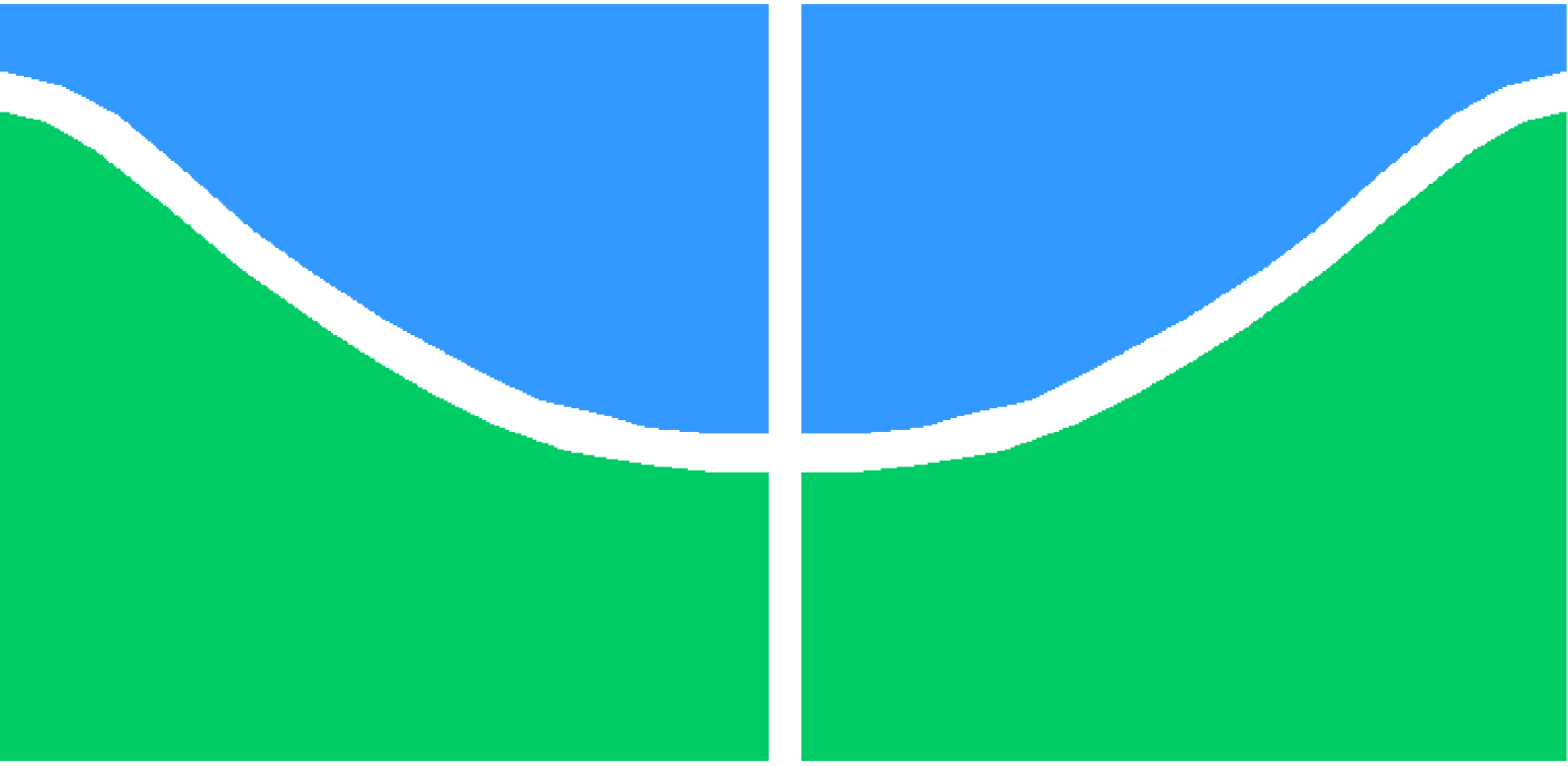

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Projeto e implementação de um balanceador de carga de servidores confiável para sistemas Linux

Autor: Matheus Souza Fonseca

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF

2015

Matheus Souza Fonseca

Projeto e implementação de um balanceador de carga de servidores confiável para sistemas Linux

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF

2015

Matheus Souza Fonseca
Projeto e implementação de um balanceador de carga de servidores confiável para sistemas Linux/ Matheus Souza Fonseca. – Brasília, DF, 2015-
104 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. Balanceamento de carga. 2. Algoritmo de escalonamento. I. Prof. Dr. Luiz Augusto Fontes Laranjeira. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Projeto e implementação de um balanceador de carga de servidores confiável para sistemas Linux

CDU

Matheus Souza Fonseca

Projeto e implementação de um balanceador de carga de servidores confiável para sistemas Linux

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 14 de Maio de 2015:

Prof. Dr. Luiz Augusto Fontes Laranjeira
Orientador

Prof. Dra. Carla Silva Rocha Aguiar
Membro convidado 1

Prof. Dr. André Barros de Sales
Membro convidado 2

Brasília, DF
2015

Este trabalho é dedicado a todas as pessoas que participaram diretamente ou indiretamente da minha trajetória acadêmica até o atual momento.

"Não force o crescimento, elimine os fatores que o limitam."
— Peter Sange.

Resumo

Com o surgimento da World Wide Web, há uma crescente e incessante demanda por serviços fornecidos por sistemas de informação das mais variadas dimensões. Desde o uso de redes sociais até o uso de sistemas de *e-commerce*, as infraestruturas de TI das organizações precisam atender importantes requisitos para alcançar a satisfação do usuário, dentre os requisitos se encontram o desempenho, escalabilidade, confiabilidade e disponibilidade. Para tentar suprir estes requisitos estão sendo continuamente desenvolvidas soluções em hardware e software, dentre elas existe o Balanceamento de carga de servidores, que constitui em um processo de distribuição uniforme de requisições a serviços utilizando um dispositivo baseado em rede, ou seja, uma técnica de distribuição de trabalho a um *cluster* de servidores via um algoritmo de escalonamento. O presente trabalho tem como ideia projetar e implementar um produto, um balanceador de carga de servidores simplificado para computadores que fazem uso de sistemas operacionais baseados em Linux, em integração conjunta com o software que fornece um serviço de *failover*, o Application Manager. O *core* do sistema foi desenvolvido como um módulo do *kernel*, fazendo uso do Netfilter, um *framework* nativo do *kernel* do Linux para manipulação de pacotes de rede.

Palavras-chaves: balanceamento de carga. algoritmo de escalonamento. linux. netfilter.

Abstract

With the emergence of the World Wide Web, there is a growing and constant demand for services provided by information systems of all sizes. Since the use of social networks to the use of e-commerce systems, the IT infrastructures of the organizations need to meet important requirements to achieve user satisfaction, among the requirements are performance, scalability, reliability and availability. To try to meet these requirements are continually being developed hardware and software solutions, among them is the Load balancing server, which is a process of uniform distribution of requests to services using a network-based device, i.e., a technique of distribution of work to a cluster of servers via a scheduling algorithm. This present work has the idea to design and implement a product, a simplified server load balancer for computers that use Linux-based operating systems, with joint integration with a software that provides a failover service, the Application Manager. The core of the system was developed as a kernel module, using Netfilter, a native Linux kernel framework for handling network packets.

Key-words: load balancing. scheduling algorithm. linux. netfilter.

Lista de ilustrações

Lista de tabelas

Lista de quadros

Lista de abreviaturas e siglas

TCC	Trabalho de Conclusão de Curso
ASICs	Application Specific Integrated Circuits
API	Application Programming Interface
OSI	Open Systems Interconnection
IP	Internet Protocol
VIP	Virtual Internet Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
PDU	Protocol Data Unit
ISO	International Organization for Standardization
ICMP	Internet Control Message Protocol
GPL	GNU Public License
CPU	Central Processing Unit
RAM	Random Access Memory
DNS	Domain Name System
KISS	Keep It Simple, Stupid
NAT	Network address translation
HTTP	Hypertext Transfer Protocol
LAN	Local Area Network
CLI	Command Line Interface
IPC	Inter-Process Communication
LVS	Linux Virtual Server

Sumário

1 Introdução

A internet em conjunto com suas ferramentas de acesso (e.g. *World Wide Web*) é a mais poderosa ferramenta de comunicação do mundo, por intermédio dessa são diariamente realizadas desde simples conversas entre pessoas, até transações financeiras de alto valor. As mudanças na forma de comunicação que esta infraestrutura causou e causa até hoje na vida das pessoas e suas relações é incalculável. Todas as vantagens que esse meio de comunicação oferece, implica no fato de que dia após dia, existe um aumento de uso dessa infraestrutura.

Como prova deste aumento de uso, vejamos a progressão do número de buscas feitas na ferramenta de busca Google no período de seis anos (STATISTIC BRAIN, 2014). No ano de 2007 foram realizadas uma média de 1.200.000.000 (um bilhão e duzentos milhões) de buscas por dia, já no ano de 2013, o dado mais recente no período de realização deste presente trabalho, foram realizadas uma média de 5.922.000.000 (cinco bilhões, novecentas e vinte e duas milhões) de buscas por dia. Um aumento aproximado de 493,5% nas buscas, ou seja, um fluxo de requisições quase cinco vezes maior em um período 6 anos.

Esta situação se repetiu e se repete em vários outros serviços disponíveis na *web*, até mesmo internamente em empresas (com uma escala menor, obviamente). Isto motivou o desenvolvimento de diversas soluções de sistemas distribuídos para amenizar os impactos causados por esta situação, uma dessas soluções é o balanceamento de cargas em servidores.

1.1 Justificativa

Dados os fatos introduzidos anteriormente, este trabalho de conclusão de curso é motivado a desenvolver uma solução que contorne o aumento do uso de serviços em uma rede. Levando-se em consideração um serviço que aumenta diariamente a sua taxa de acessos, podem surgir diversos problemas, como os a seguir:

- Diminuição de desempenho do serviço;
- Caso o serviço esteja centralizado em um único local (e.g. Servidor físico), se houver uma falha neste servidor todo o serviço ficará indisponível;
- Se for optado pelo aumento dos recursos do servidor (e.g. memória RAM, processamento, interface de rede com maior largura de banda, etc), a situação pode ser contornada temporariamente, mas se o aumento de requisições prosseguir, em algum momento será atingido o limite físico da máquina, e o problema retornará.

O balanceamento de carga em servidores é uma possível solução definitiva para estes problemas. Atualmente existem diversas soluções de balanceamento baseadas em software e hardware, disponibilizadas de forma comercial ou até mesmo gratuitamente. As soluções comerciais são normalmente ASICs (*Application Specific Integrated Circuits*), uma solução conjunta de software e hardware especializado para o balanceamento de carga, possuindo um desempenho melhor e um alto custo, exemplos de soluções são o Cisco CSS[1], o Alteon NG Load Balancer[2] e o F5 LTM[3]. Já em relação as soluções baseadas somente em software, que operam sobre sistemas operacionais padrões (e.g. Debian Linux[4], Red Hat ES[5], Windows Server[6], etc), existem versões pagas, gratuitas e até livres, exemplos de soluções são o LVS (Linux Virtual Server) [7] e o HAProxy[8] (softwares livres).

O trabalho fica justificado pelo seu fim didático e por este propor uma solução mais minimalística em relação as citadas anteriormente.

1.2 Objetivo geral

O presente trabalho tem o objetivo de projetar e implementar uma solução de balanceamento de carga de servidores confiável, sendo a sua plataforma de execução sistemas operacionais baseados em Linux.

1.3 Objetivos específicos

- Levantar material bibliográfico para suporte a toda a pesquisa e desenvolvimento relacionado a este trabalho;
- Desenvolver a arquitetura geral de uma solução de balanceamento de carga, além da devida explicação de detalhamentos técnicos, definindo as tecnologias a serem utilizadas e os componentes a serem criados. Exemplos de tais definições: Linguagem de programação, plataforma de execução, visão geral dos componentes (API's, bibliotecas, etc), camada de rede de atuação, tipo de retorno de tráfego, forma de interceptação de pacotes, forma de comunicação entre os componentes do sistema, forma de configuração do sistema, etc;
- Implementar a solução de balanceamento de carga seguindo as especificações definidas em seu projeto, além da realização de testes de desempenho da mesma.

1 <http://www.cisco.com/go/css11500/>
2 <http://www.radware.com/Products/Alteon/>
3 <https://f5.com/products/modules/local-traffic-manager>
4 <https://www.debian.org/>
5 <http://br.redhat.com/products/enterprise-linux/>
6 <http://www.microsoft.com/pt-br/server-cloud/products/windows-server-2012-r2/>
7 <http://www.linuxvirtualserver.org/>
8 <http://haproxy.1wt.eu/>

- Projetar e implementar a integração do software a ser desenvolvido (balanceador de carga) e do software Application Manager.
- Realizar uma análise sobre os resultados alcançados.

1.4 Estrutura do trabalho

A estruturação do trabalho está de acordo com o sumário deste documento. No capítulo 1 é feita uma introdução ao tema, oferecendo a devida motivação para a realização da pesquisa, juntamente com informações adicionais sobre o trabalho. No capítulo 2 são citados vários conceitos que oferecem toda a base para a realização do trabalho, ou seja, uma fundamentação teórica necessária. O capítulo 3 também oferece fundamentos teóricos, mas são fundamentos específicos sobre balanceamento de carga. No capítulo 4 é fornecida uma visão alto nível (generalista) da solução que está sendo proposta, demonstrando os componentes da arquitetura do balanceador de cargas. Já o capítulo 5 é responsável por fornecer detalhes técnicos sobre como foi realizada a implementação funcionalidades mais importantes do sistema. O capítulo 6 visa a descrição e análise sobre os resultados alcançados no projeto, demonstrando medidas coletadas sobre o uso do sistema. No capítulo 7 será feita a conclusão do trabalho, fazendo um resumo a tudo o que feito pesquisado, projetado e desenvolvido, listando também os futuros trabalhos que poderão ser desenvolvidos em cima deste trabalho de conclusão de curso.

1.5 Metodologia de trabalho

Para a realização deste trabalho a coleta de dados foi feita através de livros, artigos e documentações de software, todos no segmento de Redes de computadores, Sistemas distribuídos e *kernel* do Linux.

2 Fundamentação teórica

Este capítulo terá a função de demonstrar a pesquisa realizada acerca dos diversos temas que tangem este trabalho, fornecendo um referencial teórico para descrição e justificativa do projeto da solução. Somente assuntos com maior recorrência serão trabalhados aqui, caso um determinado assunto pertinente não seja explicado neste capítulo, significa que ele será abordado diretamente nos posteriores.

2.1 Terminologia básica

Neste trabalho surgirão diversos termos que serão recorrentemente citados e devem ter significados esclarecidos para não haver interpretações incorretas. Ressalta-se que estes termos possuem estes significados no domínio deste trabalho (balanceamento de carga), em outros domínios eles podem ter outros significados. A seguir os termos:

Servidor real: Um servidor real é um dispositivo conectado em rede que possui serviços em execução, estes serviços tem a carga compartilhada entre os outros servidores (BOURKE, 2001, p. 15).

***Cluster* de servidores:** Um *cluster* de servidores (i.e. fazenda de servidores) é um aglomerado de servidores independentes, trabalhando em conjunto como um sistema único. Neste trabalho o *cluster* de servidores será a representação de todos os servidores reais, mas diferentemente de uma solução de *cluster* típica, os servidores não se comunicarão entre si.

VIP: Um VIP (*Virtual IP*) é o endereço IP do servidor de balanceamento de carga que é acessível fora da rede (normalmente um IP público), fazendo a representação de todo o *cluster* de servidores, que são os *hosts* para quem ele despacha as requisições.

***Failover*:** *Failover* é o processo de troca de um servidor primário para um servidor secundário/redundante quando ocorre um colapso do sistema primário (JAYASWAL, 2006, p. 309). Para determinar quando ocorre o colapso, pode ser utilizada a técnica de *heartbeat*.

***Kernel*:** O *kernel* é o principal programa de computador que compõe um sistema operacional. Ele possui funcionalidades essenciais para o funcionamento da máquina, como a interação com o hardware da máquina e o provimento de ambiente de execução para outros programas (BOVET; CESATI, 2000, p. 12).

Camada de rede: Uma camada de rede é um componente de uma arquitetura de um software de rede. Em uma pilha de protocolos, os protocolos são agrupados por camadas, onde cada camada tem o objetivo de oferecer solução a um subproblema do todo.

