Elcio Luiz Pagani Bortolin

Alta Disponibilidade usando CODA e LVS

Monografia de Pós-Graduação apresentada ao Departamento de Ciência da computação da Universidade Federal de Lavras como parte das exigências do Curso ARL- Administração em Redes Linux.

Orientador
Prof. Jones Albuquerque

Co-Orientador
Prof. Joaquim Quiteiro Uchôa

Lavras
Minas Gerais - Brasil
2005

Elcio Luiz Pagani Bortolin

Alta Disponibilidade usando CODA e LVS

Monografia de Pós-Graduação apresentada ao Departamento de Ciência da computação da Universidade Federal de Lavras como parte das exigências do Curso ARL- Administração em Redes Linux.

Aprovada em *10 de setembro de 2005*

Prof. Joaquim Quiteiro Uchôa

Prof. Herlon Ayres Camargo

Prof. Jones Albuquerque
(Orientador)

Prof. Joaquim Quiteiro Uchôa
(Co-Orientador)

Lavras
Minas Gerais - Brasil

Sumário

Sumário . . . vii
Lista de Figuras . . . x
Lista de Tabelas . . . xi

1 Introdução **1**

2 Computação Baseada em *Cluster* - Visão Geral **3**
2.1 Introdução . . . 3
2.2 Arquitetura de Computadores Paralelos Escaláveis . . . 3
2.3 Arquitetura de um *Cluster* . . . 5
2.4 Classificação dos *Clusters* . . . 7
2.5 Componentes para um *Cluster* . . . 9
2.5.1 Processador . . . 10
2.5.2 Memória e Cache . . . 10
2.5.3 Discos e Sistemas de Entrada e Saída - I/0 . . . 11
2.5.4 Barramento . . . 11
2.5.5 Interconexão do *Cluster* . . . 12
2.5.6 O Sistema Operacional . . . 12
2.6 Ambientes de Programação e Ferramentas . . . 13
2.6.1 *Threads* . . . 13
2.6.2 Sistemas de Passagem de Mensagem - MPI e PVM . . . 13
2.6.3 Sistema de Memória compartilhada - DSM . . . 14
2.6.4 Ferramentas para Análise e Visualização de Desempenho . 14
2.6.5 Ferramentas de Administração do *Cluster* . . . 16

3 Clusters de Alta Disponibilidade **17**
3.1 Introdução . . . 17
3.2 Computação de Missão Crítica . . . 17
3.3 Conceitos de Confiabilidade . . . 19
3.3.1 Falhas, Erros, Defeitos . . . 19

3.3.2 Atributos de Confiabilidade 20
3.3.3 Significado da Confiabilidade 20
3.4 Arquitetura do *Cluster* 20
3.4.1 Sem Compartilhamento x Armazenamento Compartilhado 20
3.4.2 Ativo/*Standby* x Ativo/Ativo 21
3.4.3 Interconexão 23
3.5 Detectando e Mascarando Falhas 24
3.5.1 Auto-Teste 25
3.5.2 Processador, Memória e Barramento 25
3.5.3 *Watchdog* - Temporizadores de Hardware 26
3.5.4 **Watchdog** - *Software* 26
3.5.5 Confirmações, Verificação de Consistência e ABFT (*Algorithm Based Fault Tolerance*) 27
3.6 Recuperação de Falhas 28
3.6.1 Ponto de Verificação e Restore - Checkpointing 28
3.6.2 *Failover* e *Failback* 29

4 Cluster de Alta Disponibilidade - GNU-Linux 31
4.1 Introdução 31
4.2 Linux Virtual Server 31
4.2.1 Visão Geral do LVS 32
4.2.2 Opções para Distribuir a Carga 33
4.2.3 Modos de Balancear a Carga com LVS 35
4.2.4 Planejamento do Balanceamento de Carga 38
4.3 Alta Disponibilidade com LVS 41
4.3.1 Heartbeat + CODA 41
4.4 Instalação 43
4.4.1 Linux Virtual Server 43
4.4.2 Heartbeat 43
4.4.3 CODA 46
4.4.4 Comentários Gerais 53

5 Testes de Desempenho 55
5.1 Sistema de Arquivos CODA 55
5.2 Linux Virtual Server 58
5.3 Comentários Finais 60

6 Conclusão 63

Referências Bibliográficas 66

A Glosário 67

Lista de Figuras

2.1 Arquitetura de um *cluster* de Computador 6

3.1 Cluster sem compartilhamento 21
3.2 Cluster Ativo/Standby 22
3.3 Cluster N-Caminhos 23
3.4 Interconexão do cluster 24

4.1 Visão Geral do Linux Virtual Server, extraido de (PROJECT, 2005) 32
4.2 Exemplo de Configuração DNS 33
4.3 LVS: VS-NAT, fonte (MAGAZINE, 2005) 36
4.4 LVS: VS-NAT - Físico 37
4.5 LVS: VS-Tun, fonte (MAGAZINE, 2005) 38
4.6 LVS: VS-DR, fonte (MAGAZINE, 2005) 39
4.7 Alta Disponibilidade utilizando CODA 42
4.8 Arquivo de configuração do ldirectord 44
4.9 Arquivo de configuração ha.cf 45
4.10 Arquivo de configuração haresource 45
4.11 Arquivo de configuração authkeys 45
4.12 Ilustração sobre o funcionamento do CODA, fonte(BRAAM, 2005) 46
4.13 Terminologia CODA 48
4.14 Organização de uma célula CODA 51

5.1 Lista de arquivos a serem copiados para o volume CODA 55
5.2 Tempo obtido na copia dos arquivos 56
5.3 Tempo obtido na operação de exclusão dos arquivos copiados . . . 56
5.4 Copia dos arquivos compactados 56
5.5 Tempo resultante da descompactação do arquivo 57
5.6 Estado geral do ambiente LVS através da ferramenta IPVSADM . 58
5.7 Estado do ambiente LVS após várias requisições de serviços . . . 59
5.8 Estado do ambiente LVS após retirada do servidor real1 59

5.9 Estado do ambiente LVS após reconectar o servidor real1 60

Lista de Tabelas

2.1 características das Arquiteturas de Computadores Paralelos Escaláveis 4
2.2 Ferramentas de visualização e análise de desempenho 15

À minha esposa Elizabete e aos meus filhos Luiza e Gustavo.

Agradecimentos

Aos professores que contribuíram nesta árdua caminhada.
Ao Professor Jones que aceitou ser meu orientador.
Ao professor Joaquim Quinteiro Uchoa pela paciência na correção dos erros cometidos durante este trabalho.
À minha família.

Resumo

Este projeto visa integrar soluções de *software* livre na composição de um *cluster* de alta disponibilidade, garantindo um nível de qualidade equivalente ao das soluções proprietárias existentes no mercado. O *cluster* de alta disponibilidade é uma infraestrutura capaz de manter a disponibilidade dos serviços prestados por um sistema computacional, através da redundância de *hardware* e de *software*. Na integração destas tecnologias busca-se entendê-las individualmente para prover uma solução simples e flexível, que possa ser otimizada para as particularidades de cada aplicação. Esta solução divide-se em três blocos básicos, que são: monitoração dos *nodes*, balanceamento de carga, sincronização dos dados. Estes três blocos podem ser utilizados em conjunto ou individualmente, podendo atender a diversos propósitos e serviços onde a alta disponibilidade faz-se necessária.

Capítulo 1

Introdução

Servidores de programas para internet suportam aplicações e serviços de missão crítica como transações financeiras, acesso a banco de dados, intranets corporativas, e outras funções essenciais que precisam estar disponíveis 24 horas por dia, 7 dias por semana. Somando a isso, aplicações de rede e servidores necessitam estar aptos ao escalonamento de *performance* para suportar grandes volumes de requisições de clientes sem criar *delays* indesejados.

Clusters de Alta Disponibilidade permitem o gerenciamento de um grupo de servidores independentes como um único sistema, de fácil gerenciamento, e grande escalabilidade.

Pode-se usar serviços de balanceamento de carga para implementar alta disponibilidade e soluções escaláveis para entrega de serviços e aplicações baseadas em TCP/IP (*Transmission Control Protocol/Internet Protocol*).

Esta solução não requer *hardware* especial, pode-se utilizar qualquer computador compatível com IBM-PC neste cluster de alta disponibilidade com balanceamento de cargas.

Utilizando-se Linux e algumas ferramentas gratuitas, distribuídas sob licença GPL(General Public License) é possível construir um ambiente com ótima redundância a falhas. Isso significa que, caso ocorra algum problema com um dos servidores integrantes do cluster, um outro assume o controle e passa a executar as mesmas tarefas.

Visando-se facilidade na compreensão, este trabalho estrutura-se na seguinte forma:

1. Visão geral sobre *clusters*;
2. *Clusters* de alta disponibilidade;
3. *Clusters* de alta disponibilidade usando Linux;

4. Testes de desempenho.

Capítulo 2

Computação Baseada em *Cluster* - Visão Geral

2.1 Introdução

Cada vez mais aplicações precisam de mais poder computacional do que um computador seqüencial pode fornecer. Uma maneira evitar limitação é incrementar a velocidade de operação dos processadores e outros componentes. Todavia, isso é possível atualmente, mas, futuros incrementos vão esbarrar nos limites da velocidade da luz, nas leis da termodinâmica e nos custos proibitivos para a fabricação do processador. Uma solução viável é conectar múltiplos processadores e coordenar seus esforços computacionais. Estes sistemas são conhecidos como computadores paralelos, e eles permitem compartilhar a computação de tarefas entre múltiplos processadores.

2.2 Arquitetura de Computadores Paralelos Escaláveis

Na década passada surgiram vários sistemas de computadores que suportavam alta performance de computação. Suas taxonomias estavam baseadas em como seus diferentes componentes se interconectavam, tais como processadores, memórias, etc. Os sistemas mais comuns são:

- Processamento Massivo Paralelo **(MPP)**
- Multiprocessadores Simétricos **(SMP)**
- Cache Coerente - Memória de Acesso não Uniforme **(CC-NUMA)**

Características	MPP	SMP CC-NUMA	Cluster	Distribuídos
Número de *Nodes*	0 (100)- 0 (1000)	0 (10)-0 (100)	0 (100) ou menos	0 (10) - 0 (1000)
complexidade do *node*	Pequena/Média	Média/Alta	Média	Muito Alta
Comunicação entre *nodes*	Passagem de mensagem/Variáveis compartilhadas para memória distribuída	Compartilhamento Centralizado e distribuído de Memória (DSM)	Passagem de Mensagem	Compartilhamento de Arquivos, RPC, passagem de Mensagem e IPC
Agendamento de tarefas	Única Fila no Host	Única Fila na maioria	Múltiplas filas coordenadas	Filas Independentes
Suporte SSI	Parcialmente	Sempre em SMP e alguns em NUMA	Pretendido	Não
Tipo e cópia do SO no *node*	SOs com N Microkernels monolíticos ou em camadas (Módulos)	Um monolítico em SMP e Vários em NUMA	N Plataformas de SO - Homogêneos ou Micro-Kernel	N Plataformas de SO - Homogêneos
Espaço de Endereçamento	Múltiplo - Único para DSM	Único	Múltiplo ou Único	Múltiplo
Segurança entre *nodes*	Desnecessário	Desnecessário	Requerido se Exposto	Requerido
Proprietário	Uma única organização	Uma única organização	Uma ou mais organizações	Várias organizações

Tabela 2.1: características das Arquiteturas de Computadores Paralelos Escaláveis

- Sistemas Distribuídos
- Clusters

A tabela 2.1 mostra uma comparação das características funcionais e estruturais dessas máquinas, originalmente feita por Hwang e Xu - 1998, (BUYYA, 1999).

Um MPP é um grande sistema de processamento com arquitetura não compartilhada. Consiste em várias centenas de elementos de processamento **(*nodes*)**, cada *node* pode ter uma variedade de componentes de *hardware*, mas, geralmente compõe-se de memória principal e um ou mais processadores. Alguns *nodes* especiais podem conter periféricos especiais como discos ou sistemas de backup. Sistemas SMP possuem de 2 a 64 processadores e possuem arquitetura compartilhada. Nesses sistemas, os processadores compartilham todos os recursos disponíveis (memória, I/O, BUS, etc); uma cópia de sistema operacional executa no sistema.

CC-NUMA é um sistema com arquitetura multiprocessada escalável com cache coerente e acesso não uniforme a memória. Como no sistema SMP, todo processador tem uma visão global do sistema de memória. Este nome (**NUMA - Non**

Uniform Memory Acess) devido ao tempo não uniforme de acesso as posições de memória.

Sistemas distribuídos podem ser considerados uma rede de computadores independes. Possuem múltiplas imagens de sistema, cada *node* executa seu próprio sistema operacional. Cada máquina em um sistema distribuído pode ser por exemplo, combinações de MMPs, SMPs, *clusters*, e computadores individuais.

Um *cluster* é uma coleção de estações de trabalho ou PCs interconectados através de alguma tecnologia de rede. Se os propósitos do *cluster* for computação paralela geralmente será constituído por estações de trabalho de alta performance ou PCs conectatados através de uma rede de alta velocidade. Um *cluster* trabalha com uma coleção integrada de recursos e podem ter uma única imagem de sistema espalhada em todos os *nodes*.

2.3 Arquitetura de um *Cluster*

Um *cluster* é um tipo de sistema de processamento paralelo ou distribuído, que consiste de uma coleção de computadores individuais interconectados, trabalhando juntos como um sistema único, com recursos de computação integrados.

Um *node* pode ser uma máquina mono-processada ou um sistema com múltiplos processadores (PCs, Estações de trabalho ou SMPs) com memória, recursos de I/O e sistema operacional. Um *cluster* se refere a dois ou mais computadores (*nodes*) juntamente conectados. Os *nodes* existentes podem estar em um único gabinete ou fisicamente separados e conectados através de uma rede. Um *cluster* de computadores interconectados, baseado em rede (**LAN-Based**) podem operar como um sistema único para usuários e aplicações. É um sistema que pode fornecer serviços com rapidez e confiança a baixo custo, que anteriormente eram encontrados em sistemas proprietários de memória compartilhada a um custo proibitivo.

A seguir estão os componentes relevantes de um *cluster* de computador (Fig. 2.1):

- Múltiplos computadores de alta *performance* (PCs, Estações de Trabalho ou SMPs);
- Sistema Operacional (*Layered* ou baseado em microkernel);
- Rede de Alta velocidade / *Switches* (Gigabit Ethernet ou Myrinet);
- Interface de Rede (NICs);
- Protocolos e Serviços de comunicação rápida;

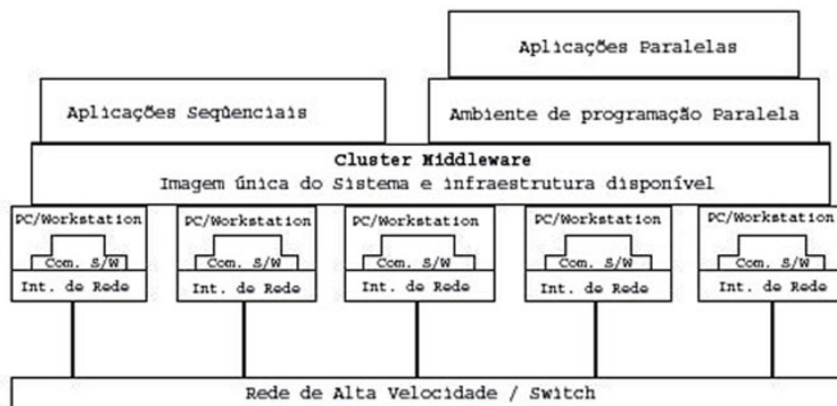

Figura 2.1: Arquitetura de um *cluster* de Computador

- *Cluster Middleware*;
- Ambiente de programação paralela e Ferramentas (Compiladores, PVM e MPI;
- Aplicações (Seqüenciais, paralelas ou distribuídas).

A interface de *hardware* atua como um processo comunicador, sendo responsável por transmitir e receber pacotes de dados entre os *nodes* do *cluster* por intermédio da rede/switch.

O *software* de comunicação oferece uma comunicação de dados rápida e confiável entre os *nodes* do *cluster* e o mundo externo. Freqüentemente, *clusters* com uma rede especial como Myrinet usam protocolos de comunicação como *Ative Messages* [1] para uma comunicação rápida entre os *nodes*. Para fazer isso, esses *clusters* contornam potencialmente o sistema operacional, removendo a crítica comunicação de overheads e fornecendo diretamente nível de usuário para acessar a interface de rede.

Os *nodes* do *cluster* podem operar em conjunto, como um recurso integrado de computação, ou como computadores individuais. O *cluster meddleware* é re-

[1] http://www.cs.cornell.edu/Info/Projects/CAM/sc96/node12.html

sponsável por oferecer uma ilusão de um sistema único e disponibilidade externa sobre uma coleção de computadores independentes mas, interconectados.

2.4 Classificação dos *Clusters*

Os *clusters* oferecem as seguintes características a um custo relativamente baixo:

- Alto Desempenho;
- Expansibilidade e Escalabilidade;
- *Throughput* Elevado;
- Alta Disponibilidade.

