka Regina Lucas Jaquie Castelo Branco

1° EXAMINADOR: _____

2° EXAMINADOR: _____

Marília, ____ de _____ de 2008.

“Invoca-me, e te responderei; anunciar-te-ei coisas grandes e ocultas, que não sabes.”

Jeremias 33.3

Willy (i.m.)

AGRADECIMENTOS

A Deus, aos meus familiares, aos meus amigos, em especial ao Tiago Ferreira Lima de David, Heverton Roberto Vialle, Caio Miguel Marques, Marcelo Augusto Zanata Alves, Micheli Caroline Rodrigues Crepaldi, Marcelo Oswaldo da Costa e Patrícia Forte Blini.

Aos irmãos do oriente de Adamantina-SP, em especial Marcos Antonio Hagui, Fabio Ribeiro Nunes, Jéferson Eduardo Marquetti Francisco, João Paulo Dállaqua, Edson Luis de Oliveira e Glauco Rodrigues Ramos. Aos companheiros de profissão, em especial, ao Diretor do Campus da UNIP de Assis-SP, Samir Saliba Murad e ao Professor Marcelo Climaites Fernandes, aos companheiros de mestrado, em especial a Giulianna Marega Marques e Silvio Ricardo Rodrigues Sanches, pela amizade e pela troca de experiências profissionais.

Aos meus professores da graduação, em especial aos Professores André Luis Olivete, Márcio Cardim e Rubens Galdino da Silva. A todos os professores do PPGCC - Programa de Pós-Graduação em Ciência da Computação do Programa de Mestrado da Fundação Eurípides Soares da Rocha pela dedicação e conhecimento passado em especial aos Professores Edward David Moreno, Antônio Carlos Sementille, Raul Junji Nakashima e Marcos Luiz Mucheroni.

Um agradecimento em especial a minha orientadora, a Professora Doutora Kalinka Regina Lucas Jaquie Castelo Branco, que com certeza contribuiu para que eu evoluísse como profissional e um pouco mais como ser humano.

"...nunca [...] plenamente maduro, nem nas idéias nem no estilo,
mas sempre verde, incompleto, experimental."

Gilberto Freyre,
Tempo Morto e Outros Tempos, 1926

LIMA, Ivair. ***AW-Kernel*: Um Protótipo para Simulação de Facilidades de Distribuição de *bytecode* em nível de *Kernel***. 2008. 106 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

RESUMO

Os sistemas distribuídos permitem explorar de forma transparente os recursos computacionais de *hosts* interligados por alguma tecnologia de comunicação. A transparência pode ser obtida utilizando-se software projetado especificamente para esse propósito. Neste trabalho trata-se o problema de distribuição do processamento entre os componentes do sistema, objetivando um balanceamento de carga ideal para se obter o melhor desempenho possível. Desenvolveu-se uma extensão de *Kernel Linux* que foi denominada extensão *AW-Kernel* com o propósito de prover um escalonamento com balanceamento de cargas com base no índice de carga de cada *host*. Para que fosse possível prover tais melhorias, foi realizada uma modificação do *Kernel Linux* com a implementação de novas funcionalidades que permitem a inicialização de máquinas virtuais *Java* nos *hosts*, o armazenamento e recuperação da carga de processamento do sistema, transferência e inicialização remota e automática de *bytecodes*. Essas novas funcionalidades foram incorporadas com a mesma prioridade do kernel, uma vez que foram implementados como um módulo do kernel.

Palavra-chave: Sistemas Distribuídos, Sistema Operacional, *Kernel*, Balanceamento de Cargas

LIMA, Ivair. *AW-Kernel*: Um Protótipo para Simulação de Facilidades de Distribuição de *bytecode* em nível de *Kernel*. 2008. 106 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

ABSTRACT

Distributed systems are designed to explore transparently computational resources of hosts connected by some communication technology. Transparency can be obtained through software designed specifically for this purpose. In this study, the problems of the distribution of the processing of the system components is investigated with the objective of improve the load balancing to obtain the best performance. In order to do that, an extension of the *Linux Kernel* denominated *AW-Kernel* extension was developed with the purpose of providing a processes scheduling load balancing according to the load rate of each host. To provide these improvements, the *Linux Kernel* have to be modified because of the implementation of new functionalities that allowed the initialization of Java virtual machines in the hosts, the storage and retrieving of the system processing load rate and the transference and automatic initialization of *bytecodes* in remote hosts. Those new functionalities have the same execution priority of *Kernel*, once they were implemented as a module of *Kernel*

Keywords: Distributed Systems, Operation System, *Kernel*, Load Balance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
AW-KERNEL	*All Working Kernel*
BSD	*Berkeley Software Distribution*
CPU	*Central Processing Unit*
CVS	*Concurrent Version System*
DTP	*Data Transfer Process*
FTP	*File Transfer Protocol*
GHZ	*Gigahertz*
GNU	*GNU is Not Unix*
GRUB	*Grand Unified Bootloader*
HURD	*Hird of Unix-Replacing Daemons*
I/O	*Input/Output*
IANA	*Internet Assigned Numbers Autorithy*
IDE	*Integrated Drive Electronics*
IP	*Internet Protocol*
ISO	*International Organization for Standardization*
LKM	*Loadable Kernel Module*
MB	*Mother Board*
MVS	*Multiple Virtual Storage*
NFS	*Network File System*
NIS	*Network Information Service*
OSI	*Open Systems Interconnection*
POSIX	*Portable Operating System Interface*
RFC	*Request for Comments*
RPC	*Remote Procedure Call*
SATA	*Serial ATA*

SO	Sistema Operacional
SOD	Sistema Operacional Distribuído
TCP	*Transmission Control Protocol*
UCP	Unidade Central de Processamento
UDP	*User Datagram Protocol*
ULA	Unidade Lógica Aritmética
VFS	*Virtual File System*
XDR	*External Data Representation*

SUMÁRIO

Capítulo 1

1. Introdução

Nos últimos anos o uso de sistemas computacionais distribuídos tem se tornado um atrativo principalmente para ambientes coorporativos e acadêmicos. Fazendo que o escalonamento de processos tornasse algo motivador e desafiador para a pesquisa (TANENBAUM, 2003).

O resultado do grande avanço das redes de computadores foi o surgimento dos sistemas distribuídos, os quais permitem agrupar o poder computacional de diversos computadores interligados por uma rede para processar colaborativamente uma determinada tarefa, de forma coerente e transparente (COULOURIS *et al.*, 2007).

Para se obter um bom desempenho nos sistemas distribuídos observou-se que todos os computadores deveriam colaborar no processamento de forma equilibrada. Para isso, os sistemas distribuídos contam com o escalonamento de processos, que pode prover o balanceamento de cargas entre os computadores que compõem o ambiente. A busca por maior desempenho fez com que o escalonamento de processos tornasse algo motivador e desafiador para pesquisa (BRANCO, 2004).

O propósito desse trabalho foi a distribuição de *bytecodes* em ambientes distribuídos, e aprimorar o escalonamento de processos para alcançar melhor desempenho, tendo como

objetivo principal a criação de uma extensão de um *Kernel* para este objetivo.

1.1. Organização do Trabalho

As etapas envolvidas na elaboração deste trabalho foram agrupadas em sete Capítulos, sendo o primeiro Capítulo esta introdução e os demais detalhados a seguir.

Inicialmente são apresentados no Capítulo 2 os conceitos de sistemas operacionais, seus tipos, bem como a relevância das chamadas ao sistemas e as linguagens utilizadas na programação destas.

Os modelos arquiteturais dos sistemas distribuídos são abordados no Capítulo 3, bem como as vantagens do escalonamento de processos e do balanceamento de carga.

No Capítulo 4 são analisados dois protocolos de gerenciamento e manipulação de arquivos, sendo estes o FTP (*File Transfer Protocol*) e o NFS (*Network File System*). Esta análise foi efetuada para que fosse possível a elaboração dos ambientes que utilizasse estes protocolos. O intuito disto foi obter o desempenho destes ambientes quando realizada a transferência de arquivos.

No Capítulo 5 são apresentados conceitos básicos referentes à implementação do *AW-Kernel* (*All Working*) processo de *boot*. São descritas suas funcionalidades e seus algoritmos.

No Capítulo 6 são apresentadas as plataformas utilizadas para os testes, metodologias, e as avaliações de desempenho, bem como a análise estatística.

Para finalizar, no Capítulo 7 são expostas algumas considerações e conclusões finais, além de sugestões para trabalhos futuros.

Capítulo 2

2. Sistemas Operacionais

Para utilizar os recursos que o computador pode oferecer, é necessário que exista um programa, ou um conjunto de programas, que controle e direcione as tarefas e os comandos enviados pelo operador ou por outros programas do computador. O Sistema Operacional (SO) é responsável por essa tarefa (CAPRON *et al.*, 2006).

Segundo (TANENBAUM, 2003), o SO pode ser considerado um conjunto de ferramentas necessárias para que um computador possa ser utilizado de forma adequada, atuando como uma camada intermediária entre os programas aplicativos e o *hardware* da máquina (Figura 1).

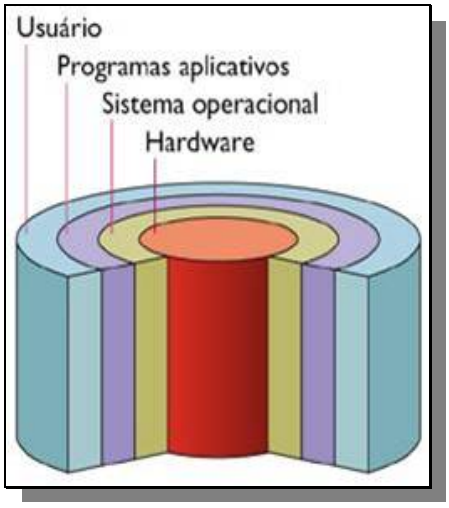

Figura 1 - Abstração de uma estrutura de um Sistema Computacional (CAPRON *et al.*, 2006).

O objetivo principal de sua utilização, segundo (SILBERSCHATZ *et al.*, 2004), é oferecer aos programas de usuários uma interface mais simples. Processadores, discos, teclado, monitores, interfaces de rede, entre outros componentes, são controlados pelo SO. A área de atuação de um sistema operacional pode ser observada na Figura 2.

Figura 2 - Área de atuação de um Sistema Operacional (TANENBAUM, 2003).

No nível 1 está localizado o *hardware*, em seguida, no nível 2, está a microarquitetura, que são registradores internos da unidade central de processamento (UCP) e um caminho dos dados contendo uma unidade lógica aritmética (ULA). A linguagem de máquina, representada no nível 3, possui instruções para mover dados, fazer operações aritméticas e comparar valores. No nível 4 encontra-se o sistema operacional, cujas funções foram citadas anteriormente, seguido pelo restante dos *softwares* do sistema. Estes *softwares*, representados no nível 5, são os compiladores, os interpretadores, os sistemas de janelas e programas similares independentes de aplicação. Os programas de usuários (processadores de

texto, planilha, programas específicos, entre outros) estão situados no nível 6, representado pela parte superior da Figura 2.

2.1. Tipos de Sistemas Operacionais

Um aspecto importante em relação a sistemas operacionais é como eles podem variar de acordo com a realização de tarefas de gerenciamento. Alguns sistemas são projetados de forma a otimizar a utilização do *hardware*, outros têm como objetivo principal dar suporte a uma maior quantidade de jogos, aplicações comerciais, entre outros (SILBERSCHATZ *et al.*, 2004).

Esta diferenciação também se tornou uma forma de classificação de sistemas operacionais proposta por (TANENBAUM, 2003), a qual permite identificar sete tipos:

- Sistemas Operacionais de Computadores de Grande Porte: normalmente encontrados em grandes corporações e projetados para obter o máximo desempenho do hardware. O OS/390 é um exemplo deste tipo de sistema.
- Sistemas Operacionais de Servidores: sistemas que permitem múltiplos usuários em rede compartilhar recursos de hardware e software. Unix e Windows 2000 são considerados sistemas operacionais típicos de servidores embora servidores *Linux* também possuam larga utilização.
- Sistemas Operacionais de Multiprocessadores: em sua grande parte constituem variações dos Sistemas Operacionais de Servidores, com aspectos especiais de comunicação e conectividade para suportar múltiplas CPUs (*Central Processing Unit*).

- Sistemas Operacionais de Computadores Pessoais: projetados com o objetivo de suportar um grande número de aplicações. Como exemplos desses sistemas têm-se toda família Windows (95/98/2000/XP/Vista), o sistema operacional da Macintosh (Mac OS), além das inúmeras distribuições *Linux* (Debian, Fedora, Ubuntu, entre outras).
- Sistemas Operacionais de Tempo Real: caracterizados por possuírem o tempo como parâmetro fundamental. Estes sistemas são normalmente utilizados em indústrias para controlar máquinas no processo de produção. Alguns sistemas que se encaixam nessa categoria são o VxWorks e QNX.
- Sistemas Operacionais Embarcados: utilizados em pequenos computadores e algumas vezes possuem características de tempo real com restrições de tamanho, memória ou consumo de energia. O PalmOS e o *Windows* CE, são exemplos destes sistemas.
- Sistemas Operacionais de Cartões Inteligentes: considerados os menores sistemas operacionais. Estes sistemas possuem restrições severas de consumo de energia e memória, operam em dispositivos que contém um *chip* de CPU ainda menor que os dos Sistemas Operacionais Embarcados.
- Sistema Operacional Distribuído (SOD): Para o usuário, parece um sistema operacional tradicional de processador único, mas que na realidade seja composto de múltiplos processadores. Os usuários não precisam saber onde seus programas estão sendo executados nem onde seus arquivos estão localizados, pois tudo é tratado

automaticamente e eficientemente por este sistema operacional (TANENBAUM e VAN STEEN, 2002).

2.2. Processos e Tarefas de um Sistema Operacional

Diversos procedimentos e tarefas são fundamentais para descrição de um sistema operacional, pode-se destacar de um modo geral: processos, *tasks* e *threads*, e de modo específico: *scheduling* e *dispatching*, monitores e semáforos, interrupções e as chamadas ao sistema (TANENBAUM and WOODHULL, 2000).

2.2.1. Processo e *Threads*

Um processo representa uma sequência de instruções única, que em sistemas paralelos ou concorrentes, são executadas paralelamente a outra sequências de instruções, tanto por particionamento de tempo, multiprocessamento ou multithread. Pode-se dizer que processo é um módulo executável único, que é executado concorrentemente com outros módulos executáveis do sistema em um ambiente multitarefa, como no Sistema Operacional Unix, que suporta processos, um processador de texto, uma planilha eletrônica e um banco de dados são processos separados que podem rodar concomitantemente. Processos são módulos separados e carregáveis, ao contrário de *threads*, múltiplas threads de execução podem ocorrer dentro de um mesmo processo, além das threads o processo também inclui certos acesso aos recursos, como arquivos e alocações dinâmicas de memória. Já uma *thread* é uma maneira de um programa dividir a si mesmo em duas ou mais tarefas simultâneas. Um exemplo é ter uma thread para prestar atenção a um ambiente, determinado processo, enquanto outras threads fazer outros “cálculos” (SILBERSCHATZ *et al.*, 2004).

No *Kernel* do *Linux* quando se olha a execução de um processo rodando sob ele, apenas um programa, do sistema operacional, pode acessar o recurso, várias tarefas são carregadas em sub-rotinas, enquanto ele define qual a próxima tarefa será executada, havendo um controle desses processos, cada processo está protegido em sua área na memória. Do ponto de vista interno do *Kernel* do *Linux* os processos têm seus privilégios de modo de usuário, onde os privilégios determinam uma seqüência de execução em memória. A Figura 3 mostra a importância disso, representada pelas setas do diagrama que mostram as possibilidades de mudanças nesses estados (BECK et al, 1999).

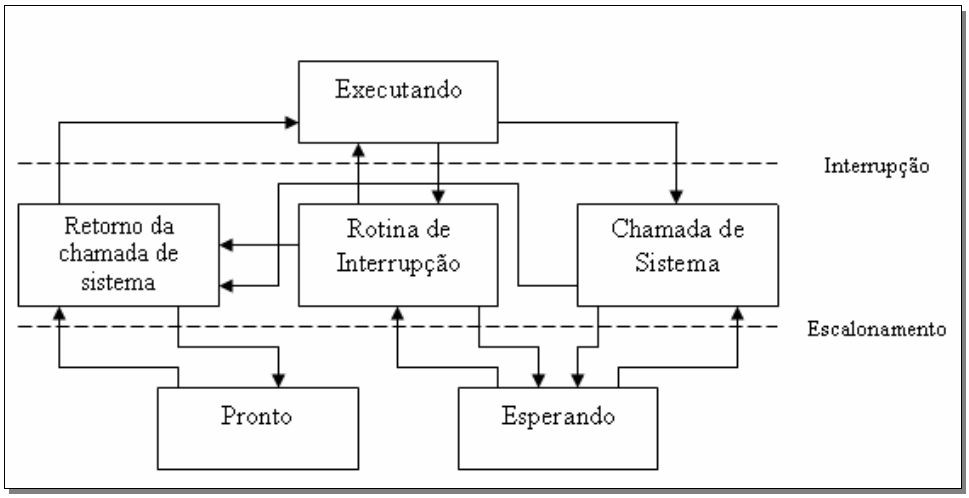

Figura 3 - Representação gráfica de estados de um processo no *Kernel* do *Linux* (BECK et al, 1999).

2.2.2. *Scheduling* e *Dispatching*

Um sistema operacional necessita de critérios para delimitar, qual o próximo processo será executado primeiramente. Isso é necessario para realizar o compartilhamento da CPU entre vários processos. Esse modo de seleção é realizado pelo *Scheduling* ou escalonador, que prepara os processos a que se refere como os processos são distribuídos para execução nos processadores em um Sistema de Computação.

> “Quando mais de um processo é executável, o Sistema Operacional deve decidir qual será executado primeiro. A parte do Sistema Operacional dedicada a esta decisão é chamada escalonador e o algoritmo utilizado é chamado algoritmo de escalonamento” (TANENBAUM, 2003).

Mais do que um simples mecanismo, o escalonamento deve representar uma política de tratamento dos processos que permita obter os melhores resultados possíveis em um sistema. A forma com que ocorre o escalonamento é, em grande parte, responsável pela produtividade e eficiência atingidas por um Sistema de Computação. O Ativador de Processos, chamado de *dispatcher*, aloca um contador de instruções (processador) a um processo no estado pronto, um *job*, tarefa ou processo, é selecionado apenas uma vez durante sua execução, o *dispatcher*, gerenciam filas de *jobs* ou de processos que esperam a sua aceitação por um processador real ou virtual, seu processo de ativação (*dispatching*) tem como função selecionar para execução o processo terminado que possui mais alta prioridade, sempre que o processador se encontra livre, um processo é chamado (SILBERSCHATZ *et al.*, 2004).

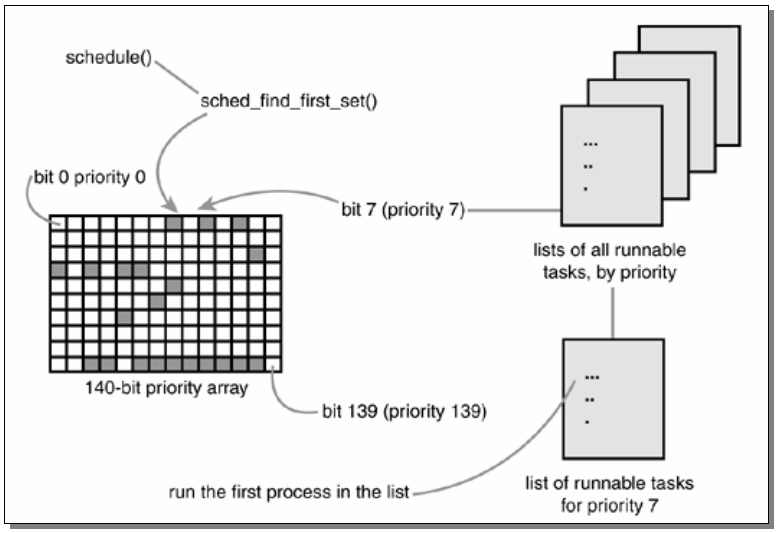

Figura 4 - Representação gráfica do *Scheduler* do *Kernel* do *Linux* (LOVE, 2005).