Pacote de rede: Em sentido estrito, pacote representa o conjunto de informações trocadas pelas camadas de rede (protocolo IP), mas este termo também é amplamente utilizado para representar qualquer PDU (*Protocol Data Unit*) de qualquer camada de rede, seja ele um quadro *ethernet*, um pacote IP ou um segmento TCP. Neste trabalho um pacote terá este significado mais amplo.

2.2 Protocolos de rede

Logo após o começo da pesquisa e desenvolvimento em redes de computadores [1], foram desenvolvidas soluções de software de rede para redução da complexidade de comunicação do sistema. Comumente os software de rede não são um grande "amontoado" de código que efetua toda a operação de comunicação entre máquinas. Como uma boa decisão arquitetural, o software que realiza essa tarefa foi componentizado, de forma que os problemas que esse "amontoado" de código resolveria, fosse dividido de acordo com suas similaridades e cada componente ficaria responsável por resolver um pequeno conjunto de problemas (princípio de "Dividir para conquistar"[2]).

Estes componentes foram organizados de uma forma hierárquica, formando uma pilha de camadas, umas sobre as outras. Cada camada tem a função de fornecer determinados serviços as camadas superiores, e uma camada "X" de uma máquina "A" tem a função de se comunicar com a mesma camada "X" de uma máquina "B" (TANENBAUM, 2003, p. 37). Seguindo o fluxo dessa pilha de camadas a comunicação seria estabelecida, conforme a Figura 1.

A forma em que essas camadas se comunicam é determinado pelo protocolo de rede utilizado, que é uma implementação da solução dos problemas associados a camada, alinhado a um conjunto de regras e procedimentos a serem respeitados para emissão e recepção de mensagens.

2.2.1 Modelo OSI

O primeiro e mais conceituado modelo de camadas é o OSI (*Open System Interconnection*), um modelo de referência publicado pela ISO[3] (*International Organization for Standardization*) que determina sete camadas e seus objetivos (seguindo a lógica proposta

[1] Ocorreu em meados da década de 1960, resultando na rede ARPANET.
[2] <http://pt.wikipedia.org/wiki/Divis%C3%A3o_e_conquista>
[3] Organização internacional de padronização: <http://www.iso.org/iso/>

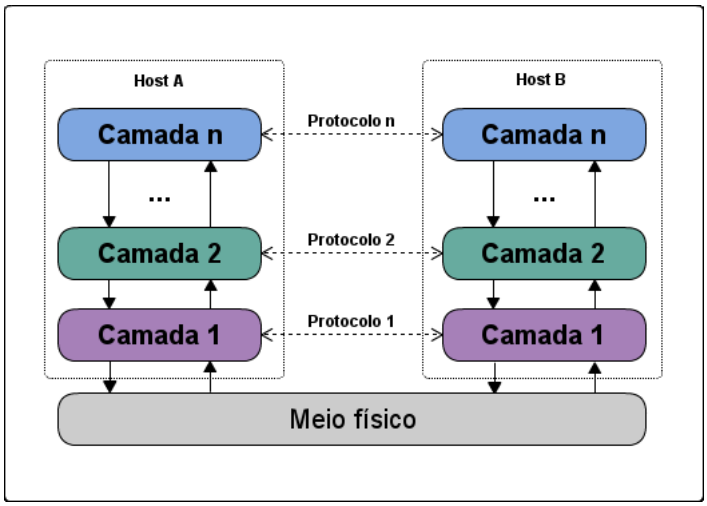

Figura 1: Camadas de rede

na seção anterior). Este é somente um modelo conceitual e não um guia de implementação, onde futuras implementações reais se basearam. A seguir uma breve descrição de cada camada de acordo com (TANENBAUM, 2003, p. 45-47):

1. **Física:** Trata da transmissão de bits (1's e 0's) por um meio físico de comunicação (e.g. cabo coaxicial, cabo de par trançado, cabo de fibra ótica, etc);

2. **Enlace de dados:** Transformar a transmissão de dados em algo mais confiável, reduzindo os erros de transmissão e agrupando as informações em pacotes (nesta camada as PDU's são chamadas de "quadro");

3. **Rede:** Roteamento dos pacotes (pacotes IP) até o seu destino, passando nas sub-redes onde eles trafegam;

4. **Transporte:** Fornecer um canal ponto a ponto livre de erros entre os dois *hosts* que se comunicam, diferente da camada de enlace esta normalmente tem a função de comunicar os *hosts* de origem e destino, enquanto as camadas inferiores comunicam os *hosts* com seus vizinhos imediatos, assim a origem e destino podem estar separados por vários roteadores;

5. **Sessão:** Estabelece um controle do diálogo que está ocorrendo entre os dois *hosts*;

6. **Apresentação:** Controle da sintaxe e semântica do diálogo, podendo controlar as diferentes formas em que diferentes arquiteturas computacionais organizam suas estruturas de dados (e.g. *Little endian* e *Big endian*);

7. **Aplicação:** É composta por protocolos para atendimento de demandas do usuário, que podem ser desde a transferência de e-mails até a uma transmissão multimídia.

2.2.2 Pilha TCP/IP

Diferentemente do modelo de referência OSI, que foi definido inicialmente por um grupo de trabalho da ISO antes mesmo de qualquer pilha de protocolos serem implementadas, o modelo de referência oficial da internet foi formalizado após os protocolos forem projetados, implementados e testados (COMER, 2013, p. 53). Este modelo oficial é chamado de TCP/IP [4], ele possui somente cinco camadas e cada uma contém n protocolos associados.

Fazendo uma relação com o modelo OSI, a pilha TCP/IP possui as camadas de Sessão e Apresentação incorporadas na camada de Aplicação, o restante continuando da mesma forma, com os mesmos objetivos (fruto da influencia do modelo OSI).

A pilha TCP/IP atualmente é implementada por padrão em todos os sistemas operacionais modernos, e como é um padrão utilizado na internet geralmente não ocorrem problemas de comunicação em máquinas com diferentes sistemas operacionais ou arquiteturas computacionais. A seguir uma breve lista de alguns dos protocolos mais famosos da pilha TCP/IP, agrupados por camada (WIKIPEDIA, 2014):

1. **Física:** Bluetooth, RS-232, USB (*Universal Serial Bus*), etc;
2. **Enlace de dados:** Ethernet, PPP (*Point to Point Protocol*), Frame Relay, etc;
3. **Rede:** IP (*Internet Protocol*), ARP (*Address Resolution Protocol*), ICMP (*Internet Control Message Protocol*), IGMP (*Internet Group Management Protocol*) etc;
4. **Transporte:** TCP (*Transmission Control Protocol*), UDP(*User Datagram Protocol*), etc;
5. **Aplicação:** HTTP(*Hypertext Transfer Protocol*), FTP(*File Transfer Protocol*), DNS(*Domain Name System*), etc.

2.3 Sistemas Linux

O Linux é um *kernel* (núcleo) de sistema operacional baseado em Unix, não sendo considerado um sistema operacional completo por não incluir por padrão outras ferramentas utilitárias necessárias para ser categorizado como tal. Dentro da "árvore genealógica" de sistemas operacionais baseados em Unix[5], o Linux hoje obtém a posição de segundo mais utilizado no mundo até o presente momento, perdendo somente para sistemas Mac OS X da Apple[6]. A Tabela 1 demonstra os dados mensais de deste ano de 2014.

[4] Faz referência aos seus dois protocolos mais famosos, TCP e IP.
[5] Imagem da árvore genealógica: <http://www.computerworld.com/common/images/site/features/2009/062009/unix_chart_775.jpg>
[6] <http://www.apple.com/br/osx/>

Tabela 1: Estatística de uso de sistemas operacionais

2014	Win8	Win7	Vista	NT	WinXP	Linux	Mac	Mobile
Abril	15.8%	55.4%	1.2%	0.2%	8.0%	5.0%	10.3%	4.0%
Março	15.0%	55.1%	1.3%	0.2%	9.4%	4.9%	9.9%	4.0%
Fevereiro	14.2%	55.0%	1.4%	0.3%	10.1%	5.0%	10.0%	4.0%
Janeiro	13.4%	55.3%	1.5%	0.3%	11.0%	4.9%	9.6%	4.0%

Fonte: W3SCHOOLS (2014).

Estes resultados englobam sistemas operacionais de todos os tipos, mas se a pesquisa fosse restrita a sistemas operacionais para uso em servidores, os que fazem uso do *kernel* Linux disparadamente ocupariam o primeiro lugar. Grande parte desse sucesso é devido a enorme comunidade que oferece desenvolvimento contínuo e grande suporte as versões do Linux, resultante da filosofia de software livre adotada.

2.3.1 Conceitos básicos

O Linux é um *kernel* de sistema operacional não-proprietário sob licensa GPL (*GNU Public License*), o que o caracteriza como um software livre. Um software livre se baseia em quatro fundamentos de liberdade principais (FREE SOFTWARE FOUNDATION, 2014):

- A liberdade de usar um software para qualquer propósito,
- A liberdade de modificar um software para necessidades próprias,
- A liberdade de compartilhar um software com seus amigos e vizinhos, e
- A liberdade de compartilhar as mudanças que você fez no software.

Quando um software atinge esse grau liberdade, ele é considerado um software livre. O Linux atende todas estas liberdades, e como resultado disso ele possui uma vasta comunidade de desenvolvedores experientes, que além de efetuarem as mais diversas correções e melhorias em seu código, conseguiram portar este para ser utilizado em diferentes arquiteturas computacionais (e.g. Intel x86, Intel x86-64, AMD 64, ARM, PowerPC, entre outras), sendo possível fazer uso do sistema operacional em servidores de grande porte, computadores pessoais, *tablets*, celulares e até em uma geladeira.

Todas as características de software livre citadas anteriormente são extremamente benéficas para os campos de ensino, pesquisa e desenvolvimento também. Afinal, as liberdades possibilitam a investigação do código fonte e maior liberdade de acesso a funcionalidades do sistema, o que facilita o desenvolvimento uma nova solução (o objetivo deste trabalho).

Como já foi dito anteriormente, o Linux é um *kernel*, sendo o principal programa de um sistema operacional e ficando responsável por: Gerenciamento de recursos (memória, processador, disco e periféricos), escalonamento de processos, provimento de ambiente de execução para aplicações, etc. Outras funcionalidades estão implementadas em aplicativos utilitários, como: Editor de texto, interpretador de comandos, compilador, ambiente de *desktop*, etc. O *kernel* Linux e os aplicativos utilitários em conjunto formam um sistema operacional por completo, onde normalmente comunidades de desenvolvedores os customizam e os distribuem com um determinado codinome, as chamadas distribuições Linux (e.g. Ubuntu, Debian, Fedora, Arch Linux, etc).

2.3.2 Modos de CPU (*kernel* e usuário)

De acordo com Atwood (2008), o *kernel* de um sistema trabalha em um modo especial chamado comumente de modo *kernel*, este oferece uma série de privilégios em relação ao acesso direto de funcionalidades do hardware da máquina (e.g. executar qualquer instrução da CPU e referenciar qualquer área da memória RAM). Se todo e qualquer processo pudesse ter acesso a essas funcionalidades, existiram diversos problemas de segurança por conta de programas mal intencionados. Por este motivo os outros aplicativos (e.g. editores de texto, compiladores, etc) atuam em um modo diferente, mais restritivo, chamado comumente de modo usuário.

Essa separação é garantida pela própria CPU, que é capaz de distinguir qual modo está sendo executado o processo corrente e então restringir determinadas instruções. Mas existem determinados momentos, onde alguns processos comuns precisam ter acesso ao hardware, como por exemplo, acesso a um arquivo que está gravado em disco. Para ter tal recurso o processo precisa fazer uma chamada ao sistema (i.e. *system call*), que é um pedido feito ao *kernel* do sistema, para que esse acesse o hardware e depois entregue o pedido de volta ao processo (novamente, uma questão de segurança). A Figura 2 ilustra essa situação.

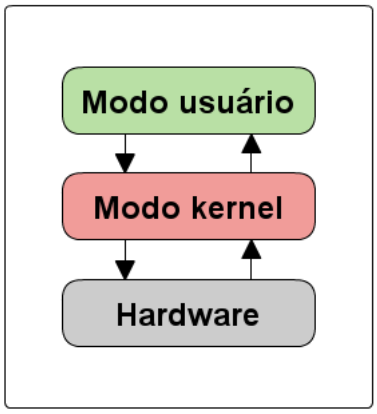

Figura 2: Interação entre modos de CPU e hardware

Existem outras características de modos de CPU específicas de sistemas Linux, por exemplo, em modo *kernel* o processo tem a maior prioridade de requisição de memória dinâmica, já em modo usuário a requisição de memória é considerada não-urgente. Outra questão, em modo *kernel* as funcionalidades são consideradas livres de erro (o *kernel* confia em si próprio), não sendo necessários mecanismos avançados de prevenção de erros de programação, já em modo usuário estes erros são esperados e tratados, por isso quando um processo em modo usuário falha, dificilmente irá travar todo o sistema (BOVET; CESATI, 2000, p. 183).

Estas características enfatizam duas coisas, o modo *kernel* preza por desempenho, já que tem acesso mais direto ao hardware alinhado de maior prioridade de acesso a memória. O modo usuário preza por estabilidade, já que o processo neste modo é supervisionado pelo *kernel*, possuindo uma maior proteção contra erros.

2.3.3 Módulos do *kernel*

O Linux contempla uma série de funcionalidades como as já descritas anteriormente, mas e quando surge a necessidade de uma funcionalidade que não foi desenvolvida no seu código fonte? Por exemplo, estabelecer a comunicação com um novo hardware desenvolvido. Isso é possível graças aos módulos do *kernel*. A seguir uma descrição de acordo com Salzman, Burian e Pomerantz (2007, p. 2):

> Módulos são pedaços de código que são carregados e descarregados no *kernel* sob demanda. Eles estendem a funcionalidade do *kernel* sem a necessidade de reiniciar o sistema [...] Sem módulos, nós precisaríamos de criar *kernels* monolíticos[7] e adicionar a funcionalidade diretamente na imagem do *kernel*. Ter grandes *kernels* ainda tem a desvantagem de nos exigir a reconstrução e reiniciação do *kernel* toda vez que queremos novas funcionalidades.

A ideia de módulos vieram de uma estratégia utilizada em *microkernels*, que são *kernels* com somente as principais funcionalidades do sistema, as outras deveriam ser acopladas ao *kernel* sempre que necessário, oferecendo uma maior flexibilidade arquitetural do sistema. Um tipo de módulo pode ser um *device driver*[8], um código que oferece uma interface ao sistema operacional para interação com um hardware. Exemplos de módulos que não são *device drivers* podem ser a implementação de um escalonador de processos personalizado, um manipulador de pacotes de rede, etc.

Os módulos sempre atuam no modo *kernel* de CPU, pois quando são carregados dinamicamente eles são acoplados ao *kernel* do sistema. O que trás a vantagem de maior desempenho e acesso a funcionalidades restritas, mas trás a desvantagem de perda de

[7] <http://pt.wikipedia.org/wiki/N%C3%BAcleo_monol%C3%ADtico>
[8] Caso ele seja inserido estaticamente a imagem do *kernel*, ele não é um módulo.

estabilidade e maior complexidade de desenvolvimento (como foi dito anteriormente na subseção 2.3.1).

2.3.4 Netfilter

Especificamente para módulos do *kernel* Linux que trabalham com aspectos de rede, existe uma série de componentes que podem ser utilizados para ter acesso a determinadas funcionalidades, como a interceptação de pacotes. Este conjunto de componentes fazem parte do *framework* chamado de Netfilter, que é composto de quatro partes (RUSSEL; WELTE, 2002):

***Hooks*:** São pontos pré-definidos na travessia de pacotes na pilha de protocolos TCP/IP de uma máquina. Toda vez que é chegado neste ponto pré-definido, o *hook* chamará o que estiver registrado nele;

Registro de *hooks*: Qualquer módulo do *kernel* pode registrar um *hook* com uma função de *callback* própria[9], ou seja, quando um pacote chegar a uma determinada camada de rede, a função de *callback* implementada e registrada no *hook* irá receber o pacote da camada e várias operações poderão ser feitas nele (aceitar, descartar, esquecer, empilhar e também efetuar alterações);

Empilhamento de pacotes: Quando em uma função de *callback*, um pacote é empilhado, ele pode ser acessado por aplicações no modo usuário de CPU, de forma totalmente assíncrona;

Documentação: Todas estas funcionalidades possuem uma boa documentação, além de comentários no próprio código do componente.