As tecnologias de *clusters* permitem às organizações incrementar o poder de processamento, utilizando *hardware* e *software* que são padrão de mercado; podendo ser adquiridos a custo relativamente baixo. Isso assegura expansibilidade e uma boa maneira de aumentar o poder computacional, preservando o investimento existente.

O desempenho das aplicações também melhora com suporte em um ambiente escalável de *software*. Outro beneficio da clusterização é redundância permitindo que um computador backup responda pelas tarefas do computador que falhou ou se tornou inoperante, designado como ***failover***.

Os *Clusters* são classificados em várias categorias baseado em vários fatores, conforme:

1. **Aplicação**

   - Alto Desempenho **HP**
   - Alta Disponibilidade **HA**.

   O *Cluster* **HA** tem a finalidade de manter um determinado serviço de forma segura o maior tempo possível. O *Cluster* **HP** é uma configuração destinada a fornecer um grande poder computacional, maior do que um único computador poderia fornecer em um capacidade de processamento (PITANGA, 2002).

   O *cluster* de Alta Disponibilidade será o foco deste trabalho.

2. **Posse do *Node*** - Possuído por node individual ou dedicado.

   - *Clusters* Dedicados

- *Clusters* Não dedicados.

A distinção entre os dois casos é baseada na posse dos *nodes* em um *cluster*. No caso de *clusters* dedicados, um *node* particular ou individual não possui uma estação de trabalho; os recursos são compartilhados de modo que a computação paralela possa se desempenhar no *cluster* inteiro. Em um cluster não dedicado, o *node* particular ou individual possui uma estação de trabalho e as aplicações são executadas através do roubo de ciclos nos tempos ociosos da CPU. Isso é baseado no fato de que a maioria dos ciclos de CPU das estações de trabalho não são usados. A computação paralela que altera a configuração de uma não dedicada estação de trabalho é chamada **Adaptative Parallel Computing**. Em *clusters* não dedicados, existe uma disputa entre os proprietários das estações de trabalho e os usuários remotos que precisam delas para executar suas aplicações. O primeiro espera uma resposta interativa e rápida de sua estação de trabalho, enquanto o último está interessado na saída rápida da sua aplicação pela utilização de qualquer ciclo sobressalente da CPU. Essa ênfase no compartilhamento nos recursos de processos corroem o conceito da propriedade do *node* e, introduz uma complexidade necessária como a migração de processos e o balanceamento de cargas. Essas estratégias permitem aos *clusters* entregar um adequado e interativo desempenho como prover compartilhamento de recursos exigidos pelas aplicações seqüenciais e paralelas.

3. **Tipo de *Hardware* do *Node*** - PC, Estação de Trabalho, ou SMP.

   - *Cluster* de PCs (CoPs) ou Pilhas de PCs (PoPs)
   - *Clusters* de Estações de Trabalho (*Workstation*)(COWs)
   - *Clusters* de SMPs (CLUMPs)

4. **Sistema Operacional do *Node*** - Linux, NT, Solaris, AIX, etc.

   - Linux *Clusters* (Ex. Beowulf [2])
   - Salaris *Clusters* (Ex. Berkeley NOW [3])
   - NT *Clusters* (Ex. HPVM [4])
   - AIX *Clusters* (Ex. IBM SP2[5])

[2]http://www.beowulf.org/
[3]http://now.cs.berkeley.edu/
[4]www-csag.ucsd.edu/projects/hpvm.html
[5]http://www.findarticles.com/p/articles/mi_m0ISJ/is_n2_v34/ai_17285758

- Digital VMS *Clusters*[6]
- HP-UX Clusters[7]
- Microsoft Wolfpack *Clusters*[8]

5. **Configuração do *Node*** - Arquitetura do *Node* e o tipo de SO carregado.

    - *Clusters* Homogêneos: Todos os *nodes* possuem a mesma arquitetura e executam os mesmos SOs.
    - *Clusters* Heterogêneos: Todos os *nodes*possuem arquitetura diferente e executam SOs diferentes.

6. **Tamanho do *Clusters*** - Baseado na localização e contagem.

    - Grupo de *Clusters* (2-99 *Nodes*): Os *nodes* são conectados por SANs (*System Area Networks*) como Myrinet e são empilhados em um lugar dentro da organização.
    - *Clusters* Departamentais (10-99 *Nodes*).
    - *Clusters* Organizacionais (Várias centenas de *Nodes*).
    - Metacomputadores Nacionais (WAN/Baseados em internet): (Vários sistemas departamentais/organizacionais ou *clusters*).
    - Metacomputadores Internacionais (Baseados em internet): (1000-Vários milhares de *Nodes*).

    Clusters individuais podem ser interconectados para formar um grande sistema (*clusters* de *clusters*), e a própria internet pode ser usada para processamento em *cluster*. O uso de recursos computacionais através de WANs, para alto desempenho de processamento, tem conduzido para o surgimento de um novo campo chamado Metacomputing.

2.5 Componentes para um *Cluster*

As melhorias nas estações de trabalho e no desempenho da rede, também a padronização e disponibilidades das programações em APIs, são responsáveis pela disseminação dos sistemas paralelos baseados em *clusters*. Na seqüência serão descritos alguns dos componentes de *hardware* e *software* mais comumentemente usados na construção de um cluster.

[6]http://whitepapers.techrepublic.com/search.aspx?dtid=2&promo=1500&scid=260
[7]www.hp.com/techservers/clusters/hp-ux_clusters.html
[8]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mscs/mscs/c_gly.asp

2.5.1 Processador

Nas duas últimas décadas ocorreram grandes evoluções na arquitetura dos microprocessadores [9] (Ex. RISC, CISC, VLIW e Vector), isso tornou um pequeno *chip* mais poderoso que supercomputadores de uma década atrás. Recentemente, os pesquisadores têm tentado integrar processador e memória ou interface de rede em um único chip. O projeto Intelligent RAM da Berkeley (IRAM) (BERKELEY, 2005) tenta integrar processador e DRAM em um único chip.

Os processadores Intel são os mais comumentemente usados em computadores baseados em PCs. A geração atual de processadores da familia x86 incluem o Pentium Pro, II, III e IV. Esses processadores não estão em uma escala de grande performance, possuem um nível médio de performance, são processadores mais destinados à estação de trabalho. Na escala de alta performance o Pentium Pro mostra-se muito melhor em operações com inteiros que o UltraSPARC da Sun de mesma velocidade de clock, entretanto as operações de ponto flutuante têm desempenho muito baixo. O Pentium II Xeon, como o Pentium II, utiliza freqüência de 100MHz no barramento de memória. É a mesma freqüência utilizada nas CPUs, superando adições no tamanho do cache L2. Para acompanhar, o chipset 450NX para o Xeon, suporta barramento PCI de 64bits e podem suportar conexões de Gigabit.

Outros processadores populares incluem variantes do x86 (AMD x86, Cyrix x86), Digital Alpha, IBM Power PC, Sun SPARC, SGI MIPS e HP PA. Sistemas de computadores baseados nesses processadores também são utilizados em *clusters*. Como exemplo tem-se o Berkeley NOW que se utiliza do processador SPARC da Sun nos *nodes* do *cluster*.

2.5.2 Memória e Cache

Originalmente a memória presente nos computadores era de 640 KBytes, geralmente incorporada a Placa Principal (*motherboard*). Hoje, um computador geralmente possui entre 128 e 512 KB de memória instalados em slots do tipo SIMM. A capacidade de memória de um computador está em torno de algumas centenas de MB.

O montante de memória requerido por um *cluster* é determinado pela sua finalidade. Programas que são paralelizados podem ser distribuídos pela memória como os processos são entre processadores permitindo escalabilidade. Assim, não é necessário possuir uma RAM que mantenha o processo todo na memória de cada sistema, mas o suficiente para evitar excesso de swapping dos blocos de memória para o disco, pois o acesso ao disco traz um grande impacto no desempenho.

[9] http://www.di.ufpb.br/raimundo/HistoriaDoPC/indice.html

Acesso à memória é extremamente lento se comparada a velocidade do processador, consumindo um tempo que um ciclo de clock da CPU. Os caches são utilizados para manter os blocos de memória recentemente acessados, permitindo um acesso extremamente rápido se a CPU se referenciar a esses blocos novamente. Todavia, o custo das memórias rápidas é alto, bem como a complexidade dos circuitos de controle do cache aumentam com o seu tamanho. Devido a essas limitações, o tamanho do cache fica compreendido entre 8KB e 2MB.

Nos computadores baseados no processador Pentium não é incomum encontrarmos uma largura de 64 bits no barramento de memória e um *chipset* que suporte 2MB de cache externo. Esses incrementos são necessários para explorar todo o poder do Pentium e tornar a arquitetura de memória muito similar ao das estações de trabalho Unix.

2.5.3 Discos e Sistemas de Entrada e Saída - I/0

As melhorias em tempo de acesso dos discos não têm mantido o ritmo de desempenho dos microprocessadores, que possuem o desempenho melhorado em torno de 50% a cada ano. A densidade das mídias magnéticas tem aumentado, reduzindo o tempo de transferência dos discos em aproximadamente 60 a 80% ao ano. No total, somando-se as melhorias no tempo de acesso dos discos e melhorias nos sistemas de mecanismos, estão abaixo de 10% ao ano.

O grande desafio são os aplicativos e a freqüente necessidade de processar grandes volumes de dados. Segundo a lei de **Amdahl**[10] (apud BUYYA,1999), o aumento da velocidade obtido por um processador é limitado pelo baixo desempenho dos componentes seqüenciais do sistema. Conseqüentemente é necessário balancear o desempenho dos sistemas de I/0 com desempenho da CPU. Uma maneira de se conseguir isso é realizar operações em paralelo, como as suportadas pelos sistemas de arquivos paralelos baseados em RAID via *software* ou *hardware*. Implementar RAID via *hardware* é muito caro e uma maneira de fugir deste alto custo é implementar RAID via *software* pela associação dos discos em cada estação do *cluster*.

2.5.4 Barramento

Inicialmente o barramento usado nos PCs possuíam uma largura de 8 bits e velocidade de processamento 5MHz. PCs são sistemas modulares e até recentemente, somente processador e memória estavam localizados na placa principal; outros componentes eram tipicamente encontrados em placas secundarias (filhas), conectadas ao sistema de barramento. O desempenho dos PCs tem aumentado desde que

[10]Gene Amdahl

o barramento ISA foi usado. Posteriormente, o barramento ISA teve sua largura aumentada para 16 bits, contudo não foi suficiente para atender a demanda dos processadores, interface de disco e outros periféricos.

Um grupo de indústrias introduziram o barramento VESA Local Bus, com largura de 32 bits que foi amplamente substituído pelo barramento PCI, que permite 133MB/s de transferência e é usado nos PCs baseados em Pentium e outras plataformas não Intel.

2.5.5 Interconexão do *Cluster*

Os *nodes* de um *cluster* se comunicam sobre redes de alta velocidade, utilizando protocolos padrão como o TCP/IP ou protocolo de baixo nível, *Ative Messages*. Tendo como principal facilidade, é provável que a interconexão seja via *Ethernet* padrão. Em termos de desempenho (latência e largura de banda) esta tecnologia mostra sua idade. Entretanto, *ethernet* é uma tecnologia barata e fácil de implementar compartilhamento de arquivos e impressoras. Uma única comunicação *ethernet* não pode ser usada em computação baseada em *cluster*; sua largura de banda e latência não é balanceada comparado ao poder computacional das estações de trabalho hoje disponíveis. Tipicamente se esperaria que as interconexões do *cluster* excedesse a largura de banda dos 10Mbps/s tendo mensagens de latência menor que 100μs, tais como: *Fast* e *Gigabit Ethernet*, ATM , SCI e Myrinet.

2.5.6 O Sistema Operacional

Um sistema operacional moderno fornece dois serviços fundamentais para os usuários. Primeiro tornar o *hardware* do computador fácil de usar. Isso cria uma máquina virtual que se diferencia da máquina real. Segundo, um sistema operacional compartilha recursos de *hardware* entre usuários. Um dos recursos mais importantes é o processador. Um sistema operacional multitarefa como o UNIX ou Windows NT dividem o trabalho a ser executado entre processos, dando a cada processo memória, recursos de sistema, pelo menos uma *thread* de execução e uma unidade executável dentro do processo. O sistema operacional executa uma *thread* por um curto período de tempo e alterna para outra, executando cada *thread* em volta. Comparado a mono-usuário, o multitarefa permite ao computador executar múltiplas tarefas de uma vez. Por exemplo, um usuário pode editar um documento enquanto outro documento é impresso em segundo plano, ou, um compilador compila um programa qualquer. Cada processo tem seu trabalho finalizado e para o usuário todos os programas parecem executar simultaneamente.

O novo conceito de serviços do sistema operacional é suportar várias *thread* de controle no próprio processo. O conceito trouxe uma nova dimensão para o

processamento paralelo, o paralelismo dentro de um processo, instanceado através de programas. Na proxima geração do *kernel* dos sistemas operacionais, espaço de endereçamento e *thread* serão separados, então, um único espaço de endereçamento pode ter múltiplas *threads* em execução. Programar um processo que possua múltiplas *threads* de controle é conhecido como **multithreading**. A interface de *threads* do POSIX é um ambiente de programação padrão para criar paralelismo e concorrência dentro de um processo.

2.6 Ambientes de Programação e Ferramentas

2.6.1 *Threads*

Threads são um paradigma popular para programação concorrente em máquinas mono e multiprocessadas. Em sistemas com multiprocessadores, as *threads* foram utilizadas primeiramente para trabalhar simultaneamente com todos os processadores disponíveis. Em sistemas monoprocessados, as *threads* são utilizados para alocar os recursos eficientemente. Isso é conseguido pelo acompanhamento assíncrono de uma aplicação para sobrepor a computação e comunicação. Aplicações utilizando-se de *multithreads* oferecem uma resposta rápida para entrada do usuário e a executam rapidamente. Ao contrário da criação de processos, a criação de *threads* é mais barata e fácil de gerenciar. *Threads* se comunicam utilizando variáveis compartilhadas como as que são criadas dentro dos espaços de endereçamento de seus processos pais.

Threads são potencialmente portáveis, pela a existência de uma padronização IEEE para a interface de *threads* POSIX, popularmente chamada de *pthreads*. A padronização da interface *multithreads* POSIX está disponível em PCs, estações de trabalho, SMPs e *clusters*. Uma linguagem de programação como Java possui embutido suporte a *multithereads* habilitando facilmente o desenvolvimento de aplicações *multithreads*. *Threads* tem sido extensivamente utilizadas no desenvolvimento de aplicações e sistemas.

2.6.2 Sistemas de Passagem de Mensagem - MPI e PVM

As bibliotecas de passagem de mensagem permitem que programas paralelos sejam escritos para sistemas de memória distribuída. Estas bibliotecas possuem rotinas para iniciar e configurar o ambiente de mensagem, bem como, o envio e recebimento de pacotes de dados. Atualmente, dois dos mais populares sistemas

de passagem de mensagem são o PVM [11] [12] (*Parallel Virtual Machine*) utilizado em aplicações científicas e de engenharia e, o MPI[13] (*Message Passing Interface*) definido pelo MPI *Forum* [14].

PVM é tanto um ambiente quanto uma biblioteca de passagem de mensagem, que pode ser usado para executar aplicações paralelas em sistemas de supercomputadores através de *clusters* de estações de trabalho. Visto que o MPI é uma especificação para passagem de mensagem designada a ser um padrão para computação paralela distribuída usando explicitamente passagem de mensagem. Esta interface é uma tentativa para estabelecer uma prática, portável, eficiente e flexível da padronização para passagem de mensagem. O MPI está disponível na maioria dos sistemas HPC, incluindo as máquinas SMP.

2.6.3 Sistema de Memória compartilhada - DSM

O paradigma da programação em sistemas de memória distribuída mais utilizado é a passagem de mensagem. O problema deste paradigma é a dificuldade e a complexidade se comparado com a programação para sistemas de memória compartilhada. Sistemas de memória compartilhada oferecem um modelo simples e geral de programação pecando em escalabilidade. Uma solução alternativa com custo reduzido é a construção de um sistema DSM (*Distributed Shared Memory*), exibindo um modelo simples e geral de programação e escalabilidade em um sistema de memória distribuída.

DSM permite a programação de variáveis compartilhadas e podem ser implementadas pelo uso de *software* ou *hardware*. As características dos sistemas de *software* DSM são: geralmente construídos como módulos em separado no topo da interface de comunicação; têm como vantagem a característica de aplicação; páginas virtuais, objetos e tipos de linguagem são unidades de compartilhamento.

2.6.4 Ferramentas para Análise e Visualização de Desempenho

O propósito básico das ferramentas para análise de desempenho é ajudar o programador a entender as características de desempenho de uma aplicação. Em particular, ela deveria analisar e localizar partes de uma aplicação que exibe um baixo desempenho. As ferramentas são mais utilizadas para entender o comportamento de aplicações seqüenciais normais e podem enormemente ajudar em muito na análise das características de desempenho das aplicações paralelas.