No *Kernel Linux*, o *scheduler* é um componente do *Kernel* que seleciona o próximo processo a ser executado, o "*process scheduler*" pode ser visto como um subsistema do *Kernel* que divide os recursos finitos do tempo de processamento do sistema em tempo de execução. Durante o desenvolvimento da versão 2.5 do *Kernel* do *Linux*, o *Kernel* recebeu um "*scheduler overhaul*", o novo *scheduler* é chamado de *o(1)*, por causa do algoritimo de *behavior*. O *scheduler* é a base para a multitarefa de um sistema operacional como o GNU/*Linux*, ele é responsável por uma melhor utilização do sistema, e ele da a impressão que múltiplos processos estão sendo executado simultaneamente (LOVE, 2005).

2.2.3. Monitores e Semáforos

Monitor é um conjunto de procedimentos, variáveis e estruturas de dados definidos dentro de um módulo, ou seja, são mecanismos de sincronização de alto nível que tentam tornar mais fáceis o desenvolvimento e a correção de programas concorrentes. Umas de suas característica mais importante é a implementação automática da exclusão mútua entre seus

procedimentos, ou seja, somente um processo pode estar executando um dos procedimentos do monitor em um determinado instante (SILBERSCHATZ *et al.*, 2004).

Toda vez que algum processo chama um desses procedimentos, o monitor verifica se já existe outro processo executando algum procedimento do monitor. Toda implementação da exclusão mútua nos monitores é realizada pelo compilador, e não mais pelo programador, como no caso do uso de semáforos. Para isso, basta colocar todas as regiões críticas em forma de procedimentos no monitor e o compilador se encarregará de garantir a exclusão mútua desses procedimentos. Assim, o desenvolvimento de programas concorrentes fica mais fácil e as chances de erros são menores.

O conceito de semáforo foi proposto como uma solução mais geral e simples de ser implementada, para os problemas de sincronização entre processos concorrentes. Um semáforo é uma variável inteira e não negativa, que só pode ser manipulada por duas instruções *down* e *up* (TANENBAUM and WOODHULL, 2000).

No caso da exclusão mútua, as instruções *down* e *up* funcionam como protocolos de entrada e saída, para que um processo possa entrar e sair de sua região crítica. O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes.

Na implementação do *Kernel* do *Linux* os semáforos são varáveis contadoras que podem sem incrementadas a qualquer tempo, entretanto, só podem ser decrementadas quando seu valor for maior zero, neste caso não será decrementada e processo será bloqueado. O uso de semáforos sob o *Linux* não deixa de ser o modelo clássico de semáforos existente no System V. Uma matriz de semáforos pode ser usada nas chamadas ao sistema, sendo possível modificar um numero da matriz de semáforos com uma simples operação, o processo pode ter qualquer escolha de valores, o programador também pode especificar que certas operações podem ser reservadas para alguns processos (BECK et al, 1999).

2.2.4. Interrupções e Chamadas ao Sistema

Um sistema de computação é formado por uma Unidade Central de Processamento (UCP) que acessa a memória para fazer busca de instruções e dados, executa as instruções e interage com os periféricos nas operações de entrada e saída, sua comunicação com os periféricos dá-se através de registradores.

Os dispositivos de entrada e saída podem ser mapeados em um espaço de endereçamento separado, distinto do espaço de endereçamento de memória, a comunicação e a troca de informações entre o processador e o controlador do periférico é feita através deste espaço, ou o espaço de endereçamento pode ser mapeado diretamente na memória, ocupando endereços no espaço de memória. O periférico, ao término de uma operação de transferência de dados, se comunica com o processador gerando um pedido de interrupção. Nesse momento o processador passa então a executar o procedimento que implementa a função correspondente à interrupção ocorrida.

O sistema operacional reage a uma ocorrência de interrupções, que provocam a execução da rotina de tratamento correspondente à fonte de interrupção. No final da execução do serviço, o controle retorna ao processo solicitante ou um novo processo é selecionado (SILBERSCHATZ *et al.*, 2004).

Esses serviços oferecidos pelo sistema operacional são acessíveis aos programas de aplicação sob a forma de chamadas ao sistema. Assim, as chamadas ao sistema são a interface entre os programas sendo executados e o sistema operacional, e geralmente, são implementados com o uso de instruções de baixo nível. Um determinado processo em execução, para abrir um arquivo utiliza uma chamada de sistema. Durante o tempo em que o sistema operacional trata a chamada de sistema o processo que faz solicitação permanece bloqueado a espera do recurso, e um novo processo é selecionado para execução.

A figura 5 mostra a transição de estados dos processos em uma chamada de sistema.

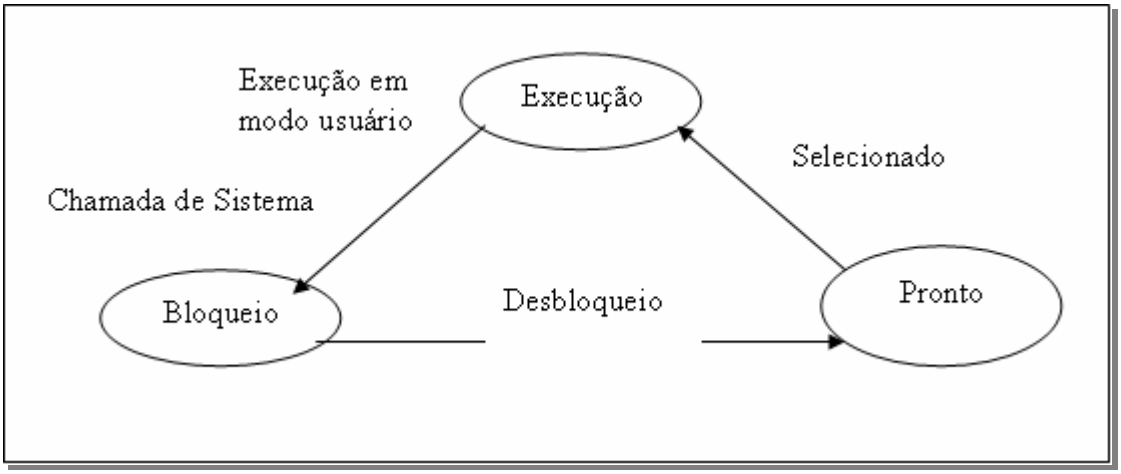

Figura 5 - Transição de estados de um processo em uma Chamada de Sistema (Silberschatz *et al.*, 2004).

Os processos em execução sofrem transição entre três estados: executando, bloqueado e pronto para executar. O processo quando é submetido para execução é inserido em uma fila de processos aptos para executar. Quando o mesmo é selecionado passa para o estado executando, que ocorre quando há uma chamada de sistema e o processo pede ao processador uma operação de entrada e saída, e passa para o estado bloqueado.

Ao término da chamada de sistema passa para o estado pronto para executar e é inserido na fila de aptos. As chamadas ao sistema podem ser relacionadas ao controle de processos, a manipulação de arquivos, a manipulação de dispositivos, à comunicação, dentre outras(SILBERSCHATZ *et al.*, 2004).

No *Kernel* do *Linux* faz-se a distinção entre o espaço de endereçamento do usuário e o espaço de endereçamento do sistema, onde não é permitido um processo do usuário acessar diretamente os serviços do *Kernel*. O processo do usuário solicita o serviço ao sistema operacional com uma chamada de sistema, dependendo a arquitetura do computador. Nos processadores Intel a troca de contexto do modo usuário para modo *Kernel* é feita com o uso da instrução int 80h. Entretanto o *Kernel* do *Linux* suporta mais de 200 chamadas ao sistema que podem ser acionadas pela instrução int 80h, nesse caso, cada chamada de sistema possui

um número que a distingue das demais, que é passado para o *Kernel* no registrador EAX (BOVET and CESATI, 2002).

Um arquivo "/usr/include/asm/unistd.h" contém o nome e o número das chamadas ao sistema. Na Tabela 1, é apresentado os números associados a algumas chamadas ao sistema existentes no *Kernel Linux*.

Tabela 1- Chamadas ao sistema existentes no *Kernel Linux* (BOVET and CESATI, 2002)..

```
#define     __NR_exit        1
#define     __NR_fork        2
#define     __NR_read        3
#define     __NR_write       4
#define     __NR_open        5
#define     __NR_close       6
#define     __NR_waitpid     7
#define     __NR_creat       8
#define     __NR_link        9
#define     __NR_unlink      10
#define     __NR_execve      11
```

2.3. *Kernel* de Sistemas Operacionais

O *Kernel* de um sistema operacional representa a camada mais baixa de interface com o *hardware*, responsável por gerenciar os recursos do sistema computacional como um todo. É nele que estão definidas funções para operação com periféricos de computadores, gerenciamento de memória, entre outros. O *Kernel* é um conjunto de programas que fornece para os programas de usuário uma interface para utilizar os recursos do sistema como um todo (BOWMAN *et al.*, 1998).

Os componentes típicos do *Kernel*, contém um escalonador para compartilhar o tempo do processador entre os diversos processos, um sistema de gerenciamento de memória para gerenciar os espaços de endereço do processo e os serviços do sistema, bem como a rede e a comunicação entre os processos (BOWMAN *et al.*, 1998).

O *Kernel* geralmente reside em um estado do sistema de alta prioridade quando

comparado às aplicações normais do usuário. Isso inclui um espaço da memória protegida e um acesso completo para o *hardware*. Esse estado do sistema e o espaço da memória são chamados juntos de espaço do *Kernel* (BOVET e CESATI, 2002).

Em oposição, as aplicações do usuário são executadas no espaço do usuário. Elas vêem um conjunto dos recursos disponíveis da máquina e são incapazes de executar certas funções do sistema.

Ao executar o *Kernel*, o sistema está no espaço do *Kernel* em oposição à execução normal do usuário no espaço usuário (BOWMAN *et al.*, 1998). As aplicações executadas no sistema se comunicam com o *Kernel* através da chamada ao sistema. Esse funcionamento pode ser observado na Figura 6.

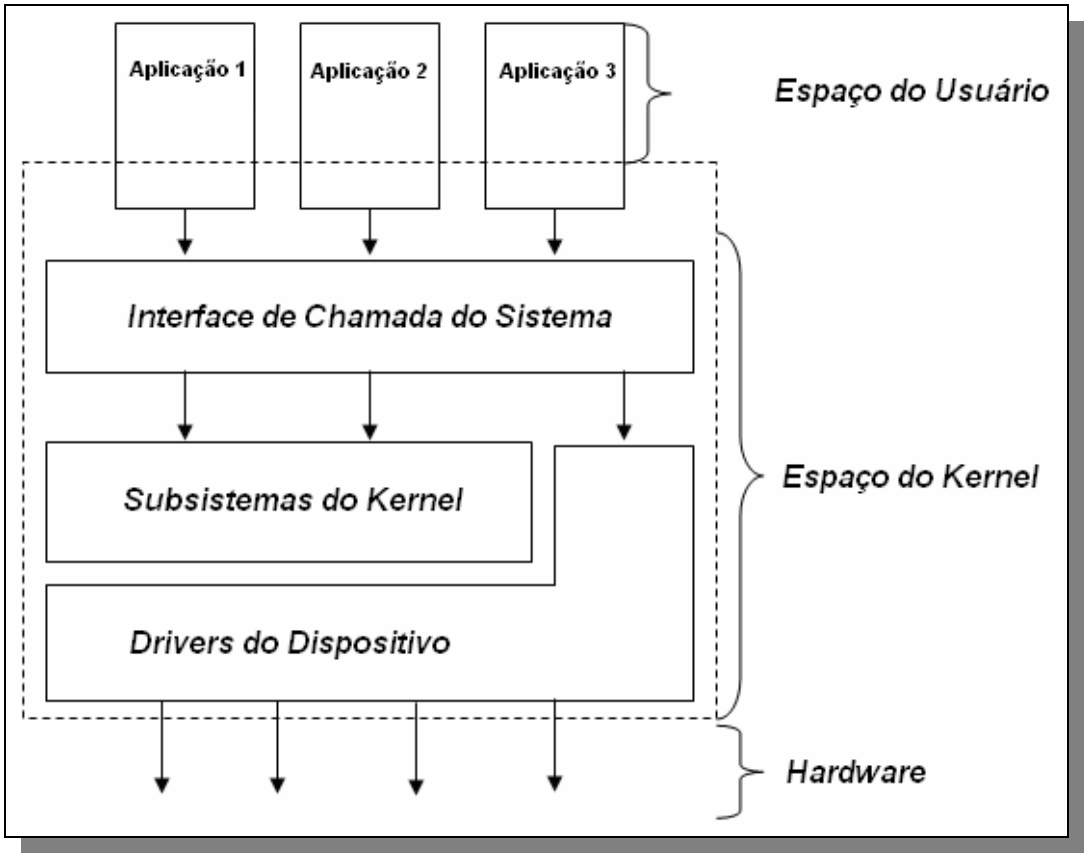

Figura 6 - Relação existente entre as aplicações, o *Kernel* e o *hardware* (BOVET and CESATI, 2002).

O *Kernel* fornece de uma interface para que os programas possam acessar os recursos do sistema de um nível mais alto e de forma transparente, ficando assim resolvido o problema da duplicação do trabalho, não sendo necessario que o aplicativo faça essa função (ACCETTA *et al*, 1988).

2.3.1. Arquitetura de *Kernel*

Um *Kernel* monolítico consiste em um único bloco, com todas as funcionalidades carregadas na memória. É um *Kernel* que implementa uma interface de alto nível para possibilitar chamadas ao sistema específicas para gestão de processos, concorrência e gestão de memória por parte de módulos dedicados que são executados com privilégios especiais. Pode-se citar que sistemas operacionais baseados em *Unix* e *Linux* utilizam esse tipo de *Kernel*.

O *Kernel* modular possui os módulos específicos para cada tarefa carregados opcionalmente (ACCETTA *et al*, 1988).

O Microkernel tem acesso a todas as instruções e a todo *hardware*, em sua definição clássica é usado para caracterizar o sistema cujas funcionalidades saíram do *Kernel* e foi para servidores, que se comunicam com um núcleo mínimo, usando o mínimo possível o "espaço do sistema" e deixando o máximo de recursos rodando no "espaço do usuário". No espaço do usuário, o *software* sofre algumas restrições, não podendo acessar alguns *hardwares*, nem tem acesso a todas as instruções. Algums sistema operacionais que utilizam o Microkernel são o Debian GNU/Hurd, o Minix e o Microsoft Windows NT (L4KA, 1994).

A estruturação de um sistema operacional em Exokernel elimina a noção de que o mesmo deve fornecer uma abstração total do hardware sobre a qual são construídas todas as

aplicações e bibliotecas do sistema operacional que usam a interface de baixo nível do Exokernel **(**ENGLER *et al*, 1995).

Este também implementa abstrações de mais alto nível e podem definir implementações especiais para melhorar a funcionalidade e desempenho das aplicações, para prover o máximo de oportunidade para o gerenciamento de recursos no nível de aplicação **(**ENGLER *et al*, 1995).

A arquitetura do Exokernel consiste de um pequeno *Kernel* embutido que faz a multiplexação e exporta os recursos físicos de forma segura através de um conjunto de primitivas de baixo nível. As aplicações que usam diretamente uma interface Exokernel não podem ser portáveis, pois a interface incluirá informações específicas de *hardware* (ENGLER, 1998).

O Nanokernel pode se dizer que é a sistema de baixo-nível do sistema operacional, vem a ser utilizado em vários dispositivos embarcados, devido a sua capacidade de executar apenas aquela função o qual foi projetado. É o mais simples de todos, só trata uma única thread por vez (MILLS e KAMP, 2000).

2.3.2. Chamadas ao Sistema

A interface entre os programas de usuários e o sistema operacional é definida por um conjunto de chamadas ao sistema (*system calls*) oferecidas por cada sistema operacional individualmente (TANENBAUM, 2003). Essa interface possui variações entre os diferentes sistemas operacionais embora seus conceitos sejam basicamente os mesmos.

As chamadas ao sistema são geralmente escritas em uma linguagem de alto nível (C ou C++) e acessadas por programas via uma API *(Application Program Interfac*e. As APIs mais comuns nos sistemas operacionais existentes são: Win32 API para *Windows*, POSIX

API para sistemas baseados em POSIX (*Portable Operating System Interface*), incluindo virtualmente todas as versões de UNIX, *Linux*, FreeBSD e Mac OS X. Na maioria dos sistemas programados em C e C++, tem em seu modelo o gerenciamento do sistema por meio de chamadas ao sistemas.

Alguns sistemas operacionais tais como Unix, Debian GNU/*Linux* e FreeBSD permitem que tais chamadas sejam feitas por programas de alto nível de maneira direta por meio da geração de uma chamada a uma rotina especial em tempo de execução que realizará a chamada ao sistema.

A linguagem *Java* não permite que chamadas ao sistema sejam feitas diretamente. Isto ocorre porque uma chamada é específica a um sistema operacional resultando em código específico da plataforma. Entretanto, existem meios de invocar métodos escritos em outras linguagens, normalmente C ou C++ para realização das chamadas ao sistema. Tais métodos são denominados métodos “nativos” (SILBERSCHATZ *et al.*, 2004).

Existem muitas chamadas ao sistema, mais especificamente, procedimentos de biblioteca que realizam essas chamadas, padronizados pelo POSIX e que são aplicadas aos sistemas UNIX, System V, BSD (*Berkeley Software Distribution*), *Linux*, entre outros. As maneiras como as chamadas ao sistemas podem ser agrupadas são de acordo com a interpretação de cada autor (LOVE, 2005).

Segundo (SILBERSCHATZ *et al.*, 2004) é possível identificar cinco grupos principais: controle de processos, manipulação de arquivos, manipulação de dispositivos, manutenção de informações e comunicações. (TANENBAUM, 2003), organiza essas chamadas em quatro grupos: gerenciamento de processos, gerenciamento de arquivo, gerenciamento de diretórios, e chamadas diversificadas. Independentemente da forma como são agrupadas, entre os sistemas operacionais que não adotam o padrão do POSIX, a grande maioria possui chamadas ao sistema que realizam funções semelhantes às padronizadas

(TANENBAUM, 2003), devido à semelhança das funções que um sistema operacional exerce.

Esta similaridade é mostrada na Tabela 2, que é listada as chamadas a API Win32 (não adota o padrão POSIX) apresentando as chamadas que correspondem às chamadas do Unix (construídas de acordo com o padrão POSIX).

Tabela 2- Chamadas da API Win32 que correspondem às chamadas UNIX (TANENBAUM, 2003)

Unix	Win32	Descrição
Fork	CreateProcess	Criar um novo processo
Waitpid	WaitForSingleObject	Poder esperar um processo sair
Execve	(none)	Criar Processo = *fork* + execve
Exit	ExitProcess	Terminar a execução
Open	CreateFile	Criar um arquivo ou abrir um arquivo existente
Close	CloseHandle	Fechar um arquivo
Read	ReadFile	Ler dados de um arquivo
Write	WriteFile	Escrever dados para um arquivo
Lseek	SetFilePointer	Mover o ponteiro de posição do arquivo
Stat	GetFileAttributesEx	Obter os atributos do arquivo
Mkdir	CreateDirectory	Criar um novo diretório
Rmdir	RemoveDirectory	Remover um diretório vazio
Link	(none)	Win32 não suporta ligações (*link*)
Unlink	DeleteFile	Destruir um arquivo existente
Mount	(none)	Win32 não suporta *mount*
Umount	(none)	Win32 não suporta *mount*
Chdir	SetCurrentDirectory	Alterar o diretório de trabalho atual
Chmod	(none)	Win32 não suporta segurança (exceto NT)
Kill	(none)	Win32 não suporta sinais
Time	GetLocalTime	Obter o horário atual

2.4. Considerações Finais

Um sistema operacional tem como objetivo criar uma camada de abstração entre o usuário e o *hardware* propriamente dito.