2.4 Balanceamento de carga de servidores

Esta seção pode ser considerada uma das mais importantes de todo o trabalho, pois um dos objetivos específicos citados anteriormente na seção 1.3 era o de levantamento bibliográfico para suporte ao trabalho, e como este trabalho irá projetar uma solução de balanceamento de carga de servidores, nada mais importante do que possuir uma definição teórica básica sobre o assunto.

2.4.1 Definição e tipos de balanceamento

De acordo com Karimi et al. (2009):

[9] Funções de *calback* são passadas como argumento de uma outra função, para serem futuramente utilizadas.

> Balanceamento de carga é a técnica de divisão de trabalho entre dois ou mais computadores, *links* de rede, CPU's, discos rígidos, ou outros recursos, a fim de obter a melhor utilização dos recursos, vazão (i.e. *throughput*) ou tempo de resposta.

Como a referência citou, o termo recurso pode ter diferentes representações, onde a definição de balanceamento de carga ainda seria passível de aplicação, por exemplo, a distribuição de carga entre trabalhadores de uma empresa. Mas as formas que estes balanceamentos são feitos contêm suas peculiaridades, a seguir diferenças de duas das principais:

Balanceamento de carga de CPU: Em um sistema com multiprocessamento, as CPU's podem estar em uma mesma máquina ou distribuídas, e o intuito do balanceamento de carga é distribuir um trabalho, mas este trabalho é divido em partes, onde os processadores atuam paralelamente (e.g. As instruções de um programa são divididas entre as CPU's).

Balanceamento de carga de servidores: Em uma arquitetura *web* podem existir diversos servidores que rodam o mesmo *website*, a carga de trabalho pode ser distribuída entre eles, mas neste caso os servidores não atuam obrigatoriamente em paralelo, provavelmente eles nem se comunicam ou sabem da existência um do outro. Neste caso cada pedido ao *website* é distribuído entre os servidores, mas um mesmo pedido não será processado em dois ou mais servidores. A Figura 3 representa este caso.

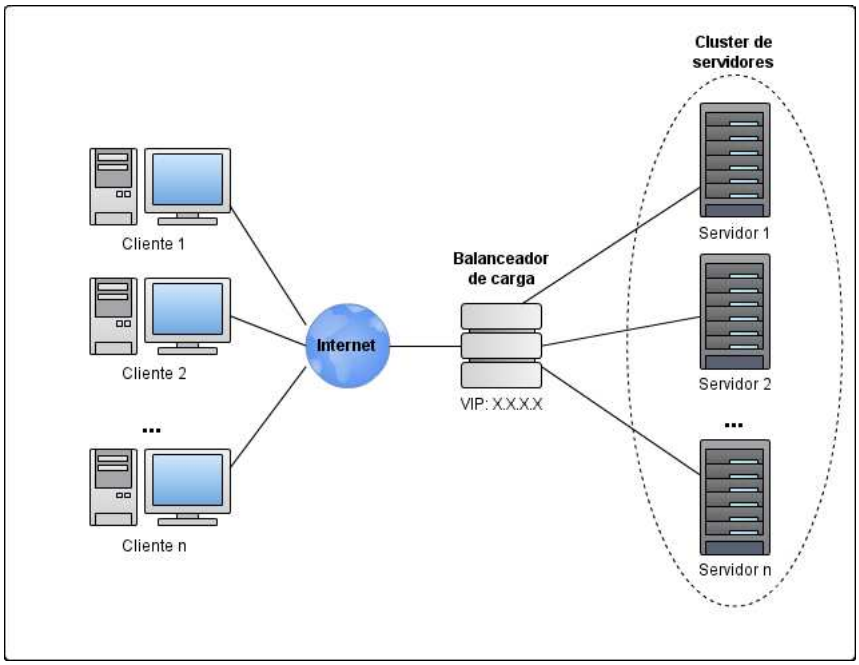

Figura 3: Balanceador de carga de servidores

Na sua forma mais simples, o processo de balanceamento de carga de servidores que ocorre na Figura 3 é o seguinte:

1. Um cliente Y requisita um serviço (e.g. HTTP, SMTP, Telnet, FTP, etc) a um determinado endereço IP (X.X.X.X) e porta;
2. O balanceador de carga que responde pelo VIP X.X.X.X recebe a requisição, determina por meio de um algoritmo qual será o servidor real a responder pelo serviço, então altera o IP de destino para corresponder com o servidor real selecionado;
3. O servidor real recebe a requisição, efetua o processamento e envia a resposta para o endereço do remetente da requisição (cliente Y) por sua rota padrão;
4. O balanceador de carga é o *gateway* da rota padrão, então ele recebe de volta a resposta do servidor real, mas para ela não ser descartada pelo cliente X o balanceador altera agora o IP do remetente, colocando novamente o VIP X.X.X.X.

Esta solução citada utiliza a técnica de retorno de tráfego via NAT, mas existem outras forma de efetuar o balanceamento que serão citadas posteriormente no decorrer do trabalho.

2.4.2 Histórico

Com o aumento do tráfego na internet foi necessário um determinado investimento na estrutura das redes e das empresas que disponibilizavam serviços. Inicialmente como primeira medida para aumentar o desempenho dos servidores, foi utilizada a abordagem de *scale-up*, que consiste em aumentar os recursos computacionais de uma máquina (CPU, memória RAM, velocidade do disco rígido, etc). Durante algum tempo essa abordagem atendeu algumas necessidades, mas rapidamente foram atingidos limites de melhoria (os recursos computacionais para uma máquinas são finitos), além disso, os recursos com maior capacidade tinham altos preços, e ao final de tudo as máquinas ainda corriam o risco de falhar em algum momento e deixar seus serviços indisponíveis.

Outra abordagem que foi iniciada para se adaptar ao aumento do tráfego foi a de *scale-out*, onde ao invés de aumentar os recursos de uma única máquina, é aumentado o número de máquinas para trabalhar em conjunto de forma distribuída. De acordo com (SHARMA; SINGH; SHARMA, 2008), um sistema computacionalmente distribuído provê compartilhamento de recursos como uma das suas maiores vantagens, o que provê melhor performance e confiabilidade do que qualquer outro sistema tradicional nas mesmas condições.

Antes dos balanceadores de carga virarem um produto real, os administradores de sistema faziam uso da técnica baseada em DNS *round-robin* para atingir resultados simi-

lares. Um DNS tradicionalmente faz associação de um domínio (e.g. www.example.com) a um IP (e.g. 187.10.10.1), mas os servidores de DNS também oferecem suporte a associação de mais de um IP a um domínio, o que possibilita que cada requisição ao domínio seja distribuída entre os n servidores reais associados.Para escolha de qual será o servidor que receberá a requisição, é utilizado o algoritmo *round-robin* (será discutido mais a frente) para escolha. Mas esta solução contém algumas desvantagens em relação as soluções de balanceamento de carga tradicionais, como suporte a um único algoritmo de decisão e inexistência de monitoramento dos servidores, onde caso ocorra o colapso de um desses servidores, o DNS não fará nada para contornar (BOURKE, 2001, p. 5-6).

2.4.3 Benefícios

Com o surgimento das soluções de balanceadores de carga baseados em servidores (operam sobre máquinas servidoras com sistemas operacionais padrão) foram alcançados diversos benefícios, vários deles já foram citados diretamente ou indiretamente neste trabalho. A seguir uma lista dos principais de acordo com Bourke (2001, p. 8):

Escalabilidade: Em uma infraestrutura com balanceamento de carga, existe um bom nível de escalabilidade dos serviços, pois para aumentar o seu poder é necessário somente adicionar mais servidores reais. Também é facilitada a remoção de servidores com defeito, ou a desativação e ativação de servidores de acordo com a demanda. Tudo com transparência para os clientes.

Desempenho: São alcançados altos níveis de performance a um custo mais baixo do que se fossem comprados computadores com recursos computacionais mais potentes. Dependendo da eficiência do algoritmo utilizado para decisão de despacho das requisições, onde serão escolhidos os que estiverem menos ocupados, melhor será o desempenho.

Disponibilidade e confiabilidade: Caso um dos servidores falhe, o balanceador de carga é capaz de detectar e redirecionar o fluxo para outros, sem que o serviço como um todo fique indisponível. Outro aspecto importante é que o balanceador de carga não deve ser um gargalo para o serviço (caso ele seja o nó a falhar), então é necessário aplicar alguma redundância no serviço de balanceamento, possuindo um servidor de *backup* adicionado de um processo de *failover* integrado, aumentando a confiabilidade do sistema.

3 Arquitetura de um balanceador de carga de servidores

A partir das definições básicas abordadas anteriormente sobre os balanceadores de carga de servidores, é possível detalhar melhor os seus componentes, mostrando de fato como eles atuam nas redes de computadores modernas e demonstrando quais são as suas possibilidades reais de funcionamento. As decisões de quais são as melhores estratégias e a forma de organização dos componentes de uma solução, determina qual é a arquitetura da mesma.

Em um projeto real de um balanceador de cargas devem ser tomadas precauções para que os objetivos almejados sejam alcançados, ou o pior pode acontecer, o que já não era o ideal pode se tornar pior, devido ao aumento da complexidade do sistema sem evidências de melhoria. De acordo com (BOURKE, 2001, p. 41), existem três aspectos que devem ser procurados em uma solução, e outros três aspectos que devem ser evitados em uma solução: Simplicidade ao invés de complexidade, funcionalidade ao invés de desperdício e elegância ao invés de irrelevância. Todas elas remetem a um princípio conhecido como KISS (*Keep It Simple, Stupid*)[1], ou seja, existirão diversas soluções que resolvem o problema de balanceamento de carga, mas cabe ao arquiteto que está projetando a solução escolher a que atenda da forma mais simples e direta.

Os detalhes arquiteturais de um balanceador podem ser divididos em quatro categorias principais:

1. De acordo com a camada de rede de atuação (segundo o modelo OSI);
2. De acordo com o retorno do tráfego de rede;
3. De acordo com a decisão de balanceamento das requisições;
4. De acordo com a forma de redundância do serviço.

3.1 Camada de rede de atuação (modelo OSI)

Uma importante decisão arquitetural na implementação ou implantação de um balanceador de carga, é decidir em qual camada de rede ele vai atuar, ficando definido qual tipo de pacote ele irá balancear o tráfego (e.g. Quadro *ethernet*, pacote IP, segmento TCP, mensagem HTTP, etc).

[1] Mantenha isso simples, estúpido: <http://pt.wikipedia.org/wiki/Keep_It_Simple>

Existe uma forma de balancear o tráfego na camada 2, para isso é utilizada a técnica de *link agreggation* que consiste união de dois ou mais *links* de rede, simulando um único *link* lógico de alta velocidade e redundância. Também pode ser realizado na camada 3, onde os pacotes IP em um dispositivo de rede são redistribuídos seguindo algum algoritmo, o problema é que via camada 3 só são conhecidos os possíveis *hosts* alvos, não tendo conhecimento da aplicação alvo (só seria acessível via a porta do serviço). Normalmente os equipamentos como *switches* e roteadores operam até a camada 3, sendo possível acessar cabeçalhos de quadros *ethernet* e pacotes IP.

Natário (2011) cita que as camadas de rede mais utilizadas são a 4 e a 7. O balanceamento de carga na camada 4 é um método mais simplista do que na camada 7, consiste na redistribuição de requisições na camada de transporte, fazendo uso normalmente dos protocolos TCP ou UDP. A sua simplicidade está ligada ao balanceador não fazer análise do conteúdo (i.e *payload*) dos segmentos TCP ou UDP, somente tendo a função de repassar a informação, resultando em um maior desempenho. Já na camada 7 é feito um extenso uso de análise de informações que estão sendo trefegadas, sendo capaz de analisar a URL (*Uniform Resource Locator*) de um pedido, os *cookies*[2] que estão sendo trefegados, o tipo de dado que foi requisitado (via *mime-type*[3]). Em cima dessas informações podem ser tomadas decisões mais inteligentes, fazendo por exemplo uma separação dos servidores da infraestrutura de acordo com os seus dados, um grupo especializado em informações estáticas (e.g. páginas HTML estáticas, imagens e vídeos) e outro grupo especializado em informações dinâmicas (e.g. scripts PHP), melhorando a organização da infraestrutura. Resumindo o balanceamento na camada 4 presa por desempenho livre do contexto das aplicações, já o balanceamento na camada 7 presa por uma análise inteligente dos conteúdos das requisições, de forma a melhorar a qualidade do serviço da aplicação.

3.2 Retorno de tráfego

Acerca da forma de retorno do tráfego dos servidores reais que estão atrás do balanceador, serão abordados aqui três estratégias que são parcialmente descritas por Bourke (2001, p. 44-45), mas principalmente baseadas nas técnicas de retorno de tráfego do LVS, descritas por Zhang (2000). As técnicas são via NAT (*Network Adress Translation*), via tunelamento IP e via DR (*Direct Routing*).

3.2.1 Retorno de tráfego via NAT

Este é a técnica de retorno de tráfego mais simples das três, mas por conta da sua simplicidade existe uma considerável perda de desempenho se comparado as outras técni-

[2] <http://en.wikipedia.org/wiki/HTTP_cookie>
[3] <http://en.wikipedia.org/wiki/Internet_media_type>

cas. Devido ao alcance de endereços IP no IPv4 [4] e também por requisitos de segurança, existem um conjunto de IP's reservados para uso interno em organizações, chamados de IP falsos ou IP privados. Isso impossibilita que este endereço seja utilizado publicamente na internet, mas possibilita que o endereço seja reutilizado por outros *hosts* em sub-redes internas de outras organizações, quantas vezes forem necessárias (dependendo da arquitetura da rede, até na mesma organização). Mas para estes *hosts* com IP privados conseguirem se comunicar com a internet, deve ser utilizada a técnica conhecida como NAT, onde o IP privado será representado pelo IP público do *gateway* que está na saída padrão da rede, sofrendo um processo de tradução sempre que for necessário.

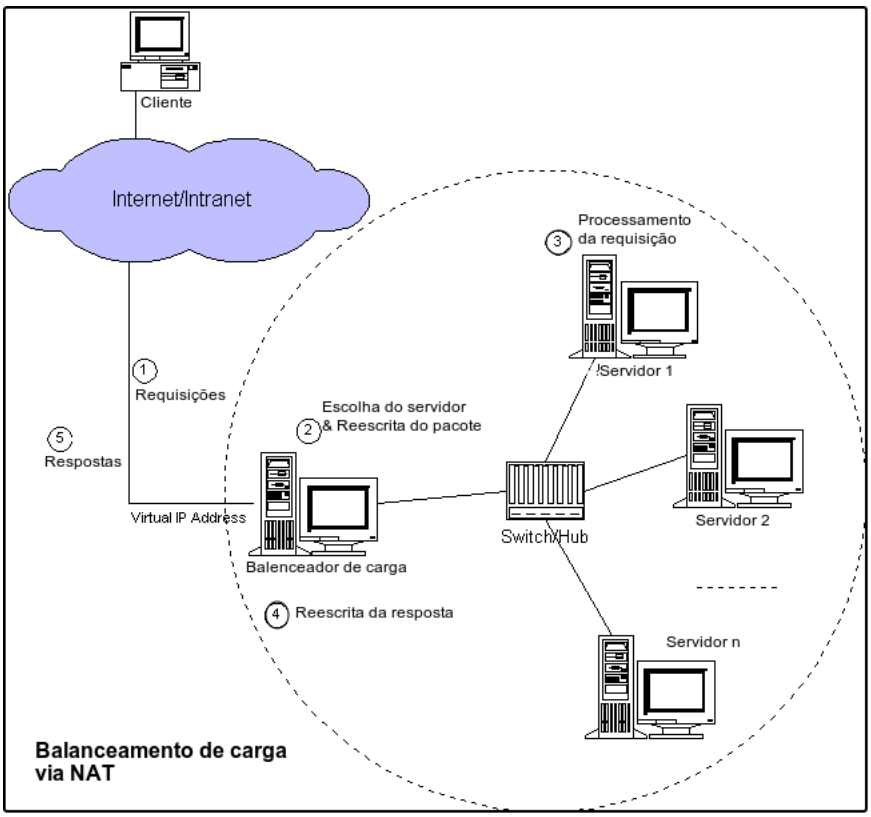

Figura 4: Retorno de tráfego via NAT

Fonte: Zhang (2000). Adaptado pelo autor.