---

[11] Desenvolvido pelo Oak Ridge National Laboratory

[12] http://www.inf.puc-rio.br/ñoemi/victal/pvm.html

[13] http://www.lam-mpi.org/mpi/

[14] http://www.mpi-forum.org

Ferramenta	Suporta	URL
AIMS	Biblioteca de instrumentação,monitoração e análise	http://science.nas.nasa.gov/*Software*/AIMS
MPE	biblioteca de log e visualização instantânea de desempenho	http://www.mcs.anl.ogv/mpi/mpich
Pablo	Biblioteca de monitoração e análise	http://www-pablo.cs.uiuc.edu/Projects/pablo
Paradyn	Instrumentação dinâmicas de analises do run-time	http://cs.wisc.edu/paradyn
SyPablo	Biblioteca integrada de instrumentos e análise	http://www-pablo.cs.uiuc.edu/Projects/pablo
Vampir	Biblioteca de monitoração e visualização de desempenho	http://www.pallas.de/pages/vampir.htm
Dimemas	Predição de desempenho para programas de passagem de mensagem	http://www.pallas.com/pages/dimemas.htm
Paraver	Programas de análise e visualização	http://cepba.upc.es/paraver

Tabela 2.2: Ferramentas de visualização e análise de desempenho

A maioria das ferramentas de monitoração de desempenho consiste em alguns ou todos os componentes abaixo:

- Meios de introduzir chamadas de instrumentação para monitoramento de desempenho de rotinas nas aplicações de usuários.

- Uma biblioteca de desempenho para o *run-time*, esta consiste em conjunto de rotinas de monitoração que medem e armazenam vários aspectos do desempenho de um programa.

- Um conjunto de ferramentas para processar e mostrar os dados do desempenho.

É muito importante notar que a instrumentação afeta as características de desempenho das aplicações paralelas assim, o resultado pode ser uma falsa visualização de comportamento do desempenho. A tabela 2.6.4 mostra as ferramentas mais comuns utilizadas para análise de desempenho dos programas que utilizam passagem de mensagem.

2.6.5 Ferramentas de Administração do *Cluster*

A monitoração do *cluster* é uma tarefa desafiadora que pode ser facilitada por ferramentas que permitam observá-lo por inteiro em diferentes níveis utilizando uma interface.

Existem vários projetos que investigam os sistemas de administração de *clusters* suportando computação paralela. NOW, (apud BUYYA, 1999) , a ferramenta de administração da Berkeley faz recolhimento e a armazena os dados em um banco de dados relacional. Ela possui um *applet* Java permitindo aos usuários monitorarem o sistema através de um *browser*. A ferramenta de administração SMILE é chamada de K-CAP, (apud BUYYA, 1999). Este ambiente consiste em *nodes* de cálculo, *node* de gerenciamento e um cliente para controlar e monitorar o *cluster*. K-CAP usa um *applet* Java para conectar o *node* de gerenciamento através de uma URL predefinida. O *node* Relatório de *Status* (NSR) provê um mecanismo padrão para medidas e acessos as informações de status. As aplicações paralelas ou ferramentas podem acessar o NSR através de sua interface. PARMON (INDIA, 2005)[15] é um ambiente detalhado para monitorar grandes *clusters*usando técnicas de cliente-server para prover acesso transparente a todos os *nodes* monitorados.

[15]http://www.buyya.com/parmon/

Capítulo 3

Clusters de Alta Disponibilidade

3.1 Introdução

Os sistemas de computação baseados em *clusters* surgiram há mais de uma década e seguiram duas diferentes linhas da computação por motivações ligeiramente diferentes: Alto Desempenho (HP - *High Performance*) e Alta disponibilidade (HA - *High Availability*).

Clusters são geralmente definidos como um Sistema Paralelo ou Distribuído, consistindo em uma coleção de computadores inteiramente interconectados, em que os recursos unificadados são utilizados como um único computador. *Clusters* **HP** são vistos pela perspectiva do Alto Desempenho e Escalabilidade, são referidos como *Clusters* de Estação de Trabalho (WSCs - *Workstation Clusters*) ou Rede de Estação de Trabalho (NOWs - *Network of Workstations*); *Clusters* **HA** possuem a perspectiva da Alta Disponibilidade.

Clusters de Alta Disponibilidade são construídos a partir de dois *nodes* com necessidades crescentes de desempenho; estão aptos a suportar mais de dois *nodes* e fornecer mecanismos adicionais para agregar ao desempenho dos *nodes* participantes através de balanceamento de carga.

3.2 Computação de Missão Crítica

Atualmente os sistemas de informações são essenciais às instituições, aumentando a demanda para aplicações de alta disponibilidade. Esta classe de programas sensíveis inclui processamento de transações on-line, comércio eletrônico, serviços de HTTP, sistemas de controle, aplicações cliente/servidor, etc. Estes servidores precisam operar continuamente, atendendo requisições de seus clientes, conseqüentemente necessitando de alta disponibilidade. A dependência das organizações pelas

aplicações de missão crítica tem aumentado o custo total de propriedade (TCO - *Total Cost of Ownership*), especialmente dos sistemas de **HA** que protegem o sistema contra paradas, mas é menor que o custo dos sistemas ficarem indisponíveis. Um meio efetivo de implementar **HA** para esses sistemas é um modelo de *cluster*.

Clusters são construídos sob o avanço da computação segura ocorrido na última década. Históricamente um dos primeiros mecanismos de proteção para aplicações críticas foi o *backup offline* dos dados (*Cold Backup* - Backup a Frio). *Backups* não tornam o sistema mais disponível, mas permitem uma rápida recuperação do sistema caso ocorra algum desastre.

Uma evolução do conceito de *backup* é o ***hot-backup*** (Backup a Quente). O *Backup* a Quente protege os dados, não apenas em um certo período do dia, mas, continuamente utilizando-se de um sistema de espelhamento onde todos os dados são replicados. A tecnologia RAID (*Redundant Arrays of Inexpensive Disks*) foi introduzida para proteger a integridade dos dados. Novamente não garante disponibilidade. Todavia, se o servidor falhar, pode-se repará-lo manualmente para que o sistema volte a estar disponível.

Proteções ambientais foram empregadas também com a finalidade de proteger os sistemas dos perigos inerentes do ambiente. Incluem-se proteção contra surtos de energia e UPSs (*Uniterruptible Power Supplies*), para permitir operações durante faltas de energia. Evidentemente que isso é efetivo em faltas transientes. Faltas mais longas ou permanentes devem ser supridas através de redundância.

Até alguns anos atrás, depois de montar um sistema como descrito acima, esperava-se que tudo funcionasse sem problemas, o que geralmente não ocorria. Como alternativa consiste-se em utilizar redundância avançada de *hardware*, para aplicações que exigem extrema alta disponibilidade. A redundância nesses sistemas é explorada em todos os níveis incluíndo-se fontes de energia, portas I/O, CPUs, discos, adaptadores de rede e redes físicas com o objetivo de eliminar ponto de falha dentro da plataforma. Na ocorrência de uma falha no hardware no qual funciona uma aplicação crítica, o componente duplicado estará sempre disponível para assegurar que a aplicação tenha o recurso para manter-se em execução. Esses sistemas também são chamados de Tolerante a Falhas (*Fault Tolerant*), sendo capaz prover uma disponibilidade quase continua; em torno de 99.999% de disponibilidade ou um tempo de parada de até 5 minutos por ano.

Esse nível de disponibilidade geralmente requer o uso de *hardware* e *software* proprietário, afastando os usuários comuns desta tecnologia. A solução é combinar computação altamente escalável e altamente de confiança com componentes abertos disponíveis. Isso pode ser feito através de *clusters*. O primeiro passo nessa direção é empregar dois sistemas similares conectados através de uma LAN

(*Nodes* primário e secundário). O *node* secundário possui um *backup* dos dados e processos do servidor primário, podendo então substituí-lo em caso de falhas.

Além da Alta Disponibilidade, a habilidade de escalar transformou-se em exigência. O alto desempenho juntou-se a alta disponibilidade tornado-se um requerimento adicional em sistemas de missão crítica.

O Fato é que os *clusters* possuem a habilidade de fornecer alto desempenho e escalabilidade. Em contraste com os esquemas de *failover* puro, um *node* do *cluster* que aguarda por uma falha de um outro *node* não precisa ficar inativo enquanto aguarda a falha ocorrer. Sua capacidade de processamento pode ser aproveitada. Utilizando-se de *software* adequado, a carga pode ser compartilhada com outros membros do *cluster*. Esta evolução marca a ascensão dos ***Clusters* de alta disponibilidade escaláveis**.

Os clusters de missão crítica resultam da combinação das seguintes potencialidades:

- **Disponibilidade:** Capacidade de um sistema manter-se disponível mesmo em caso de falhas.
- **Escalabilidade:** Capacidade de ampliação do número de *nodes* conforme o aumento da carga.
- **Desempenho:** Capacidade efetuar distribuição da carga de trabalho.
- **Gerenciabilidade:** Capacidade se ser gerenciado como um sistema único.

3.3 Conceitos de Confiabilidade

3.3.1 Falhas, Erros, Defeitos

Quando um programa executando em um *cluster* apresenta resultados incorretos, isso ocorre devido a defeitos. Defeitos são o que se tenta prevenir. Para fazer isso, é necessário receber algum tipo de aviso para se tomar as medidas necessárias evitando-se a ocorrência do defeito. Felizmente antes que o comportamento do sistema externo seja afetado internamente pode-se chegar a um estado inválido (um estado inválido é chamado de erro). Se o erro for detectado, o qual, pode ser um valor incorreto de variável, talvez seja possível compensar isso reparando antes que o sistema venha a falhar. O tempo entre a primeira ocorrência do erro e o seu momento de detecção é chamado de latência de detecção. Quanto antes o erro for detectado mais efetiva será a correção. Obviamente, erros podem ser provocados por fontes de diversas origens como interferências eletromagnéticas e ainda, possuir tempo de duração variados, como transientes e pequenos surtos de energia, que nesse caso são chamados de falhas.

3.3.2 Atributos de Confiabilidade

A confiabilidade pode ser vista sob diferentes perspectivas, dependendo da aplicação. Assim como o sistema provê disponibilidade, mede-se a probabilidade de tempo que o sistema consegue fornecer de maneira adequada os serviços. São atributos da confiabilidade o Tempo Médio entre as Falhas - **MTBF** (*Mean Time Between Failures*) que é proporcional a confiabilidade e o Tempo Médio para Reparar a Falha - **MTTR** (*Mean Time To Repair*).

A disponibilidade pode ser expressa por **MTBF /(MTBF + MTTR)**.

3.3.3 Significado da Confiabilidade

Existem essencialmente duas maneiras de resolver o problema da confiabilidade. Uma é prevenir a ocorrência de falhas - Prevenção de Falhas (***Fault Prevention***), através de projeto conservador, provas formais, exaustivos testes, etc. Outra alternativa é impedir que os erros de tornem defeitos - Tolerante a Falhas (***Fault Tolerance***). Estas duas aproximações são em fato complementar. Um sistema de qualidade ruim não deve ser executado dentro de um sistema de alta disponibilidade apenas por ser tolerante a falhas. A prevenção de falhas não deixa um sistema totalmente livre de falhas.

3.4 Arquitetura do *Cluster*

Os *clusters* podem ser configurados de várias maneiras observando-se a interconexão do sistema de armazenamento, clientes, monitorização, nível de compartilhamento de recursos, número máximo de *nodes*, granularidade do *failover*, capacidade de utilizar simultaneamente todos os recursos do cluster, entre outros. Entretanto, podem ser classificados de acordo com alguma característica especifica como *failover*, interconexão, e arquitetura de armazenamento.

3.4.1 Sem Compartilhamento x Armazenamento Compartilhado

No modelo de *cluster* não compartilhado, cada *node* possui sua própria memória e também seu próprio recurso de armazenamento (Figura 3.1). Para que o *cluster* sem compartilhamento permita aos *nodes* acessar recursos ou dispositivos comuns, esses recursos na totalidade são possuídos e gerenciados por um sistema único. Isso evita a complexidade dos esquemas de *cache* coerente e gerenciamento de bloqueio distribuído (*Distributed Lock Managers* - DLM).[1] Um dos benefícios desse

[1]DLMs são utilizados para serializar requisições de acesso e negociar conexões para recursos compartilhado

modelo é a eliminação do ponto único de falha dentro do sistema de armazenamento do *cluster*. Se o sistema de compartilhamento falhar, todos os servidores perderão o acesso aos dados. Em virtude da falta de disputa e custo adicional, este modelo pode ser completamente escalável. Existem várias estratégias visando reduzir o custo adicional de largura de banda quando discos locais são espelhados através da rede. Uma segunda placa adaptadora de rede pode ser usada para dar suporte ao espelhamento e isolar o tráfego da rede por onde os clientes se conectam.

No modelo de armazenamento compartilhado, ambos os servidores compartilham o mesmo recurso de armazenamento, como conseqüência precisam sincronizar-se entre os acessos a discos para manterem a integridade dos dados. O gerenciamento de bloqueio distribuído é utilizado nesses casos, comprometendo a escalabilidade.

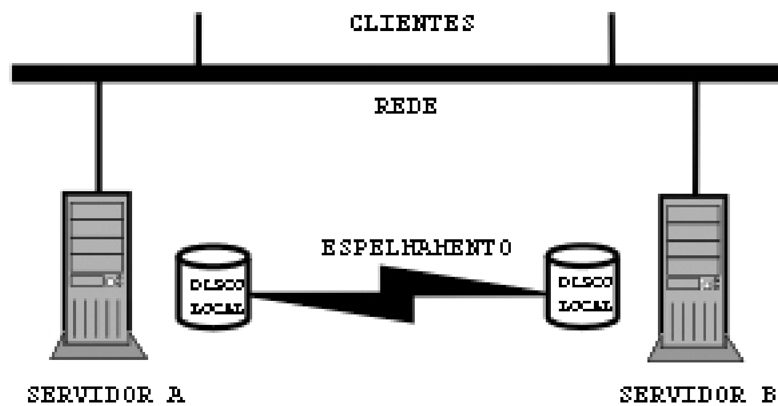

Figura 3.1: Cluster sem compartilhamento

3.4.2 Ativo/*Standby* x Ativo/Ativo

A configuração Ativo/*Standby* é chamada de *backup* a quente (*Hot Backup*) - Figura 3.2. Neste modelo é no servidor primário que a aplicação crítica é executada, e o servidor secundário é usado como um *backup* ou sobressalente a quente e, normalmente, está em modo *standby*. As duas máquinas não precisam necessariamente serem idênticas: a máquina *backup*, precisa ter recursos (memória, disco, conectividade, etc) o suficiente para atender pelo menos os requisitos mínimos da(s) aplicação(ões) que está(ão) executando no servidor primário. Na configuração ativo/ativo, todos os servidores do *cluster* estão ativos e não ficam em estado

ocioso esperando uma falha ocorrer para uma retomada. Esta solução algumas vezes é utilizada para proporcionar *failover* bidirecional pois, aplicações críticas podem ser executadas em alguns servidores como melhor suporte do que em outros.

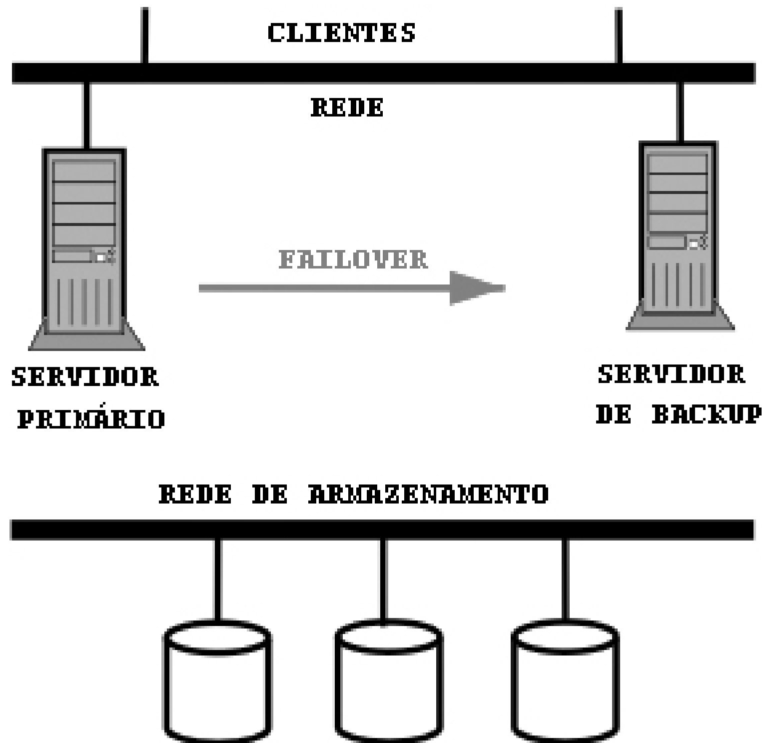

Figura 3.2: Cluster Ativo/Standby

Uma outra configuração que é basicamente uma extensão do modelo ativo/ativo é denominada N-Caminhos (N-*Way*) - Figura 3.3, possui vários servidores ativos que fazem *backup* de um para outro. Por exemplo, o servidor A pode fazer *backup* das aplicações dos servidores B e C, enquanto estes fazem *backup* de suas aplicações. Quando um defeito ocorrer em qualquer um dos servidores, as aplicações protegidas são transferidas do servidor com defeito para os servidores de *backup*.