O coração de qualquer sistema operacional é o conjunto de chamadas ao sistema que ele pode tratar. Essas chamadas dizem o que o sistema operacional realmente faz, organiza o Unix em quatro grupos de chamadas ao sistema. O primeiro grupo relaciona-se com a criação

e a finalização de processos, o segundo grupo é para a leitura e escrita de arquivos, o terceiro é voltado ao gerenciamento de diretórios, e o quarto grupo contém chamadas diversas (TANENBAUM, 2003).

Um sistema operacional tem toda a informação sobre o estado do sistema o qual esta instalado, entretanto, os sistemas distribuídos e centralizados são muito diferentes em pontos fundamentais, como por exemplo, é comum que sistemas distribuídos permitam que aplicações sejam executadas em vários processadores ao mesmo tempo, o que exige algoritmos mais complexos de escalonamento de processadores para otimizar o paralelismo. Atrasos de comunicação na rede muitas vezes significam que esses e outros algoritmos devam ser executados com informações incompletas, desatualizadas ou até mesmo incorretas. Essa situação é radicalmente diferente de um sistema monoprocessador, em que o sistema operacional gerenca os recursos locais (TANENBAUM e VAN STEEN, 2002).

O capítulo a seguir descreve com maiores detalhes as características de um sistema computacional distribuído.

Capítulo 3

3. Sistema Computacional Distribuído

Os sistemas distribuídos podem ser definidos como um conjunto de computadores independentes que aparecem para os usuários do sistema como um único computador (TANENBAUM, 1995); ou como um sistema em que componentes de *hardware* e *software* localizados em computadores em rede se comunicam e coordenam suas ações por meio de passagem de mensagens (COULOURIS *et al.*, 2001), isto é, permitem agrupar o poder computacional de diversos *hosts* interligados por uma rede, para processar colaborativamente determinada tarefa de forma coerente e transparente.

Esses sistemas possuem muitas vantagens em relação aos sistemas centralizados, como por exemplo, economia, velocidade, distribuição inerente, confiabilidade, e crescimento incremental (TANENBAUM, 1999).

Apesar dessas vantagens, existem alguns aspectos que devem ser analisados com cuidado antes de empregá-los. Em relação ao *software* há pouca disponibilidade para sistemas distribuídos; segurança há dificuldades para evitar acesso indevido; e a rede de intercomunicação pode não dar vazão a demanda (levando a saturação).

Em aspectos de projeto, seguindo a literatura (TANENBAUM, 1995)(COULOURIS *et al.*, 2001), os sistemas computacionais distribuídos devem apresentar:

- transparência: como se apenas um único computador estivesse executando a tarefa;
- flexibilidade: representa a facilidade de fazer reconfigurações (inter-operabilidade);
- confiabilidade: que caracteriza a disponibilidade, tolerância a falhas, e a segurança;
- *performance*: é a avaliação do paralelismo e da comunicação para obter o nível de granularidade; e a
- escalabilidade: que refere-se a capacidade do sistema expandir-se com o mínimo de degradação de desempenho.

Os sistemas distribuídos podem ser compostos por computadores homogêneos conectados por uma rede, isto é, todos possuem a mesma capacidade de *hardware* e mesma configuração de *software*; ou, o mais comum, compostos por computadores heterogêneos, com uma grande diversidade de tecnologia em relação ao *hardware*, *software*, linguagens de programação, entre outros.

3.1. Modelos de Arquiteturas

Os principais modelos de arquiteturas onde realiza a distribuição de os componentes de um ambiente de sistema distribuído são conhecidos como modelo Cliente/Servidor, *Peer-to-Peer* e Híbridos, que é a combinação do modelo Cliente/Servidor com o modelo *Peer-to-Peer* (COULOURIS *et al.*, 2001).

O modelo Cliente/Servidor, representado na Figura 7, consiste em estruturar o

sistema operacional como um grupo de processos cooperantes (servidores) que oferecem serviços aos usuários (clientes). Os servidores podem, por sua vez, ser clientes de outros servidores (COULOURIS *et al.*, 2001).

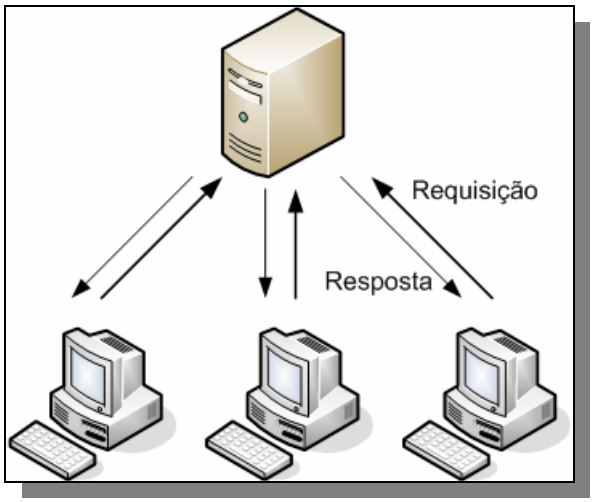

Figura 7 - Modelo Cliente/Servidor.

Este modelo utiliza o protocolo requisição/resposta que é projetado para associar respostas as suas correspondentes requisições. Possui como vantagem a simplicidade e a eficiência, e como desvantagem a falha do servidor, a qual pode deixar o sistema inoperante, porém é possível evitar quando utilizadas técnicas de replicação de servidores.

O modelo *Peer-to-Peer* fornece uma alternativa à arquitetura cliente/servidor tradicional, a qual implica na comunicação direta entre *hosts*. Neste contexto o par (*peer*) age tanto como um cliente quanto como um servidor (COULOURIS *et al.*, 2001). Esse modelo é representado na Figura 8.

Os aplicativos *Peer-to-Peer* fornecem compartilhamento de arquivos, uso de *cache web*, distribuição de informação entre outros, apresentando maior eficácia quando usados para armazenar conjuntos muito grandes de dados imutáveis (COULOURIS *et al.*, 2001).

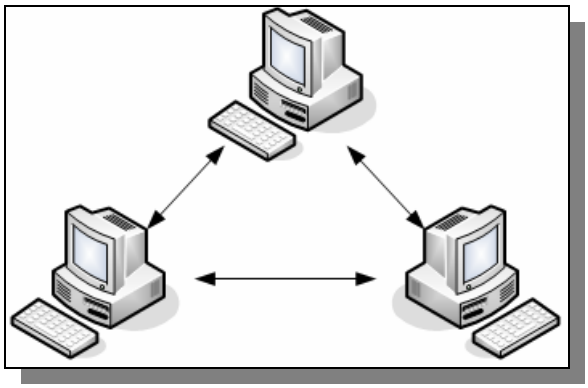

Figura 8 - Modelo *Peer-to-Peer*.

Um problema que se destaca é referente às sobrecargas indevidas que determinados *hosts* possam vir a sofrer, visto que além do gerenciamento local, cada *host* deve também tratar os acessos remotos. A vantagem é a capacidade de explorar recursos ociosos, a estabilidade para suportar grandes números de clientes e *host*, entre outros.

Um modelo híbrido busca somar as vantagens do *Peer-to-Peer* com a proteção oferecida pelos paradigmas Cliente/Servidor.

3.2. Escalonamento de Processos

Quando mais de um processo precisa ser executado, o sistema operacional precisa decidir qual deles deve ser executado primeiro e quem realiza a escolha de qual processo deve executar, recebe o nome de escalonador (TANENBAUM, 2001). A tarefa do escalonador não é trivial, entretanto é bastante conhecida e sedimentada.

O escalonador é de vital importância para sistemas distribuídos também, mas por outro lado, pode ser considerado um dos problemas mais desafiantes nesta área (SHIVARATRI *et al.*, 1992).

Segundo (CASAVANT e KUHL, 1988), o escalonamento refere-se à atividade de

alocar os recursos disponíveis entre as tarefas que compõem cada aplicação. Portanto, o escalonador é responsável pelo gerenciamento dos recursos dos *hosts* e pela atribuição dos consumidores (processos ou tarefas que compõem uma aplicação) entre os recursos disponíveis.

O escalonamento exige atenção com a rede de interconexão para que não ocorra degradação de desempenho causada pelo tráfego. Uum bom algoritmo de escalonamento deve prover: justiça para garantir que todos os processos do sistema terão chances iguais de uso do processador; eficiência para manter os recursos ocupados 100% do tempo; baixo tempo de resposta para os usuários interativos; *turnaround* para minimizar o tempo que os usuários *batch* devem esperar pela saída; *throughput* para maximizar o número de tarefas processadas na unidade de tempo (TANENBAUM, 1992).

O escalonador de processos faz sua decisão baseado em uma política de escolha utilizando os algoritmos de escalonamento e implementando mecanismos, isto é, as políticas determinam quais, quando e como os mecanismos serão empregados para que o escalonamento seja efetuado (SHIVARATRI *et al.*, 1992)(BRANCO, 2004).

Nas políticas e nos mecanismos específicos são definidos os objetivos do escalonamento. Dentre os objetivos, pode-se destacar a diminuição do tempo médio de resposta, diminuição dos atrasos na comunicação, maximização da utilização dos recursos disponíveis, e o balanceamento das cargas entre os elementos de processamento.

As políticas de escalonamento definem critérios e regras para a ordenação das tarefas a serem realizadas para que ocorra o escalonamento, e são divididas da seguinte forma (SHIVARATRI *et al.*, 1992):

- política de transferência: determina se um elemento de processamento está apto a participar de uma transferência como transmissor ou como receptor conforme a carga do elemento de processamento;

- política de seleção: está relaciona da escolha de qual tarefa será transferida após a definição de qual elemento de processamento será o transmissor (geralmente é a tarefa iniciada mais recentemente);
- política de localização: assim que a política tenha decidido qual máquina é emissora ou receptora, é responsável por definir quais elementos de processamento serão parceiros de transferência;
- política de informação: é responsável por decidir em que momento as informações sobre o estado dos elementos de processamento do sistema devem ser coletadas, de onde serão coletadas, e quais informações serão coletadas (utilizadas). Existem três tipos de políticas de informação:
  - política orientada à demanda: onde uma máquina coleta o estado das demais somente quando ela se torna emissora ou receptora;
  - política periódica: as informações são coletadas de tempos em tempos;
  - política orientada à mudança de estado: as informações das máquinas são coletadas de acordo com a mudança de seu estado.

Os mecanismos, por sua vez, são responsáveis pela definição de como o escalonamento deverá ser efetuado, e estes são divididos em três categorias (SHIVARATRI *et al.*, 1992)(BRANCO, 2004):

- mecanismo de métrica da carga: define o método utilizado para medir a carga de cada elemento de processamento;
- mecanismo de comunicação da carga: determina o método por meio do qual será efetuada a comunicação das informações de cargas entre as máquinas

disponíveis;

- mecanismo de migração: delibera o protocolo que deverá ser utilizado quando ocorrer migração de processos entre máquinas.

A classificação hierárquica da composição dos algoritmos de escalonamento é representada na Figura 9.

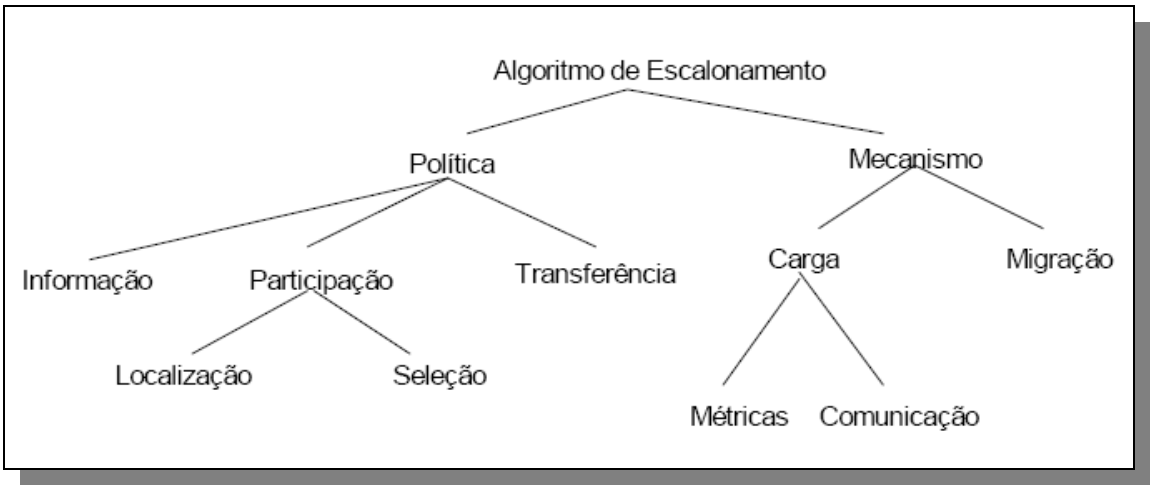

Figura 9 - Classificação Hierárquica da Composição dos Algoritmos de Escalonamento (BRANCO, 2004).

Levando em consideração as características da aplicação executada grande parte das políticas de escalonamento procura tratar adequadamente aplicações com diferentes requisitos. As aplicações consideradas no escalonamento de aplicações paralelas distribuídas são: (SHIVARATRI *et al.*, 1992)

- CPU-Bound: aplicações que possuem alta demanda por processamento e pouca atividade de entrada e saída;
- Memory-Bound: aplicações que possuem alta demanda por memória;
- I/O-Bound: (*Input /Output*) aplicações que possuem alta demanda por entrada e saída. E podem ser chamadas também de aplicações Disk-Bound, que são

aplicações que possuem alta demanda por disco, isso é, necessitam muito acesso ao disco, tanto para leitura quanto para escrita;

- Network-Bound: aplicações que possuem alta demanda por comunicação entre processos, o que implica em grande tráfego na rede

A aferição de carga é um elemento essencial para a política de escalonamento, e esta é feita por meio dos índices de carga e de desempenho.

3.2.1. Balanceamento de Cargas

Em um ambiente paralelo distribuído, na maioria das vezes, alguns elementos de processamento completam suas tarefas antes de outros, tornando-se assim ociosos. Uma causa pode ser o fato de alguns elementos de processamento serem mais rápidos que os demais, e/ou a carga não ter sido distribuída de forma balanceada e coerente com o potencial do elemento de processamento.

Como já tratado anteriormente, o escalonamento de processos tem como um de seus objetivos propostos o balanceamento de carga, que consiste em selecionar uma tarefa e definir o local onde esta será executada (ZALUSKA, 1991) buscando manter a mesma carga em cada elemento de processamento (SHIVARATRI *et al.*, 1992).

É altamente indesejável, quando se almeja o balanceamento de cargas, que existam alguns processadores com carga de processamento muito elevada e outros em estado quase ocioso, como no exemplo da Figura 10.

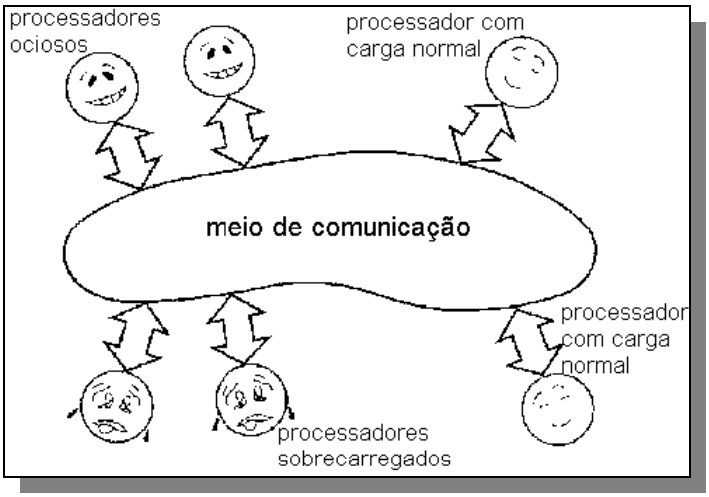

Figura 10 - Sistema Distribuído sem Balanceamento de Carga (SHIVARATRI *et al.*, 1992).

Deve ser considerada a heterogeneidade arquitetural e configuracional entre os *hosts* ao efetuar uma alocação de tarefa, uma vez que possuem potências computacionais diferentes, para isso, é preciso escolher o que melhor se aplica às restrições de determinada tarefa (BRANCO, 2004).

Para obter o melhor aproveitamento em ambientes heterogêneos, a distribuição de carga deve ser suficiente e compatível com a capacidade total do sistema, de maneira que todos os elementos de processamento tenham uma carga equilibrada entre si (nem sobrecarregados e nem ociosos).

Para isso, o escalonador de processos faz a distribuição de processos (tarefas) entre os elementos de processamento, e utiliza o balanceamento de carga para diminuir o efeito das diferenças de velocidade e capacidade dos *hosts* heterogêneos. O resultado desta administração e redistribuição é uma melhora do desempenho e eficiência de execução de aplicações (SHIVARATRI *et al.*, 1992).

O escalonamento pode realizar a distribuições de processos nos elementos de processamento durante a execução do programa, segundo o que este *host* pode oferecer de *hardware* e de *software*. Em geral este método adota como critério o balanceamento da carga entre os *hosts*, com o objetivo de melhorar o desempenho e a eficiência da aplicação. Estes

métodos realizam a distribuição de processos entre os *hosts* quando possui sobrecarga ou mesmo ociosidade nos mesmos, e retira o trabalho dos que estão sobrecarregados transferindo (se houver transferência de processos) para os que possuem menos tarefas, o que resulta em uma rede balanceada.

A distribuição ideal não é uma tarefa simples, e caracteriza o problema de balanceamento de carga (PLASTINO, 2000). Várias são as causas da falta de balanceamento de carga, entre elas é possível citar a falta de conhecimento sobre a carga de trabalho que envolve cada tarefa, a criação dinâmica de novas tarefas, a variação da carga externa à aplicação em um ambiente não dedicado, além da própria heterogeneidade da arquitetura e do sistema operacional (CASAVANT e KUHL, 1988)(BRANCO, 2004).

3.3. Considerações Finais

Com o grande avanço dos computadores e das redes foram criados os sistemas distribuídos. Os sistemas distribuídos apresentam vantagens sobre os sistemas centralizados, o que permitiu que se tornassem ainda mais difundidos.

Os sistemas distribuídos permitem agrupar o poder computacional de diversos *hosts* interligados por uma rede de comunicação a fim de processar colaborativamente determinada tarefa (ou tarefas) de forma coerente e transparente.

Contam, dessa forma, com o escalonador de processos objetivando o balanceamento de cargas a fim de se obter um bom desempenho.

Os sistemas de arquivos distribuídos são importantes em praticamente todas as redes coorporativas, acadêmicas e até mesmo domésticas. Neste cenário, o NFS é um dos sistemas de maior importância, não só pela importância histórica em ter sido um dos pioneiros, mas por ser largamente utilizado em todo o mundo.

O uso de um sistema de arquivo distribuído torna-se algo ainda mais interessante quando se pensa em efetuar um balanceamento de cargas de modo a prover o gerenciamento e manipulação mais efetiva dos arquivos.

Desse modo, o capítulo seguinte descreve diferentes formas de se efetuar o gerenciamento e manipulação de arquivos em computadores interligados, a fim de permitir, posteriormente, uma avaliação de desempenho por meio de diferentes protocolos.