A Figura 4 Ilustra o retorno de tráfego via NAT. A seguir uma explicação da técnica seguindo a numeração dos passos na Figura 4:

1. Uma requisição realizada por um cliente da internet chega na rede;

2. O balanceador de cargas recebe a requisição, escolhe qual será o servidor que irá processá-la e então reescreve o endereço de destinatário do pacote IP, substituindo pelo IP privado do servidor real escolhido;

[4] O IP versão 4 é a versão do protocolo IP mais utilizada atualmente na internet. Mas a versão 6 (IPv6) aos poucos vem crescendo e tende a substituir o IPv4 por conta dos seus benefícios.

3. A requisição chega a um servidor real, ele a processa e manda a resposta para o balanceador de volta, já que ele é a rota padrão e o endereço IP do cliente não possui rota definida para o servidor real;

4. O balanceador recebe o pacote de resposta, então ele novamente altera o pacote, inserindo o seu IP como remetente para que o cliente não o descarte;

5. O pacote segue de volta ao cliente da requisição.

Essa mesma estratégia pode ser utilizada também na camada de transportes, a diferença principal é que as portas de remetente e destinatário também deveriam ser alteradas, como é feito no LVS via PAT (*Port Address Translantion*). Um problema associado a essa técnica é que o balanceador é um potencial gargalo para o sistema, pois ele atua tanto como porta de entrada da rede como porta de saída, podendo ficar sobrecarregado de funções. As próximas técnicas diminuem um pouco essa sobrecarga pelo balanceador se tornar somente porta de entrada da rede, mas em compensação elas aumentam a complexidade do sistema pois definem restrições a rede e/ou ao *cluster* de servidores.

3.2.2 Retorno de tráfego via tunelamento IP

Tunelamento IP é uma técnica que, de forma resumida, encapsula um pacote IP dentro de outro pacote IP e o repassa para outro *host* processá-lo (ZHANG, 2000). Para ter suporte a este tipo de técnica o balanceador de carga e os servidores reais devem ter suporte ao protocolo de tunelamento IP, além disso, deve existir uma interface de rede virtual específica para tunelamento, e ela deve estar configurada de modo que se estabeleça um túnel entre o balanceador de carga com o VIP e os servidores reais. Normalmente são utilizados dois protocolos de tunelamento IP em conjunto, o L2TP (*Layer 2 Tunneling Protocol*) na camada 2 em conjunto com o IPsec (*Internet Protocol Security*) na camada 3, opções amplamente utilizadas em VPN's (*Virtual Private Networks*). A Figura 5 demonstra o uso da técnica.

Observando a Figura 5 vemos algumas diferenças entre a técnica usando NAT. Até o momento da chegada da requisição no balanceador tudo ocorre da mesma forma, a escolha do servidor real que irá processar também ocorre igualmente, mas no momento em que deveria ser feita a reescrita do pacote ocorre algo diferente. O balanceador fazendo uso do seu protocolo de tunelamento, insere o pacote IP dentro de um outro pacote, então sendo possível enviá-lo ao servidor escolhido via o túnel (anteriormente configurado entre o balanceador e os servidores reais). No momento em que o servidor escolhido recebe o pacote, faz uso do seu protocolo de tunelamento para desencapsular o pacote IP e então pode processar a requisição normalmente. A grande diferença ocorre agora, ao analisar o pacote o servidor real consegue averiguar que o pacote era destinado ao VIP, que está

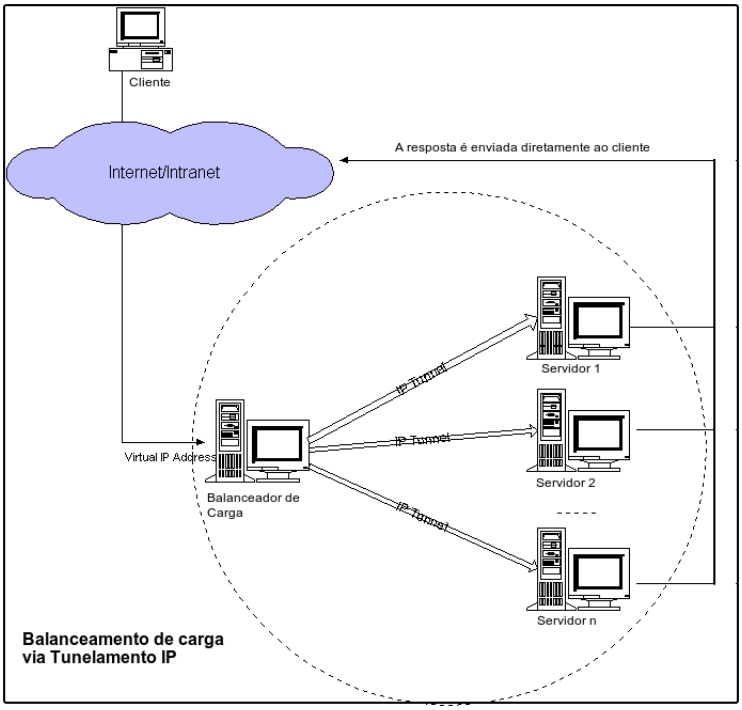

Figura 5: Retorno de tráfego via tunelamento IP

Fonte: Zhang (2000). Adaptado pelo autor.

configurado em seu túnel, então ao gerar uma resposta ele insere o VIP como remetente, sendo possível enviá-la diretamente ao cliente. Outra característica marcante da técnica é que somente nela é possível que os servidores reais estejam em uma sub-rede diferente da do balanceador de carga, isso por conta do pacote da requisição poder estar encapsulado em outros pacotes.

A técnica remove o *overhead* do balanceador, agora ele somente precisa interceptar os pacotes que entram na rede, em sua saída os pacotes podem ser enviados diretamente para o cliente, via uma rota padrão diferente. Mas em compensação existe também um *overhead* neste tipo de técnica, pois os pacotes agora são inseridos dentro de outros pacotes, existindo dois cabeçalhos para cada conjunto de dados, agora serão necessários trafegar alguns pacotes a mais. Também existem as tarefas adicionais de encapsular e desencapsular os pacotes IP, mas mesmo assim, o desempenho da solução por tunelamento IP é superior a via NAT.

3.2.3 Retorno de tráfego via DR

A implementação da técnica via DR (*Direct Routing*) é a terceira e última técnica de retorno de tráfego explicada por Zhang (2000). Para aplicação desta técnica é necessário que o balanceador e os servidores reais tenham uma de suas interfaces de rede ligadas por um segmento interrupto, ou seja, ou devem estar ligados via um cabeamento direto, ou por um HUB/*switch*, tudo em uma única LAN (*Local Area Network*). Outra característica é

que todos os servidores reais devem possuir um endereço IP igual ao VIP, mas sem remover o endereço IP que identifica unicamente os servidores reais dentro da LAN.

Um endereço IP às vezes é erroneamente definido como uma forma de identificação de uma máquina na rede, mas na verdade o endereço IP não está associado ao *host*, ele é associado a uma interface de rede deste, possibilitando que um mesmo *host* tenha n IP's para suas n interfaces de rede. Na técnica via DR, como os servidores reais e o balanceador estão interligados em uma LAN, não há a necessidade de incluir outra interface de rede para ser possível associar o VIP nos servidores reais, é possível utilizar a interface de *loopback*[5]. A Figura 6 ilustra o uso da técnica.

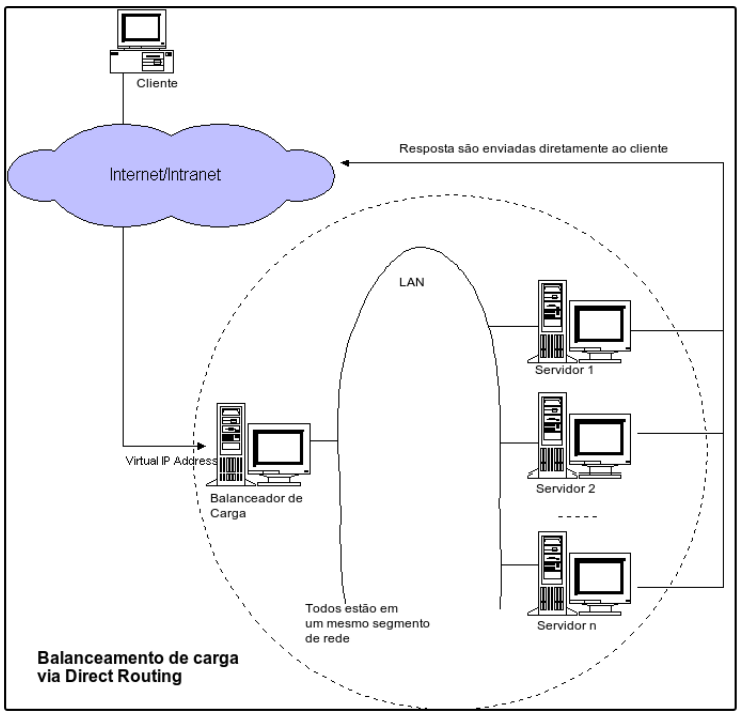

Figura 6: Retorno de tráfego via DR

Fonte: Zhang (2000). Adaptado pelo autor.

A Figura 6 demonstra que a arquitetura deve estar configurada de modo que todos os *hosts* estejam conectados em um mesmo segmento de rede. O processo que ocorre nesta técnica é o seguinte:

1. Um cliente envia a requisição ao balanceador de carga;

2. Após decisão de qual servidor real irá efetuar o processamento, o balanceador troca o endereço MAC (*Media Access Control*) do destinatário (que originalmente era o seu próprio MAC). Isso deve ocorrer para que o endereço VIP de destino do pacote permaneça inalterado;

[5] Interface de rede virtual utilizada para serviços locais na máquina

3. O servidor real recebe a requisição, verifica que ele é o destinatário por causa do seu VIP, processa a requisição e envia uma resposta para a saída padrão da rede. O campo de destinatário da resposta é o endereço do cliente e o campo de remetente é o VIP do balanceador, que estava configurado em sua interface de *loopback*;

4. O cliente recebe a resposta e a aceita, pois o endereço de remetente é realmente um endereço a quem ele fez uma requisição.

Um problema aparente desta solução é com o protocolo ARP, este protocolo tem a função de encontrar um endereço MAC de uma interface em uma rede, informando qual o endereço IP da interface que se deseja a informação. No caso da técnica via DR isto representa um problema por conta de todos os *hosts* da LAN compartilharem um mesmo IP, podendo ocasionar problemas na rede. Para resolver a situação é necessário que o sistema operacional hospedeiro dos *hosts* possibilite a desabilitação do protocolo ARP nas interfaces de *loopback*.

3.2.4 Comparação entre as técnicas de retorno de tráfego

A Tabela 2 mostra uma comparação entre as técnicas de retorno de tráfego, levando-se em consideração: Especificações do servidor, forma de configuração da rede, número de servidores reais aconselhados na rede e gateway da rede.

Tabela 2: Comparação entre técnicas de retorno de tráfego

	Via NAT	**Via tunelamento IP**	**Via Direct Routing**
Especificações	Qualquer	Protocolo e Túnel configurado	Interface de loopback com VIP e ARP desabilitado
Rede	LAN	LAN/WAN	LAN
Quantidade	Baixo (10 a 20)	Alto (apróx. 100)	Alto (apróx. 100)
Rota padrão	Balanceador de carga	Roteador da rede	Roteador da rede

Fonte: Zhang (2000). Adaptado pelo autor.

3.3 Algoritmos de escalonamento

Esta seção trata de um dos mais importantes aspectos do balanceamento de carga, não somente de balanceamento de cargas de servidores, mas grande parte das técnicas que serão abordadas a seguir são aplicáveis em todos os tipos de balanceamento de carga (e.g. CPU e disco). O balanceamento de carga é um problema de escalonamento de trabalho, trabalho que originalmente deveria ser realizado por um único indivíduo (e.g. *Host* de rede ou núcleo de uma CPU) poderá ser dividido e realizado por outros n indivíduos, para tomar a decisão de como organizar este trabalho é necessário um algoritmo de escalonamento.

Casavant e Kuhl (1988) definiram uma taxonomia para caracterização de algortmos de balanceamento de carga, a seguir algumas das categorias componentes dessa taxonomia:

Estático vs. Dinâmico: Um algoritmo estático tem como premissa o desempenho dos indivíduos que irão realizar o trabalho, ou seja, antes de se iniciar o escalonamento carga de trabalho é definida a forma que ele será distribuído, não sendo alterado em tempo de execução. Já um algoritmo dinâmico possui poucas ou nenhuma informação sobre os indivíduos, todo a decisão de escalonamento será realizada em tempo de execução;

Cooperativo vs. Não-cooperativo: Em um algoritmo não-cooperativo os indivíduos realizam suas tarefas independente dos outros indivíduos do sistema, já em um cooperativo um indivíduo pode alterar a forma de realização de sua tarefa devido a uma possível cooperação com outro indivíduo (neste caso os indivíduos realmente se conhecem).

Adaptável vs. Não-adaptável: Em uma solução adaptável a forma de escalonamento da carga de trabalho poderá ser modificada em tempo de execução, isso ocorre principalmente devido a análise de um comportamento anterior do sistema, onde se descobre que uma possível alteração no escalonamento pode aumentar o desempenho do sistema. Já em um não-adaptável não existe modificação na forma de escalonamento em tempo de execução.

Foram omitidas algumas outras categorias dessa taxonomia, pois as selecionadas são a que tem maior importância quando se trata especificamente do balanceamento de carga em servidores. A seguir serão descritos um total de cinco algoritmos de balanceamento de carga de servidores, três estáticos e dois dinâmicos.

a) *Round-robin* e Aleatório

É um dos mais antigos algoritmos na área da computação, aplicado originalmente em escalonamento de processos. No balanceamento de carga de servidores foi originalmente aplicado no DNS *round-robin* (como explicado na seção 2.4.2). É um algoritmo estático, não-cooperativo e não-adaptável que consiste em uma simples técnica que distribui de forma homogênea as requisições aos servidores que foram previamente ordenados, similar a um sentido horário. Este algoritmo trata todos os servidores de forma igualitária, somo se tivessem a mesma configuração de hardware, não diferenciado-os, caso isso seja a realidade do *cluster* este algoritmo tende a trabalhar bem (MADHURAVANI; SUMALATHA; SHANMUKHI, 2012).

Um cenário explicativo desta situação a seguir: Existem três servidores (S1, S2 e S3) e foram enviadas em sequência quatro requisições (R1, R2, R3 e R4) ao sistema,

usando o algoritmo round-robin, R1 seria destinado ao S1, R2 ao S2, R3 ao S3 e R4 ao S1.

A única diferença do algoritmo *round-robin* com o aleatório é que ao invés da requisição ser distribuída seguindo a ordem dos servidores (1, 2, 3, ..., n), ela é distribuída de forma aleatória, mas sem atribuição de duas requisições ao mesmo servidor antes que um ciclo completo de requisições tenha sido distribuído aos outros servidores.

b) *Ratio*

O algoritmo *ratio*, também chamado de *round-robin* com pesos em algumas soluções de balanceamento de carga, é outro algoritmo estático, não-cooperativo e não-adaptável. O *round-robin* tradicional é aconselhado para *clusters* com servidores homogêneos, mas isso nem sempre é a realidade de uma organização, podendo existir servidores mais antigos e com configuração mais básica atuando em conjunto com servidores mais novos com maior poder computacional, nestes casos o *round-robin* tradicional pode sobrecarregar alguns servidores enquanto outros estão ociosos (JAYASWAL, 2006, p. 224).

O algoritmo *ratio* é indicado para *clusters* com servidores heterogêneos, onde são distribuídos pesos entre eles, os que tiverem maior poder computacional possuem maior peso e os com menor poder um peso menor. Um cenário explicativo poderia ser o seguinte: Quatro servidores reais (S1, S2, S3 e S4) com os pesos 10, 5, 5 e 7, respectivamente. Com a soma dos pesos sendo 27, para o servidor S1 seriam direcionadas 37% das requisições, para o S2 e S3 aproximadamente 18% cada, já pro S4 cerca de 26% das requisições.

c) Menos conexões

Como explicado anteriormente, os algoritmos estáticos tem o seu comportamento determinado previamente, não possuindo mudanças em tempo de execução. Já os algoritmos dinâmicos podem tomar as decisões baseadas em novas informações do sistema (MADHURAVANI; SUMALATHA; SHANMUKHI, 2012), estas podendo ser coletadas via monitoramento dos servidores ou via análise de informações mantidas pelo próprio balanceador.