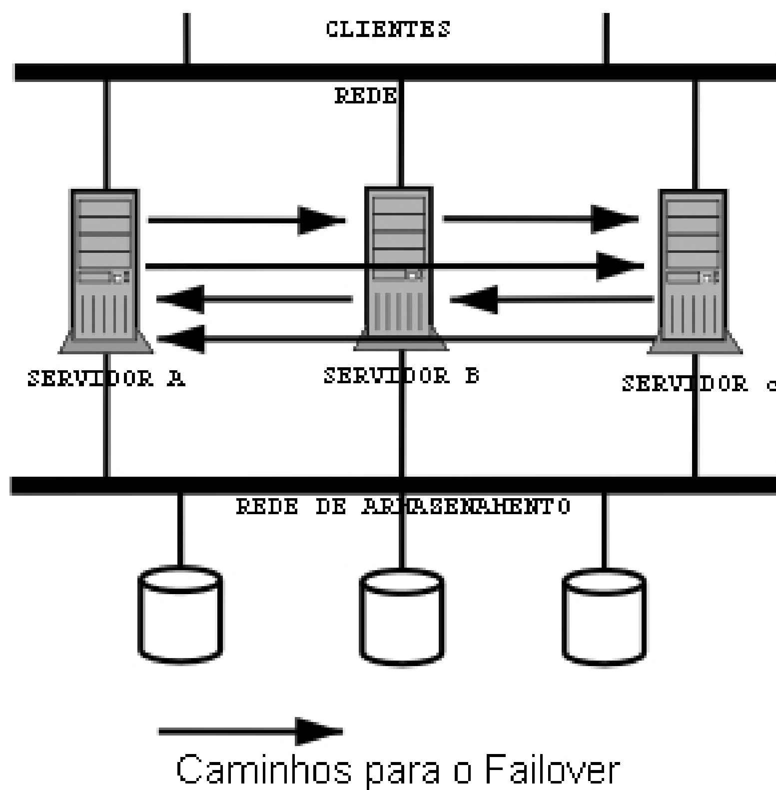

Figura 3.3: Cluster N-Caminhos

3.4.3 Interconexão

Existem três tipos de interconexão a se considerar em um *cluster*: rede, armazenamento e monitoração - Figura 3.4. A interconexão de rede se refere a comunicação entre os sistemas clientes e o *cluster*. Geralmente é sobre *ethernet* ou *fast ethernet*. A interconexão de armazenamento se refere a tecnologia que provê as conexões de I/O entre os *nodes* do *cluster* e o sistema de armazenamento em disco. Tipicamente este tipo de tecnologia fica entre SCSI e *Fiber Channel*.

Fiber Channel Arbitrated Loop (FCAL), permite conexões rápidas - 100 Mbps e comprimento de cabos maiores que 10 Km. SCSI é mais popular, entretanto SCSI tem um inconveniente importante que é o limite de distância que efetivamente acaba limitando a distância entre os servidores clusterizados. Esta limitação pode ser superada pela utilização das tecnologias estendidas do SCSI, mas definiti-

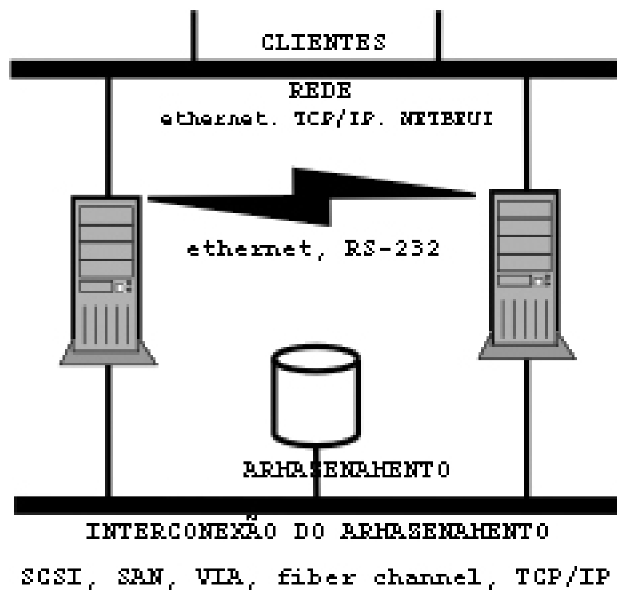

Figura 3.4: Interconexão do cluster

vamente a tecnologia *Fiber Channel* é mais eficiente nas estratégias de recuperação de desastre.

A interconexão de monitoração se refere a uma mídia de comunicação adicional utilizada para os propósitos de monitoração. As interconexões de rede e armazenamento também podem ser utilizadas para monitoração. Entretanto, exige-se pelo menos uma conexão serial entre os servidores para este propósito.

3.5 Detectando e Mascarando Falhas

A confiabilidade é construída sobre diversas camadas. Essas camadas compreendem várias técnicas baseadas em qualquer *software* ou *hardware*, entretanto o ponto inicial para tolerar erros é sempre detectá-los. Os mecanismos de detecção de erros podem ser implementados tanto por software quanto por hardware em vários níveis de abstração. Esses mecanismos podem ser classificados de acordo com duas importantes características: a cobertura de erros e latência de detecção. A primeira corresponde a porcentagem de erros que são detectados, enquanto a segunda, representa o tempo gasto para detecção de um erro.

Em uma latência de detecção maior, cresce a probabilidade do erro se estender para outras partes do sistema e contaminar aplicações relevantes e o sistema de

dados. Após a detecção do erro, a camada de diagnostico tenta coletar o máximo de informações quanto possível sobre a localização, origem e quais as partes do sistema foram afetadas. O diagnóstico é de máxima importância, principalmente em *clusters*. Em *clusters* um erro detectado em um *node* geralmente causa a transferência da aplicação hospedada para a máquina de *backup*.

Depois do diagnóstico, as ações de recuperação e reconfiguração podem ser tomadas.

3.5.1 Auto-Teste

Erros gerados por falhas permanentes podem ser detectados por auto-teste. Usualmente, esses mecanismos de detecção de erros consistem na execução de programas específicos exercitando diferentes partes do sistema (processador, memória, I/0, comunicações, etc.) e validam a saída na expectativa de encontrar resultados conhecidos. Testes podem ser executados periodicamente depois da maquina ser ligada, ou depois da ocorrência de um erro para localizar sua origem. Auto-testes não são muito efetivos na detecção de falhas transientes pois os efeitos tendem a desaparecer com o tempo.

3.5.2 Processador, Memória e Barramento

Nas modernas estações de trabalho e PCs utilizados na construção de *clusters* existem recursos avançados de detecção e correção de erros. Memórias primárias possuem ao menos alguns *bits* de paridade, as quais detectam uma única inversão de *bit*. O sistema de barramento também pode checar a paridade do circuito, com especial inclusão do microprocessador. A correção e detecção de erro (EDAC - *Erro Detection And Correction*) tornaram-se *chips* e são cada vez mais comuns no *hardware* das estações de trabalho. Estes *chips* monitores procuram no sistema de barramento por erros e possuem potencial para detectar até dois *bits* de erro, mas só corrigem erros de um *bit* [2].

Os microprocessadores dentro dos *node* do *cluster* também possuem vários mecanismos de detecção de erros, denominados detecção de instrução ilegal, acesso ilegal e instruções privilegiadas entre vários outros.

Estes tipos de mecanismos são bons para detectar certos tipos de erros, tais como erros de controle de fluxo, mas falham quando o problema ocorre em níveis mais altos ou afetam dados de alguma outra maneira. Por exemplo, se uma aplicação parar porque entrou em um laço infinito, não será identificado pela detecção de erro embutida no hardware.

[2] Também conhecido como SECDED (Single Error Correction Double Error Detection)

3.5.3 *Watchdog* - Temporizadores de Hardware

Um temporizador do *watchdog* é uma maneira apropriada de rastrear as funções de processos. Uma temporização é mantida como um processo separado daquele que é checado. Se o temporizador não resetar antes de expirar o processo correspondente provavelmente travou ou falhou de alguma maneira. Supõe-se que qualquer falha ou corrupção do processo checado fará com que falhe o *reset* do *wachdog*. Entretanto, a eficácia desse recurso é limitada pela falta de checagem nos dados e resultados. A função dos temporizadores é indicar possíveis falhas em processos. Um processo pode falhar parcialmente produzindo erros e até dar *reset* em seu temporizador.

O conceito de temporizador *watchdog* podem ser implementados em *software* ou *hardware*; tanto os processos monitorados quanto temporizadores podem ser executados no mesmo *hardware*.

3.5.4 Watchdog - *Software*

Um *software watchdog* é um processo que monitora outro(s) processos em busca de erros. A monitoração é realizada geralmente com exatidão diferente, pois, necessita que o processo monitorado esteja ciente da monitoração do *watchdog* e coopere em sua tarefa.

O *watchdog* monitora a aplicação até que eventualmente deixe de funcionar, e sua única ação é o relançamento da aplicação. Se o processo monitorado é instruído a cooperar com o *watchdog*, então, a eficiência cresce substancialmente. A cooperação com o *watchdog* pode ser realizado através de vários mecanismos como descrito a seguir:

Heartbeats

O *heartbeat* é uma notificação periódica enviada pela aplicação para o *watchdog* para assegurar que esta permanece ativa. Isto pode consistir em uma aplicação inicializar uma mensagem "Eu estou viva.- "*I'm Alive*.", ou um esquema de requisição/resposta no qual o *watchdog* faz uma requisição a aplicação - "Você está viva?- "*Are you Alive*?", para se certificar que esta permanece ativa aguarda por um reconhecimento. Em ambos os casos, quando um *timeout* especificado expira pelo lado do *watchdog*, este assume que a aplicação tenha travado ou deixado de funcionar. Então, a ação é matar o processo e novamente iniciar a aplicação. *Watchdogs* podem coexistir no mesmo sistema das aplicações protegidas ou em outro sistema. No primeiro caso, o *watchdog* fica sem uso se o próprio sistema falhar. Todavia, se o *watchdog* estiver em outro sistema irá detectar o problema.

Algumas soluções de *cluster* fornecem vários caminhos alternativos para o *heartbeat*, por exemplo, **LifeKeeper** (STEELEYE, 2005), que suporta *heatbeats* através de **TCP/IP, RS232 e Barramento SCSI compartilhado**. Todavia, se a interface de rede falhar, o LifeKeeper irá noticiar isto e obterá melhor informação para se decidir a fazer ou não a comutação.

Notificações de Inatividade

A idéia por trás das notificações de inatividade de uma aplicação é informar o *watchdog* sobre os períodos em que a aplicação fica inativa ou, não esteja fazendo nenhum trabalho útil (um servidor sem requisição de clientes). O *watchdog* pode então tomar ações preventivas como simplesmente reiniciar a aplicação. Esta ação que parece sem sentido no primeiro momento, entretanto, o *software* envelhece (HUANG C. KINTALA; FULTON, 1995), isto é, decorrido um longo tempo da sua inicialização, o seu desempenho pode degradar e apresentar ineficiência na utilização dos recursos computacionais que o suportam, tais como, estouro de memória, bloqueio indevido de arquivos, fragmentação de dados com perda de integridade, perda de apontadores, uso indevido de memória virtual, etc.

3.5.5 Confirmações, Verificação de Consistência e ABFT (*Algorithm Based Fault Tolerance*)

A verificação de consistência é uma técnica simples de detecção de falhas que pode ser implementada tanto em nível de *software* quanto em *hardware*. uma verificação de consistência é executada verificando que os resultados intermediários ou finais da mesma operação são razoáveis, em uma base absoluta, ou como apenas uma função de entrada de outros dados usados que serão usados para derivar o resultado. No nível de *hardware* está embutido a verificação de consistência de endereço, *opcodes* e operações aritméticas. A confirmação de endereço consiste no simples verificar se o endereço que está sendo acessado existe.

Outra forma simples de verificar a consistência é a verificação de escala - confirmando que o valor computado está dentro de uma escala válida de valores. Um exemplo simples, é uma válvula onde a abertura possa variar dentro de uma escala válida de 0 a 100

A nível de algoritmo, uma técnica similar é usada baseando-se preferencialmente no algoritmo de tolerância a falhas - ABFT. Nesta aproximação, após ter executado um algoritmo, a aplicação executa uma verificação de consistência específica para esse algoritmo permitindo uma verificação rápida e fácil quanto a exatidão dos resultados (HUANG; ABRAHAM, 1984).

3.6 Recuperação de Falhas

3.6.1 Ponto de Verificação e Restore - Checkpointing

O ponto de verificação é uma técnica que permite um processo salvar seu estado em intervalos regulares durante a execução normal, de modo que possa ser restaurado caso ocorra uma falha, visando reduzir a quantidade de trabalho perdido. Usando pontos de verificação, quando uma falha ocorre, o processo afetado pode simplesmente ser reiniciado do último ponto de verificação (estado) salvo, evitando iniciar o processo desde o início. Esta técnica é especialmente interessante para proteger aplicações que executam por períodos de tempo muito longos, onde podem ocorrer falhas transientes.

Técnicas do tipo pontos de verificação, geralmente são classificadas de acordo com as seguintes características:

- Transparência;
- Dados a serem inclusos no ponto de verificação;
- Intervalos de ocorrência do ponto de verificação.

Pontos de verificação podem ser transparentes e incluídos automaticamente em tempo de execução ou pelo compilador, ou incluídos manualmente pelo programador. Em aproximações transparentes, o ponto de verificação consiste em um instantâneo global de estado dos endereços do processador, incluindo todos os dados dinâmicos do Sistema Operacional. Outras aproximações de transparência incluem somente o contexto interno do processador, a pilha e os segmentos estáticos e dinâmicos de dados. Em outros casos, a transparência do ponto de verificação faz com que uma grande quantidade de dados sejam salvos desnecessariamente, por não ser possível saber quais dados realmente são críticos para a aplicação que esta sendo executada.

No modo manual, o programador é responsável por definir quais dados realmente são críticos para aplicação e como deve ser o ponto de verificação.

O intervalo do ponto de verificação é importante e, consiste simplesmente em um intervalo de tempo entre pontos de verificação consecutivos. Um ótimo intervalo de tempo para o ocorrência do ponto de verificação não é muito fácil de ser definido, depende de vários fatores como freqüências da falhas, carga de trabalho do sistema, tempo total de execução, da carga adicional imposta pela própria execução do ponto de verificação e do nível de segurança desejado. Este problema é crítico quando o ponto de verificação é inserido automaticamente. Quando o ponto de verificação é inserido pelo programador da aplicação, se torna mais fácil

de controlar todas essas variáveis. Em outras palavras, o ponto de verificação pode ser inserido em um ponto chave do algoritmo.

O lugar onde o ponto de verificação armazena os dados também é crítico para o sucesso de uso desta técnica. No mínimo designar um armazenamento estável com redundância de *hardware* e *software* e imunes a deterioração de memória. Sendo ainda recomendado que as operações de leitura e escrita sejam feitas de maneira atômica. Para pontos de verificação muito grandes, pode-se também utilizar dispositivos de armazenamento em fita.

3.6.2 *Failover* e *Failback*

O processo de *failover* é o núcleo do *cluster* de alta disponibilidade. É uma situação simples, em que a falha de um *node* causa o chaveamento para um *node* alternativo ou *node* de *backup*. o *failover* pode ser totalmente automático e transparente, sem necessitar da intervenção do administrador ou reconexão manual do cliente.

O processo reverso é o *failback* e consiste basicamente em mover as aplicações clientes para o servidor original após o reparo. Como o *failover* este processo pode ser automático e transparente.

Os serviços de manutenção podem ser desempenhados, simplesmente chaveando as aplicações protegidas para um segundo servidor. Manutenções on-line reduzem o tempo de parada para manutenção, tais como *upgrade* de *software* e do Sistema Operacional.

Capítulo 4

Cluster de Alta Disponibilidade - GNU-Linux

4.1 Introdução

Como descrito no capítulo 3, a estratégia para se conseguir disponibilidade ininterrupta 24 horas ao dia e 365 dias ao ano será replicar várias partes do sistema, tantas quantas seja possível e possibilitar que certas partes deste sistema tomem lugar de outras que venham a falhar de forma automática e transparente.

4.2 Linux Virtual Server

O projeto Linux Virtual Server - (**LVS**) fornece os softwares necessários e toda informação para montar um **servidor virtual** e facilmente escalável em um *cluster* utilizando o sistema operacional Linux. Para o usuário final existirá somente um servidor e não saberá tratar-se de um *cluster*.