Capítulo 4

4. Gerenciamento e Manipulação de Arquivos Distribuídos

Com a evolução das redes de computadores foram criadas protocolos para que o usuário pudesse acessar, ou até mesmo transferir, um arquivo de uma máquina para outra por meio da rede. Um protocolo muito utilizado para prover a transferência de arquivos é o FTP (*File Transfer Protocol*).

Com o objetivo de eliminar a distinção entre um arquivo local e um remoto foi criado o NFS (*Network File System*). Este permite que um arquivo de uma máquina remota possa ser utilizado como se estivesse na máquina local de modo transparente para o usuário e independente do *hardware* destas. É possível prover a transparência de localização, pois não é necessário determinar o caminho dos dados armazenados no servidor, diferente de quanto se utiliza o protocolo FTP.

4.1. FTP (*File Transfer Protocol*)

A transferência de dados em redes de computadores envolve normalmente transferência de arquivos e acesso a sistemas de arquivos remotos. O protocolo FTP surgiu

em 1971, antes da popularização da Internet, e ainda é muito utilizado, sendo considerado o padrão da Internet para transferência de arquivos (STEVENS, 1994). Teve sua especificação oficial definida na RFC (*Request for Comments*) 959 de outubro de 1985 (POSTEL e REYNOLDS, 1985).

O FTP é um protocolo da camada de aplicação do modelo TCP/IP (*Transmission Control Protocol / Internet Protocol*) e tem como função tornar transparente para os usuários a heterogeneidade de *hardware* e de *software* presente em uma rede de computadores, além de prover o compartilhamento de arquivos e a transferência destes.

Os computadores que compartilham os arquivos para transferência são denominados servidores FTP, e os computadores que efetuam o acesso a estes servidores são denominados clientes FTP.

Numa típica sessão de uso do protocolo FTP, é permitido que a conexão cliente especifique o tipo e o formato dos dados armazenados, por exemplo, se o arquivo contém texto ou inteiros binário (KUROSE e ROSS, 2001).

Como segurança, o protocolo FTP implementa um processo de autenticação e outro de permissão. A autenticação é verificada por meio de usuário e senha, já a permissão, é dada em nível de diretórios e arquivos.

O protocolo FTP difere de outras aplicações, pois se utiliza de duas conexões TCP "paralelas" para transferir um arquivo: uma conexão de controle e uma conexão de dados (STEVENS, 1994).

O FTP possui portas de conexão registradas na autoridade central IANA (*Internet Assigned Numbers Autorithy*) (COULOURIS *et al.*, 2007). Como pode ser observado na Figura 11, o cliente e o servidor FTP possuem um interpretador de protocolo (*Protocol Interpreter* ou PI) responsável por iniciar as conexões de controle, interpretar comandos e controlar o processo de transferência de dados (*Data Transfer Process* ou DTP) (STEVENS,

1994).

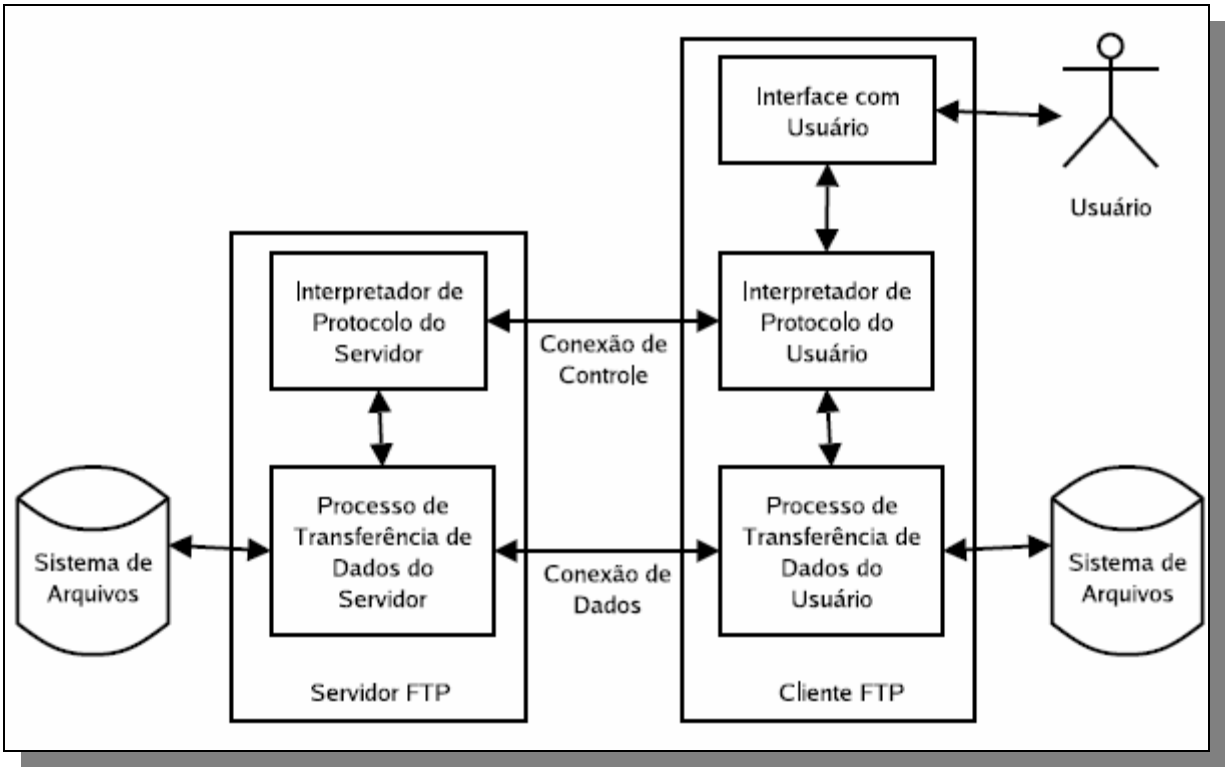

Figura 11 - Um modelo de conexão do Protocolo FTP (STEVENS, 1994).

A conexão de controle é utilizada para a transmissão de sinais de controle. Nela o cliente envia os comandos FTP e recebe as respostas do servidor. Esta conexão é estabelecida da seguinte forma: o servidor “escuta" uma porta conhecida, a porta 21, esperando por uma conexão do cliente. Os clientes, por sua vez, se conectam nessa porta e estabelecem a conexão de controle. A conexão de controle é mantida durante todo o tempo de comunicação entre o cliente e o servidor (STEVENS, 1994).

A conexão de dados é utilizada para trafegar os dados da comunicação, ou seja, os arquivos a serem transferidos. Esta conexão é estabelecida a cada vez que um arquivo é transferido e assim, não necessariamente aberta durante todo o tempo de comunicação. As conexões de dados são iniciadas pelo servidor em uma porta determinada pelo cliente por meio de comandos do FTP, de modo que é importante que o cliente esteja “escutando” esta

porta. A porta responsável por “escutar” a conexão de dados pode estar em um computador diferente do que iniciou a conexão de controle (STEVENS, 1994).

A maioria das transferências é feita pelo método chamado FTP anônimo, isto significa que em vez de identificar-se com uma conta apropriada no computador remoto, basta efetuar o *login* como anônimo, e em vez de uma senha, há apenas um endereço eletrônico de e-mail. Entretanto, o acesso anônimo é restrito a diretórios públicos que foram especificados pelo administrador da rede (COULOURIS *et al.*, 2007).

4.2. NFS (*Network File System*)

Um sistema de arquivos de rede, ou NFS (*Network File System*) é uma aplicação cliente/servidor que possibilita aos clientes um acesso transparente a arquivos remotos em um servidor com o desempenho e a segurança ao acesso a arquivos armazenados em discos locais (CALLAGHAN, 2000), isto é, permite centralizar a administração de discos fornecendo uma cópia única de diretórios comuns a todos os sistemas em uma rede de computadores (STERN *et al.*, 2001).

O NFS foi originalmente projetado e desenvolvido pela Sun Microsystems apoiada pela IBM, Hewlett-Packard, Apollo Computer, Apple Computer e várias outras corporações fornecedoras de sistemas Unix (RUPPERT e GEUS, 2006). Adotado na indústria e nos ambientes acadêmicos desde seu início em 1985, é atualmente suportado por muitos fornecedores.

Três aspectos do NFS são interessantes: a arquitetura, o protocolo e a implementação (TANENBAUM, 2003).

O protocolo NFS (versão 3) é um padrão da Internet definido no RFC 1813 (COULOURIS *et al.*, 2007) *apud* (CALLAGHAN, 1997).

Para reduzir a complexidade de projetos, a maioria das redes é organizada como uma pilha de camadas ou níveis. O objetivo de cada camada é oferecer determinados serviços às camadas superiores, isolando essas camadas dos detalhes de implementação desses recursos (TANENBAUM, 2002). A idéia fundamental é que um determinado item de *software* ou *hardware* forneça um serviço a seus usuários, mas mantenha ocultos os detalhes de seu estado interno e seu algoritmo.

O modo como são “empilhadas” estas camadas exige que os dados vindos de outras máquinas passem por camadas de níveis mais baixo até atingir as camadas de níveis mais altos e que os dados transmitidos a outras máquinas façam o caminho inverso na máquina de origem. Nestas camadas atuam os protocolos que, basicamente, podem ser considerado um acordo entre as partes que se comunicam estabelecendo como se dará a comunicação (TANENBAUM, 2002).

O modelo de camadas é amplamente utilizado devido à possibilidade de protocolos de níveis inferiores serem inteiramente substituídos sem afetar protocolos de níveis mais altos desde que ofereçam serviços compatíveis com a camada que atuam (STERN *et al.*, 2001).

O modelo de camadas padrão para redes e aplicações distribuídas é o OSI (*Open Systems Interconnection*). Camadas OSI ou Interconexão de Sistemas Abertos é um conjunto de padrões ISO (*International Organization for Standardization*) que estabelece um modelo, na forma de camadas de abstração, para o projeto de redes de computadores. Na Tabela 3 são ilustradas as camadas do modelo OSI e os protocolos utilizados pelo NFS em cada nível (STERN *et al.*, 2001).

Tabela 3 - O NFS em funcionamento no modelo OSI (STERN *et al.*, 2001).

Camada	Nome	Protocolos utilizados
7	Aplicação	NFS
6	Apresentação	XDR
5	Sessão	RPC
4	Transporte	TCP ou UDP
3	Rede	IP
2	Enlace	Ethernet
1	Física	CAT-5

Embora o NFS utilize-se de todas as seis camadas mais baixas, não existe a necessidade do usuário possuir conhecimento a respeito dos níveis subseqüentes devido às características do modelo de camadas descritas anteriormente.

Vários serviços de arquivo distribuídos foram desenvolvimentos anteriormente ao NFS, mas foram utilizados apenas em universidades e laboratórios de pesquisa. O NFS foi o primeiro serviço de arquivo projetado como um produto (COULOURIS *et al.*, 2007).

O projeto e a implementação do NFS obtiveram um sucesso tanto técnico como comercial. A partir da disponibilização das principais interfaces do NFS como domínio público, juntamente com o código fonte de uma implementação de outros fabricantes, passou-se a produzir novas versões (COULOURIS *et al.*, 2007).

O projeto NFS é independente de sistema operacional e existem implementações em máquinas multiprocessadores de alto desempenho (COULOURIS *et al.*, 2007). Embora independente de sistema operacional, originalmente o NFS foi desenvolvido para utilização em redes UNIX. Por este motivo, a implementação UNIX do protocolo NFS (versão 3) é descrito com detalhes em (COULOURIS *et al.*, 2007). É ilustrada na Figura 12 a arquitetura do NFS.

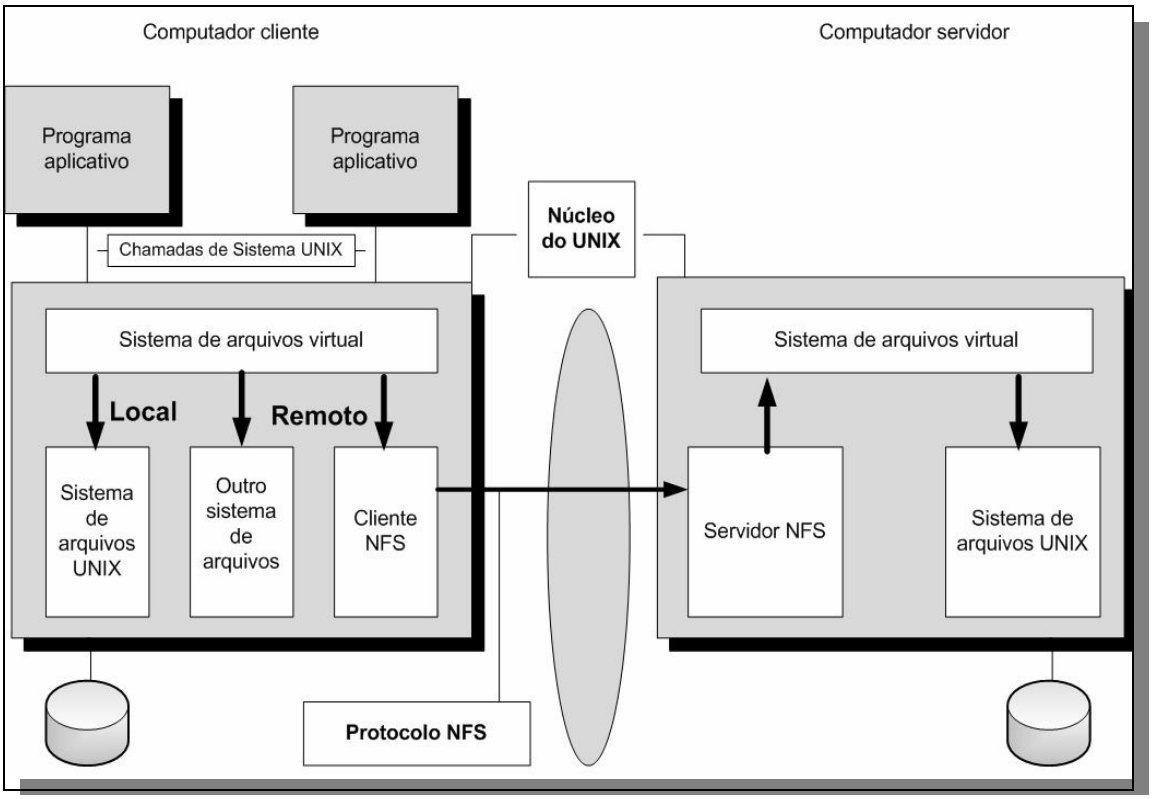

Figura 12 - Arquitetura do NFS (COULOURIS *et al.*, 2007).

Em redes UNIX o módulo servidor NFS está localizado no núcleo de cada servidor e as requisições a arquivos remotos são transformadas em operações do protocolo NFS pelo módulo cliente e, em seguida, passadas para o módulo servidor NFS no computador que contém o sistema de arquivos em questão.

O NFS pode ser utilizado baseado em um relacionamento cliente-servidor simétrico: cada computador pode atuar como cliente e como servidor tendo seus arquivos disponíveis ao acesso de outras máquinas embora a utilização de servidores dedicados seja mais comum (COULOURIS *et al.*, 2007).

A comunicação entre os módulos (cliente e servidor) é feita por meio de RPCs (*Remote Procedure Call*). O RPC da Sun foi desenvolvido para ser utilizado no NFS e suporta os protocolos UDP (*User Datagram Protocol*) e TCP (*Transmission Control Protocol*), portanto, o protocolo NFS é compatível com ambos. Um mapeador de porta é incluído para

permitir que os clientes se associem, por meio de um nome, aos serviços de um determinada máquina (COULOURIS *et al.*, 2007).

A configuração dos procedimentos do RPC utiliza a representação externa de dados XDR (*External Data Representation*) que é encarregada de passar argumentos entre o cliente e o servidor (STERN *et al.*, 2001). A interface RPC do servidor é aberta, portanto, qualquer processo pode enviar requisições para um servidor NFS. Níveis adicionais de segurança como exigência de envio de credenciais e criptografia dos dados podem ser configurados (COULOURIS *et al.*, 2007).

A implementação do servidor NFS é sem estado (*stateless*), permitindo que clientes e servidor retornem a execução após uma falha sem a necessidade de quaisquer procedimentos de recuperação. Os objetivos do projeto NFS são (COULOURIS *et al.*, 2007):

- escalabilidade: servidores NFS podem manipular cargas reais muito grandes, de maneira econômica e eficiente. O desempenho em um servidor pode ser aumentado com a adição de processadores, discos e controladores, além da possibilidade de adição de novos servidores quando esses processos são atingidos.
- replicação de arquivos: meios de armazenamento de arquivos somente para leitura podem ser aplicados em vários servidores, mas replicação de arquivos com atualizações não é suportada pelo NFS. O NIS (*Network Information Service*) (STERN *et al.*, 2001) é utilizado, normalmente, junto com o NFS para essa finalidade garantindo que a informação de configuração seja propagada para todos os *hosts* e o NFS assegura que os arquivos de um usuário possam ser acessados por estes *hosts*.
- eficiência: o desempenho medido de várias implementações do NFS,

e sua adoção para uso em situações de cargas de trabalho muito pesadas, são indicações claras da eficiência com que o protocolo pode ser implementado (COULOURIS *et al.*, 2007).

- transparência de acesso: o módulo cliente fornece uma interface de programação de aplicativos para processos locais idêntica à interface do sistema operacional local. Portanto, o acesso a arquivos remotos são realizados utilizando as chamadas ao sistemas comuns.
- transparência de localização: os sistemas de arquivos remotos podem ter diferentes nomes de caminhos em diferentes clientes, mas um espaço de nomes uniforme pode ser estabelecido pelo uso de tabelas de configuração apropriadas em cada cliente para fornecer transparência de localização.
- transparência e mobilidade: os sistemas de arquivos podem ser movidos entre servidores, mas as tabelas de montagem de cada cliente devem ser atualizadas separadamente para permitir que os clientes acessem os arquivos em sua nova localização. Portanto a transparência de mobilidade não é obtida totalmente (COULOURIS *et al.*, 2007).

O sistema de arquivos montado[1] usando o NFS fornece dois níveis de transparência, pois aparenta estar montado em um disco acoplado ao sistema local, ou seja, um disco fixo na

[1] Montado: Definido como um dispositivo montado em um sistema baseado em UNIX utilizando uma analogia a maneira a qual os dispositivos são mapeados em sistemas operacionais baseados na plataforma WINDOWS.

máquina local, e todas as entradas do sistema de arquivo e diretórios são vistos da mesma maneira, sendo eles local ou remoto. O NFS esconde a posição dos arquivos na rede.

O sistema de arquivos NFS montado remotamente não contêm informações sobre o usuário do arquivo. O servidor de arquivos NFS pode ter uma arquitetura ou executar de um sistema operacional com uma estrutura de arquivos diferente. Por exemplo, uma máquina Sun que funciona com Solaris pode montar um sistema de arquivos NFS de um sistema Windows NT ou um *mainframe* IBM MVS (*Mutiple Virtual Storage*), usando permissões do usuário NFS para cada um destes sistemas. O NFS esconde diferenças na estrutura remota subjacente do sistema de arquivo e faz o sistema de arquivo remoto parecer ter exatamente a mesma estrutura do cliente (STERN *et al.*, 2001).

O NFS consegue o primeiro nível da transparência definindo um local genérico para as operações do sistema de arquivos que são executadas em um sistema de arquivos virtual VFS (*Virtual File System*). O Segundo nível vem da definição de "nós" virtuais, que são relacionadas às estruturas mais comuns do *i-node*[2] do sistema de arquivos do Unix, mas esconde a estrutura real do sistema de arquivos físico abaixo delas. O local de todos os procedimentos que podem ser executados em arquivos é a definição da relação do *vnode*. O *vnode* e as especificações do VFS definem junto ao protocolo NFS (STERN *et al.*, 2001).