Uma informação que o balanceador pode e deve armazenar é o número de conexões que cada servidor real esta processando. Uma conexão aqui pode ser tratada como o processamento de requisições de um único cliente, por exemplo, caso um servidor real esteja tratando 3 requisições, sendo que duas destas pertencem a um mesmo cliente, o servidor possuí um total de 2 conexões então. A partir do número de conexões de cada servidor real, o balanceador poderá dirigir as novas requisições ao servidor com o menor número de conexões no momento (JAYASWAL, 2006, p. 224). Este algoritmo é chamado de “Menos conexões”, ele é um algoritmo dinâmico, não-cooperativo e adaptável.

d) Menor latência

Outro algoritmo dinâmico, não-cooperativo e adaptável, é o "Menor latência", semelhante a estratégia do ultimo algoritmo apresentado, onde é tomada a decisão de balanceamento em cima de informações do sistema, só que neste caso ela é coletada a partir do monitoramento do *cluster*. A latência (i.e. *ping*), é a quantidade de tempo que um dispositivo de rede deve esperar pela resposta de outro (JAYASWAL, 2006, p. 131). Em um período de tempo pré-determinado é feita a coleta do tempo de resposta destes servidores, onde normalmente se utiliza o protocolo ICPM para tal. Os servidores com menor latência serão os escolhidos para receber requisições.

Um problema deste protocolo é que ele está fazendo uma medida relacionada a máquina servidor, mas o que realmente importa no ponto de vista de um cliente é a performance do serviço que este servidor está provendo. Algo ainda pior pode ocorrer, o servidor real pode estar em pleno funcionamento, mas os serviços dele podem se encontrar inoperantes ou sobrecarregados, neste cenário caso a latência do servidor ainda esteja mais baixa em relação a outros, ele ainda sim poderia ser selecionado para receber novas requisições (que seriam frustradas). Uma possível solução para o problema seria checar os tempos de resposta dos serviços ao invés de ser capturada a latência do servidor, e usá-los para o os valores coletados para escalonamento. A forma de checar a latência de um serviço pode ser via a diferença de tempo entre uma requisição TCP SYN e o retorno TCP SYN ACK[6], já que o uma conexão TCP está ligada a uma porta/serviço (KOPPARAPU, 2002, p. 34). A mesma solução não é válida para serviços que operam sobre o protocolo UDP já que este não é orientado a conexão, não possuindo mensagens ACK[7].

3.4 Redundância do serviço

Clientes da infraestrutura de TI[8] com o passar dos anos aumentaram a sua dependência de serviços disponibilizados na internet e esse crescimento é incessante (e.g. 24h por dia e 7 dias por semana). Então existe uma necessidade de garantir que os clientes desses serviços *online* não fiquem desapontados com um serviço fora do ar por causa de uma falha em hardware ou software. Existem requisitos não funcionais[9] que tratam paralelamente esta necessidade, os requisitos de disponibilidade e confiabilidade.

Para garantia destes requisitos existe uma área de estudo na computação chamada de Tolerância a falhas. De acordo com Jayaswal (2006, p. 233):

> Tolerância a falhas é a capacidade de um *host* ou subsistema de recuperar a falha de um componente sem a interrupção do serviço. Um

[6] Parte do que é conhecido como *handshake* de três vias: <http://andreysmith.wordpress.com/2011/01/02/three-way-handshake/>

[7] Abreviação de *acknowledge*. É a mensagem de reconhecimento.

[8] Acrônimo para Tecnologia da Informação.

[9] A norma ISO/IEC 9126 trata de conceituar estes requisitos e outros: <http://pt.wikipedia.org/wiki/ISO/IEC_9126>.

> *host* tolerante a falhas ou dispositivo tem componentes redundantes que monitoram uns aos outros.

Para implementar esta redundância em balanceadores de cargas de servidores, existem dois cenários típicos que serão descritos a seguir. O cenário mais comum de redundância é o cenário ativo-*standby*, onde existem dois servidores (instâncias) de balanceador de cargas, uma sendo o servidor primário, e outro o servidor secundário, também chamado de servidor de backup. Quando se inicia a execução do sistema, o servidor primário entra em estado de ativo do sistema (tendo o comportamento normal de um balanceador), já o servidor secundário entra em estado de *standby* (em prontidão). Quando ocorre uma falha no servidor em estado de ativo, o servidor em estado de *standby* detecta isso usando algum tipo de técnica, neste momento o servidor secundário entra em estado de ativo, e o servidor primário (com falha) entra em estado de *standby* (CISCO SYSTEMS, 2010, p. 33.1-33.2). Esse processo de troca é chamado de *failover*.

Outro cenário é o ativo-ativo, onde existem dois servidores que atuam de forma igualitária, os dois se encontram ativos e fazem o balanceamento de carga. Caso ocorra a falha em um dos servidores, o outro irá continuar executando as suas operações e o serviço ainda continuará online.

Para que o servidor detecte a falha em outro, seja em ativo-*standby* ou ativo-ativo, deve ser utilizado algum protocolo de monitoramento em conjunto com um cabo de *heartbeat*. Esse cabo é configurando ligando os dois servidores diretamente por suas interfaces (e.g. cabo RS232 ligando interfaces seriais DB9 ou um cabo de par trançado ligando interfaces ethernet) e então o protocolo de *heartbeat* estabelece uma comunicação por pulsos entre os dois servidores, estes pulsos determinam se o servidor está operando normalmente, caso os pulsos se encerrem significa que alguma falha ocorreu e o servidor entrou em colapso (JAYASWAL, 2006, p. 334-336). A detecção de erros do sistema está completamente ligada a sua confiabilidade. A Figura 7 ilustra a organização da técnica em um cenário de redundância ativo-*standby*:

No cenário ativo-*standby* é valido ressaltar que ao ocorrer o processo de *failover*, as configurações do servidor falho devem ser incorporadas no novo servidor ativo (e.g. endereço IP, tabela de conexões, etc), para isso elas devem ser periodicamente transferidas entre os servidores. Também deve ser ressaltado que no cenário ativo-ativo, os dois servidores devem possuir alguma política de configuração do VIP, já que existirão duas instâncias do balanceador de carga trabalhando paralelamente.

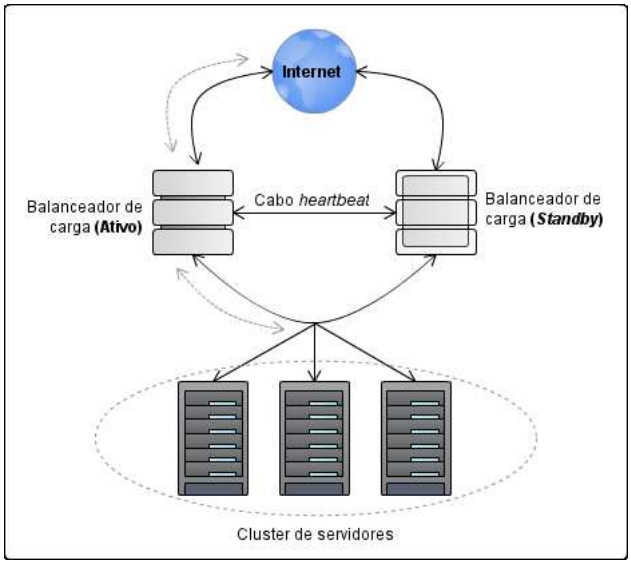

Figura 7: Cenário de redundância ativo-*standby*

4 Projeto arquitetural

Este capítulo tem uma grande importância para este trabalho, neste capítulo que são descritas as estratégias que foram utilizadas pelo autor para a solução que foi implementada. Para construção do conhecimento abordado a seguir, foi necessária a pesquisa prévia de todo o referencial teórico em que este trabalho foi baseado (capítulos 2 e 3).

Especificamente neste capítulo será tratada a arquitetura do software proposto, será descrito um modelo conceitual que facilitará a transição dos requisitos de um balanceador de carga para a sua implementação, ou seja, as informações explanadas neste capítulo serviram como um guia para o desenvolvimento do sistema. A arquitetura de software é uma das subáreas de conhecimento da engenharia de software de acordo com SWEBOK[1] (*Software Engineering Body of Knowledge*).

A arquitetura em grosso modo pode ser descrita como a subdivisão de um todo em partes, estabelecendo relações específicas entre as partes. Garlan e Shaw (1994) definem a motivação da arquitetura de software como:

> Conforme o tamanho e a complexidade de sistemas de software aumentam, o problema de design vai além dos algoritmos e das estruturas de dados da computação. A projeção e a especificação da estrutura geral do sistema emergem como um novo tipo de problema. As questões estruturais incluem organização total e estrutura de controle global; protocolos de comunicação, sincronização e acesso a dados; atribuição de funcionalidade a elementos de design; distribuição física; composição de elementos de design; escalonamento e desempenho; e seleção entre as alternativas de *design*.

Para uma documentação que siga as melhores práticas empreendidas na indústria atualmente, o projeto arquitetural do balanceador de cargas foi feito tendo como uma das bases o *template* de um documento de arquitetura de software do RUP - *Rational Unified Process* (IBM RATIONAL SOFTWARE, 2001), utilizando-o de forma simplificada.

4.1 Representação da arquitetura

Neste capítulo serão indicadas as decisões de uso das tecnologias que foram explicadas no referencial teórico, e pelo seu baseamento em um *template* de documento de arquitetura de software, este capítulo também conterá subseções que tratam das visões de arquitetura aplicáveis neste contexto (personalizadas para melhor atender o trabalho),

[1] Arquitetura de software está contida dentro de uma outra área de conhecimento mais genérica que é a de *design* de software: <http://www.computer.org/portal/web/swebok>

e ao final listando restrições de qualidade do sistema. A seguir uma breve descrição de cada item citado:

Tecnologias e técnicas utilizadas: Uma listagem e justificativa de quais tecnologias, técnicas e estratégias foram utilizadas no balanceador de carga (e.g. qual modo de CPU será utilizado, qual a camada de rede de atuação, qual a técnica de retorno de tráfego, etc).

Visão lógica: Uma descrição da arquitetura lógica do balanceador, explicando uma decomposição em módulos e camadas.

Visão de implantação: Uma descrição da decomposição do balanceador em termos físicos, ou seja, explicação de quais são os hosts da rede, forma de interconexão dos hosts (meio físico e protocolos de comunicação) e mapeamento dos programas que atuarão nos hosts.

4.2 Tecnologias e técnicas utilizadas

Seguindo ainda o princípio KISS introduzido por Bourke (2001, p. 41), nesta seção serão descritos e justificados quais tecnologias, técnicas e estratégias foram utilizadas na implementação do balanceador de carga de servidores (grande parte das alternativas já foram listadas nos capítulos 2 e 3), sempre prezando pela simplicidade do sistema.

a) Camada de rede de atuação

O balanceador implementado é baseado em servidores comuns, fazendo uso de um computador comum com um sistema operacional padrão, isso porque este tipo de solução requer somente o desenvolvimento do software, não sendo necessário o desenvolvimento da solução de hardware como em balanceadores baseados em AISIC's.

Os balanceadores na camada 2 são em sua maioria baseados em AISIC's, ficando descartada a sua utilização. A camada 3 também tem grande parte das suas soluções baseadas as AISIC's, isso ocorre por essa camada ser a mais alta camada de atuação de dispositivos de rede como os roteadores (atuam normalmente na 1, 2 e 3), o que colabora com o desenvolvimento de um balanceador com software e hardware personalizados, mas mesmo assim existem algumas soluções baseadas em servidores comuns na camada 3. Balanceadores na camada 7 estão ligados a aplicações específicas (e.g. *web*, e-mail, servidor de arquivos, etc), o que não é o foco deste trabalho, já que se almeja um sistema mais genérico que não faça distinção do tipo de aplicação, ficando também descartada a solução na camada 7.

Ficaram como opções as camadas 3 e 4 do modelo OSI para atuação. Depois de uma análise de vantagens e desvantagens de atuação nestas duas, foi escolhida a camada 4, a seguir as justificativas:

- O protocolo IP, que é o principal protocolo da camada 3 é não orientado a conexão, o que significa que este protocolo tem uma baixa confiabilidade.
- O protocolo TCP, um dos principais protocolos da camada 4, é categorizado como orientado a conexão e fornece um *stream* confiável de dados. Isso significa que uma conexão prévia é estabelecida entre os *hosts* comunicantes (*handshake* de três vias), e para cada segmento de dados enviado existe uma resposta de reconhecimento (ACK), que afirma que o segmento chegou ao seu destino. Ao final também existe um momento de finalização da conexão. A Figura 8 explica esse processo de comunicação.

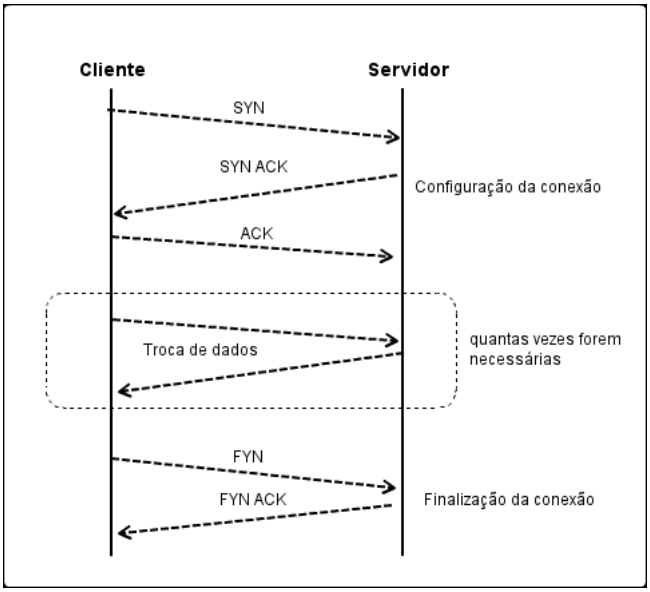

Figura 8: Processo de comunicação TCP

- O protocolo TCP oferece um mecanismo de prevenção de entrada de dados duplicados, porque cada segmento possui um campo `SEQUENCE NUMBER` e outro `ACKNOWLEDGEMENT NUMBER`, o primeiro número identifica o segmento e o segundo número serve como um indicador do que o remetente espera como `SEQUENCE NUMBER` da resposta, oferecendo um maior controle dos pacotes recebidos (COMER, 2013, p. 211).
- Os protocolos da camada 4 de rede estão relacionados a portas, o que os remete a comunicação entre aplicações de dois hosts. Isso facilita a prevenção o envio de pacotes indesejados do balanceador até o *cluster*, porque somente serão enviados pacotes destinados aos serviços anteriormente cadastrados pelo balanceador. Por exemplo, em um *cluster* de servidores *web* somente é desejado o acesso as portas

80 e 443[2] (HTTP e HTTPS), qualquer requisição de acesso a outra porta não será enviado. Em um protocolo da camada 3 não existe a distinção de tipos de aplicação, podendo somente ser definido o endereço de um *host* e não um serviço que se deseja acesso, isso poderia ocasionar o envio de requisições a todo e qualquer serviço a um *host*, desde que ele esteja cadastrado.

b) Modo de CPU

A decisão do modo de CPU que uma aplicação irá atuar é uma ponderação sobre desempenho e estabilidade, como já foi abordado na seção 2.3.2. Este trabalho está sendo extensamente baseado na implementação da solução em software livre LVS, e os seus escalonadores são módulos do *kernel* Linux, ou seja, atuam no modo *kernel* de CPU (HORMAN, 2003). Para maior desempenho dos sistema, que é um dos focos do balanceamento de carga, a solução proposta foi implementada como um módulo do *kernel*, mas não completamente. Uma outra parte do sistema foi implementado como programas para fins de administração e de suporte para a solução, atuando no modo usuário de CPU.

O módulo do *kernel* é responsável por efetuar a manipulação dos pacotes de rede, já a aplicação de administração é formada por um serviço em *background* e uma interface com o usuário por linha de comando, onde por intermédio dela é possível configurar o módulo (maiores explicações nas seções de visões arquiteturais). A forma mais recomendada de manipulação destes pacotes de rede no modo *kernel* é fazendo uso do *framework* Netfilter.

c) Linguagens de programação

Como foi tomada a decisão de projeto que o balanceador seria implementado como um módulo do *kernel* para sistemas Linux, obrigatoriamente a linguagem de programação deverá ser C ou C++, que são as únicas linguagens utilizadas. O desenvolvimento de módulos usando C++ é possível, mas é desencorajada de acordo com o próprio Torvalds (2004), criador do Linux. Algumas das justificativas dadas são:

- O tratamento de exceções do C++ são falhos, principalmente em módulos do *kernel*;
- Qualquer tipo de compilador ou linguagem que abstrai coisas como a alocação de memória, não é uma boa opção para módulos do *kernel*, o que é o caso do C++ por ser uma linguagem nativamente orientada a objetos;
- Bibliotecas como STL e Boost do C++ trazem abstrações que podem dificultar a depuração de um módulo, além de trazerem falhas que podem ser catastróficas em um módulo do *kernel*;

[2] Lista de portas padrão por protocolo: <http://pt.wikipedia.org/wiki/Anexo:Lista_de_portas_de_protocolos>

- É possível simular a orientação a objetos na linguagem C, fazendo uso de *structs* e ponteiros para função, melhorando a modularidade do código sem trazer os malefícios citados anteriormente (mas perdendo os diversos benefícios da orientação a objetos formal).