O Linux Virtual Server executa em maquinas Linux tanto para servidores como para os equipamentos que fazem balanceamento de cargas (O ponto de entrada de todo *cluster* que direcionará o trafego para cada um dos servidores reais). O LVS assume o endereço IP principal do *cluster* e passa responder por ele perante aos clientes. O *software* do Linux Virtual Server está disponível sob a licença GNU General Public Licence (GPL). O Linux Virtual Server atualmente é utilizado em sites como o portal Linux (http://www.linux.com), no portal de download Sourceforge (http://www.sourceforge.net) e muitos outros.

4.2.1 Visão Geral do LVS

O *layout* mostrado na figura 4.1 nos da uma idéia geral de um *cluster* usando LVS.

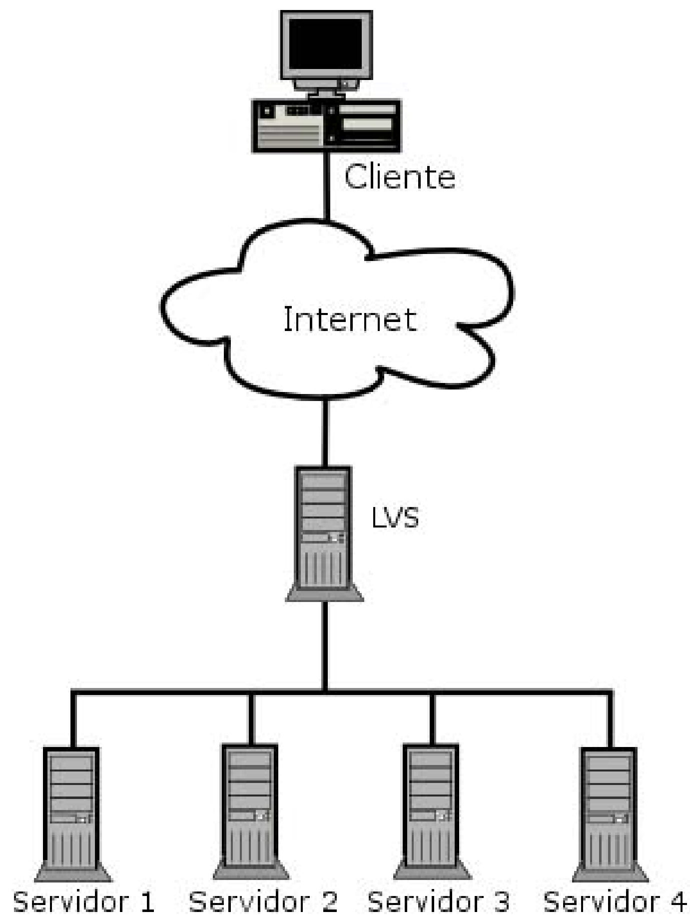

Figura 4.1: Visão Geral do Linux Virtual Server, extraido de (PROJECT, 2005)

O usuário se conecta através da internet ao *cluster* que está ligado ao balanceador de carga. Este equipamento está conectado através da rede LAN ou WAN aos demais *nodes* do *cluster* (os servidores reais) e se encarregará de encaminhar a requisição ao *node* que terá a melhor condição de atendimento (menor carga se esta for a opção escolhida). Cada *node* poderá ter acesso direto a internet e responder direto ao cliente ou será feita através do balanceador de carga.

Pode-se conseguir escalabilidade muito facilmente, bastando para isso adicionar mais equipamentos a LAN, aumentando assim o número de servidores reais (*nodes* do *cluster*). Consegue-se aqui também alta disponibilidade por se possuir vários servidores e, se algum falhar outro assume sua carga. Existe nessa configuração um ponto de falha que é o próprio balanceador de carga. Pode-se nesse caso utilizar mais de um equipamento para fazer o balanceamento de carga, dessa forma aumentando o grau de disponibilidade.

4.2.2 Opções para Distribuir a Carga

Existem várias formas de montar um *cluster* e distribuir a carga entre os *nodes*. O método mais simples é mediante um DNS Round-Robin. Para se obter um balanceamento de carga com DNS faz-se necessário definir várias entradas tipo A no arquivo de configuração do domínio em questão. A figura 4.2 mostra um exemplo para o arquivo de configuração de um servidor DNS:

```
@       IN      SOA     dns1.exemplo.com.br.    email.exemplo.com.br.    (
                        01      ; serial
                        03H     ; refresh
                        15M     ; retry
                        1W      ; expiry
                        1D      ) ; minimum
        IN      NS      dns1
        IN      NS      dns2
        IN      MX      10     mail
dns1    IN      A       1.2.3.41
dns2    IN      A       1.2.3.42
mail    IN      A       1.2.3.50
www     IN      A       1.2.3.1 # Servidor 1
        IN      A       1.2.3.2 # Servidor 2
        IN      A       1.2.3.3 # Servidor 3
        IN      A       1.2.3.4 # Servidor 4
ftp     IN      A       1.2.3.1 # Servidor 1
        IN      A       1.2.3.2 # Servidor 2
        IN      A       1.2.3.3 # Servidor 3
        IN      A       1.2.3.4 # Servidor 4
```

Figura 4.2: Exemplo de Configuração DNS

Para cada requisição de resolução de nomes para www.exemplo.com.br ou ftp.exemplo.com.br, o servidor em princípio devolverá um IP distinto. Porém essa facilidade aparente encontra um inconveniente, o *cache* de DNS. Os servidores DNS possuem *cache* com as últimas requisições realizadas visando agilizar o processo de resolução. Isso pode ser um problema porque o balanceamento não seria feito de forma justa para todos os servidores WWW ou FTP. Uma "solução"que apenas ameniza o problema é a diminuição do TTL (time to live), que é o tempo de vida de um registro em uma *cache* de DNS em segundos. Então com um TTL baixo faz-se com que os outros servidores DNS não mantenham por muito tempo o registro em seus caches. Desta forma consegue-se distribuir a carga entre os nodes de

uma forma pseudo-aleatória conforme a chegada das requisições. Este método encontra também o problema de não levar em conta a carga real de cada *node*, sendo possível que todas as requisições sejam passadas sempre para o mesmo node o que acabaria por saturá-lo, mesmo que os demais estivessem servindo requisições triviais.

Uma solução melhor estará em utilizar um balanceador de carga para distribuir as conexões entre os servidores. Com esta solução pode se aumentar a noção de "unidade"do *cluster*, pois para o usuário existirá um único endereço IP para o qual serão dirigidas todas as requisições.

A granularidade da distribuição pode se dar por conexão (cada requisição de cliente se dirige ao node que melhor tenha condição de atender) ou por sessões (armazena-se em uma tabela qual *node* recebe a conexão do cliente e as envia ao mesmo *node* enquanto permanecer ativa a sessão). Quando algum *node* falha, é mais fácil mascarar o erro, tendo nesse caso o balanceador de carga mecanismos necessários para detectar a falha do node e eliminá-lo da sua lista, de forma a não encaminhar nenhuma requisição. Por esse mesmo motivo, a administração do *cluster* se torna simplificada, pode-se retirar a qualquer momento qualquer dos *nodes* para realizar tarefas de manutenção sem que sejam provocadas interrupções de serviços.

O balanceamento de carga pode ser feito em dois níveis: a nível de conexão e IP e a nível de protocolo. Para o segundo caso o balanceador seria uma espécie de *proxy*, o qual seria programando para receber conexões em uma determinada porta, podendo inspecionar o pacote para ver se trata do protocolo correto ou extrair algum tipo de dado do protocolo, podendo ainda filtrar requisições incorretas e encaminhar as conexões para um dos *nodes* do *cluster*. O problema com essa aproximação com *proxy* é a necessidade de se analisar o protocolo em todas a conexões entrantes, o programa balanceador é muito complexo e poderia causar um gargalo na entrada do *cluster* (estima-se que o número que o número de nodes para servir sem problemas de congestionamento seria em torno de 4 a 6). Neste tipo de balanceadores conta-se com Reverse-Proxy e pWEB (*Parallel Web Server*) (HANSEN, 2005).

O balanceamento a nível de conexão IP, é muito mais eficiente já que o processo a ser realizado é muito mais simples. Quando chega uma requisição ao balanceador ele somente aceita a conexão e encaminha para um dos nodes. Nesse nível, o número de nodes atrás do balanceador pode oscilar entre 25 e 100 observando-se é claro a potência dos equipamentos. O balanceador Linux Virtual Server - LVS funciona neste nível.

4.2.3 Modos de Balancear a Carga com LVS

Balanceamento por NAT (VS-NAT)

Este tipo de balanceamento aproveita a possibilidade do *kernel* do Linux funcionar como um roteador com NAT (*Network Address Translation*). A única rota padrão do *cluster* será o balanceador e, neste, quando um pacote externo chega, modificará sua rota para que chegue a um dos *nodes* do *cluster* e, a de origem para que seja devolvido a ele e reenviado ao cliente que iniciou a conexão.

De acordo com a documentação do LVS o seu funcionamento ocorre da seguinte maneira - Figura 4.3:

1. O Cliente faz uma requisição de serviço ao balanceador de carga executando o LVS;

2. O Balanceador decide a que *node* vai enviar a requisição, reescreve o cabeçalho do datagrama TCP/IP e envia ao *node* correspondente;

3. O *Node* recebe a requisição de serviço, processa, gera a resposta e envia ao balanceador de carga;

4. O balanceador reescreve novamente o cabeçalho TCP/IP do datagrama e envia de volta ao cliente como se ele próprio tivesse gerado a resposta.

No esquema físico mostrado na figura 4.4, apenas um endereço publico de IP é necessário, no caso 201.10.75.50 que está associado ao servidor LVS. O resto dos endereços são privados. A grande vantagem dessa técnica reside no fato de que os *nodes* que compõem o *cluster* poderiam executar qualquer sistema operacional que suporte TCP/IP, sendo que toda manipulação de endereços e gestão do *cluster* é feito pelo balanceador de carga. Entretanto este método encontra também uma grande desvantagem, a mesma encontrada nos balanceadores a nível de protocolo anteriormente descrita.

Balanceamento por Encapsulamento IP (VS-Tun)

Este método permite escalar um número maior de *nodes*, 100 ou mais, porém todos deverão ser capazes de trabalhar com encapsulamento IP (IP Tunneling). O encapsulamento IP consiste em se fazer trafegar um datagrama TCP/IP, com seus endereços de origem e destino, dentro de um outro datagrama com origem e destino distintos e, quando esse datagrama chegar ao seu destino seja desencapsulado o datagrama original para ser reroteado a partir dali. É uma forma de se utilizar um roteamento alternativo, por exemplo uma rede A para uma rede B passando por

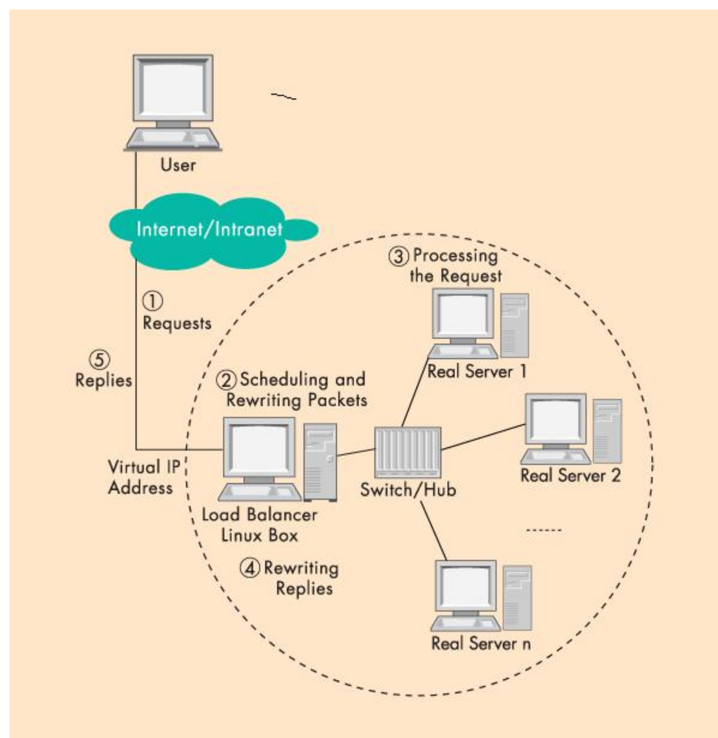

Figura 4.3: LVS: VS-NAT, fonte (MAGAZINE, 2005)

uma rede C. O problema deste método é que nem todos os sistemas operacionais suportam.

Com a utilização deste método de balanceamento todos os *nodes* do *cluster* precisam ter configurada em alguma interface um endereço IP publico se os *nodes* estão distribuídos em uma WAN. O ponto de entrada do cluster é o balanceador de carga, porém uma vez que o tráfego chega a um dos *nodes*, este roteará as respostas diretamente ao cliente sem passar pelo balanceador - Figura 4.5.

Nesse método todos os *nodes* do *cluster* necessitam possuir IPs públicos, podendo se tornar um ponto negativo. Entretanto têm-se uma grande vantagem, os *nodes* podem estas dispersos em uma área muito ampla. Ao poder separa geograficamente os servidores agregamos mais um ponto a alta disponibilidade, visto, que é muito mais difícil a ocorrência de problemas em todas as localizações dos *nodes*.

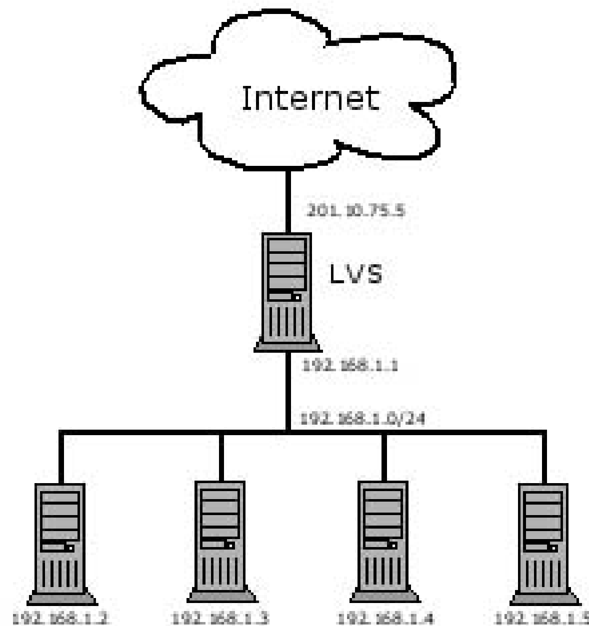

Figura 4.4: LVS: VS-NAT - Físico

Balanceamento por Roteamento Direto (VS-DR)

Este método requer que todos os *nodes* do *cluster* tenham um IP público compartilhados virtualmente com o balanceador de carga e, que se encontrem na mesma rede física (mesmo segmento) do balanceador de carga. A carga imposta ao balanceador de carga será menor, visto que não será necessário distribuir os pacotes (como no caso do NAT) e nem encapsular (no caso do Encapsulamento IP). O balanceador nesse caso não será um gargalo para o tráfego. O trafego unicamente passará pelo balanceador e será roteado diretamente ao node e, este roteará a resposta diretamente ao cliente. O balanceador de carga, como sempre será o ponto de entrada para o *cluster*. Entretanto as interfaces dos *nodes* deverão estar configuradas para não responder a comandos ARP para não interferir com outros protocolos (todos os equipamentos respondem pelo mesmo endereço IP com MACs distintos) - Figura 4.6. Quando uma requisição chega ao balanceador, este decide para qual *node* será enviada e envia o pacote a nível de enlace para o endereço MAC do *node* eleito. Quando chega ao *node* com a MAC destino e esta é analisada em nível de

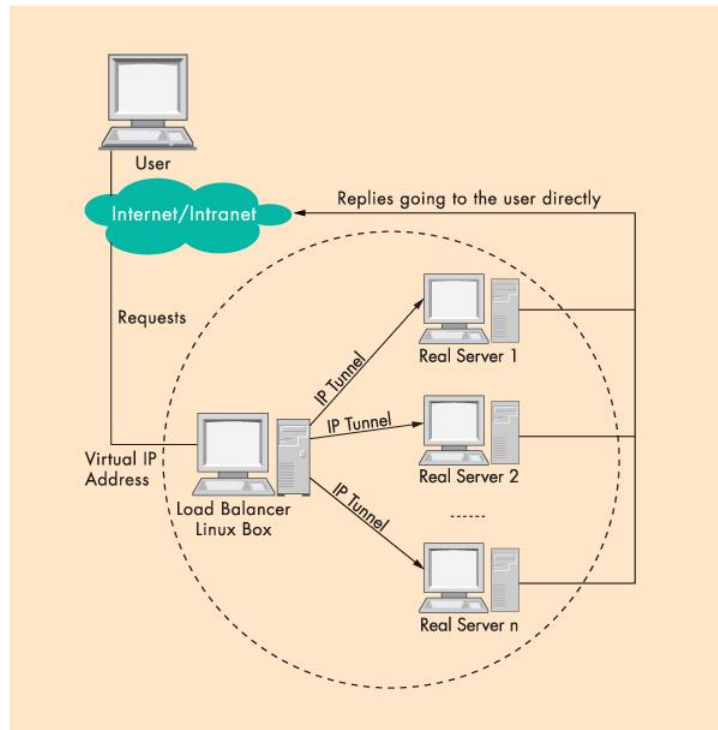

Figura 4.5: LVS: VS-Tun, fonte (MAGAZINE, 2005)

rede (TCP/IP), como o *node* também tem o endereço IP publico do *cluster* aceita o pacote, processa e envia a resposta diretamente ao cliente.