4.3. Considerações Finais

Os protocolos de gerenciamento e manipulação de arquivos FTP e o NFS são muito utilizados para transferência de arquivos entre *hosts*, por este motivo, estudos avançados no desempenho dos ambientes que os utilizam para transferência de arquivos tornaram-se base

[2] i-node: Estrutura de dados armazena informações básicas sobre arquivos regulares, diretórios, ou outros objetos do sistema de arquivo.

de comparação com o projeto proposto por este trabalho.

O próximo capítulo descreve a modificação em nível de *Kernel* para prover transferência de arquivos, mais especificadamente de *bytecodes.*

Capítulo 5

5. *AW-Kernel* - Protótipo para Sistemas Distribuídos

Os sistemas distribuídos são sistemas projetados para que seja possível o aproveitamento otimizado e transparente dos recursos computacionais disponíveis. Um dentre os diversos objetivos possíveis é obter o máximo de desempenho de processamento.

Para um sistema distribuído obter um bom desempenho é necessário utilizar um escalonador de processos projetado para isso, ou seja, um que pode prover o balanceamento de cargas entre os *hosts* participantes do ambiente. A busca por maior desempenho fez com que o estudo e implementação de novos algoritmos de escalonamento de processos tornasse algo motivador e desafiador para pesquisa (COULOURIS *et al.*, 2001), (BRANCO, 2004).

Neste trabalho, considerando-se o objetivo de se obter um maior desempenho em um sistema distribuído, criou-se uma extensão de um *Kernel*, específico para ambientes distribuídos, denominada *AW-Kernel*. Ela tem como propósito prover um escalonamento de processos com balanceamento de cargas entre os *hosts* da plataforma distribuída.

A *AW-Kernel* é uma extensão do *Kernel* do sistema operacional *Debian GNU/Linux versão 4 r0*, que contém a implementação de módulos com chamadas ao sistema que permitem prover a transparência na distribuição e execução do *bytecodes* entre os *hosts* do sistema; a inicialização automática da máquina virtual *Java* em cada *host* cliente do sistema, o

monitoramento da carga de processamento dos *hosts* cliente do sistema; a transferência automática do código *bytecode*, por meio do redirecionamento de chamadas do sistemas, utilizando *sockets* para o acesso remoto; monitoramento do tempo de execução dos processos do sistema.

As chamadas da *AW-Kernel* são implementadas com a linguagem C utilizando as bibliotecas padrão de programação do *Kernel GNU/Linux*, *a glibc*, disponível na distribuição Debian GNU/*Linux*.

5.1. Extensão *AW-Kernel*

Para se obter um escalonamento com balanceamento de cargas fizeram-se necessários algoritmos de hierarquia cliente/servidor programados em módulo para serem adicionados ao *Kernel*, e utilizou-se como carga[3], programas binários no formato *bytecode*. Desenvolveu-se e implementou-se como *daemons*, um algoritmo para ser executado no *host* servidor (*awk-server*) e um outro para ser executado nos *hosts* clientes (*awk-client*). É apresentado na Figura 13 um diagrama do fluxo de execução do algoritmos necessarios para o *AW-Kernel*, e a seguir são explicados os algoritmos usados.

[3] O termo carga refere-se a processos executando no sistema distribuído.

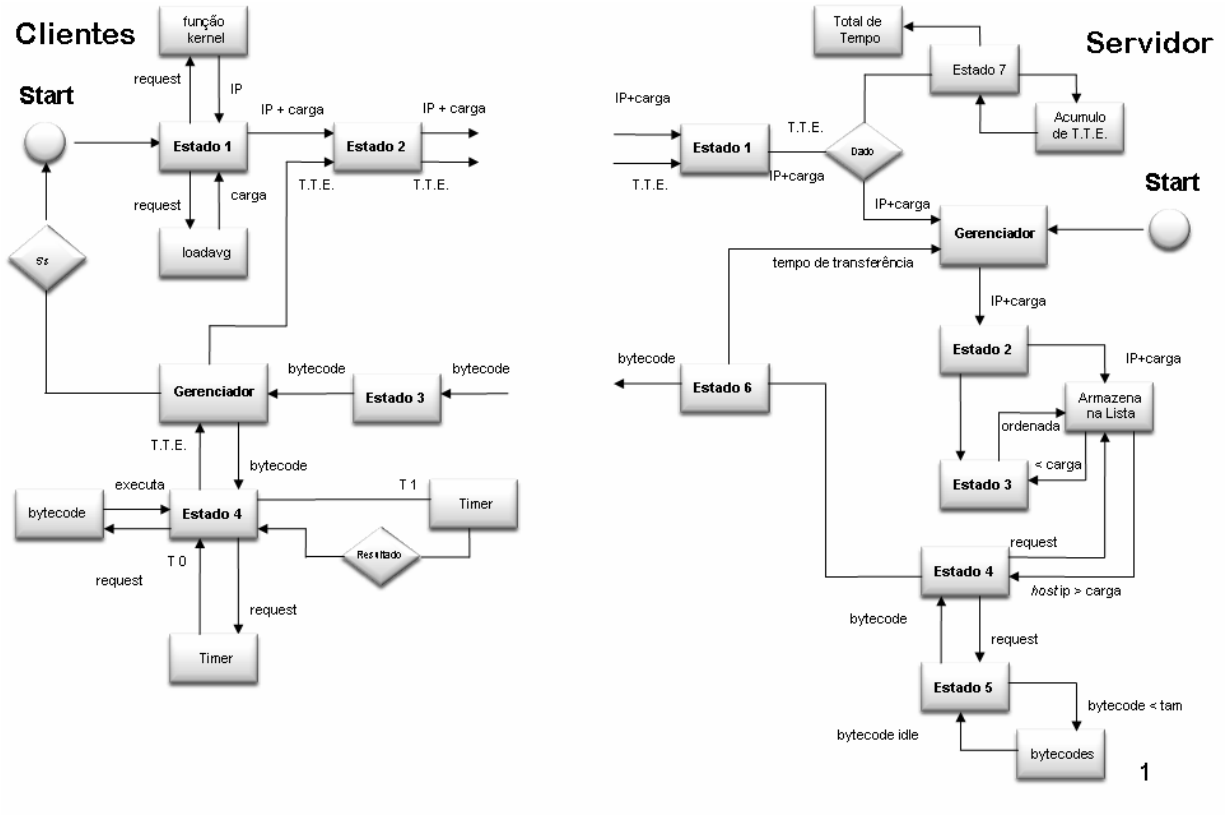

Figura 13 - Estrutura dos Algoritmos de Escalonamento e Balanceamento do Aw-Kernel.

5.1.1. Algoritmo *AW-Kernel* do *Host* Servidor

O algoritmo *awk-server* tem como função gerenciar o ambiente distribuído, além de verificar as cargas da CPU (*Central Processing Unit*) enviadas pelos *hosts* clientes e classificá-las. Quando iniciados, os *hosts* clientes informam seu endereço IP (*Internet Protocol*) e sua carga da CPU ao *host* servidor. As informações dos clientes são recebidas no *host* servidor por uma k*ernel thread* que utiliza *socket* conectado na porta 534 (trecho de código representado na Figura 14).

```
…
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/socket.h>
#include <asm/page.h>
…
int ServerPortMsg = 534;
…
int Conecta(void *unused)
{
…
…
}
int SocketCria(void)
{
      struct sockaddr_in server;
      int error;
           error =
sock_create(PF_INET,SOCK_STREAM,IPPROTO_TCP,&MainSocket);
…
…
      error = MainSocket->ops->bind(MainSocket,(struct
sockaddr*)&server,sizeof(server));
…
      error = MainSocket->ops->listen(MainSocket,1);
…
int awk_init_module(void)
{
      int binding, i;
       binding = SocketCria();
…
       pid[i] = kernel_thread(Conectado, NULL, CLONE_SIGHAND);
…
…
module_init(awk_init_module);
…
```

Figura 14 - Trecho de código referente à definição das *threads* e *sockets* do hosts servidor.

O *host* servidor recebe as informações e as armazena em uma lista ordenada de modo ascendente pelo índice de carga. O primeiro *host* cliente a receber *bytecodes* é aquele que apresentar a menor carga. Utiliza-se de *socket* por meio da porta 536 para efetuar o envio dos *bytecodes* aos *hosts* clientes, conforme representado no trecho de código na Figura 15.

```
…
int ServerPortBcj = 536;
…
char * acha_arquivo(char *buffer)
{
      char *start_name, *end_name, *arquivo;
      arquivo=kmalloc(25, GFP_KERNEL);
      start_name = strstr(buffer, "/");
…
…
      strcat(arquivo, start_name);
      return arquivo;
}
…
…
int Conecta(void *unused)
{
…
…
    fd = filp_open(nomearquivos, O_RDONLY, 0);
           if (IS_ERR(fd)) {
             …
             …
                 oldfs = get_fs(); set_fs(KERNEL_DS);
                 error = sock_sendmsg(newsock, &msg,
sizeof(awheader));
                 set_fs(oldfs);
             …
             …
           else {
             …
             …
                 oldfs = get_fs(); set_fs(KERNEL_DS);
                 error = sock_sendmsg(newsock, &msg,
sizeof(awheader));
                 set_fs(oldfs);
             …
             …
…
```

Figura 15 - Trecho de código referente a seleção e envio dos *bytecodes*.

Após a execução do *bytecode*, o *host* cliente informa ao *host* servidor o término da execução, para que o mesmo calcule o tempo total de resposta, que representa a soma do tempo de transferência (envio e recebimento) com tempo de execução.

5.1.2. Algoritmo *AW-Kernel* do *Host* Cliente

O algoritmo *aw-client* tem como função enviar o endereço IP e a carga da CPU dos *hosts* clientes ao *host* servidor que se conectam, receber e executar o *bytecode* transferido pelo *host* servidor, realizando tomadas de tempos.

A informação da carga de CPU é retirada do arquivo *loadavg*, que se localiza no diretório */proc*. A função do algoritmo responsável pela obtenção da carga, não levando em consideração as outras informações existentes no arquivo a não ser a de CPU. Para obter o endereço IP, utiliza-se outra função. Representam-se as funções citadas na Figura 16.

```
…
load_t load = { 0, -1, -1, -1, -1, -1, -1 };
int procreadload(load_t * load)
{
    FILE *fp;
    fp = fopen(LOADAVG, "r");
    if (!fp)
      return 0;

    if (fscanf(fp, "%f %f %f %d/%d %d", &load->tone, &load-
>tfive,&load->tfifteen, &load->awrunning, &load->ttotal,
            &load->tlast_id) != 6) {
      fclose(fp);
      return 0;
    }

    fclose(fp);
    load->last = procnow();
    return 1;
}
…
in_addr_t get_my_IP(){
      char name[80];
         struct hostent * hostent_ptr;
      int ret;
      ret = gethostname (name, 80);
      if(ret == -1)return 0;
      hostent_ptr = gethostbyname(name);
      if(hostent_ptr == NULL)    return 0;
      return  ((struct  in_addr  *)hostent_ptr->h_addr_list[0])-
>s_addr;
}
…
```

Figura 16 - Trecho de código referente coleta do endereço IP e carga de CPU feitas pelos hosts clientes.

A coleta das informações ocorre de 5 em 5 segundos, tempo esse determinado por meio da utilização da função *gettimeofday()* da API do *Kernel*. Utiliza-se um contador para computar o tempo, dado o tempo total, aplica-se a função *sock_sendmsg()* para enviar as informações ao servidor. As informações são enviadas com a mesma freqüência em que são coletadas por meio do *socket* na porta 534, aberta pelo servidor.

O *host* cliente executa o *bytecode* logo após o seu recebimento. Ao concluir a execução, o cliente retorna ao servidor o tempo de execução, para que seja efetuado o cálculo do tempo total de resposta. Para obter o tempo total de execução, armazena-se o tempo inicial da execução do *bytecode* em uma variável, depois de executado, registra-se o tempo final da execução em outra variável. O tempo total de resposta refere-se à soma do tempo de envio do *bytecode*, do tempo de execução e do tempo de recebimento da confirmação de execução.

Para essa troca de informações entre os *host* servidor e os *hosts* clientes, o uso das funções *sock_sendmsg()* e *sock_recvmsg()* faz-se necessárias. (trecho de código representado na Figura 17).

```
…
…
           oldfs = get_fs(); set_fs(KERNEL_DS);
           error = sock_recvmsg(newsock, &msg, sizeof(buf), 0);
           set_fs(oldfs);
           if (error<0)
           nome = acha_executa(buf);
            fd = filp_open(nome, O_RDONLY, 0);
           if (IS_ERR(fd)) {
…
…
     iov.iov_base = (void *)header_ok;
                iov.iov_len = sizeof(awheader_ok);
                msg.msg_iov = &iov;
                oldfs = get_fs(); set_fs(KERNEL_DS);
                error = sock_sendmsg(newsock, &msg,
sizeof(awheader_ok));
…
…
```

Figura 17 - Representação do trecho de código que demonstra uso das funções de envio e recebimento.

5.2. Gerenciamento do Projeto

Como todo projeto de *Kernel,* este projeto passou por versões estáveis e instáveis e a utilização do CVS (*Concurrent Version System*), o que facilitou a manutenção dessas versões e a organização do processo de desenvolvimento como um todo. Este processo recebe o nome de *branching* e possibilita um desenvolvimento baseado em um ramo de evolução do projeto, partindo-se de qualquer ponto.

Os vários *branches* de um projeto ficam registrados no repositório e o desenvolvimento do ramo alternativo prossegue em paralelo e sem interferência do ramo original. Mais tarde, se necessário, é possivel unir o *branch* alternativo com o ramo principal (*merging*).

Para criar no repositório um novo *branch*, identificado por uma nova *tag* de uma versão antiga do projeto, deve-se utilizar o comando da Figura 18.

```
# cvs checkout -r aw kernel-0-10 aw10 ; cvs tag -b aw kernel-0-10-bug-fix-variant
```

Figura 18 - Criando um novo *branch.*

Quando necessário carregar um *branch* existente criando uma cópia da nova versão do *Kernel*, utiliza-se o comando da Figura 20.

```
# cvs checkout -r aw kernel-0-10-bug-fix-variant aw10
```

Figura 19 - Carregando um *branch.*

Para carregar um *branch* específico atualizando uma cópia de trabalho existente,

utiliza-se o comando da Figura 20.

```
# cvs update -r awkernel-0-10-bug-fix-variant aw10
```

Figura 20 - Carregando um *branch.*

A Figura 21 representa o comando para realizar um *merging* de um ramo inteiro.

```
cvs commit -m "Fixed variant";
cvs checkout awkernel-development ;
cvs update -j awkernel-0-10-bug-fix-variant ;
cvs commit -m "Included bug fix”
```

Figura 21 - *Merging* de um *branch* inteiro.

Não há limite para o número de *branches* que pode ser criado ou para o nível de dependência entre eles, podendo haver “*branches* de *branches* de *branches”*.

O *AW-Kernel* segue os mesmos padrões de compilação de *Kernels Linux*, a imagem é descompactada no diretório /usr/src e é criado um *link* simbólico para que o diretório /usr/src/linux aponte para a árvore do novo *Kernel* descompactada.

Para a primeira compilação desta árvore do *Kernel*, fez-se necessário excluir os arquivos gerados. O comando *make* (com as opções *clean*, *mrproper* e *distclean*) foi utilizado, e em seguida o novo *Kernel* foi configurado. Outras opções disponíveis do comando *make*: *config, oldconfig, menuconfig e xconfig.*

O *debian-installer* instala o pacote de imagem de *Kernel* GNU/LINUX adequado para processador detectado no momento da instalação com o uso de um *parsing* no */proc*/cpuinfo. A imagem *zImage* possui um formato antigo voltados para arquiteturas Intel com o tamanho máximo permitido de 512KB, caso seja necessário utilizar uma imagem de

Kernel superior a este tamanho, faz-se o uso da imagem *bzImage*.

A *bzImage* é uma imagem que pode atingir um tamanho limite de 2,5MB, por este motivo, foi adotada para a criação da imagem do *Kernel* com as modificações implementadas (*AW-Kernel*). Os *patches*, atualizações ou correções pequenas que podem ser aplicadas para *Kernels* específicos, não foram utilizados no projeto devido ao grande número de alterações realizadas. Essa imagem foi descompactada no diretório adequado, para que fosse possível configurá-la e a compilá-la.

No *AW-Kernel* é permitido o carregamento e o descarregamento de módulos em tempo de execução, como em qualquer *Kernel* GNU/*Linux*. Essa facilidade foi utilizada para adicionar novas funcionalidades ao *Kernel* sem a necessidade de uma nova compilação.

Alguns módulos considerados desnecessários foram retirados, tornando assim o novo *Kernel* mais eficiente devido a redução nas linhas de código, e a imagem do *Kernel* criada no diretório */usr/src/linux/arch/i386/boot/bzImage* com o comando *make bzImage*. Para manter o padrão do *Kernel* do *Linux* a imagem deve ser criada no diretório /boot com o padrão de nome *vmlinuz-versão*.

O arquivo */usr/src/linux/System.map* foi copiado para o diretório */boot*. Os módulos devem ser compilados por meio do comando *make modules* e instalados utilizando o comando *make modules_install*. O gerenciador de *boot GRUB* foi configurado com a finalidade de carregar a nova imagem no momento da inicialização do sistema operacional.

O *Kernel*, antes de seu início, utiliza o *initrd* para carregar alguns *drivers*, o que possibilita o acesso ao disco rígido tornando possível o carregamento dos próximos módulos. Os *drivers* que não faziam parte da arquitetura dos *hosts* utilizados nesse trabalho foram retirados do *Kernel*. A criação de imagens *initrd* foi efetuada no Debian devido ao suporte a RAM *disks* e dispositivos *loop*, necessários. Durante o boot, o sistema faz a checagem dos arquivos de configuração de módulos (*/etc/modprobe.conf ou /etc/modules.conf*) para verificar

quais os módulos são necessários ao carregamento do sistema.

Para a geração da imagem *initrd* deve-se utilizar o suporte *cramfs* para o *Kernel*. Em sistemas Debian GNU/*Linux*, o comando *mkinitrd* utiliza a configuração determinada no arquivo */etc/mkinitrd/mkinitrd.conf* e o comando representado na Figura 22 para gerar a imagem *initrd* no Debian.

```
# mkinitrd -o /boot/initrd-aw-kernel0.10 /lib/modules/aw-kernel-0.10
```

Figura 22 - Comando utilizado para gerar uma imagem de boot no Debian.

5.3. Sistema de Inicialização

Para que se fosse possível iniciar a extensão *AW-Kernel* foram necessárias alterações nos *scripts* de inicialização, parâmetros de *boot* e no nível de execução do *init* do sistema operacional.

Na inicialização do *AW-Kernel* o processo de *boot* foi customizado, desta forma, alterou-se os *scripts* de inicialização, parâmetros de *boot* e o nível de execução do *init*. Na montagem do sistema de arquivos raiz, o *AW-Kernel* executa o programa */sbin/init* que realiza a leitura do arquivo */etc/inittab* para definir as próximas tarefas a serem executadas. O arquivo */etc/inittab* descreve quais, como e quando os processos serão executados durante o *boot*.

O *AW-Kernel* mantém o padrão do sistema de inicialização *System V*, utilizado na maior parte das distribuições *Linux*. O sistema de inicialização é dividido em 7 níveis de execução, os quais podem ser encontrados na Tabela 4.

Tabela 4 - Níveis de Inicialização do Sistema.

Nível	Inicialização
0	Sistema desligado
1	Sistema mono-usuário
2	Sistema multiusuário sem acesso a recursos de rede
3	Sistema multiusuário com acesso a recursos de rede – *AW-Kernel* (modificado)
4	Não padronizado/utilizado
5	Sistema multiusuário com acesso a recursos de rede e ambiente gráfico
6	Sistema em modo de reinicialização

O nível padrão fixado para o *AW-Kernel* foi o nível 3, ou seja, um sistema com suporte a rede, sem interface gráfica, carregando somente os módulos básicos.