Resumindo, o desenvolvimento em C é mais propício pelo programador ter maior controle das funções do sistema (principalmente nas alocações dinâmicas de memória), o que na teoria reduz os problemas com estabilidade do código, pois é necessário lembrar-se que uma falha em um programa que atua no modo *kernel* de CPU pode fazer todo o sistema operacional entrar em colapso. Por conta destes fatores o módulo foi desenvolvido usando a linguagem C.

Já o aplicativo com fins de administração do sistema não possuí essa necessidade eminente de ser desenvolvido em C, pelo fato de atuar modo usuário de CPU. Então foi ponderado, analisado e definido que ele utilizaria linguagem de programação Python para sua implementação, a seguir algumas justificativas:

- O interpretador Python vem nativamente instalado em grande parte das distribuições Linux (e.g. Ubuntu, Debian e Fedora), contendo várias funcionalidades da própria distribuição implementadas em Python;

- A linguagem Python é a 8ª mais utilizada do mundo, de acordo com dados de maio de 2014 (TIOBE SOFTWARE, 2014). Este sucesso se faz por conta de vários benefícios que a linguagem oferece: Possui uma sintaxe de fácil compreensão e aprendizado, tem estruturas de dados que facilitam o desenvolvimento (e.g. listas, tuplas e dicionários) e é uma linguagem de alto nível, interpretada, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte;

- Por conta da sua vasta comunidade de usuários, existem diversas bibliotecas e *frameworks* desenvolvidos e distribuídos sob licenças livres para os mais diversos domínios de conhecimento (e.g. física aplicada, jogos de computadores, sistemas distribuídos, computação gráfica, etc).

d) Retorno de tráfego

A técnica de retorno de tráfego via NAT é de longe a mais simples de ser implementada, por conta de não exigir qualquer configuração adicional nos servidores do sistema, tanto o balanceador quanto os servidores reais que irão desempenhar o processamento das requisições. Mas ela possuí uma grande desvantagem de necessitar que o as respostas dos servidores reais passem de volta pelo balanceador, o que o torna um gargalo, diminuindo o desempenho do sistema como um todo.

As técnicas de retorno de tráfego via DR e via tunelamento IP necessitam de um esforço de configuração adicional do balanceador e servidores reais como já foi descrito nas seções 3.2.2 e 3.2.3. A técnica via tunelamento oferece a capacidade o envio de carga de trabalho até para servidores reais que não se encontram na rede, diferentemente do DR que necessita que todos os *hosts* estejam na mesma LAN, mas comparando com a técnica via DR, esta oferece maior desempenho por não ter o *overhead* de desencapsulamento dos pacotes que o tunelamento oferece. A técnica via DR oferece uma maior complexidade de implementação, pois além do balanceador de carga ter que acessar os pacotes relacionados a sua camada de rede de atuação (e.g. Segmento TCP), ele ainda precisa acessar os quadros *ethernet* da LAN para direcionar a carga de trabalho ao *cluster* de servidores.

Por motivos de simplicidade, a técnica via NAT foi utilizada, pois mesmo tendo um menor desempenho relacionado as outras duas, ela elimina uma série de fatores que aumentam a complexidade do projeto, que poderiam eventualmente até arriscar a viabilidade do trabalho.

e) Algoritmos de escalonamento

Ficaram definidos três algoritmos a serem implementados, um da categoria de algoritmos de escalonamento estáticos e outros dois da categoria de algoritmos dinâmicos. O algoritmo estático a implementado foi o *round-robin*, que apesar de sua simplicidade tende a ter bons resultados em um *cluster* heterogêneo com aplicações de propósito específico (MADHURAVANI; SUMALATHA; SHANMUKHI, 2012). Os algoritmos dinâmicos escolhidos foram o “menor latência”, que consegue identificar servidores reais menos sobrecarregados pelo seu tempo de resposta dos seus serviços, e também o algoritmo de “menos conexões”, que determina o destino da requisição levando em consideração os servidores reais com menos conexões simultâneas em execução.

f) Forma de redundância

Parte do esforço deste trabalho se destinou ao projeto de integração do balanceador de cargas com o software fornecido pelo professor orientador deste trabalho, o Application Manager. Este software implementado na linguagem Java é um gerenciador de múltiplas aplicações em cenário ativo-*standby*, destinado a ambientes com múltiplos *hosts* distribuídos (LARANJEIRA, 2011). Por intermédio dele uma mesma aplicação (e.g. servidor *web*, servidor de arquivos, etc) pode possuir duas instâncias, uma ativa em um *host* de uma rede, a outra inativa (em *standby*) em outro *host* da mesma rede, e elas podem ser gerenciadas via um utilitário de linha de comando (amcli). Em caso de um colapso da aplicação ativa, a outra que estava inativa seria iniciada para exercer as mesmas funções da aplicação que falhou.

Por conta deste objetivo específico de pesquisa e desenvolvimento pré-definido, o único cenário de redundância utilizado neste trabalho foi o ativo-*standby*.

4.3 Visão lógica

A visão lógica é única para todo o sistema, ela visa ilustrar os subsistemas, módulos e tudo mais que possuir comportamento significativo no ponto de vista da arquitetura (IBM RATIONAL SOFTWARE, 2001). A Figura 9 a seguir demonstra um diagrama de pacotes[3] da UML (*Unified Modeling Language*) do balanceador de carga projetado.

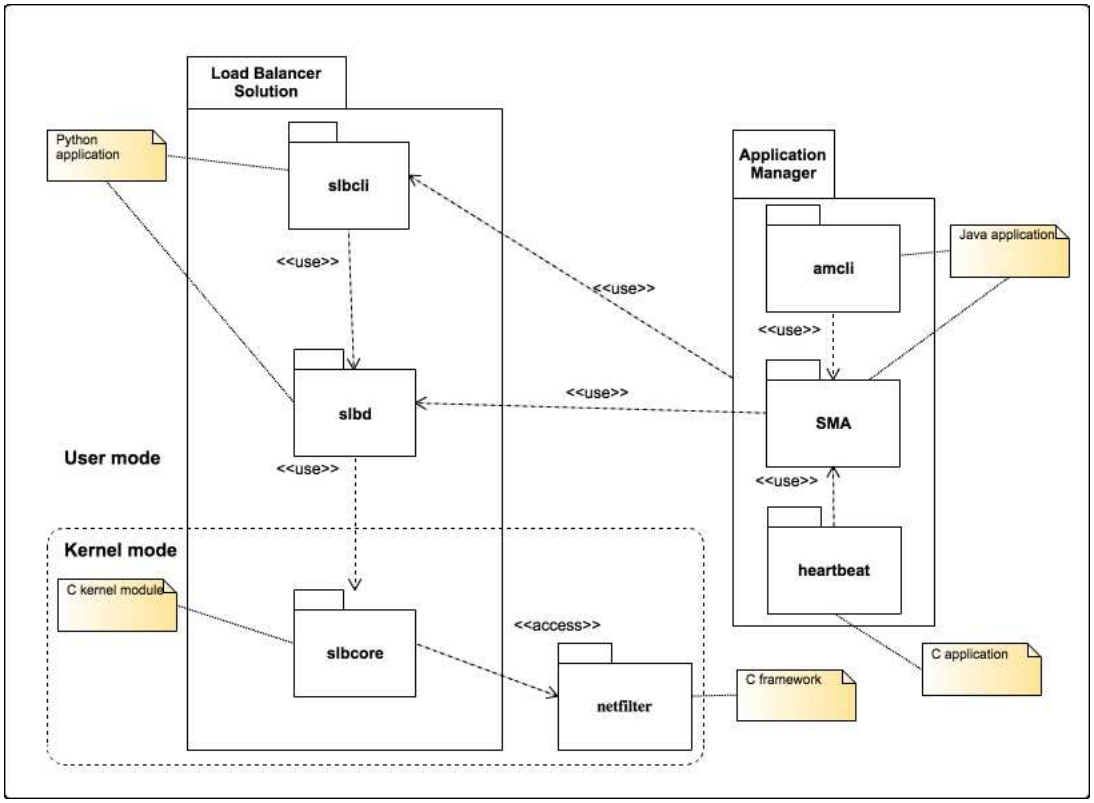

Figura 9: Visão lógica da solução de balanceamento de carga

A seguir será feita uma descrição de cada componente pertencente ao diagrama.

a) Load Balancer Solution

Este é o conjunto de programas que constituem o balanceador de cargas de servidores. Após sua instalação em um servidor Linux é possível efetuar a configuração e uso da solução de balanceamento de carga.

b) slbcli

O componente slbcli (*Server Load Balancer CLI*) é um programa do tipo CLI (*Command Line Interface*). Ele contém uma série de funcionalidades administrativas do serviço que poderão ser acessadas via o interpretador de comandos do Linux (Shell). Este programa é a interface do balanceador de cargas com o usuário administrador do sistema, onde é possível iniciar, parar e configurar o balanceador de carga.

[3] <http://pt.wikipedia.org/wiki/Diagrama_de_pacotes>

A seguir alguns dos comandos implementados (a lista completa é encontrada no apêndice A):

 # slbcli.py --load : Carrega o módulo do balanceador no *kernel* e inicializa as funções de balanceamento de carga.

 # slbcli.py --unload : Descarrega o módulo do balanceador no *kernel* e para as funções de balanceamento de carga.

 # slbcli.py --reload : Recarrega o módulo do balanceador no *kernel* e inicializa as funções de balanceamento de carga.

 # slbcli.py --with-redundancy : Flag que indica que o sistema deverá trabalhar em um modo redundante, com um servidor primário e um de *backup* inicialmente como ativo e standby, respectivamente.

 # slbcli.py --backup-slb : Define as informações para o servidor de *backup* no sistema de redundância, são passados como parâmetro as informações sobre o host e as credenciais de acesso ao sistema operacional via SSH. Utiliza a notação "ENDEREÇO,USUÁRIO,SENHA,PORTA_SSH" (e.g. "192.168.0.2,root,1234,22").

 # slbcli.py --status : Checa o status do balanceador.

 # slbcli.py --clear-config : Limpa a configuração em uso do balanceador de carga (não remove as definições do arquivo de configuração padrão).

 # slbcli.py --address : Define o endereço de rede IPv4 para recebimento de requisições dos clientes (VIP).

 # slbcli.py --address-dev : Define o nome da interface de rede para recebimento de requisições dos clientes (e.g. eth0).

 # slbcli.py --gateway : Define o endereço de rede IPv4 para distribuição de requisições aos servidores reais.

 # slbcli.py --gateway-dev : Define o nome da interface de rede para distribuição de requisições aos servidores reais (e.g. eth1).

 # slbcli.py --algorithm : Define o algoritmo de escalonamento (ROUND_ROBIN, LOWER_LATENCY ou LEAST_CONNECTIONS).

 # slbcli.py --cluster-network : Define o endereço de rede para o *cluster* de servidores reais utilizado utilizando a notação CIDR (e.g. 172.16.0.0/24). Esse parâmetro irá possibilitar a execução do NAT.

 # slbcli.py --add-server : Adiciona um serviço de um servidor real ao *cluster* utilizando a notação "ENDEREÇO:PORTA" (e.g. 172.16.0.1:80).

O símbolo "`#`" significa que o comando está sendo executado em um *shell*[4] pelo usuário *root* do sistema Linux. A lista de comandos provavelmente sofrerá alterações na futura implementação do balanceador pela mudança ou descoberta de novos requisitos. O slbcli é um programa implementado em Python, ele interage com o serviço de balanceamento de carga que estará executando em *background*, fazendo um repasse dos comandos para que este execute-os.

c) slbd

O componente slbd (*Server Load Balancer Daemon*) é um programa do tipo *daemon*, um tipo de processo que é executado em *brackground* e não possui interação direta com nenhum tipo de usuário. Esse processo é o responsável por ser a aplicação no modo usuário de CPU que interage com o módulo do *kernel*, tendo as funcionalidades de carregá-lo, descarregá-lo, configurá-lo, entre outras. Este componente poderia ter sido subdividido em outras partes, pois ele comporta funcionalidades de monitoramento do *cluster*, sincronização de configuração, gerenciamento da redundância, entre outros. Em relação ao monitoramento do *cluster*, este componente checa os servidores reais em tempo de execução. Ao identificar possíveis falhas nos servidores ou em seus serviços, ele remove o servidor/serviço das opções de balanceamento de forma temporária, checando periodicamente o seu possível retorno. O componente também tem a função de monitorar a latência dos serviços quando estiver sendo utilizado o algoritmo "menor latência", servindo como parâmetro para suas decisões de escalonamento.

d) slbcore

O componente slbcore (*Server Load Balancer Core*) é o núcleo de todo o sistema, é um LKM (*Loadable Kernel Module*), um código objeto carregável que irá estender as funcionalidades do *kernel* Linux. Este módulo irá fazer o balanceamento no modo NAT, que foi o tipo de retorno de tráfego escolhido para ser implementado na solução. Ele é responsável por interceptar os segmentos TCP e UDP que contém as requisições a serviços que os servidores reais irão desempenhar, e após decisão de qual será o servidor escolhido para processamento da requisição (por intermédio do algoritmo de escalonamento), este módulo repassará o segmento.

e) netfilter

É o componente que representa o *framework* Netfilter a ser utilizado pelo slbcore para acesso a funcionalidades de interceptação de pacotes de rede, fazendo a importação de suas devidas bibliotecas de forma privada (por isso o uso da associação com estereótipo "access" no diagrama UML).

i) Application Manager

Este é o conjunto de programas que constituem o software Application Manager,

[4] Iterpretador de comandos Unix: <http://en.wikipedia.org/wiki/Unix_shell>

um um gerenciador de múltiplas aplicações em cenário ativo-*standby*, destinado a ambientes com múltiplos *hosts* distribuídos. No contexto da solução que está sendo projetada, existirão duas instâncias do software, uma no balanceador de carga primário e outra no secundário. A Figura 10 ilustra a estrutura do sistema.

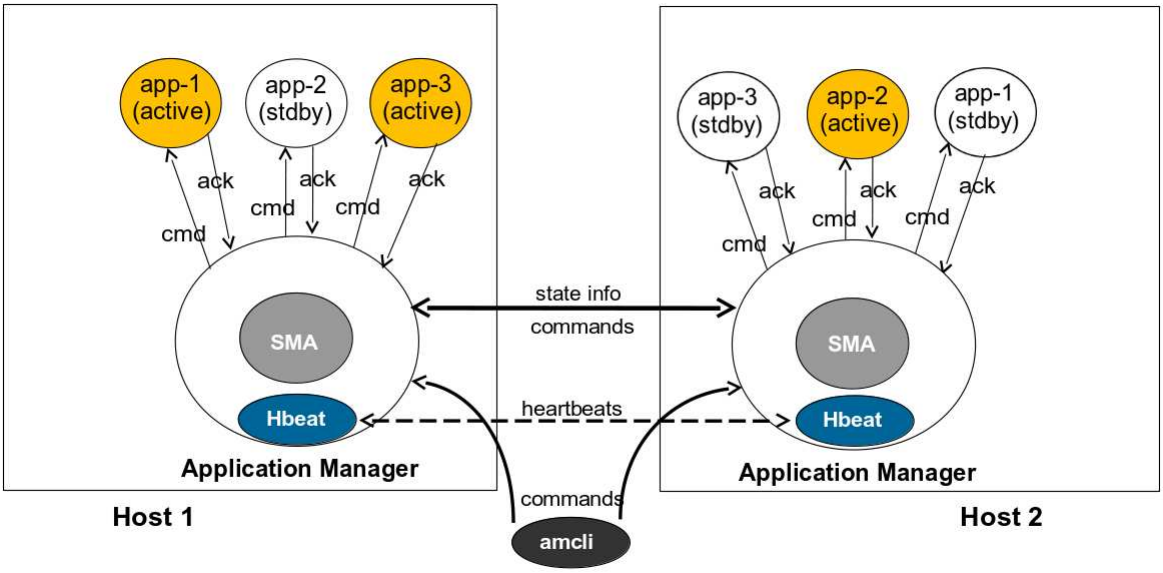

Figura 10: Estrutura do Application Manager

Fonte: Laranjeira (2011). Adaptado pelo autor.