Os problemas desse método são:

1. Nem todos os sistemas operacionais permitem configurar um IP ou um dispositivo de modo a não responder a comandos ARP;

2. Todos os *nodes* devem estar no mesmo segmento físico para poder encaminhar os datagramas a nível de enlace para os endereços MAC, perdendo assim a possibilidade de dispersar geograficamente o *cluster* como no método anterior.

4.2.4 Planejamento do Balanceamento de Carga

Quando da configuração do balanceador de carga pode-se escolher entre uma série de algoritmos para se distribuir a carga entre os *nodes* do *cluster* e como será eleito

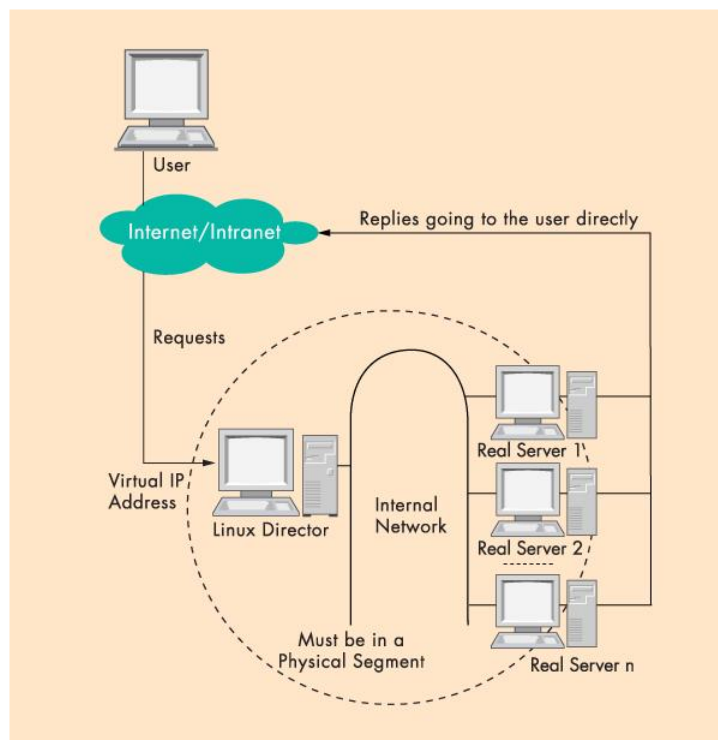

Figura 4.6: LVS: VS-DR, fonte (MAGAZINE, 2005)

o node ao qual se enviam as requisições. O Linux Virtual Server permite utilizar os seguintes algoritmos:

Round Robin

Cada requisição será enviada a um *node* do *cluster* e, a requisição seguinte será enviado ao próximo *node* da lista, até chegar ao último, o qual volta a enviar ao primeiro. É a solução mais simples e que menos recurso consome, apesar de não ser a mais justa (como no balanceamento por DNS, a carga pesada pode recair sempre sobre o mesmo *node*). A diferença com a solução baseada em DNS reside na distribuição de carga pelo balanceador, conseguindo assim uma granularidade por pacote, não existindo nesse caso a persistência de um cache como no método com DNS.

Outro problema encontrado neste algoritmo, é que todos os *nodes* receberam o mesmo número de requisições independente da sua capacidade de processamento.

Round Robin Ponderado

Este algoritmo é igual ao anterior, porém atribui a cada servidor um peso. Esse peso é um inteiro que indica a capacidade de processamento do *node*, de maneira que o algoritmo Round Robin se modificará para que os *nodes* com maior capacidade de processamento recebam mais requisições (ao menos mais que os demais de menor capacidade de processamento).

Servidor com menos Conexões Ativas

Este algoritmo de distribuição consulta os *nodes* a cada momento para ver quantas conexões abertas cada um tem com os clientes, e envia a requisição ao servidor que possui menos conexões no momento. É uma forma de distribuir as requisições que chegam aos *nodes* com menos carga. Este método falha na prática quando a capacidade de processamento dos *nodes* forem diferentes. Um *node* pode estar com poucas conexões, porém sua capacidade de processamento atrasa no atendimento de uma requisição levando a espera a *TIME_WAIT*.

Servidor com menos Conexões Ativas (Ponderado)

É igual a estratégia de Round Robin Ponderado, atribuindo aos *nodes* pesos que de alguma forma medem suas capacidades de processamentos, para modificar a preferência na hora de escolher um ou outro.

Menos Conectado Baseado em Serviço

Este algoritmo dirige todas as requisições a um mesmo servidor até que haja sobrecarga (se o número de conexões ativas for maior que o seu peso) então, passa para uma estratégia de menos conexões ativas ponderada para o resto dos *nodes* do *cluster*. Este método de planejamento pode ser útil quando se oferece vários serviços distintos e quer dedicar-se a cada *node* a um serviço, porém, todos os *nodes* são capazes de desempenhar os demais serviços.

Tabelas Hash por Origem e Destino

Este algoritmo dispõe de uma tabela de entrada fixa onde deverão estar assinalados os endereços IPs de origem e destino e indicar qual servidor deverá atender a requisição. O Balanceador compara os endereços dos datagramas TCP/IP que recebe com essa tabela e age de acordo.

Conexões Persistentes

Pode ser adicionado a todos os algoritmos citados permitindo que uma vez o que o cliente conectou em um *node*, seja sempre enviado ao mesmo. Isso pode ser útil, para por exemplo, se a aplicação web faz uso de sessões HTTP e precisa-se manter a conexão, ou se queira permitir conexões persistentes.

4.3 Alta Disponibilidade com LVS

4.3.1 Heartbeat + CODA

Nesta solução a alta disponibilidade é obtida utilizando-se redundância de *hardware* e *software* e, monitorando recursos para detectar qualquer falha e reorganizar o *cluster* para que outras máquinas possam assumir a carga de trabalho daquela que tenha vindo a falhar. Para se conseguir este objetivo serão necessário os seguintes programas:

- Heartbeat, para monitorar se determinado equipamento está funcionando ou falhou. Caso algum servidor falhe, outro pode tomar seu lugar, assumindo seu endereço IP e os serviços monitorados;

- CODA, para disponibilizar um sistema de arquivos distribuído com redundância de servidores e *caches* locais nos servidores.

Na figura 4.7, está representado a montagem do *cluster*, bem como o lugar que ocupa cada um dos servidores e *softwares*.

O cluster se divide em três partes:

1. Como primeira linha, tem-se o balanceador de carga. E, para que esta máquina não se converta em um ponto único de falha(SPOF), se duplica a mesma com outra igual que a princípio estará inativa. Ambas monitoram seu funcionamento através do *heartbeat*. Se o balanceador primário falhar, o secundário se utilizará do *heartbeat* para tomar o seu lugar e suplantar sua identidade e obter o endereço IP de entrada do *cluster* e tomar o controle da distribuição de carga.

2. Em uma segunda linha, temos os servidores reais do *cluster*. A alta disponibilidade se consegue aqui, instalando vários servidores e monitorá-los no distribuidor de carga, tanto a nível de *hardware* quanto de serviços para retira-los do *cluster* caso ocorra alguma falha total ou de algum dos serviços monitorados.

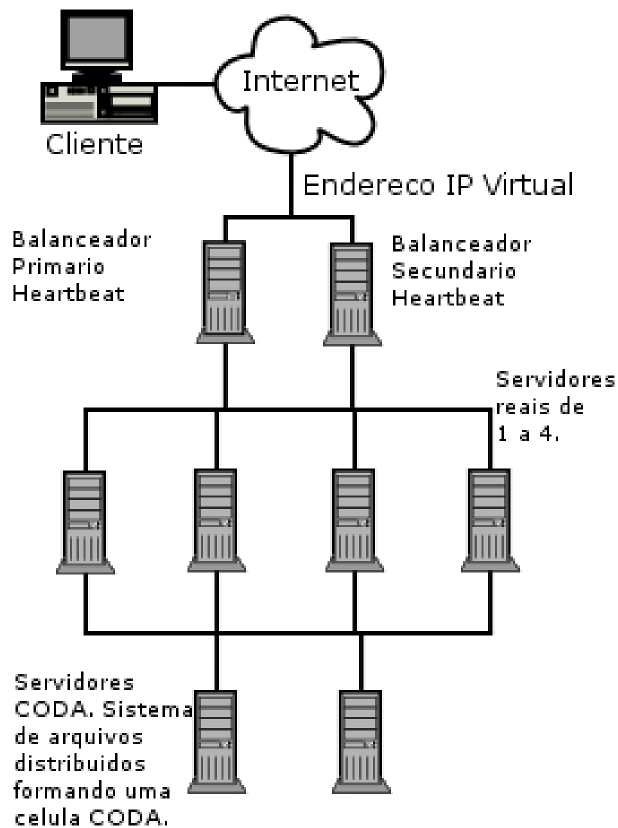

Figura 4.7: Alta Disponibilidade utilizando CODA

3. Na terceira linha, temos o sistema de arquivos distribuídos, nesse caso, CODA. Entretanto, aqui pode-se utilizar NFS. Todavia, CODA traz uma vantagem sobre o NFS que é a possibilidade do servidor real continuar a trabalhar utilizando-se do cache local em caso de falha ou perda da conexão com o servidor de arquivos. Poderia-se montar vários servidores em uma célula CODA para assegurar sua disponibilidade em face a qualquer falha.

Além do descrito acima, é necessário levar em conta a necessidade de uma estrutura de rede redundante, com vários roteadores de saída e várias redes internas para conectar os equipamentos e, múltiplas placas de rede para cada servidor para que se possa sair por uma ou outra caso alguma falhe.

4.4 Instalação

4.4.1 Linux Virtual Server

O Linux Virtual Server é um servidor altamente escalável e com alta disponibilidade construído sobre um *cluster* de servidores reais, com um balanceador carga rodando sobre o sistema operacional Linux. A arquitetura do *cluster* é totalmente transparente para os clientes. Os clientes vêem como se fosse apenas um único servidor. Das técnicas de balanceamento de IP (métodos packet forwarding) existentes no LVS, a escolhida foi roteamento direto.

Antes de iniciar a instalação e qualquer configuração do LVS, é importante entender que os recursos de kernel disponíveis para o balanceamento de carga não são nativos nas versões das distribuições mais antigas. Nesta implementação, especificamente, não será necessário recompilar o kernel, visto que a distribuição Slackware versão 10.1 será utilizada.

Os servidores precisam estar configurados de modo a ver o tráfego para o endereço virtual como local, mesmo não estando atuando como balanceador de carga ativo. Isto pode ser feito utilizando um alias IP para o dispositivo de loopback. O arquivo responsável pela configuração do LVS é o ldirectord.cf mostrado na figura 4.8, arquivo em que são definidas todas as diretivas para os servidores virtuais.

4.4.2 Heartbeat

O Heartbeat desenvolvido e mantido pelo Linux-HA é utilizado para construir *clusters* de altíssima disponibilidade. Com ele é possível fazer com que dois *nodes* do *cluster* obtenham a capacidade de assumirem os recursos e serviços de um número ilimitado de interfaces IP. Ele trabalha enviando um '*heartbeat*' entre dois *nodes* através de uma conexão serial, *ethernet*, ou ambas. Se o *heartbeat* falhar, a maquina secundária irá detectar que a primária falhou, e assumir aos serviços que estavam rodando na máquina primária. Na implementação proposta utilizou-se duas interfaces *ethernet* para troca dos pacotes de *heartbeat*. Uma interface disponível aos clientes e uma interface dedicada de conexão entre os dois *nodes* do *cluster*, neste caso, entre os balanceadores de carga. O *heartbeat* será o responsável pela alta disponibilidade do serviço de balanceador de carga (*ldirectord*). Ignorando a documentação da Linux-HA que sugere a utilização de uma interface serial para *backup* da interface *ethernet* do *heartbeat*, esta, não será utilizada aqui. Em uma implementação de produção é necessária, pois qualquer problema na pilha IP fará com que ambas as conexões ethernet sejam interrompidas. A versão utilizada é a 1.x.x sendo que algumas distribuições já o possuem de forma nativa ao sistema.

```
virtual=192.168.1.30:443
    real=100.100.100.1:443 gate 1
    real=100.100.100.2:443 gate 1
    real=100.100.100.3:443 gate 1
    real=100.100.100.4:443 gate 1
    fallback=127.0.0.1:443
    service=https
    request="index.html"
    receive="happy"
    scheduler=wrr
    persistent=120
    netmask=255.255.255.0
    protocol=tcp
virtual=192.168.1.30:80
    real=100.100.100.1:80 gate 1
    real=100.100.100.2:80 gate 1
    real=100.100.100.3:80 gate 1
    real=100.100.100.4:80 gate 1
    fallback=127.0.0.1:80
    service=http
    request="index.html"
    receive="happy"
    scheduler=wrr
    persistent=120
    netmask=255.255.255.0
    protocol=tcp
```

Figura 4.8: Arquivo de configuração do ldirectord

Os arquivos de configuração do Heartbeat, estão localizados em /usr/local/etc/ha.d/. São eles: ha.cf, haresources, authkeys. Em outras instalações eles podem estar localizados em /etc/ha.d.

ha.cf

O ha.cf é o responsável pelas configurações de funcionamento do heartbeat. A figura 4.9 mostra o arquivo onde pode-se observar as opções utilizadas.

haresources

O arquivo haresource define basicamente a lista de recursos que serão migrados de um *node* para outro. Ele deve ser idêntico em ambos os *nodes* monitorados.

```
debugfile /var/log/ha-debug
logfile /var/log/ha-log
logfacility     local0
keepalive 2
deadtime 20
warntime 10
initdead 30
baud    19200
bcast eth1
auto_failback on
node    servidor
node    cliente
ping 192.168.1.30
respawn hacluster /usr/local/lib/heartbeat/ipfail
```

Figura 4.9: Arquivo de configuração ha.cf

```
#haresource file
servidor IPaddr::192.168.1.30 ldirectord
```

Figura 4.10: Arquivo de configuração haresource

O *node* preferencial é utilizado para identificar para qual node o recurso será migrado em uma configuração com *nice-failback* desabilitado, ou nos casos em que ambos os *nodes* serem inicializados simultaneamente.

authkeys

O terceiro arquivo de configuração é o authkeys, mostrado na figura 4.11. Ele determina a autenticação entre os *nodes*. Os três métodos de autenticação disponíveis são crc, md5 e sha1. Se o *heartbeat* irá rodar sobre uma rede segura, uma conexão dedicada, o crc é o método mais barato do ponto de vista de recursos. Se a rede é insegura deve-se optar por utilizar o md5 ou sha1 atentando-se ao fato de que o consumo de CPU necessário para esses métodos é mais intenso.

```
3 md5 Hello!
auth 1
1 crc
```

Figura 4.11: Arquivo de configuração authkeys

É necessário configurar as permissões deste arquivo de forma segura, como 600 (direitos de leitura e gravação somente para o usuário privilegiado do sistema e dono do arquivo).

4.4.3 CODA

CODA é um sistema de arquivos distribuídos extremamente avançado, com suas origens no AFS2, que vem sendo desenvolvido desde 1987 na Universidade Carnaige Mellon pela equipe do professor M. Satyanarayanan.

Entretanto coda não é tão conhecido como seus alternativos (GFS - *Global File System*, NFS - *Network File System*, etc). De certa maneira poderia-se dizer que praticamente não saiu do ambiente acadêmico, o que não quer dizer que seja ruim, muito pelo contrário, conta com muitas características desejáveis para um sistema distribuído (especialmente para um *cluster*), figura 4.12.

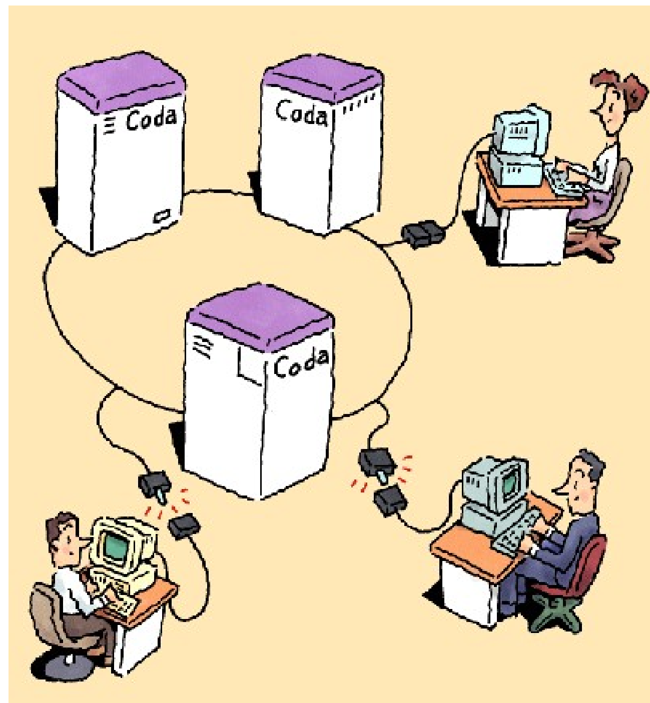

Figura 4.12: Ilustração sobre o funcionamento do CODA, fonte(BRAAM, 2005)

- O cliente é capaz de funcionar sem problemas desconectado do servidor. Se o servidor for desconectado ou falhar a conexão, quando a conexão for

reestabelecida os sistemas do cliente e servidor serão sincronizados de forma automática.