Configurações foram efetuadas para que a máquina virtual *Java* fosse iniciada juntamente com o sistema operacional. Na Figura 23 é representada a linha de comando do arquivo */etc/rc.d/rc.sysinit* que indica qual o *script* de inicialização que o *init* utiliza para realizar as primeiras verificações em seu sistema, antes mesmo que o *script* de nível seja executado. Este script é executado em todos os níveis, pois não existe entre os dois pontos um número que identifique o nível.

```
si::sysinit:/etc/rc.d/rc.sysinit
```

Figura 23 - Configuração Utilizada para Verificações do Sistema.

A escolha da distribuição *Debian GNU/Linux*, como base para o projeto, deve-se entre outros fatores a utilização de um sistema de inicialização em níveis, sendo considerado modular (conforme demonstrado na Figura 24), diferente de outras distribuições, classificadas como monolítico nesse processo.

```
l0:0:wait:/etc/rc.d/rc                    0
l1:1:wait:/etc/rc.d/rc                    1
l2:2:wait:/etc/rc.d/rc                    2
l3:3:wait:/etc/rc.d/rc                    3
#l4:4:wait:/etc/rc.d/rc                   4
#l5:5:wait:/etc/rc.d/rc                   5
#l6:6:wait:/etc/rc.d/rc 6
```

Figura 24 - Sistema de Inicialização do Debian.

As linhas identificadas com os números de l0 até l6 fazem com que o *init* execute os *scripts* presentes no nível de destino. Os *scripts* estão localizados em */etc/rc.d/rcX.d*, sendo "X" o número do nível.

Foi configurado também um terminal virtual de acesso por meio do *mingetty*, que é a aplicação responsável por realizar a autenticação dos usuários em uma interface "Modo Texto", conforme demonstrado na Figura 25. Foi necessária a otimização do sistema para que apenas os recursos necessários fossem utilizados, liberando os recursos para carregamento da máquina *Java* e as rotinas feitas para o projeto.

```
1:345:respawn:/sbin/mingetty              tty1
#2:2345:respawn:/sbin/mingetty            tty2
#3:2345:respawn:/sbin/mingetty            tty3
#4:2345:respawn:/sbin/mingetty            tty4
#5:2345:respawn:/sbin/mingetty            tty5
#6:2345:respawn:/sbin/mingetty tty6
```

Figura 25 - Configuração da Autenticação dos Usuários em uma Interface Texto.

A localização desses *scripts* de inicialização dos *daemons* estão no diretório */etc/init.d*, esses *scripts* são executados pelos *links* simbólicos localizados nos diretórios correspondentes ao nível de execução ou nos *scripts* de inicialização, que são chamados */etc/init.d/rc.S* no *AW-Kernel*. Nesse caso o *script /etc/init.d/rc.S* realiza a execução de todos

os *links* simbólicos de */etc/rcS.d* com a finalidade de efetuar uma checagem de todos os sistemas de arquivos locais.

Os *daemons* são novos serviços inseridos no processo de inicialização do sistema, podem ser configurados manualmente por meio da criação de *links* simbólicos nos diretórios adequados utilizando o padrão de nomes para os *links*. Para o início da máquina virtual *Java* com as novas rotinas criadas para a chamada de sistema, fez-se o uso dos comandos representados na Figura 26.

```
# update-rc.d aw-njvm defaults 99
# update-rc.d –f aw-njvm remove
# update-rc.d aw-njvm stop 99 0 1 6 . start 60 2 3 4 5 .
```

Figura 26 - Comando para definir o serviço que vai ser ativado durante o boot.

Na Tabela 5 é demonstrada a seqüência em que o sistema é carregado.

Tabela 5 - Seqüência de Carregamento do Sistema.

Seqüencia	Carregamento
1	O gerenciador de *boot* carrega o *AW-Kernel* e o *initrd* em memória.
2	Na inicialização, o *AW-Kernel* descompacta e copia o conteúdo da *initrd* no dispositivo */dev/ram0* e então livra a memória utilizada pelo *initrd*. Este processo transforma a *initrd* em um RAM disk.
3	O *AW-Kernel* monta como leitura e escrita o dispositivo */dev/ram0* sendo a raiz do sistema.
4	O arquivo */linuxrc* é executado.
5	No *AW-Kernel*, o script */linuxrc* é interpretado pelo *dash*, ou mais comum, *ash*. Quando o script */linuxrc* é executado, as configurações escritas pelo comando *mkinitrd* no momento da criação da imagem *initrd* do arquivo /linuxrc.conf são lidas.
6	O script */linuxrc* do *AW-Kernel* não executa o comando *pivot_root* como de padrão, primeiramente, ao detectar que o script */linuxrc* foi terminado, o *Kernel* executa o */sbin/init*. Até o momento o sistema raiz verdadeiro ainda não foi montado. O *init* realiza a execução dos *scripts* do diretório */scripts* na ordem definida, e depois realiza a execução do *pivot_root* para a mudança da raiz do sistema.
7	*/linuxrc* monta o sistema raiz verdadeiro.
8	*/linuxrc* coloca o sistema raiz real no diretório raiz com o comando *pivot_root*.
9	A seqüência de *boot* normal é executada no sistema de arquivo raiz.
10	O sistema de arquivo da *initrd* é removido.

Embora possua uma arquitetura monolítica, o *Kernel Linux* suporta o carregamento dinâmico de funções na forma de módulos dinamicamente carregáveis (LKM – *Linux Kernel Modules*), os quais são implementados como código de *Kernel*, e seguem um modelo padronizado, com funções especificas para seu inicio e seu término. Como pode ser observado na Figura 27, estes módulos são carregados por meio da chamada de sistema *init_module(),* e pode ser utilizados parâmetros ao serem iniciados, ou seja, parâmetros de inicialização como se fossem rotinas programadas na linguagem de programação C e não em uma especifica para *Kernel* (BOWMAN *et al.*, 1998).

```
#include <linux/kernel.h>
…
#define NUM_THREAD 5
…
int Socket_Create(void)
{
…
}

int awk_init_module(void)
{
     int binding, i;
     binding = Socket_Create();
…
}
…
module_init(awk_init_module);
module_exit(awk_cleanup_module);
```

Figura 27 - Representação do trecho de código referente à arquitetura de programação de módulos.

Quando um módulo é carregado, ele fica ligado dinamicamente ao *Kernel*, isto é, todas as suas dependências de símbolos são traduzidas a endereços de memória consultando a tabela de símbolos públicos do *Kernel*. Um módulo também pode exportar seus próprios símbolos para tabela do *Kernel*, fazendo com que ele utilize suas funcionalidades.

É possível ainda que um módulo utilize o código de outros módulos carregados, criando assim uma rede de dependências. A remoção de um módulo pode ser efetuada apenas se não estiver sendo utilizado (BOWMAN *et al.*, 1998).

5.3.1. Máquina Virtual *Java* e Execução de *Bytecodes*

A máquina virtual *Java* (*Java Virtual Machine*) emula uma máquina real possuindo um conjunto de instruções próprias e atua em áreas de gerenciamento de memória, independente de *hardware* ou de sistema operacional. A máquina virtual *Java* permite que o mesmo código possa ser executado em várias plataformas sem a necessidade de recompilação. O código escrito na linguagem *Java* é compilado para uma forma intermediária de código denominada *bytecode*, o qual é interpretado pela máquina virtual *Java*.

O *Kernel Linux* permite multitarefa e é reentrante, fazendo com que vários processos executem em modo *Kernel* simultaneamente. Para que os *bytecodes* fossem executados diretamente, sem a necessidade de invocar constantemente a máquina virtual *Java*, foi escrito um algoritmo para mantê-la presente na memória. O trecho de código desse algoritmo é apresentado na Figura 28.

```
import java.io.*;

public class Teste{

 public static void main(String Args[]){
  int byteslidos=0;
  try{
         System.out.println("Aguardando...");
         byteslidos = System.in.read();
         System.out.println((char)byteslidos);
    }
    catch (IOException e){
        System.err.println( e.toString() );
  }
   }
}
```

Figura 28 - Trecho que permite a manutenção da maquina *Java* na memória.

Foi realizada uma configuração que permitiu que o próprio *Kernel* chamasse um interpretador adequado para os *bytecodes*, deste modo a dependência do interpretador *Java* continua existindo, mas não é necessário invocá-lo toda vez que se for executar um *bytecode*, pois este é executado como se fosse um binário do sistema.

Para tanto se fez necessário configurar o ambiente *Java* e recompilar e instalar o *AW-Kernel* em todos os *hosts* para que fosse habilitada a opção para executar arquivos binários distintos dos já conhecidos pelo *Kernel Linux "padrão"*. A configuração efetuada no *make.configure* pode ser observada na Figura 29.

```
kernel 2.6 > Executable file formats -> Kernel support for MISC binaries
```

Figura 29 - Configuração do menu do *make .configure.*

O diretório */proc* é conhecido uma interface com o *Kernel Linux* que informa o que está acontecendo no sistema. No entanto, também é possível por meio deste diretório

configurar parâmetros do *Kernel Linux*, ou seja, ele pode ser configurado enquanto está em execução sem a necessidade de recompilação do mesmo.

Foi por meio deste recurso provido pelo diretório */proc* que se tornou possível iniciar o *AW-Kernel* com o suporte a diversos formatos binários, além da associação ao seu respectivo interpretador de *bytecode*. Os procedimento detalhados a seguir foram os responsáveis por se conseguir esse suporte a diferente formatos a partir do */proc*.

Primeiramente, montou-se o */proc* conforme Figura 45.

```
# mount binfmt_misc -t binfmt_misc /proc/sys/fs/binfmt_misc
```

Figura 30 - Montagem do sistema de arquivo */proc* para alteração do *Kernel*.

Para verificar se houve sucesso na configuração executa-se, o comando da Figura 31. Caso a resposta deste comando seja a palavra *enabled*, o suporte a *binfmt_mis* foi configurado com sucesso.

```
# cat /proc/sys/fs/binfmt_misc/status
```

Figura 31 - Teste de montagem.

Para que o *AW-Kernel* seja notificado de que existem novos formatos binários e que é possível carregar os *bytecodes*, altera-se o *register* conforme demonstrado na Figura 46.

```
# echo :JAVACLASS:E::class::/usr/local/bin/javawrapper.sh:>/proc/sys/fs/binfmt_misc
/register
```

Figura 32 - Alteração do *register* dentro do */proc*.

Finalmente, utiliza-se um *wrapper*, que é um script utilizado para invocar o interpretador com o argumento correto, e registra-se esse *wrapper* no *AW-Kernel* como um interpretador.

```
#!/bin/bash
NOMECLASSE=`echo $1 | tr / . | awk -F. '{print $(NF-1)}'`
/usr/lib/java/bin/java $NOMECLASSE
```

Figura 33 - *Wrapper* para registro da máquina virtual *Java* no *Kernel.*

5.4. Considerações Finais

O *AW-Kernel* foi desenvolvido com o propósito de prover um escalonamento com balanceamento de cargas, objetivando um melhor desempenho. Para que fosse possível prover tais melhorias, fez-se necessário a otimização do *Kernel* do sistema operacional *GNU/Linux*, a implementação de novos módulos e ajuste de diversos scripts de configuração do sistema.

As alterações realizadas tiveram como premissa que num sistema distribuído um melhor desempenho do tempo de total de resposta da aplicação pode ser obtido por meio de atribuições de cargas proporcionalmente inversa a carga do *host*. Ou seja, as cargas mais pesadas devem ser atribuídas para os menos carregados e, conseqüentemente, as cargas mais leves devem ser atribuídas para os mais carregados.

O capítulo a seguir apresenta a avaliação de desempenho do *AW-Kernel* no tocante à transferência de arquivos utilizando protocolos de transferência como o FTP e o NFS e primitivas em nível de *Kernel*, bem como, uma avaliação de desempenho quando da execução de *bytecodes* seqüencialmente ou em uma plataforma distribuída fazendo uso de escalonamento de processos.

Capítulo 6

6. Avaliação de Desempenho

Este capítulo apresenta alguns estudos de caso com o intuito de avaliar o desempenho, em diversos aspectos, do *AW-Kernel*.

Foram efetuados 6 estudos de casos:

a) Transferências de arquivos utilizando NFS com 2 hosts homogêneos;

b) Transferências de arquivos utilizando FTP com 2 hosts homogêneos;

c) Transferências de arquivos utilizando *AW-Kernel* com 2 hosts homogêneos;

d) Distribuição de arquivos utilizando *AW-Kernel* com 4 hosts homogêneos;

e) Distribuição de arquivos utilizando *AW-Kernel* com 4 hosts heterogêneos;

Os experimentos foram realizados no Laboratório da UNIP “Universidade Paulista” Campus de Assis, SP, em uma rede de 13 computadores pessoais utilizando o sistema operacional GNU Debian/*Linux* (*Kernel* 2.6) conectados por uma rede padrão *ethernet* 1000Mbps.

Para avaliar o desempenho do *AW-Kernel* referente ao tempo de transferência de *bytecodes*, criou-se um ambiente FTP e um NFS de modo adaptado como visto no Capítulo 4

para que fosse possível efetuar comparações, alem das comparações do tempo total de resposta dos *bytecodes* em diferentes situações.

Os *bytecodes* têm o objetivo de realizar multiplicação de diversas matrizes, sendo que o conteúdo das mesmas já está definido no código fonte.

6.1. Análise Estatística Considerada

O objetivo de realizar uma análise estatística neste trabalho é verificar se as diferenças de desempenho dos ambientes, considerando o tempo de transferência nos mesmos, são significativas. Desse modo, utilizou-se a técnica estatística de Teste de Hipóteses (FRANCISCO, 1995)(ACHCAR; RODRIGUES, 1995).

O primeiro passo para a utilização do Teste de Hipóteses consiste na definição dessas duas hipóteses que, após realizar o cálculo dos testes, uma delas será aceita e a outra rejeitada. A hipótese H_0 ou hipótese de nulidade é geralmente formulada procurando-se "discordar" dos valores obtidos com o experimento. A hipótese H_1 ou hipótese alternativa é aquela que geralmente "concorda" com os valores obtidos no experimento quando comparados "in-loco". Assim, a hipótese H_0 é a negação da hipótese H_1.

A hipótese H_0 é utilizada quando os valores obtidos com o ambiente AW-K são menores que os obtidos com os ambientes FTP e NFS, isso é, o desempenho do AW-K é melhor que o desempenho do FTP e do NFS utilizando os mesmos *bytecodes* para transferência.

Para se realizar a análise estatística dos tempos coletados fazem-se, portanto, os seguintes testes de hipóteses:

- para amostras onde o tempo do AW-K < tempo do NFS:

H0: μAW-K ≥ μNFS

H1: μAW-K < μNFS

- para amostras onde o tempo do AW-K > tempo do NFS:

H0: μ AW-K ≤ μNFS

H1: μ AW-K > μNFS

onde: μAW-K e μNFS representam respectivamente os ambientes de transferência dos *bytecodes*.

Considerando-se que os 30 tempos obtidos para cada média comparada possuem uma distribuição normal, a estatística dos testes de hipóteses acima é dada por:

$$Z = \frac{\overline{X}_{AW-K} - \overline{X}_{NFS}}{\sqrt{\frac{S^2_{AW-K}}{n_{AW-K}} + \frac{S^2_{NFS}}{n_{NFS}}}}$$ Equação 4.1

onde: $\overline{X}_{AW-K}$ e $\overline{X}_{NFS}$ são as médias amostrais dos tempos obtidos; S^2_{AW-K} e S^2_{NFS} representam o desvio padrão amostral; n_{AW-K} e n_{NFS} representam o tamanho das amostras (neste caso 30).

Para um nível de significância (α) igual a 0.05 (probabilidade de estar correto 95% das vezes que a análise estatística for feita), rejeita-se a hipótese nulidade quando Z ultrapassar o limite fornecido por $Z_{0,05}$, o qual é 1.96. O valor de $Z_{0,05}$ = 1.96 é fornecido pela Tabela de Distribuição Normalizada (ACHCAR; RODRIGUES, 1995). Rejeita-se a hipótese nulidade H_0 caso Z ≤ -1.96, ou então, Z ≥ 1.96.

Para serem feitas às comparações na plataforma do FTP e as comparações na

plataforma NFS, foi aplicado o Teste da Hipótese em ambas as plataformas.

6.2. Análise de Desempenho dos Ambientes nas Transferências dos *Bytecodes*

Na avaliação de desempenho dos ambientes na transferência dos *bytecodes*, utilizou-se uma rede com 2 *hosts* (Celeron D 2.66 GHz, Memória 512 MB, Disco Rígido 80 GB SATA), sendo um o servidor e um cliente. Fez-se o uso de 10 *bytecodes* de tamanhos distintos na transferência e obteve-se o tempo de transferência de cada *bytecode* por meio do algoritmo apresentado no Apêndice A.

Conforme representado na Figura 34, o *host hiram* (IP 172.16.0.6) foi configurado para ser servidor FTP utilizando o servidor *proftpd*, e o *host abiff* (IP 172.16.0.7) foi configurado para ser cliente FTP. Para transferência de arquivos do servidor para o cliente utiliza-se o comando *wget*.

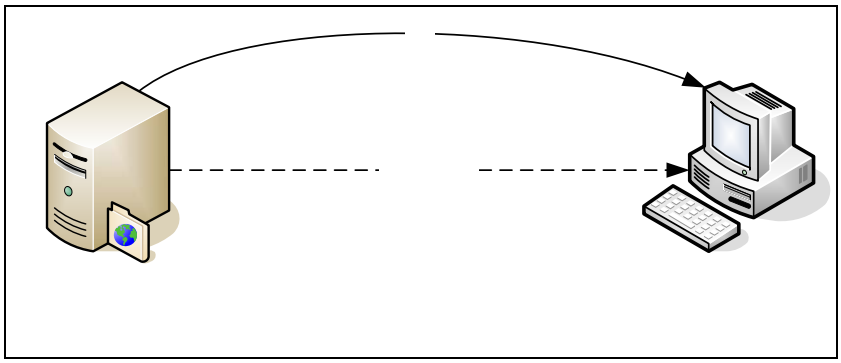

Figura 34 - Transferência dos *Bytecodes* utilizando o protocolo FTP.

De modo análogo a transferência foi feita com o protocolo NFS. Conforme ilustrado na Figura 35, o *host apolo* (IP 172.16.0.8) foi configurado para ser servidor NFS utilizando o servidor *nfsd*, e o *host willy* (IP 172.16.0.9) foi configurado para ser cliente NFS. Para

transferência de arquivos do servidor para o cliente utiliza-se o comando *cp*.

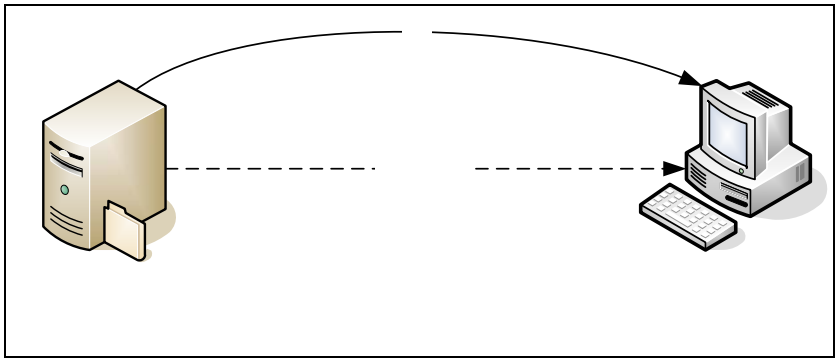

Figura 35 - Transferência de *Bytecodes* no protocolo NFS.