Como pode se observar em cada *host* podem existir mais de uma aplicação sendo gerenciada (e.g. app-1, app-2, app-3), o que oferece a capacidade de serem gerenciados tanto a solução de balanceamento de carga, quanto o componente configuration-synchronizer, que são duas aplicações independentes. Mais detalhes sobre os subcomponentes do Application Manager serão dados a seguir.

j) SMA

O SMA é o subcomponente do Application Manager responsável por gerenciar o estado de cada aplicação, ou seja, para cada aplicação que está sendo gerenciada existirão duas instâncias do SMA que administram os seus estados, um para a aplicação ativa e outro para a aplicação em *standby*. Na Figura 11 é exemplificada a máquina de estados[5] que o SMA atribui a cada aplicação, e na Tabela 3 são descritos os eventos para a transição dos estados.

A forma que este componente gerencia o estado de uma aplicação é estabelecendo uma comunicação local com ela. Para isso a aplicação gerenciada deve implementar uma interface com o mecanismo de identificação das mensagens de troca de estados, e também um mecanismo para informar que está ativa.

k) amcli

[5] <http://pt.wikipedia.org/wiki/M%C3%A1quina_de_estados_finitos>

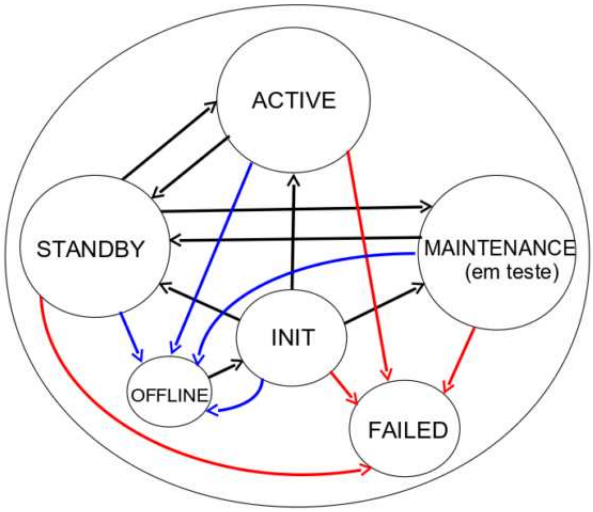

Figura 11: Máquina de estados do SMA

Fonte: Laranjeira (2011). Adaptado pelo autor.

Tabela 3: Eventos para troca de estados do SMA

Estado atual	Novo estado	Evento
OFFLINE	INIT	Aplicação iniciada
INIT	STANDBY	Aplicação inicializada, configuração BACKUP e aplicação dual em ACTIVE
INIT	ACTIVE	Aplicação inicializada, configuração PRIMARY e aplicação dual não em ACTIVE
INIT	MAINT.	Aplicação inicializada e configuração INHIBITED
STANDBY	MAINT.	Comando TEST
MAINT.	STANDBY	Comando UNTEST
STANDBY	ACTIVE	Falha do aplicação, dual ACTIVE ou comando SWITCH
Todos	FAILED	Falha da aplicação
Todos	OFFLINE	Comando SHUTDOWN ou SHUTDOWN-F

Fonte: Laranjeira (2011). Adaptado pelo autor.

O subcomponente amcli (*Application Manager CLI*), é um programa do tipo CLI. Este é o responsável por interagir com o administrador do sistema, aceitando uma grande gama de comandos para alterações nos estados das aplicações gerenciadas ou configurar o próprio Application Manager.Uma lista completa dos comando está no Apêndice B.

l) heartbeat

Programa do tipo *daemon* que possui uma instância por *host* do Application Manager. Eles implementam um protocolo de comunicação *heartbeat* para verificação das atividades do seu par, caso não consiga estabelecer a comunicação.

4.4 Visão de implantação

A visão de implantação tem a função de descrever a decomposição física do sistema, ficando visível o conjunto de hosts onde ocorrerá o processamento, inclusive a distribuição

dos componentes/pacotes de software que em um ambiente de execução se tornarão processos/threads nesses hosts (IBM RATIONAL SOFTWARE, 2001). A seguir na Figura 12 representa o diagrama de implantação[6] da UML para balanceador de carga projetado.

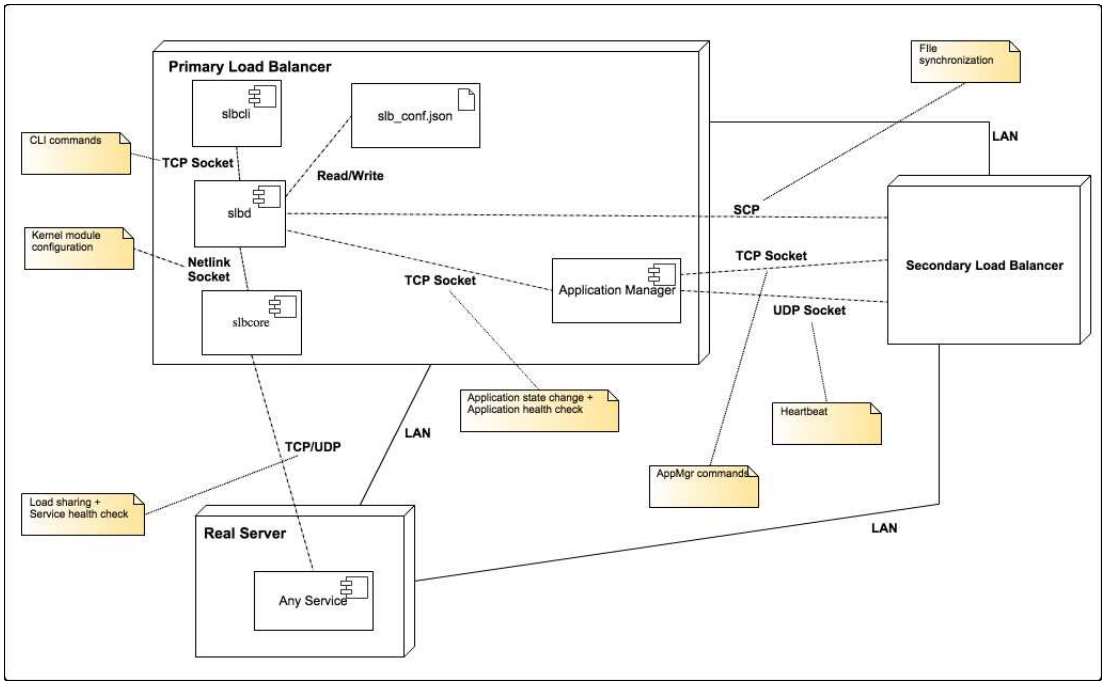

Figura 12: Visão de implantação da solução de balanceamento de carga

Neste diagrama estão dispostos os nós (forma geométrica de paralelepípedo), os componentes de software (retângulos internos aos nós) e suas associações. A seguir uma descrição de cada nó:

Primary Load Balancer: Balanceador de carga primário da solução;

Real Server: Servidor real. No diagrama foi colocado como um único nó, mas na solução real existirão n nós;

Secondary Load Balancer: Balanceador de carga secundário ou de *backup*. Seus componentes internos não foram ilustrados por possuírem a mesma disposição dos componentes do Primary Load Balancer.

Grande parte dos componentes já foram descritos anteriormente no diagrama de pacotes da visão lógica do sistema, com exceção do artefato slb_conf.json que representa o arquivo de configuração do balanceador de carga, e o componente Any Service que faz a representação de qualquer tipo de serviço que um servidor real pode prover (e.g. servidor *web*, servidor de e-mail, etc).

[6] <http://pt.wikipedia.org/wiki/Diagrama_de_instala%C3%A7%C3%A3o>

As associações ilustradas no diagrama são categorizadas em dois tipos, associações físicas e associações lógicas. As associações físicas associam dois nós do sistema, ou seja, são os meios físicos de rede (e.g. cabo de rede *ethernet*, rede sem fio, etc), representados no diagrama por linhas contínuas. As associações lógicas normalmente associam um componente a outro componente, seja ele local ou remoto, podendo até ser utilizado para associar um componente a um nó. As associações lógicas são protocolos, ou até mesmo programas completos, e foram representadas no diagrama por linhas tracejadas. A seguir a descrição das associações:

LAN: Conexão física entre os *hosts* em uma rede local. Serão um conjunto de *switches* de rede conectando os *hosts* com cabos par trançado cat 5e para redes *ethernet*.

TCP/UDP: Como a camada de rede de atuação do balanceador de carga é a camada 4, no ponto de vista do sistema serão transmitidos e recebidos segmentos TCP e/ou UDP entre o balanceador de carga e os servidores reais. Também representa a estratégia com TCP SYN / SYNACK para checagem da latência de serviços baseados no protocolo TCP.

SCP: A associação entre o configuration-synchronizer e o Secondary Load Balancer representa o protocolo SCP (*Secure Copy Protocol*), que será o responsável pela transferência segura do arquivo do Primary Load Balancer para o Secondary Load Balancer.

TCP Socket: Essa forma de comunicação via a API de comunicação TCP Socket[7] ocorre em três pontos do sistema. (*i*) A associação entre o Application Manager e o slbd representa uma comunicação interna no servidor primário e secundário, fazendo o envio de alterações de estados do serviço de balanceamento de carga (e.g. ativo, em *standby*, em manutenção), e também a checagem constante da execução processo slbd, para verificar se está em execução ou parou mediante alguma falha. (*ii*) A associação entre as duas instâncias do Application Manager, uma no Primary Load Balancer e outra no Secondary Load Balancer, onde são enviados e processados comandos que normalmente são solicitados pelo administrador do sistema ou enviados automaticamente ao ocorrer um processo de *failover*, sendo necessário iniciar o slbd no Secondary Load Balancer. (*iii*) A associação entre o slbcli e o slbd representa uma comunicação interna do servidor, onde são repassadas comandos inseridos por um usuário para o *daemon* slbd, já que o mesmo não possui interação direta com o usuário.

[7] Um *socket* é uma API (*Application Programming Interface*) oriunda de sistemas BSD Unix para comunicação entre aplicativos sobre a pilha TCP/IP, normalmente é baseado em TCP (*Stream*) ou UDP (Datagrama).

UDP Socket: Essa comunicação ocorre entre os componentes heartbeat do Application Manager no Primary Load Balancer e no Secondary Load Balancer, sendo uma implementação de uma comunicação *heartbeat*.

Netlink Socket: Essa associação entre o slbd e o slbcore significa a forma de comunicação entre o programa no modo usuário de CPU e o módulo do *kernel* em modo *kernel* de CPU. Um *netlink socket* [8] é uma família de sockets Linux utilizada para IPC (*Inter-Process Communication*, i.e. Comunicação inter-processos) em modos de CPU *kernel* e usuário, servindo como meio de comunicação entre aplicações nos dois modos. Maiores explicações no capítulo 5.

Read/Write: A associação entre o slbd e o slb_conf.json é de leitura e escrita de arquivos. Isso ocorre porque o slbd irá ser o responsável por atualizar este arquivo de configuração sempre que o slbcli indicar novos comandos de configuração (inseridos via linha de comando pelo usuário), e ele também faz a leitura do arquivo sempre que o módulo do *kernel* for carregado ou recarregado, para passagem dessa configuração via netlink. O motivo pelo qual o módulo do *kernel* não realiza acesso direto a esse arquivo será descrito no capítulo 5.

[8] netlink - communication between kernel and user space (AF_NETLINK): <http://man7.org/linux/man-pages/man7/netlink.7.html>

5 Detalhes técnicos da solução

O último capítulo ofereceu uma abordagem arquitetural sobre o balanceador de cargas que foi implementado, fornecendo uma visão geral do comportamento do sistema para que seja possível observar suas principais funcionalidades e características. Propositalmente a arquitetura de software não tende a entrar em pequenos detalhes da solução, já que ela pretende fornecer uma visão alto nível do sistema e não um projeto detalhado.

Este capítulo tem uma abordagem diferente, ele visa tanto demonstrar como foi a implementação do software quanto oferecer um certo detalhamento sobre alguns dos componentes do sistema. Esse detalhamento é necessário para um esclarecimento de forma mais técnica de como foram implementadas determinadas características, ou como foram solucionados determinados problemas que foram evidenciados. Para viabilidade da descrição destes detalhes, serão descritos somente alguns dos considerados mais importantes, porque seria inviável detalhar totalmente todas as características de um sistema.

5.1 Organização do código-fonte do projeto

O código fonte do projeto pode ser encontrado no repositório online do GitHub [1], via o link: <https://github.com/matheus-fonseca/slb>.

A figura 13 demonstra a estruturação do projeto a nível de diretórios e arquivos via a saída do programa Linux `tree` [2], quando se achou necessário foi colocado ao lado um comentário para descrever o item (após o carácter "#").

5.2 Bibliotecas e códigos-fonte de terceiros

Nem todos os códigos dispostos foram implementados pelo autor do presente TCC, alguns destes foram reutilizados como bibliotecas de terceiros, todas essas estão disponíveis na internet sob licenciamento livre para reuso. A seguir a descrição dos fontes reutilizadas no projeto:

checksum: Um conjunto de funções em C para recalcular o *checksum* dos pacotes IP e segmentos TCP/UDP que são modificados no slbcore após escalonamento da requisição (e.g. mudança de endereço destinatário). Caso não fosse realizado esse recálculo então os servidores reais descartariam as requisições, pois é o comportamento padrão

[1] GitHub é um serviço compartilhado de *hosting* para artefatos (i.e. arquivos em geral) que fazem uso do sistema de controle de versão Git

[2] tree - lista o conteúdo de diretórios no formato de árvore. Fonte: <http://linux.die.net/man/1/tree>

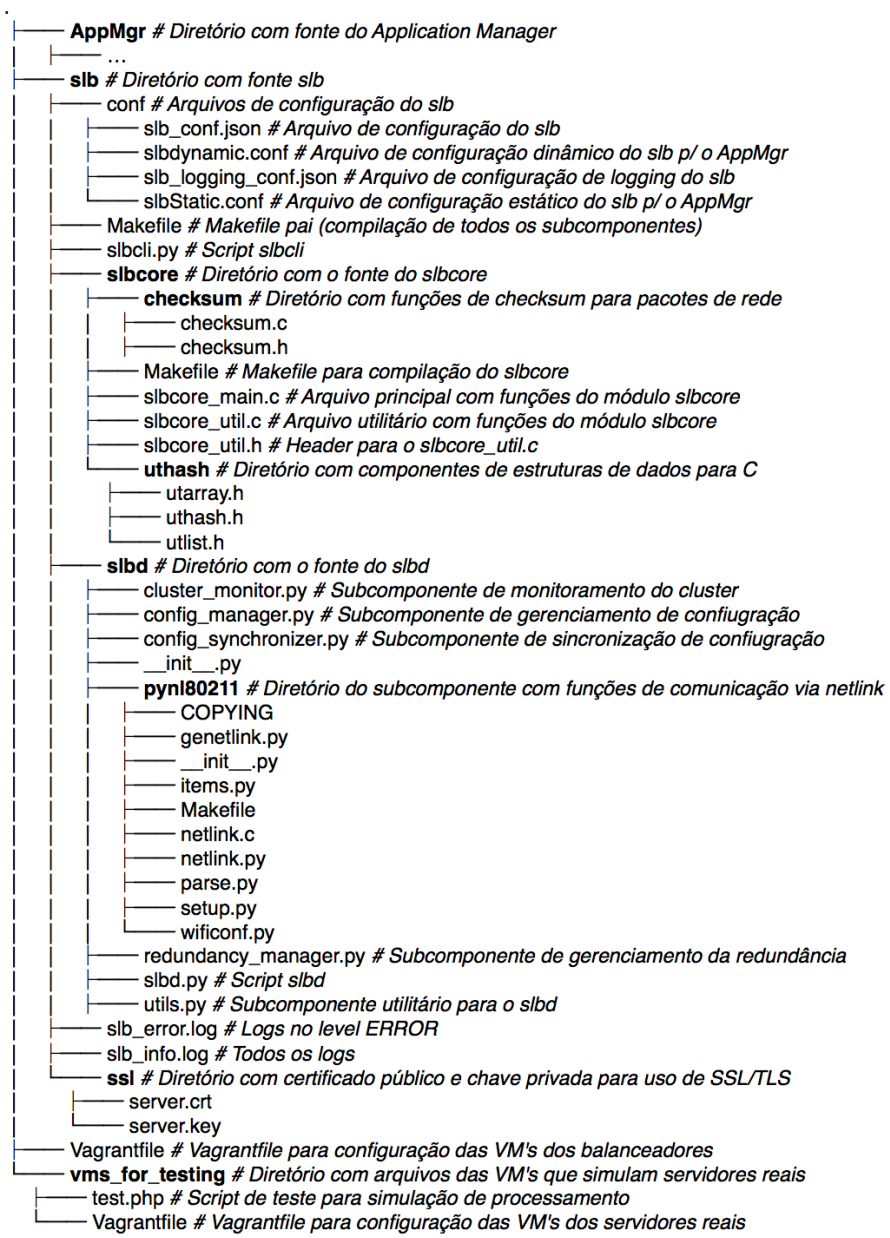

Figura 13: Estruturação do projeto slb a nível de diretórios e arquivos

dos softwares de rede ao verificarem um *checksum* errado. Este código foi desenvolvido pelo desenvolvedor Liu Feipeng <liuf0005@gmail.com> e está disponível no website: <http://www.roman10.net>. Os devidos créditos foram dados nos fontes.