- Alto rendimento proporcionado por um *cache* local na máquina cliente: Na primeira vez que acessa um arquivo será armazenado no disco local, e, os seguintes acesso ocorrem localmente. Entretanto, no acesso ocorre a verificação se a copia é válida (que não tenha sido alterada no servidor). Dessa maneira, se aumenta o rendimento de duas formas: por um lado o acesso a disco local é muito mais rápido que no servidor através da rede, por outro lado diminui o congestionamento da rede evitando-se colisões.

- Replicação automática de servidores. Coda proporciona os mecanismos necessários para realizar réplicas automáticas entre servidores, para que os clientes possam acessar um ou outro de forma automática e transparente para o usuário caso algum servidor venha a falhar.

- Modelo de segurança próprio e independente do sistema operacional para identificação dos usuários, permissões (ACLs) e criptografia de dados.

- Excelente escalabilidade.

- Licença aberta (GPL).

Todas essas características fazem de CODA um sistema distribuído idôneo para um *cluster* de alta disponibilidade.

Terminologia CODA

CODA é bastante diferente em vários aspectos de todo resto dos sistemas de arquivos disponíveis no mundo UNIX. Esta diferenças acarretam uma terminologia distinta da que habitualmente encontra-se no dia-a-dia. A figura 4.13 traz a idéia desta terminologia. Na seqüência a descrição do funcionamento:

- **Espaço de nomes unificado**: as partições compartilhadas com CODA se encontram situadas no mesmo diretório em todos os equipamentos que as compartilham; no diretório /coda na raiz do sistema de arquivos principal. Esta é uma diferença importante do restante dos sistemas de arquivos UNIX/Linux que permitem montar os diretórios compartilhados em qualquer lugar da árvore de diretórios. O que se busca com esta limitação é unificar o aspecto dos diretórios compartilhados em todas as máquinas, para assegurar que um determinado diretório se encontra em todos no mesmo lugar.

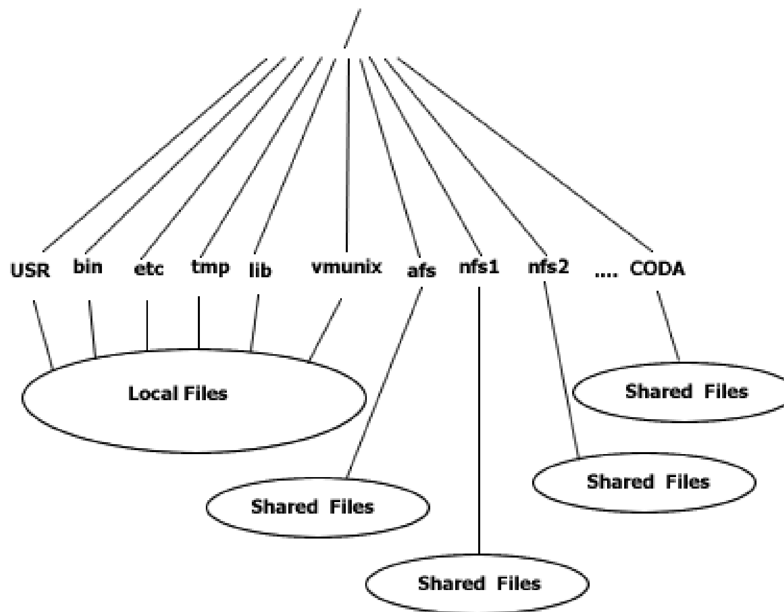

Figura 4.13: Terminologia CODA

- **Célula CODA**: uma célula CODA é um conjunto de servidores com a mesma configuração. Um dos servidores é o **SCM** (*System Control Machine*), é o servidor mestre onde se guarda a copia principal, ocorrem as modificações e se propagam aos demais servidores. Um Cliente pode pertencer a uma única célula CODA, e podem se conectar a qualquer servidor que a compõe.

- **Volumes CODA**: Os servidores agrupam os diretórios compartilhados em volumes, que tipicamente possuem mais de um diretório. Cada volume contem uma raiz e uma série de subdiretórios. Todos os volumes se montam abaixo do diretório /coda em todos os clientes e podem agrupar (montar um volume dentro do outro). O grupo de servidores que servem o mesmo volume é denominado de **VSG** (*Volume Storage Group*).

- **Pontos de Montagem de Volume**: O volume raiz é um volume especial, deve ser montado diretamente em /coda. O resto dos volumes de uma célula

CODA são montados em subdiretórios de /coda utilizando o comando `cfs mkmount`.

- **Armazenamento de Dados**: um servidor CODA precisa manter mais dados sobre os arquivos que um servidor NFS ou SAMBA. Por isso que os arquivos compartilhados tal qual sobre o disco, guardam-se indexados por número abaixo do diretório /vicepa e se armazena toda uma série de meta-dados e informação transacional de forma similar ao RaiserFS. Isso é feito na partição de **RVM** (*Recoverable Virtual Memory*).
- **RVM**: é uma biblioteca para realizar ações de forma atômica, armazenando informações transacionais em disco para recuperar mais tarde em face de qualquer erro. CODA utiliza esta biblioteca para conseguir atomicidade em suas operações e usa uma partição para armazenar os dados transacionais.
- **Cache do Cliente**: Os cliente mantém um cache local (não volátil) dos arquivos que foram acessados recentemente de forma similar ao que acontece nos servidores: geralmente os meta-dados de **RVM** em /usr/coda/DATA e os arquivos normais indexados por número no diretório /usr/coda/venus.cache. Este cache local permite o acesso e uso dos arquivos remotos de uma forma muito mais rápida que NFS ou SAMBA.
- **Validação**: toda vez que altera-se ou cria-se um arquivo no cliente, ou este reconecta ao servidor depois de um problema com a rede, CODA valida os dados do cache local pra ver se estão de acordo com as versões que existem no servidor, e os atualiza se for necessário.
- **Autenticação**: CODA gestiona a autenticação dos usuários através de um "token" que deve ser entregue quando da autenticação, similarmente ao protocolo Kerberos. Este token se associa a uma identidade de usuário no servidor e, a autenticação quando concedida tem uma duração em torno de 24 horas.
- **Proteção**: quando se tenta acessar um arquivo compartilhado com um determinado token, o servidor confirma se a identidade associada ao token tem permissão consultando uma série de tabelas (ACL - *Access Control List*).

OS Servidores

Uma célula de servidores CODA é uma entidade muito complexa onde podem coexistir centenas de servidores oferecendo uma variedade de serviços a todo um conjunto de clientes.

O protocolo CODA se sustenta sobre três serviços:

- **Servidor de Arquivos**: o servidor codasrv interage com o processo venus nos clientes. Esses dois processos são os que realizam toda troca de dados entre as máquinas.
- **Servidor de Autenticação**: o servidor auth2 é executado em todos os servidores validando os usuários e controlando seus *tokens* de identidade. Sem bloqueios as contra-senhas só podem ser trocadas no **SCM**, porque a copia da base de dados é somente de leitura em todas as máquinas, exceto no **SCM**. As atualizações de contra-senha se realizam de forma automática pelos *daemons* updateclnt/updatesrv.
- **Servidores de Atualização**: o processo updateclnt trabalha junto com o updatesrv (que se executa no SCM) para manter atualizada a base de dados de contra-senhas em todos os servidores com a copia original no **SCM**. Para isso, o *deamon* updateclnt confirma a cada certo tempo se os arquivos do **SCM** foram atualizados. É por isso que as atualizações não são imediatas, dependem do tempo de comprovação do updateclnt.

Um mesmo servidor pode oferecer os três serviços simultaneamente ou não. De qualquer maneira, o *deamon* updateclnt deve ser executado sempre para manter atualizadas suas copias dos arquivos do sistema e da base de dados das contra-senhas.

Como exemplo, poder-se-ia ter a organização da figura 4.14, tendo-se:

- Três servidores com os três serviços. O servidor mais acima a esquerda é o **SCM**, está executando o deamon updatesrv.
- Um servidor unicamente de autenticação (acima a direita) que não possui serviços de compartilhamento de arquivos.
- Um servidor unicamente para compartilhamento de arquivos, sem serviço de autenticação.

Um servidor CODA deve pelo menos ter três partições no disco:

- Uma partição para o sistema operacional e os dados do CODA (esta poderia ser divida em duas ou mais partições ou distribuídas em vários discos para otimizar os tempos de acesso aos dados).
- Uma partição para os dados de **RVM**, que deve ser em torno de 4% do tamanho da partição que contem os arquivos compartilhados pelo CODA.

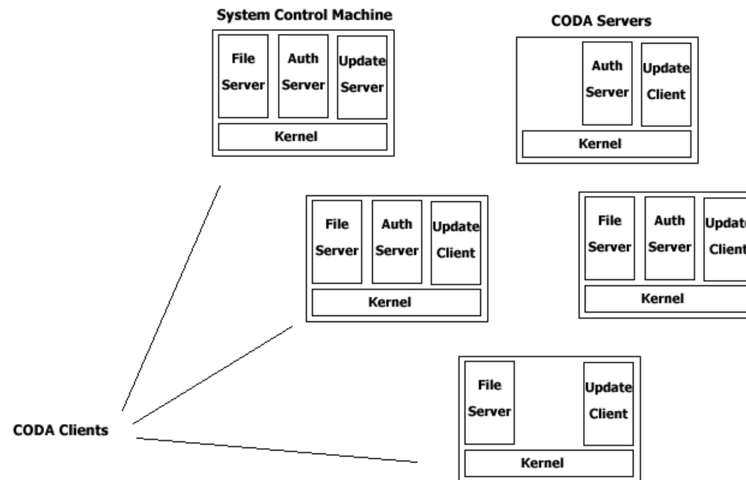

Figura 4.14: Organização de uma célula CODA

- Uma partição para os logs das transações que ocorrem na partição **RVM**, com um tamanho aproximado de 20MB, atende a demanda de espaço.

Um dos inconvenientes de CODA é o conteúdo da partição RVM que se mantém sempre em memória. Em outras palavras, para um servidor que possui 2GB de dados, teremos 80MB de memória sempre ocupados pelos dados de RVM. O problema é que está quantidade aumenta conforme o tamanho da partição que será compartilhada. Para uma partição de 40GB, precisa-se 1.6GB de memória somente para os dados de RVM. Este é um ponto importante a ser considerado ao se implantar o sistema de arquivos CODA.

Em uma instalação típica, os dados ficam armazenados no servidor nos seguintes diretórios:

- `/vicepa[a-z]`: Dados compartilhados pelo sistema CODA.
- `/vice/auth2`: Neste diretório se encontra as informações pertinentes ao serviço d autenticação, entre eles o arquivo de log do *deamon*.
- `/usr/local/sbin`: Contem os executáveis para os serviços do SCM.
- `/vice/db`: Armazena os logs dos processos de atualização, bem como as copias das bases de dados dos servidores.

- `/vice/srv`: Contem a informação do servidor de arquivo e seus arquivos de logs.

- `/vice/vol`: Armazena as informações sobre os volumes do sistema de arquivos CODA.

- `/vice/vol/remote`: Este diretório existe apenas no servidor SCM e, contem informação sobre todos os volumes existentes nessa célula CODA.

- `/vice/misc`: Nesse diretório são executados os serviços updateclnt e updatesrv.

OS Clientes

O código do cliente CODA se divide em duas partes: um controlador no núcleo do sistema operacional (já disponível no Linux a partir do *kernel* 2.2) e uma série de utilitários no espaço do usuário.

Os clientes CODA executam o *deamon* venus, que está encarregado dialogar com o servidor, realizar a troca de dados e, com o controlador do *kernel* na máquina local para passar os dados e gerar o conteúdo do diretório /coda.

Os arquivos nos clientes se organizam da seguinte forma:

- `/u/usr/local/etc/coda`: Arquivos de configuração do cliente.

- `/usr/coda/venus.cache`: Cache local dos arquivos compartilhados.

- `/usr/coda/spool`:Dados referente a sincronização de arquivos entre o cliente e servidor, durante o uso normal ou desconectado.

- `/usr/coda/tmp`: Dados temporários.

- `/coda`: Ponto de acesso a todos os volumes remotos.

No cliente também se instala as ferramentas necessárias para manipular a autenticação, controle de ACLs e gerenciamento do estado geral dos serviços, etc.

Características Avançadas

Além do já exposto, CODA tem uma série de características avançadas, muito interessantes para um cluster HA:

- **Controle de Caches**: Nos clientes CODA é possível marcar certos arquivos para que sejam obtidos do servidor e se mantenham sempre na cache. Dessa forma é possível assegura-se que uma cópia de certos arquivos importantes vão estar sempre disponíveis. Também é possível deixar alguns arquivos mais tempo em cache.

- **Replicação de servidores**: CODA dispõe de software necessário para manter replicas idênticas dos servidores de uma mesma célula. Está replica é efetuada também pelo *daemons* updateclnt e updatesrv, da mesma maneira que a replicação da base de dados de contra-senha. Desta maneira é possível garantir segurança mediante a redundância de servidores, tendo várias cópias de dados em máquinas distintas, sabendo-se que os clientes acessam indistintamente um ou outro servidor da mesma célula. Contudo, CODA controla quando um servidor esta com problemas e dirige a conexão a outro servidor da mesma célula.

- **Imagens Virtuais**: CODA também oferece a possibilidade de criar durante alguns momentos um volume virtual, no qual se mantém uma imagem de um volume em um dado momento. Podendo então prosseguir trabalhando com o volume original sem alterar o conteúdo do volume virtual. Assim, é possível efetuar cópias de segurança do volume com ferramentas externas ao CODA, mesmo que o sistema esteja trabalhando.

- **Cópias de segurança**: CODA tem um conjunto completo de ferramentas para o sistema de cópias de segurança. Podendo ser efetuadas cópias diárias tanto incrementais quanto completas.

4.4.4 Comentários Gerais

Este capítulo abordou três importantes componentes para implementação de um *cluster* HA, selecionados para este projeto devido sua escalabilidade e por propiciar altíssima disponibilidade. O Heartbeat executa a monitoração dos balanceadores de carga. Por sua vez, o LVS tem a tarefa de fazer o balanceamento de carga e prover escalabilidade para o *cluster*. A replicação dos dados entre os servidores reais ficou a cargo do sistema de arquivos CODA que operando em conjunto com o LVS aumenta a escalabilidade do sistema.

Capítulo 5

Testes de Desempenho

5.1 Sistema de Arquivos CODA

Com os clientes e os servidores já instalados e conectados, foram iniciados os testes para avaliação de como CODA se comportaria em relação a criação e exclusão de arquivos. Como a manipulação de arquivos ocorre na rede uma queda no desempenho em relação as operações em discos normais é esperada. Conforme comentado no capítulo 4 o volume do sistema de arquivos CODA é montado na máquina local em /coda. É nesse diretório onde se realizam todas as operações com arquivos do cliente CODA. Foi criado um *alias* para o diretório /var que agora passa ser /coda/servidor.nwise. Os testes serão realizados em /var/diversos.

O primeiro teste a ser realizado foi a copia de 4282 arquivos ocupando algo em torno de 134MB conforme apresentado na figura 5.1.

```
root@real1:~# ls
coda-6.0.11/           loadlin16c.txt  rpc2-1.27/         sendmail-8.12.8/
coda-6.0.11.tar.gz     loadlin16c.zip  rpc2-1.27.tar.gz  sendmail.8.12.8.tar.gz
linux-coda-6.2/        lwp-2.0/        rvm-1.11/          sendmail.mc
linux-coda-6.2.tar.gz  lwp-2.0.tar.gz  rvm-1.11.tar.gz
root@cliente:~# ls -R |wc -l
4282
root@real1:~#
```

Figura 5.1: Lista de arquivos a serem copiados para o volume CODA

Normalmente, essa copia que não levaria mais que alguns segundos em um disco normal consumiu 5':38" conforme a figura 5.2. Ao excluir os arquivos criados obteve-se um tempo de 20,64" conforme a figura 5.3.