E o último teste foi efetuado com o *AW-Kernel* conforme ilustrado na Figura 36, o *host branca* (IP 172.16.0.10) foi configurado para ser servidor *AW-Kernel* sem efetuar a análise de cargas, e o *host cota* (IP 172.16.0.14) foi configurado para ser cliente *AW-Kernel* sem a função de enviar o endereço IP e a carga. A transferência de arquivos do servidor para o cliente é iniciada automaticamente.

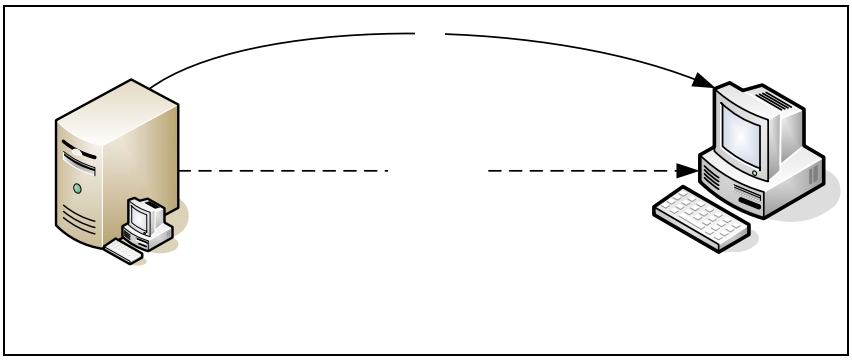

Figura 36 - Transferindo *bytecodes* usando o *AW-Kernel*.

Realizou-se 30 transferências dos 10 *bytecodes* de tamanhos distintos, obteve-se a média do tempo de transferência de cada plataforma (FTP, NFS, *AW-Kernel*), e efetuou-se a comparação dos mesmos. Na Figura 37 é demonstrada as comparações de desempenho dos ambientes referente ao tempo de transferência dos *bytecodes*, assim como as estatísticas são apresentadas na Tabela 6, Tabela 7 e Tabela 8.

Servidor
172.16

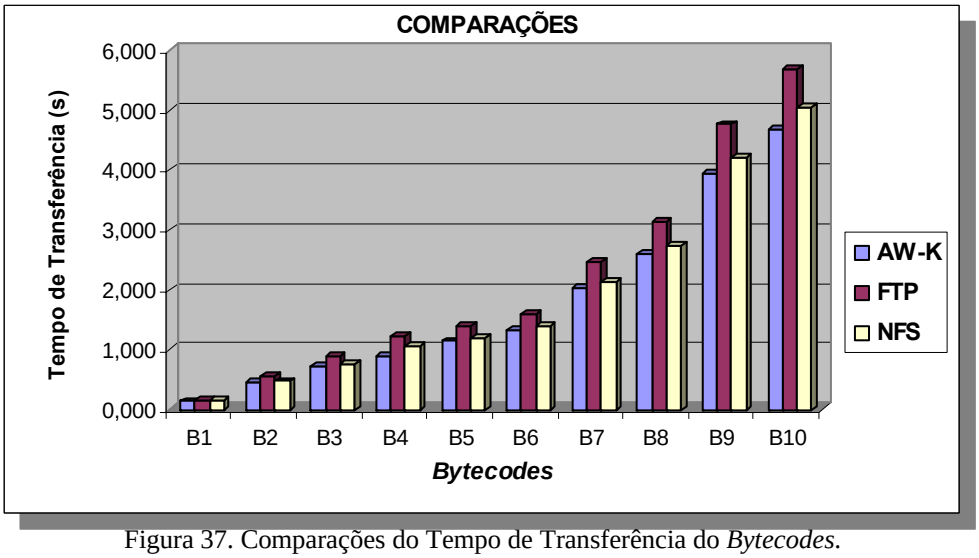

Figura 37. Comparações do Tempo de Transferência do *Bytecodes*.

Na analise verificou que a instrumentação do *Kernel*, prove a transferência de arquivos em nível de *Kernel*, apresenta melhoras significativas quando comparadas a protocolos específicos para transferência de arquivos. Principalmente em se tratando de arquivos de grande tamanho essa diferença se torna ainda mais evidente.

Observou-se que independente do tamanho do *bytecode* o *AW-Kernel* apresentou um desempenho superior ao FTP e o NFS referente ao tempo de transferência dos mesmos. Obteve-se um ganho médio de 23,01% quando comparado ao *AW-Kernel* com o FTP e de 7,09% quando comparado ao NFS.

Tabela 6 - Estatísticas Referentes ao Tempo de Transferência dos 10 *Bytecodes* no Ambiente *AW-Kernel*.

AW-Kernel										
Bytecode	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**
Tamanho (Kb)	**65359**	**387960**	**646173**	**839835**	**1098051**	**1291713**	**1937253**	**2582793**	**3809319**	**4519413**
Média(s)	**0,139545**	**0,451997**	**0,728437**	**0,880016**	**1,142106**	**1,327204**	**2,030009**	**2,591253**	**3,925153**	**4,684065**
Desvio Padrão	**0,023204**	**0,045040**	**0,043436**	**0,054126**	**0,031367**	**0,051412**	**0,060127**	**0,043522**	**0,134196**	**0,132777**
Variância	**0,000538**	**0,002029**	**0,001887**	**0,002930**	**0,000984**	**0,002643**	**0,003615**	**0,001894**	**0,018009**	**0,017630**
%Ganho (sobre FTP)	**21,03%**	**21,06%**	**20,73%**	**40,35%**	**21,18%**	**20,99%**	**21,12%**	**21,24%**	**21,15%**	**21,25%**
%Ganho (sobre NFS)	**4,84%**	**4,97%**	**4,47%**	**21,47%**	**4,96%**	**4,76%**	**4,90%**	**5,67%**	**7,43%**	**7,41%**

Tabela 7- Estatísticas Referentes ao Tempo de Transferência dos 10 *Bytecodes* no Ambiente FTP.

FTP										
Bytecode	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**
Tamanho (Kb)	**65359**	**387960**	**646173**	**839835**	**1098051**	**1291713**	**1937253**	**2582793**	**3809319**	**4519413**
Média(s)	**0,168896**	**0,547199**	**0,879428**	**1,235127**	**1,383973**	**1,605768**	**2,458783**	**3,141675**	**4,755471**	**5,679239**
Desvio Padrão	**0,028250**	**0,053907**	**0,051714**	**0,358412**	**0,036095**	**0,061400**	**0,071888**	**0,052156**	**0,157502**	**0,160577**
Variância	**0,000798**	**0,002906**	**0,002674**	**0,128459**	**0,001303**	**0,003770**	**0,005168**	**0,002720**	**0,024807**	**0,025785**
Hipótese α = 0,05	**-0,708721**	**-1,657707**	**-2,681065**	**-3,028257**	**-5,100426**	**-4,542636**	**-6,463659**	**-9,746536**	**-8,420495**	**-10,063851**

Tabela 8 - Estatísticas Referentes ao Tempo de Transferência dos 10 *Bytecodes* no Ambiente NFS.

NFS										
Bytecode	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**
Tamanho (Kb)	**65359**	**387960**	**646173**	**839835**	**1098051**	**1291713**	**1937253**	**2582793**	**3809319**	**4519413**
Média(s)	**0,146304**	**0,474463**	**0,761006**	**1,068944**	**1,198811**	**1,390345**	**2,129560**	**2,738236**	**4,216954**	**5,031196**
Desvio Padrão	**0,024567**	**0,047125**	**0,043915**	**0,307702**	**0,031450**	**0,053389**	**0,061950**	**0,042453**	**0,151724**	**0,254066**
Variância	**0,000604**	**0,002221**	**0,001929**	**0,094680**	**0,000989**	**0,002850**	**0,003838**	**0,001802**	**0,023020**	**0,064550**
Hipótese α = 0,05	**-0,169389**	**-0,405318**	**-0,603575**	**-1,720312**	**-1,239192**	**-1,068286**	**-1,560588**	**-2,745625**	**-2,988992**	**-3,056938**

6.3. Análise de Desempenho do Tempo de Execução dos *Bytecodes AW-Kernel*

Como teste do balanceamento de cargas provido pelo escalonador do *AW-Kernel* efetuaram-se comparações do tempo total de resposta dos *bytecodes* em diferentes situações, sendo:

Local (1 *host*):

- 10 *bytecodes* de tamanhos distintos;
- 40 *bytecodes*, sendo 4 vezes os 10 *bytecodes* anteriores;
- 80 *bytecodes*, sendo 8 vezes os 10 *bytecodes* anteriores;

Distribuído (4 *hosts*):

Ambiente Homogêneo:

- 10 *bytecodes* de tamanhos distintos sendo escalonados;
- 40 *bytecodes*, sendo 4 vezes os 10 *bytecodes* anteriores sendo escalonados;
- 80 *bytecodes*, sendo 4 vezes os 10 *bytecodes* anteriores escalonados;

Ambiente Heterogêneo:

- 10 *bytecodes* de tamanhos distintos sendo escalonados;
- 40 *bytecodes*, sendo 4 vezes os 10 *bytecodes* anteriores sendo escalonados;
- 80 *bytecodes*, sendo 4 vezes os 10 *bytecodes* anteriores sendo escalonados;

O ambiente elaborado para testes de desempenho do *AW-Kernel* é ilustrado na Figura 38

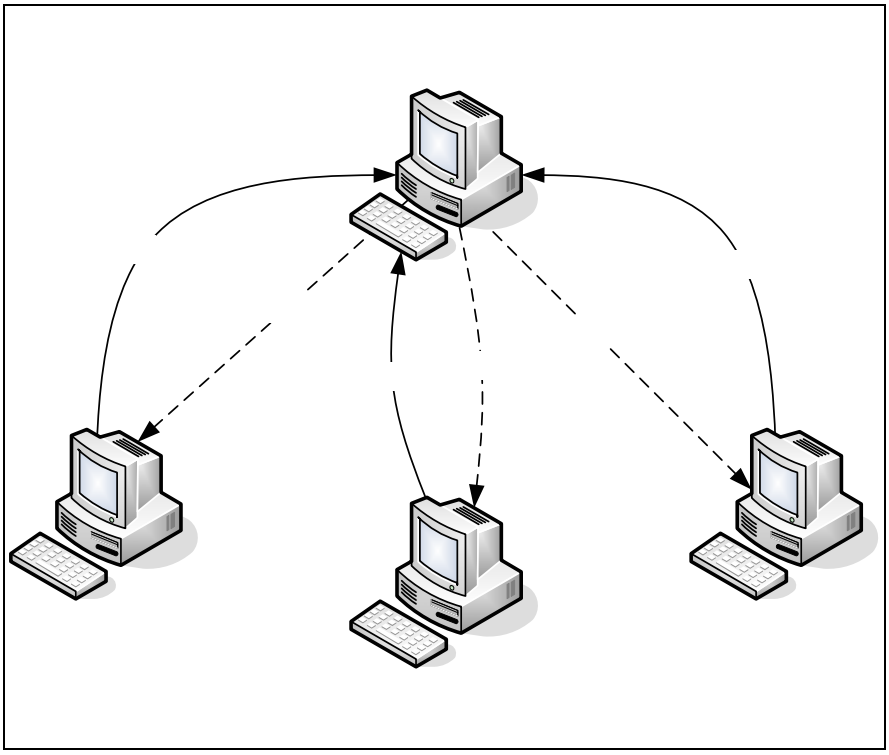

Figura 38 – Ambiente *AW-Kernel*.

O ambiente homogêneo é composto por 4 *hosts* com Processador Celeron D 2.66 GHz, Memória 512 MB, Disco Rígido 80 GB SATA. Realizou-se 30 execuções dos *bytecodes* de cada das situações propostas para ambiente homogêneo, obteve-se a média do tempo total de execução (tempo de transferência somado ao tempo de execução), e efetuou-se a comparação dos mesmos. É apresentado na Tabela 9, Tabela 10 e Tabela 11 o desempenho do *AW-Kernel* referente ao tempo de execução dos *bytecodes*.

Tabela 9 - Estatísticas Referentes ao Tempo Médio de Execução de 10 *Bytecodes* em Ambiente Homogêneo.

AW-Kernel - 10 *Bytecodes* - Ambiente Homogêneo			
	Local	Distribuído	Ganho %
Média(s)	59,106195	44,584550	32,57%
Desvio Padrão	1,297350	1,508080	
Variância	1,683117	2,274306	
Hipótese α = 0,05	0,000000	-47,487218	

Tabela 10 - Estatísticas Referentes ao Tempo Médio de Execução de 40 *Bytecodes* em Ambiente Homogêneo.

AW-Kernel - 40 *Bytecodes* - Ambiente Homogêneo			
	Local	Distribuído	Ganho %
Média(s)	266,190492	196,415040	35,52%
Desvio Padrão	0,397113	0,293019	
Variância	0,157699	0,085860	
Hipótese α = 0,05	0,000000	-460,041684	

Tabela 11 - Estatísticas Referentes ao Tempo Médio de Execução de 80 *Bytecodes* em Ambiente Homogêneo.

AW-Kernel - 80 *Bytecodes* - Ambiente Homogêneo			
	Local	Distribuído	Ganho %
Média(s)	528,655850	382,127454	38,35%
Desvio Padrão	0,309406	0,223647	
Variância	0,095732	0,050018	
Hipótese α = 0,05	0,000000	-1.099,251587	

O ambiente heterogêneo é composto de 4 *hosts,* sendo: dois clientes Celeron 1.2 GHz, Memória 256 MB, Disco Rígido 40 GB IDE; um cliente Celeron D. 2.66 GHz, Memória 512 MB, Disco Rígido 80 GB SATA; e um servidor Celeron D. 2.66 GHz, Memória 512 MB, Disco Rígido 80 GB SATA. Realizou-se 30 execuções dos *bytecodes* nas situações propostas para o ambiente heterogêneo, obteve-se a média do tempo total de execução, e efetuou-se a comparação dos mesmos. É apresentado na Tabela 9, Tabela 10 e Tabela 11 o desempenho do *AW-Kernel* referente ao tempo total de execução dos *bytecodes*.

Tabela 12 - Estatísticas Referentes ao Tempo de Execução de 10 *Bytecodes* em Ambiente Heterogêneo.

***AW-Kernel* - 10 *Bytecodes* - Ambiente Heterogêneo**			
	Local	**Distribuído**	**Ganho %**
Média(s)	**59,106195**	**51,305449**	**15,20%**
Desvio Padrão	**1,297350**	**7,028147**	
Variância	**1,683117**	**49,394847**	
Hipótese α = 0,05	**0,000000**	**-14,807839**	

Tabela 13 - Estatísticas Referentes ao Tempo de Execução de 40 *Bytecodes* em Ambiente Heterogêneo.

***AW-Kernel* - 40 *Bytecodes* - Ambiente Heterogêneo**			
	Local	**Distribuído**	**Ganho %**
Média(s)	**266,190492**	**230,612913**	**15,43%**
Desvio Padrão	**0,397113**	**26,702155**	
Variância	**0,157699**	**713,005083**	
Hipótese α = 0,05	**0,000000**	**-37,433310**	

Tabela 14 - Estatísticas Referentes ao Tempo de Execução de 80 *Bytecodes* em Ambiente Heterogêneo.

***AW-Kernel* - 80 *Bytecodes* - Ambiente Heterogêneo**			
	Local	**Distribuído**	**Ganho %**
Média(s)	**528,655850**	**448,627440**	**17,84%**
Desvio Padrão	**0,309406**	**51,672897**	
Variância	**0,095732**	**2.670,088269**	
Hipótese α = 0,05	**0,000000**	**-60,796287**	

Na Tabela 15 e na Tabela 16 são apresentadas às comparações de desempenho do *AW-Kernel* referente ao tempo total de execução local e distribuída dos *bytecodes*. É possível observar na Figura 39 as diferenças encontradas entre os ambientes.

Tabela 15 - Comparações do Tempo de Execução no Ambiente Homogêneo.

Estatísticas do Ambiente Homogêneo			
	Local	**Distribuído**	**Ganho %**
10 *Bytecodes* (s)	**59,106195**	**44,584550**	**32,57%**
40 *Bytecodes* (s)	**266,190492**	**196,415040**	**35,52%**
80 *Bytecodes* (s)	**528,655850**	**382,127454**	**38,35%**

Tabela 16 - Comparações do Tempo de Execução no Ambiente Heterogêneo.

Estatísticas do Ambiente Heterogêneo			
	Local	**Distribuído**	**Ganho %**
10 *Bytecodes* (s)	**59,106195**	**51,305449**	**15,20%**
40 *Bytecodes* (s)	**266,190492**	**230,612913**	**15,43%**
80 *Bytecodes* (s)	**528,655850**	**448,627440**	**17,84%**

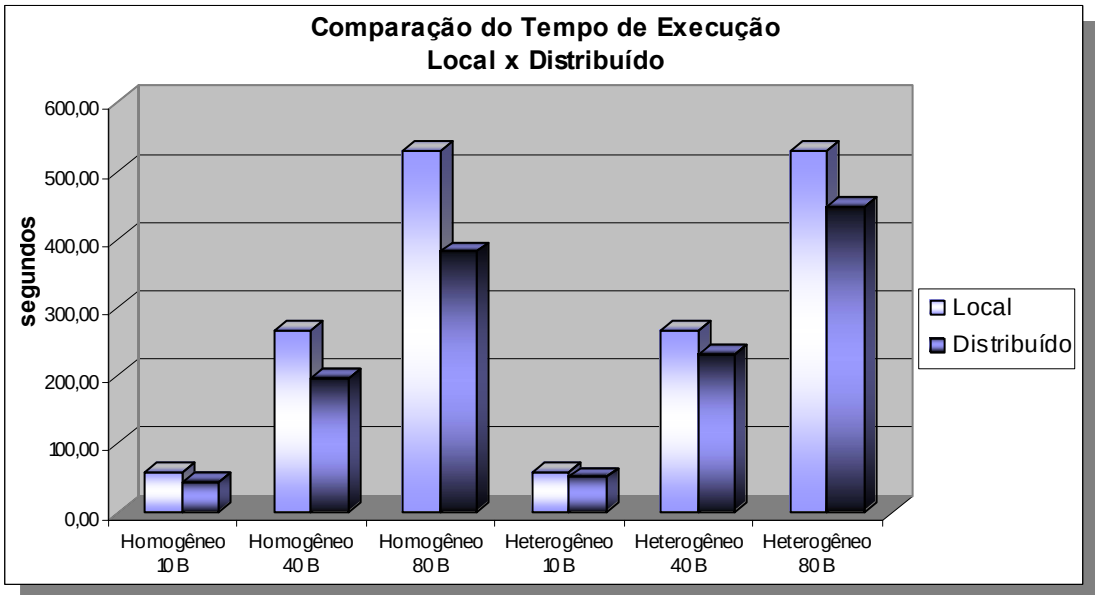

Figura 39 - Comparação do Tempo de Execução Local com o Distribuído.

Observou-se que independente da carga atribuída ao ambiente *AW-Kernel*, o desempenho obtido na execução distribuída (tempo de transferência adicionado ao tempo de execução) mostrou-se superior a execução dos *bytecodes* localmente. Obteve-se um ganho médio de 35,48% em ambientes homogêneos e de 16,16% em ambientes heterogêneos quando comparados à execução local.

É possível observar na Figura 40 o quanto à heterogeneidade do ambiente pode influênciar no desempenho final.

Pode-se observar nesse estudo especifico, que quando o ambiente é mais homogêneo têm-se uma melhora no desempenho final do sistema quando comparado a um ambiente heterogêneo composto por um mesmo numero de maquinas. O problema e a queda no

desempenho do ambiente heterogêneo devem-se ao fato da maquina mais lenta do sistema ser mais lenta que as maquinas que compõe a plataforma homogênea, de modo que a potencia computacional da plataforma heterogênea é menor que a da plataforma homogênea.

É possível avaliar também que o uso de um escalonador permite que tanto um ambiente homogêneo quanto em um ambiente heterogêneo se possa obter uma melhora no desempenho do sistema como um todo, principalmente se comparado a um único *host*.