Esta lentidão pode ser devida as atualizações constantes do servidor. Supõe-se que CODA vai avisando o servidor a cada arquivo que esta sendo copiado, um a um dos 4282 arquivos ao invés de atualizar mais tarde, haja visto que o cliente

```
root@real1:~# time cp -r * /var/diversos/

real    5m38.074s
user    0m0.70s
sys     0m1.880s
root@real1:~#
```

Figura 5.2: Tempo obtido na copia dos arquivos

```
root@real1:~# time rm -R -f /var/diversos/*

real    0m20.648s
user    0m0.020s
sys     0m0.120s
root@real1:~#
```

Figura 5.3: Tempo obtido na operação de exclusão dos arquivos copiados

dispõe de um cache local. Isso indica um ponto fraco de CODA - é muito lento na criação de arquivos pequenos. Por outro lado, a rapidez com que os arquivos foram excluídos leva a crer que a exclusão ocorre no cache local e o servidor é apenas informado que os arquivos foram apagados.

O teste seguinte foi concatenar os arquivos anteriormente copiados individualmente e expandir o conteúdo do arquivo no diretório /var/diversos conforme mostra a figura 5.4.

```
root@real1:~# tar -cf todos.tar *
root@real1:~# ls -alh todos.tar
-rw-r--r--  1 root root 128M 2005-09-22 09:12 todos.tar
root@real1:~# time cp todos.tar /var/diversos/

real    0m23.190s
user    0m0.030s
sys     0m0.840s
root@real1:~#
```

Figura 5.4: Copia dos arquivos compactados

Quando se trata da criação de arquivos grandes CODA é mais rápido, 23" não é um tempo excessivo para copiar 128MB pela rede, principalmente ao se levar em conta as sobrecargas do protocolo e as sincronizações do cliente e servidor.

Entretanto ao expandir o conteúdo concatenado CODA mostrou-se mais lento que a copia individual dos arquivos, de 5':38" o tempo subiu para 7':24", figura 5.5. Para maior certeza, o teste foi executado várias vezes e os resultados obtidos sempre ficaram em torno dos tempos citados acima.

```
root@real1:/var/diversos# time tar -xf todos.tar

real    7m24.776s
user    0m0.220s
sys     0m1.550s
root@real1:/var/diversos#
```

Figura 5.5: Tempo resultante da descompactação do arquivo

Como finalidade deste *cluster* é o seu emprego em um ambiente internet, o teste final foi sua utilização na prática. Para isso os servidores apache e sendmail foram escolhidos, tendo em vista que nesse tipo de aplicação são os serviços mais utilizados. Nesse caso específico foram comparados o desempenho dos servidores apache e sendmail em relação ao seu funcionamento no sistema de arquivos **EXT3** e utilizando o diretório /coda/servidor.nwise como um *alias* para /var através do sistema de arquivos CODA.

Surpreendentemente, CODA não mostrou degradação no desempenho. O envio e recebimento de mensagens e acesso a paginas tiveram o mesmo desempenho quando do uso do sistema de arquivos **EXT3**. Foram efetuados vários testes com mensagens com e sem anexo onde não foi possível observar diferenças no tempos para envio e recebimento entre ambos os sistemas de arquivos. Pode-se afirmar que para um ambiente internet não ocorre a degradação do desempenho.

A prova final foi desconectar o cabo da interface de rede do servidor CODA. Nessa operação verificou-se que para os clientes o volume continuava intacto e operando. Foram removidos vários arquivos e criados outros com a finalidade de provar a sincronização entre cliente e servidor quando da reconexão. Após a reconexão do servidor a sincronização foi executada e o que havia sido feito no cliente propagou-se para o servidor e os demais clientes. Este experimento foi repetida inúmeras vezes e todas lograram sucesso.

Em todos os testes realizados CODA não falhou. Pode-se dizer que a pesar de ser um sistema de arquivos que praticamente não saiu do ambiente acadêmico, é um sistema robusto e que na sua versão atual a 6.0.11 está bem estável podendo sim ser empregado em um ambiente de produção. CODA é um sistema de arquivos adequado para um *cluster* **HA**. Ao utilizá-lo como sistema de arquivos a replicação de dados torna-se mais fácil e transparente.

5.2 Linux Virtual Server

A ferramenta IPVSADM faz a administração do *cluster*. Entende-se administração aqui como o ato de acompanhar o estado de cada *node* do *cluster*, de saber quantas requisições estão sendo atendidas por cada um dos integrantes, bem como checar as configurações do servidor virtual que estão sendo utilizadas. O IPVSADM presente no balanceador de cargas é que apresenta a composição do cluster, mostrando quais são os servidores reais que formam o *cluster*, assim como quantas requisições estão sendo atendidas individualmente por cada um.

O teste do LVS foi divido em três etapas:

- Fazer várias requisições de serviço ao endereço IP do balanceado neste caso 192.168.1.30.

- Retirar de operação o servidor real1, analisar o estado do ambiente e reconectá-lo novamente.

- Retirar de operação os servidores real2, real3 e real4; procedendo como no caso do servidor real1.

Para analisar o estado do ambiente em cada etapa, a ferramenta IPVSADM será utilizada. Para o cluster em condições normais, a saída apresentada pela ferramenta será como mostra a figura 5.6.

```
root@cliente:~# ipvsadm -L
IP Virtual Server version 1.0.12 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  192.168.1.30:smtp rr
  -> 100.100.100.4:smtp            Route   1      0          0
  -> 100.100.100.3:smtp            Route   1      0          0
  -> 100.100.100.2:smtp            Local   1      0          0
  -> 100.100.100.1:smtp            Route   1      0          0
TCP  192.168.1.30:http rr
  -> 100.100.100.4:http            Route   1      0          0
  -> 100.100.100.3:http            Route   1      0          0
  -> 100.100.100.2:http            Local   1      0          0
  -> 100.100.100.1:http            Route   1      0          0
TCP  192.168.1.30:pop3 rr
  -> 100.100.100.4:pop3            Route   1      0          0
  -> 100.100.100.3:pop3            Route   1      0          0
  -> 100.100.100.2:pop3            Local   1      0          0
  -> 100.100.100.1:pop3            Route   1      0          0
root@cliente:~#
```

Figura 5.6: Estado geral do ambiente LVS através da ferramenta IPVSADM

A primeira linha mostra que estamos na máquina cliente (o balanceador de cargas). Na segunda linha mostra a versão da ferramenta, no caso 1.0.12. Por outro lado, a quarta linha mostra o protocolo utilizado (**TCP**), o endereço IP virtual do

cluster (192.168.1.30), o serviço disponibilizado (http) e o algoritmo de escalonamento utilizado (rr = round robin). As outras linhas mostram os servidores reais ativos no momento, o peso de cada um no algoritmo de escalonamento (para este caso = 1), o número de conexões ativas (0) e o número de conexões inativas (0). Estes dois últimos iguais a zero mostram que nenhuma conexão está sendo feita ou foi terminada no momento.

Para verificar o atendimento dos serviços, efetuamos requisições de uma máquina cliente (exemplo : digitamos o endereço IP 192.168.1.30 no seu navegador e forçamos a atualização, totalizando 20 requisições). Utilizando a ferramenta IPVSADM, logo após essas requisições, o resultado obtido esta mostrado na figura 5.7.

```
IP Virtual Server version 1.0.12 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  192.168.1.30:80 rr
  -> 100.100.100.4:80            Route   1      0          4
  -> 100.100.100.3:80            Route   1      0          4
  -> 100.100.100.2:80            Route   1      0          4
  -> 100.100.100.1:80            Route   1      0          4
```

Figura 5.7: Estado do ambiente LVS após várias requisições de serviços

As requisições aparecem como inativas, pois ao requisitar páginas web, o cliente abre a conexão, descarrega a página inteira para o sua máquina e fecha a conexão. Como o tempo de abrir, descarregar a página de teste e fechar a conexão é muito pequeno, o IPVSADM sempre mostrará a conexão como inativa. Como exemplo, no caso do SSH, a conexão apareceria como ativa até que a sessão fosse finalizada. O numero de requisições foi distribuído por igual, pois se definiu o peso como sendo 1 para todos os servidores reais. Como o resultado apresentado foi dentro do esperado, conclui-se que o teste foi bem sucedido.

O servidor com endereço IP 100.100.100.1 foi desconectando da rede. Neste caso é esperado que o balanceador de cargas retire este servidor do cluster e redistribua a carga apenas aos computadores aos outros computadores. Para visualizar o estado atual do cluster, outra vez a ferramenta IPVSADM foi utilizada. A saída é apresentada na figura 5.8.

```
IP Virtual Server version 1.0.12 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  192.168.1.30:80 rr
  -> 100.100.100.4:80            Route   1      0          0
  -> 100.100.100.3:80            Route   1      0          0
  -> 100.100.100.2:80            Route   1      0          0
```

Figura 5.8: Estado do ambiente LVS após retirada do servidor real1

Ao reconectarmos o servidor. O balanceador deverá colocá-lo novamente no cluster. Usamos a ferramenta IPVSADM e obtemos a seguinte saída mostrada na figura 5.9.

```
IP Virtual Server version 1.0.12 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  192.168.1.30:80 rr
  -> 100.100.100.4:80            Route   1      0          0
  -> 100.100.100.3:80            Route   1      0          0
  -> 100.100.100.2:80            Route   1      0          0
  -> 100.100.100.1:80            Route   1      0          0
```

Figura 5.9: Estado do ambiente LVS após reconectar o servidor real1

Repetindo-se os testes acima para os demais servidores, logrou-se sucesso em todos os casos.

O balanceador de cargas é um SPOF[1]. Para aumentar a disponibilidade pode-se adicionar um segundo balanceador em paralelo que será controlado pelo *heartbeat*. No caso, o ldirectord não será iniciado diretamente, mas sim pelo *heartbeat*. Para isso o ldirectord utiliza o arquivo de configuração ldirectord.cf onde define-se o endereço virtual do *cluster* e o endereço dos servidores reais, bem como as portas dos serviços controlados.

O serviço oferecido pelo conjunto de LVS e Heartbeat podem ser comparados a outras soluções proprietárias que existem no mercado a um preço muito alto.

O único problema encontrado é que quando o balanceador primário cai e o secundário assume mediante o heartbeat, o conteúdo da tabela hash, assim como as conexões ativas, e todas as informações do balanceador primário se perdem. Isso produz aos clientes, que tenham uma conexão em curso, um erro causado pela perda da conexão.

5.3 Comentários Finais

Como foi visto neste capítulo, o Software Livre, principalmente GNU/Linux, oferece todas as ferramentas necessárias para instalar um *cluster* com balanceamento de carga, alta disponibilidade e escalável. Estas ferramentas vão desde as que asseguram a proteção dos dados em cada equipamento como RAID e os sistemas de arquivos distribuídos que permitem compartilhar dados eficientemente como no caso do CODA; as que ajudam a recuperar-se frente a problemas, como o Heartbeat; as que permitem o balanceamento de cargas e escalabilidade como o LVS.

Os testes realizados comprovaram a eficácia e eficiência dessas soluções, o que vêm a demonstrar que é possível obter alta confiabilidade em sistemas computa-

[1] Single Point of Failure

cionais utilizando-se apenas de soluções livres. Com os resultados desses testes, foi possível ainda verificar que há probalidade implantar de forma bem sucedida uma solução de alta disponibilidade usando as ferramentas aqui apresentadas.

Capítulo 6

Conclusão

O inicio deste projeto foi voltado para identificação de soluções que permitissem a um provedor de internet ou empresa manter-se o maior tempo possível on-line com o menor investimento possível. O trabalho centrou-se na implementação de uma solução de alta disponibilidade com a utilização de Software Livre, se encaixando portanto na idéia inicial de se obter soluções de qualidade á um baixo custo.

A primeira dificuldade surgiu no momento de se encontrar soluções em Software Livre que implementadas trouxessem os resultados esperados. Viu-se que as possibilidades para implementação da alta disponibilidade eram diversas, e que a escolha por um conjunto de soluções dependeria da real necessidade da aplicação a ser clusterizada. A solução escolhida baseia-se em três componentes básicos (CODA, LVS e Heartbeat), sendo o passo seguinte foi a integração dos mesmos para que o objetivo do trabalho fosse atingido. Um dos problemas enfrentados foi a documentação desatualizada do CODA. Na maioria das vezes a documentação fazia referência a versões antigas do sistema, completamente incompatíveis com a recente versão 6.0.11 utilizada na implementação.

Embora o Heartbeat tenha uma limitação de poder monitorar apenas 2 *nodes* simultaneamente, nesta implementação pode ser desconsiderada, haja visto que o mesmo foi empregado apenas para monitorar os balanceadores de cargas primário e secundário.

Com a utilização do sistema de arquivos CODA, a sincronização dos dados foi resolvida e conseqüentemente, devido as características do CODA, o tempo de *failover* foi muito reduzido. Mesmo perdendo a conexão com o servidor, os *nodes* clientes podem continuar operando através do cache local. Caso o problema seja no próprio *node*, esse pode ser retirado do *cluster* e após reparado reconectar ao servidor para novamente obter seus dados, e voltar a operação de forma transparente. Embora CODA apresentou-se lento na escrita de arquivos pequenos

em grande quantidade, seu emprego no ambiente de internet não traz nenhuma degradação ao sistema.

No final, conseguiu-se uma solução de altíssima disponibilidade amplamente escalável, podendo ser otimizada de acordo com as particularidades de cada aplicação.

Referências Bibliográficas

BERKELEY. *The Berkeley Intelligent RAM Project*. [S.l.], 2005. Disponível em: <http://iram.cs.berkeley.edu>.

BRAAM, P. *The Coda Distributed File System*. [S.l.], 2005. Disponível em: <http://www.coda.cs.cmu.edu/ljpaper/lj% -.html>.

BUYYA, R. *High Performance Cluster Computing*. New Jersey: PRENTICE HALL PTR, 1999.

HANSEN, C. *Parallel Web Server*. [S.l.], 2005. Disponível em: <http://www.cs-.umd.edu/projects/hpsl/chaos/ResearchAreas/hpio/ad/pweb.html>.

HUANG C. KINTALA, N. K. Y.; FULTON, N. Software rejuvenation: Analysis, module and applications. *Proceeedins from the 25th Simposium on Fault Tolerant Computing*, 1995.

HUANG, K.; ABRAHAM, J. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. on Computers*, C, n. 33(6), p. 518–528, 1984.

INDIA, A. S. S. of the Department of Electronics Government of. *PARMON Product Brochure*. [S.l.], 2005. Disponível em: <http://www.buyya.com/parmon-/parmonckslash_brochure.pdf>.

MAGAZINE, L. *Linux Virtual Servers Clusters*. [S.l.], 2005. Disponível em: <http://www.linux-mag.com/2003-11/clusters_01.html>.

PITANGA, M. *Construindo Supercomputadores com Linux*. 1. ed. São Paulo: BRASPORT, 2002.

PROJECT, L. V. S. *Linux Virtual Server Project*. [S.l.], 2005. Disponível em: <http://www.austintek.com/LVS/LVS-HOWTO/mini-HOWTO/LVS-mini-HOWTO-pt.html>.

STEELEYE. *TLifeKeeper for Linux*. [S.l.], 2005. Disponível em: <http://www-.steeleye.com/literature>.

Apêndice A

Glosário

ABFT - Algorithm Based Fault Tolerance.
ATM - Asynchronous Trasnfer Mode.
CC-NUMA - Cache-Coherent Nouniform Memory Access.
CISC - Complex Instruction Set Computing.
CLUMPs - Cluster of SMPs.
COWs - Cluster of Workstation. Cluster
Node - A host in a cluster.
CoPs - Cluster of PCs.
DIMM - Dual In-Line Memory Module.
DRAM - Dynamic Random Access Memory.
DSM - Distributed Shared Memory.
EDO - Extend Data Out.
EIDE - Enhanced Integrated Drive Electronics: the standard disk.
FCAL - Fiber Channel Arbitrated Loop.
HA - High Availability.
HP - High Peformance.
HPC - High Performance Computing.
HT - High Throughput.
Heartbeat - Communication method in a loosely coupled cluster to
IPAT - Abbreviation for IP Address Takeover
ISA - Integrated Systems Architecture.
Idle Standby -
LAN - Local Area Network.
LVS - Linux Virtual Server.
MAC - Address Ethernet hardware address.
MIPS - Million instructions per second.

MPI - Message Passing Interface.
MPP - Massively Parallel Processors.
Mutual Takeover - A redundancy setup where both nodes do business.
NIC - Network Interface Card.
PC - Personal Computer. PCI - Peripheral Component Interconnect.
PVM - Parallel Virtual Machine.
RAID - Redundant Arrays of Independent Disks.
RAM - Random Access Memory.
RISC - Reduced Instruction Set Computer.
Raw Disk Devices - Unbuffered I/O devices which make sure data is.
Rotating Standby - A redundancy setup where one node does business.
SCI- Scalable Coherent Interface.
SCSI - Small Computer System Interface: the standard disk.
SGI - Silicon Graphics, Inc.
SIMM - Single In-line Memory Module.
SMP - Symmetric Multiprocessors.
SPOF - Single Point of Failure: a part which renders an entire.
SSA - Serial Storage Architecture.
URL - Uniform Resource Locator.
standby - Estado de espera.
VESA - Video Electronics Standards Association.
VLIW - Very Long Instruction Word.
VLSI - Very Large Scale Integration.
WAN - Wide Area Network.