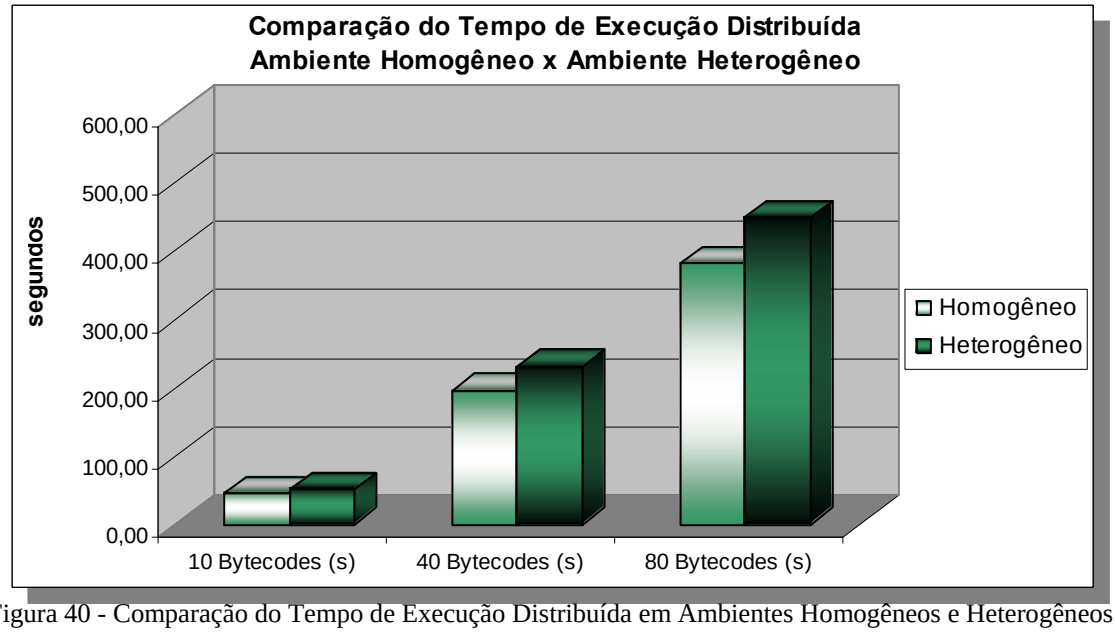

Figura 40 - Comparação do Tempo de Execução Distribuída em Ambientes Homogêneos e Heterogêneos.

6.4. Considerações Finais

A análise de desempenho dos ambientes foi realizada por meio de diferentes testes reais utilizando 10 tipos de *bytecodes*, o que permitiu efetuar uma análise estatística dos resultados obtidos.

A avaliação do ambiente *AW-Kernel* referente ao tempo de transferência apresentou resultados satisfatórios quando comparados ao tempo de transferência de ambientes que

utilizam FTP e NFS.

Conforme abordado anteriormente, o propósito principal do *AW-Kernel* é prover o escalonamento de processos objetivando o balanceamento de cargas entre os *hosts* participantes de ambientes distribuídos, função possibilitou um desempenho satisfatório tanto em ambientes homogêneos quanto heterogêneos quando comparados ao tempo de execução local de *bytecodes*.

Dessa maneira, o próximo Capítulo apresenta as conclusões obtidas como a realização dos estudos efetuados neste trabalho.

Capítulo 7

7. Conclusão

Com o propósito de equacionar alguns problemas existentes na atividade de escalonamento de processos em plataformas distribuídas, desenvolveu-se uma extensão de um *Kernel* específico para ambientes distribuídos, denominada *AW-Kernel*, a qual tem como propósito prover o escalonamento de processos objetivando o balanceamento de cargas entre os *hosts* da plataforma envolvida.

Para que fosse possível prover um bom desempenho, fez-se necessário a otimização do *Kernel* do sistema operacional, a implementação de novos módulos e algumas alterações nas configurações padrões.

Estudos referentes a sistemas distribuídos, escalonamento de processos, instrumentação de *Kernel* foram necessários.

Diversos foram os obstáculos encontrados, como a implementação da coleta da carga de CPU em nível de *Kernel*, que não é trivial quanto à coleta em nível de usuário. O estudo de sistemas operacionais, bem com seu funcionamento foram essências para permitir a realização satisfatória deste trabalho.

Problemas na configuração e utilização das máquinas, instalação e configuração do NFS também constituíram obstáculos a serem transpostos, além da definição de estudos de

caso coerentes que consumiu tempo considerável.

Entretanto, partindo-se dos resultados obtidos pode-se verificar o desempenho obtido no ambiente *AW-Kernel*, desempenho este referente ao tempo de transferência. Obteve-se resultados satisfatórios quando comparados os tempo de transferência do *AW-Kernel* e o protocolos como FTP e NFS.

A execução distribuída (tempo de transferência adicionado ao tempo de execução) utilizando o *AW-Kernel* mostrou-se viável e com um desempenho superior à execução local. Observou-se que a heterogeneidade do sistema é fator consideravel no desempenho do sistema como um todo, mas esta característica não foi um empecilho para se obter um bom desempenho, uma vez que o escalonador estava provido com características para prover um balanceamento de cargas.

7.1. Sugestões de Trabalhos Futuros

Como trabalhos futuros, é sugerido o aprimoramento da implementação realizada, pois há lógicas que podem ser melhoradas. Desenvolver utilitários de instalação, configuração e compilação automática da extensão, bem como a elaboração da documentação do *AW-Kernel*.

Outras possibilidades de trabalhos futuros sugeridas:

- implementar outros algoritmos de escalonamento para ambientes distribuídos, assim como o uso de índices de carga e de desempenho (BRANCO, 2004);
- realizar o tratamento de entrada e saída dos *bytecodes* transferidos, provendo assim a transparência de transferência e a transparência de execução dos *bytescodes*;

- alterações na *Java Virtual Machine*, devido a realização do tratamento de entrada e saída remoto;
- viabilidade de implementação de serviços de criptografia em nível de *Kernel*;
- implementar um cachê nos *hosts* clientes para efetuar a conferência de *bytecodes* transferidos.
- efetuar mais estudos de caso, tanto em ambientes homogêneos quanto heterogêneos, provendo balanceamento de cargas e sem prover balanceamento, a fim de avaliar o impacto do uso do *Kernel* na captura ou não de cargas de trabalho para o escalonamento visando ao balanceamento de cargas;
- averiguar o limite de máquinas para a execução de *bytecodes* (em termos quantitativos);
- fazer uso das premissas e dos resultados obtidos para permitir o levantamento de parâmetros para o uso dessas técnicas em dispositivos móveis, que possuem em geral pouca memória e pouco poder de processamento, permitindo que os mesmos possam executar aplicações remotamente.

Referências

(ACCETTA *et al*, 1988) ACCETTA, M.,; BARON, R.,; BOLOSKY, W.; GOLUB, D.; RASHID, R.; TEVANIAN, A.; YOUNG, M. A New *Kernel* Foundation For UNIX Development, Computer Science Department Carnegie Mellon University, 1988.

(ACHCAR; RODRIGUES, 1995) ACHCAR, J.A., RODRIGUES, J. Introdução à Estatística para Ciências e Tecnologia. ICMSC-USP, São Carlos - Apostila de Consulta, 1995.

(BECK et al, 1999) Beck, M.; BOHME, H.; DZIADZKA, M.; KNITZ, U.; MAGNUS, R.; VERWORNER, D. *Linux Kernel* Internals Second Edition, Addison Wesley, 1999

(BOVET e CESATI, 2002) BOVET, P. D., CESATI, M.: Understanding the *Linux Kernel*, 2nd Edition, O'Reilly, 2002.

(BOWMAN *et al.*, 1998) BOWMAN, I., SIDDIQI, S., TANUAM, M. C.. Concrete Architecture of the *Linux Kernel*: Department of Computer Science University of Waterloo, 1998.

(BRANCO, 2004) BRANCO, K. R. L. J. C. Índice de Carga e Desempenho em Ambientes Paralelos/ Distribuídos – Modelagem e Métricas. Dissertação de mestrado. ICMC-USP. 2004.

(CALLAGHAN, 1997) CALLAGHAN, B.. WebNFS The Filesystem for the Internet, Sun Microsystems, Inc., 1997.

(CALLAGHAN, 2000) CALLAGHAN, B. NFS Illustrated, Addison-Wesley, 2000.

(CAPRON *et al.*, 2006) CAPRON, H. L.; JOHNSON, J.A. Introdução a Informática, 8nd Edition, Pearson Prentice Hall, 2006.

(CASAVANT e KUHL, 1988) CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Transactions on Software Engineering. 1988.

(COULOURIS *et al.*, 2001) COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. Distributed Systems: Concepts and Design (Third Edition), Addison-Wesley, 2001.

(COULOURIS *et al.*, 2007) COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. Sistemas Distribuídos Conceitos e Projetos, 4nd Edition, Bookman 2007.

(ENGLER *et al*, 1995) Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr.: Exokernel: An Operating System Architecture for Application-Level Resource Management M.I.T. Laboratory for Computer Science.

(ENGLER, 1998) Dawson R. Engler: The Exokernel Operating System Architecture: MASSACHUSETTS INSTITUTE OF TECHNOLOGY: Submitted to the Department of Electrical Engineering and Computer Scienc, 1998.

(FERRARI e ZHOU, 1987) FERRARI, D.; ZHOU, S. An Empirical Investigation of Load Indices for Load Balancing Applications. In Proceedings of Performance'87, the 12th Int'l Symposium on Computer Performance Modeling, Measurement, and Evaluation. 1987.

(FRANCISCO, 1995) Francisco, W.. Estatística Básica, Unimep. Piracicaba, 2 ª Edição, 1995.

(GRAMA *et al.*, 2003) GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. Introduction to Parallel Computing. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

(KUNG *et al.*, 1991) KUNG, H. T. *et al.* Network-Based Multicomputers: an emerging parallel architecture. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing – Albuquerque, New Mexico, United States. New York, NY, USA: ACM Press, 1991.

(KUNZ, 1991) KUNZ, T. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. IEEE Transactions on Software Engineering. 1991.

(KUROSE e ROSS, 2001) KUROSE, James F.; ROSS, Keith W. Computer Networking, A top-down approach featuring the internet, Addison-Wesley, 2001.

(L4KA, 1994) L4Ka System Architecture Group Universität Karlsruhe (TH) Department of Computer Science: disponivel em http://l4ka.org/publications/, acessado

em 06 de Fevereiro de 2006.

(MCKUSICK *et al.*, 1996) MCKUSICK, M. K.; BOSTIC, K.; KARELS, M. J. ; Quarterman, J. S. The Design and Implementation of the 4.4BSD Operating System, Addison-Wesley, 1996.

(LOVE, 2005) LOVE, ROBERT. *Linux Kernel* Development 2nd Edition, Sams Publishing, 2005.

(MULLENDER, 1993) MULLENDER, S. J. Distributed Systems. 2nd Edition. ACM-Press. Addison-Wesley. 1993.

(MILLS e KAMP, 2000), David L. Mills, Poul-Henning Kamp: The Nanokernel University of Delaware and FreeBSD Project, 2000.

(O'SHANAHAN, 2000) O'SHANAHAN, D. P. CryptosFS: Fast Cryptographic Secure NFS, University of Dublin, 2000

(PAWLOWSKI *et al.*, 1994) PAWLOWSKI, B.; JUSZCZAK, C.; STAUBACH, P. NFS Version 3 Design and Implementation, In Proceedings of the Summer USENIX Technical Conference, p. 137-52, June 1994.

(PLASTINO, 2000) PLASTINO, A. Balanceamento de Carga de Aplicações Paralelas SPMD. Tese (Doutorado). Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, 2000.

(POSTEL e REYNOLDS, 1985) POSTEL, J. REYNOLDS, J. File Transfer Protocol, RFC 959.

(REWINI *et al.*, 1995) REWINI, H. E.; ALI, H. H.; LEWIS, T. Task Scheduling in Multiprocessing Systems. IEEE Computer. 1995.

(RFC1094, 1989) RFC1094. NFS: Network File System Protocol Specification, 1989

(RFC1813, 1995) RFC1813. NFS Version 3 Protocol Specification, 1995

(RFC1831, 1995) RFC1831. RPC: Remote Procedure Call Protocol Specification Version 2, 1995

(RFC2624, 1999) RFC2624. NFS Version 4 Design Considerations, 1999

(RUPPERT e GEUS, 2006) RUPPERT, G. C. S.; GEUS, P. L. Uma Analise de Segurança das Versões de Protocolo NFS, Unicamp, 2006.

(RUPPERT, 2006) RUPPERT, G. C. S. NFS Guard: Uma Solução de Segurança para o Protocolo NFS, Dissertação de mestrado, UNICAMP, 2006.

(SHIVARATRI *et al.*, 1992) SHIVARATRI, N. G.; KRUEGER, P.; SINGHAL, M. Load Distribution for Locally Distributed Systems. IEEE Computer. 1992.

(SILBERSCHATZ *et al.*, 2004) SILBERSCHATZ, A.; GALVIN, P. B.; GAGME, G. Sistemas Operacionais Conceitos e Aplicações, 6nd Edition, Elsevier 2004.

(SPRITE, 2007) The Sprite Operating System. disponivel em http://www.eecs.berkeley.edu/ Research/Projects/CS/sprite/, acessado em 06 de Junho de 2007.

(STERN *et al.*, 2001) STERN, H.; EISTER, M.; LABIARGA, R. Managing NFS and NIS, O´REILLY, 2001.

(STEVENS, 1994) STEVENS, W. Richard. TCP/IP Illustrated, 1nd, Addison-Wesley, 1994.

(TANENBAUM e RENESSE, 1985) TANENBAUM, A.; RENESSE, R. V. Distributed Operating Systems Computing Surverys, ACM, Vol 17. No4, pgs 419-70.

(TANENBAUM e VAN STEEN, 2002) TANENBAUM, Andrew S.; VAN STEEN, Maarten. Distributed systems: principles and paradigms. Upper Saddle River, New Jersey: Prentice Hall, 2002.

(TANENBAUM e WOODHULL, 2000) TANENBAUM, A. S.; WOODHULL, A. S. Sistemas Operacionais Projeto e Implementação, Bookman, 2000.

(TANENBAUM, 1992) TANENBAUM, Andrew S. Modern Operating Systems. New Jersey, Prentice Hall International, Inc. 1992.

(TANENBAUM, 1995) TANENBAUM, Andrew S. Distributed operating systems. New Jersey: Prentice Hall, 1995.

(TANENBAUM, 1999) TANENBAUM, Andrew S. Sistemas operacionais modernos. Rio de Janeiro: Livros Técnicos e Científicos, 1999.

(TANENBAUM, 2001) TANENBAUM, A. S. Modern Operating Systems. 2. ed. Upper Saddle River, New Jersey: Prentice-Hall, Inc., 2001.

(TANENBAUM, 2002) TANENBAUM, Andrew S.: Computer Network, 4nd Edition, Prentice Hall, 2002.

(TANENBAUM, 2003) TANENBAUM, Andrew S. Sistemas Operacionais Modernos. Prentice Hall, 2003.

(ZALUSKA, 1991) ZALUSKA, E. J. Research Lines in Distributed Computing Systems and Concurrent Computation. Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, ICMC/USP, São Carlos/SP. 1991.

(ZHOU *et al.*, 1993) Zhou, S.; Zheng, X.; Wang, J.; Delisle, P. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. Software: Practice and Experience, v.23, 1993.

Apêndice A - Configurações do FTP e NFS para Transferência de *Bytecodes*

Configurações para transferência usando o FTP

A seguir nessa seção são apresentados os processos necessários para que essa plataforma de testes no protocolo FTP estivesse disponível para a realização dos testes. Inicialmente, mostramos as configurações do FTP necessárias para que as transferências dos *bytecodes* fossem realizadas.

```
…
…
…
ServerName                  "ProFTPD Default Installation"
#ServerType                 standalone
ServerType                  inetd
DefaultServer               on

# Port 21 is the standard FTP port.
Port                        21
# Umask 022 is a good standard umask to prevent new dirs and files
# from being group and world writable.
Umask                       022
…
…
…
```

Figura 41 - Configuração do arquivo *proftpd.conf* para inicializar o serviço de FTP.

Configurações para transferência usando o NFS

A seguir nessa seção mostramos as configurações do protocolo NFS necessárias para que as transferências dos *bytecodes* fossem realizadas nessa plataforma.

O arquivo texto */etc/fstab* possui a configuração necessária para a montagem de volumes, este arquivo é consultado pelo comando *mount* para realizar a montagem dos dispositivos. Os arquivos */etc/mtab* e */proc/mounts* contém basicamente a mesma informação, porém apenas os volumes que já foram montados em seu sistema.

Utilizou-se o comando do NFS, representado na Figura 42, para montar todos os volumes que estão especificados no arquivo */etc/fstab*, desta forma, tornou-se possível acessar os sistemas de arquivos remotos.

```
# mount -a
```

Figura 42 - Montagem de todos os volumes especificados no */etc/fstab*.

As configurações de NFS feitas no */etc/exports* do *host* devem ser mostradas conforme representado na Figura 43. As configurações dos *hosts* são iguais, a diferença é o nome do diretório que foi compartilhado.

```
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync) hostname2(ro,sync)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt)
# /srv/nfs4/homes  gss/krb5i(rw,sync)
#
/home/hd_ willy/ 172.16.0.9 (rw,no_root_squash,sync)
```

Figura 43 - Configuração dos sistemas de arquivos remotos.

É representada na Figura 44 as configurações do sistema de arquivo local com os diretórios configurados anteriormente e compartilhados para leitura e gravação.

```
# /etc/fstab: static file system information.
#
# <file system> <mount point>       <type>  <options>                    <dump>  <pass>
Proc            /proc               proc    defaults                     0       0
/dev/sda1       /                   ext3    defaults,errors=remount-ro   0       1
/dev/sda5       none                swap    sw                           0       0
/dev/hdc        /media/cdrom0       udf,iso9660    user,noauto           0       0
/dev/fd0        /media/floppy0      auto    rw,user,noauto               0       0
172.16.0.8:/home/hd_apolo /home/mnt/apolo/  nfs   users,auto,nodev,noexec 0 0
```

Figura 44 - Configuração do sistema de arquivo local.

Algoritmo de Tomada de Tempo do FTP e NFS

Com o auxílio de *headers* (arquivos .h que são incluídos pelo pré-processador) fornecidos por meio dos compiladores C, desenvolveu-se um algoritmo de tomada de tempo para cálculo do tempo das transferências dos *bytecodes* (representados na Figura 45 e Figura 46).

```
…
…
…
004 #include <sys/time.h>
005 #include <stdlib.h>
006
007 int main(int argc,char *argv[])
008 {
009
010   int pid;
011   int estado;
012   struct timeval tv1, tv2;
013   double time_if;
…
…
…
```

Figura 45 - Representação do trecho de código referente à definição dos headers e variáveis.

```
…
…
…
015   if (pid == 0)
016   {
017      printf("pid(processo-filho)=%d\n",getpid());
018      execlp(argv[1]);
019   }
020   else
021   {
022      gettimeofday(&tv1,NULL);
023      wait(&estado);
024      gettimeofday(&tv2,NULL);
025      printf ("pid(processo-pai)=%d\n",getpid());
026        time_if = ((double)(tv2.tv_sec) +
           (double)(tv2.tv_usec)/1e6) - ((double)(tv1.tv_sec) +
           (double)(tv1.tv_usec)/1e6);
027      printf("O tempo de execucao do processo %d e. %lf,",
pid, time_if);
028      printf(" terminando com estado=%d\n", estado>>8);
029   }
030   printf ("Processo %d terminou!\n", getpid());
031   exit(30);
032 }
…
…
…
```

Figura 46 - Representação do trecho de código referente a mostrar o tempo do processo de transferências.