uthash: A biblioteca padrão C disponibiliza poucas estruturas de dados e seus algoritmos por padrão, então para trabalhar com essas técnicas ou seria necessário implementar estas funcionalidades do zero, ou então reutilizar alguma de biblioteca para tal. O uthash é um conjunto de macros [3] e estruturas de dados que preza pela simplicidade e desempenho, exatamente por este ter sido implementado

[3] Uma macro é um fragmento de código que possui um nome, onde este fragmento for utilizado ele é substituído pelo conteúdo da macro. Fonte: <https://gcc.gnu.org/onlinedocs/cpp/Macros.html>

usando macro-substituições e não funções, como é o cumum. A biblioteca disponibiliza *hashes*, *arrays* e listas (ligadas, duplamente ligadas e circulares) que foram utilizadas no projeto pelo slbcore para compor a tabela de serviços e outras estruturas auxiliares, demais detalhes melhor descritos na subseção 5.7. Este conjunto de macros e estruturas de dados foram desenvolvidas pelo desenvolvedor Troy D. Hanson <tdh@tkhanson.net> e estão disponíveis no website: <http://troydhanson.github.io/uthash>, registrados sob licença permissiva pelo próprio autor.

pynl80211: É um projeto em Python desenvolvido para interação com os serviços de Wi-Fi (definido no padrão IEEE 802.11) para sistemas Linux. Internamente este programa utiliza a comunicação com as funções do subsistema de *networking* do *kernel* via um *socket* Netlink, e como já foi citado anteriormente esta é a técnica que foi utilizada no projeto para comunicação entre os componentes slbd e slbcore (mais detalhes na subseção 5.6). Por este programa já conter classes e métodos Python que fornecem uma API alto nível para realizar a comunicação utilizando Netlink, então os módulos Python que provêem esta funcionalidade foram importados e reutilizados no slbd. O projeto é um software livre e pode ser encontrado no seguinte website: <http://git.sipsolutions.net/?p=pynl80211.git;a=summary>, se encontra sob licença GPLv2 (GNU Gerneral Public License version 2) [4].

5.3 Cenário para simulação e testes

Em primeiro lugar para conseguir simular um cenário real de balanceamento de carga de servidores seriam necessários recursos financeiros para tal, como este trabalho tem objetivo simplesmente acadêmico então foi usado um ambiente de simulação a partir de ferramentas livres e do harwdare à disposição do autor.

No trabalho foram utilizados três notebooks e equipamentos de rede. A seguir a configuração de cada um com sua respectiva função desempenhada no trabalho:

Notebook Macbook Pro 15" Early 2011 :

CPU: Intel Core i7 2.2GHZ

Disco: 120GB SSD

RAM: 8GB

SO: Mac OS X Mavericks

Função: Uma máquina virtual para o balanceador de carga primário e outra para o de *backup*

[4] Texto da GPLv2: <http://www.gnu.org/licenses/old-licenses/gpl-2.0.html>

Notebook Macbook Pro 13" Retina Late 2013 :

CPU: Intel Core i5 2.6GHZ

Disco: 500GB SSD

RAM: 8GB

SO: Mac OS X Mavericks

Função: Oito máquinas virtuais para simular servidores reais

Notebook HP Pavilion DV7-6C00 :

CPU: Intel Core i7 2.2GHZ

Disco: 750GB HD

RAM: 8GB

SO: Fedora 20 "Heisenbug"

Função: Seis máquinas virtuais para simular servidores reais. Uma instância da ferramenta de simulação de tráfego de requisições por clientes (JMeter)

Switch de rede D-Link DES-1008A :

Descrição: Switch Fast Ethernet 10/100 Mbps 8 portas

Função: Criar um enlace de rede entre os três notebooks

Cabos de rede Ethernet :

Descrição: Cabeamento par trançado UTP (Par Trançado sem Blindagem) Cat. 5

Função: Conexão entre os notebooks e o switch

Roteador Wi-Fi :

Descrição: Modem e roteador Wi-Fi ARRIS TG862

Função: Simulação de um roteador de borda conectado aos balanceadores de carga, oferencendo o IP fixo ao sistema de balanceamento (simulando um VIP).

Todas as máquinas virtuais operaram com o sistema operacional Debian 7, por ser uma das distribuições Linux mais antigas e mais famosas para uso em servidores, amplamente conhecida pela sua grande estabilidade (o que é importante para ambientes de produção de sistemas). Além disso também foi padronizado a quantidade de CPU's, memória e disco de cada máquina virtual: 1 CPU, 512 de RAM e 3GB de HD.

A Figura 14 ilustra o projeto simples da rede simulada (máquinas virtuais foram representadas como hosts reais), baseado em uma configuração descrita por Kopparapu

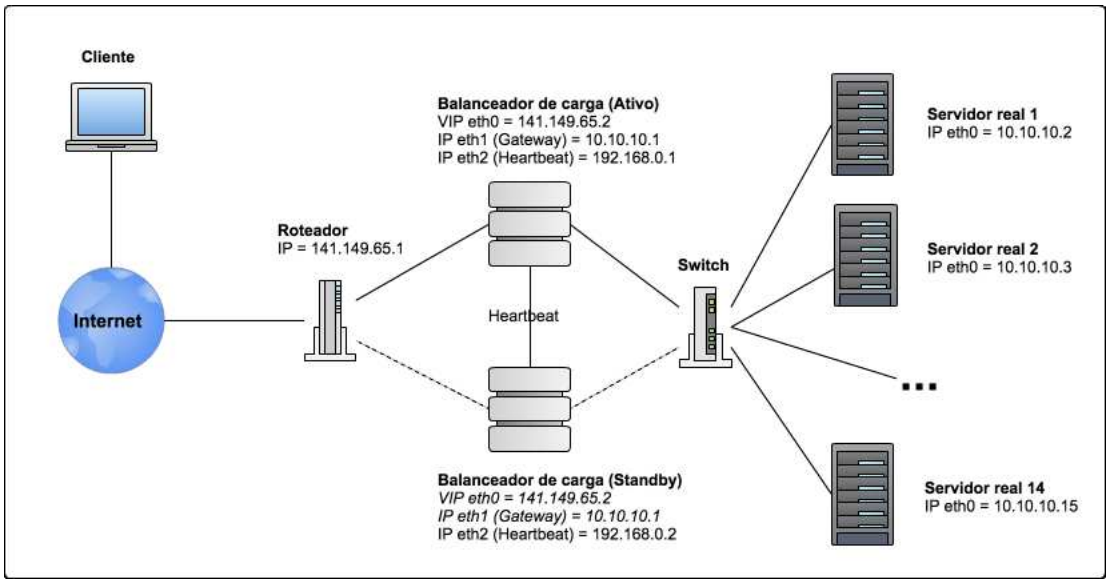

Figura 14: Projeto da rede de computadores da solução

(2002, p. 90). Neste cenário existe um roteador com IP fixo, onde ele poderia ter uma WAN (*Wide Area Network*) configurada diretamente com sua ISP (*Internet Service Provider*), isso possibilitaria que demandas da internet cheguem na rede configurada. Conectados ao roteador estarão as duas instâncias de balanceadores de carga, uma ativa e outra em *standby*. O balanceador ativo tem um VIP fixo, configurado em uma de suas interfaces de rede (e.g. eth0 em um servidor Linux), ele também possui um IP privado e único na rede configurado em outra interface (e.g. eth1) para comunicar-se entre os servidores reais e servir como *gateway* dos mesmos, também é necessário um terceiro IP privado para uso do *heartbeat* entre os balanceadores (e.g. eth2). Na teoria, uma forma mais adequada de configuração do *heartbeat* para detecção de falhas era também aplicar uma redundância nessa comunicação, podendo se utilizar, por exemplo, uma comunicação serial entre os dois balanceadores como cabo de *heartbeat* redundante. Essa redundância iria evitar que no caso de uma falha exclusiva do cabo, os balanceadores interpretassem esse evento como uma falha do seu par e então os dois entrassem no estado "ativo", mas esse comportamento não foi previamente implementado no móduolo *heartbeat* do Application Manager.

O balanceador em *standby* é configurado da mesma forma que o ativo, com exceção que seu VIP e o IP para comunicar-se entre os servidores (que terão os mesmos endereços do balanceador ativo) somente serão configurados em um processo de *failover*, quando o balanceador ativo falhar. Enquanto esse processo não ocorrer, estas interfaces de rede ficarão sem IP.

Para possibilitar a comunicação entre o balanceador e os servidores existe um *switch* da camada 2 que estará conectado aos balanceadores e os servidores reais. Neste cenário existem algumas deficiências, como os gargalos que podem se tornar o roteador e o *switch*, pois uma falha em algum dos dois dispositivos tiraria o serviço do ar. Para

resolução desse problema seria necessário uma solução com múltiplos *switches* e roteadores, aumentando a redundância da rede, mas este cenário não foi planejado por ser muito custoso, o que inviabiliza a sua realização no contexto deste trabalho.

Outro aspecto importante a se levar em consideração é que a rota padrão para os servidores reais deverá ser o IP do balanceador de carga ativo.

5.3.1 Ferramentas para virtualização

O Virtualbox [5] foi a ferramenta escolhida para virtualização dos ambientes, por ser uma ferramenta livre e de fácil configuração. Mas criar, configurar e manter todas as máquinas virtuais (16 no total) ainda sim se tornaria um trabalho árduo, então foi utilizada a ferramenta Vagrant para facilitar o processo de desenvolvimento. O Vagrant é recorrentemente chamado de "wrapper" (i.e. invólucro) para sistemas de virtualização e sistemas de gerência de configuração, ou seja, ele facilita a manipulação de máquinas virtuais (e.g. Virtualbox, VMWare, Parallels, etc) e de gerenciadores de configuração (e.g. Puppet, Chef, etc).

A partir de um arquivo chamado `Vagrantfile` é possível definir os parâmetros para criação da VM ao Virtualbox, além do que executar todas as configurações possíveis na máquina via o seu provisionador (no caso foram utilizados simples comandos em *shell script* [6] para configurar as máquinas). Como foi mostrado na figura 13, existem dois `Vagrantfile`'s no projeto, um para os balanceadores de carga e outro para os servidores reais. Segue no quadro 1 uma parte do `Vagrantfile` de configuração dos balanceadores de carga, a sintaxe do arquivo é baseado na linguagem de programação Ruby [7], aumentando a flexibilidade da sua configuração (possibilita o uso de cláusulas condicionais, cláusulas de repetição, etc).

Com o arquivo devidamente configurado é possível criar a máquina virtual e toda a sua configuração com um simples comando `vagrant up` em um terminal Linux com o Vagrant instalado e com o `Vagrantfile` presente no diretório. Para acessar a máquina virtual via SSH [8] (*Secure Shell*) basta um comando `vagrant ssh`.

5.4 Monitoramento do *cluster*

O monitoramento do *cluster* de servidores é necessário por dois motivos, (*i*) verificar se cada servidor real está em seu normal funcionamento, para que não sejam enviadas

[5] Virtualbox: <https://www.virtualbox.org/>
[6] *Shell script* é uma linguagem de *scripting* executado pelos interpretadores de comandos Linux, o mais conhecido e utilizado é o BASH (Bourne Again SHell)
[7] Ruby: <https://www.ruby-lang.org/en/>
[8] O protocolo SSH provê a estrutura para conexão remota entre hosts: <http://pt.wikipedia.org/wiki/SSH>

Quadro 1: Arquivo de confiugração Vagrantfile

```
1   Vagrant.configure(2) do |config|
2       # Specific configuration
3       config.vm.define "slb_primary" do |slb|
4           slb.vm.network "public_network", :adapter=>2,  :bridge => '
                en4: USB Ethernet', :ip => '172.16.0.100'
5           slb.vm.network "public_network", :adapter=>3,  :bridge => '
                en0: Wi-Fi (AirPort)'
6           slb.vm.hostname = "slb-primary-vm-debian-7"
7
8           slb.vm.provider "virtualbox" do |vb|
9               vb.name = "slb-primary-vm-debian-7"
10          end
11
12          config.vm.provision "shell", inline:
13          "
14              echo '----- CONFIGURING SLB HOSTNAMES:';
15              if ! grep -q '# slb' /etc/hosts; then
16                  echo -e '\n# slb backup host \n172.16.0.101 slb-
                        backup-vm-debian-7' >> /etc/hosts;
17              fi
18              if ! grep -q '\.local' /etc/hosts; then
19                  sed -i.bak 's/slb-primary-vm-debian-7/slb-primary-vm
                        -debian-7 slb-primary-vm-debian-7.local/' /etc/
                        hosts;
20              fi
21          "
22      end
23
24      # OTHER CONFIGURATIONS OMITTED ...
25  end
```

requisições a servidores inoperantes, e (*ii*) verificar a latência dos serviços providos pelos servidores reais, fornecendo parâmetros para a decisão do algoritmo de "menor latência".

Para verificar se um servidor está operante, em teoria pode ser utilizado o protocolo ICMP, destinado principalmente para troca de mensagens de controle, o protocolo trabalha juntamente com o IP e seus pacotes são inseridos dentro de pacotes IP para envio de relatórios sobre a comunicação (BENVENUTI, 2005, p. 585). Mas realizar uma verificação específica do estado do servidor real não faz tanto sentido já que também será necessária a verificação dos serviços, caso todos os serviços de um servidor estejam em um estado *offline*, então se pode-se deduzir que o servidor também se encontra em um estado *offline*, sendo necessário então somente a verificação da latência dos serviços para também verificar o estado de servidores/serviços.

Para verificar a velocidade de resposta de um serviço, pode ser utilizado parte do *handshake* de três vias que ocorre no estabelecimento de uma conexão TCP. Pode ser criar um *raw packet* de um segmento TCP com a *flag* do tipo SYN ativada (bit = "1"), e então enviá-lo ao servidor e esperar como resposta um TCP segmento TCP com as flags

SYN e ACK ativadas, a diferença de tempo entre esses dois eventos indica a latência do serviço (KOPPARAPU, 2002, p. 30). Caso o serviço seja baseado em UDP, por exemplo, essa técnica não surtirá efeito já que o protocolo não possui segmentos de reconhecimento (ACK).

Como foi dito anteriormente este módulo foi construído usando a linguagem Python (é uma parte do componente slbd), de forma a facilitar o desenvolvimento deste monitoramento foi utilizada um técnica mais alto nível e simples que as anteriormente citadas. A verificação dos serviços é feita utilizando o módulo `socket` [9], este provê uma API com classes e funções de acesso a interface BSD socket. O quadro 2 contém um método que realiza essa operação, é recebido como parâmetro o endereço do servidor e a porta do serviço e é retornada uma tupla onde o primeiro elemento indica um valor booleano de sucesso na checagem (`True` = sucesso, `False` = falha) e o segundo elemento a latência em milisegundos.

Quadro 2: Código fonte Python: Checagem da latência de um serviço

```
1
2  def service_latency(self, address, port):
3      # Cria um socket STREAM (TCP)
4      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5      # Define o valor de timeout da conexao de acordo com o valor
           definido no atributo timeout
6      sock.settimeout(self.timeout/1000)
7      elapsed_time = 0
8      try:
9          # Realiza uma simples conexao e eh verificado o tempo de
               execucao
10         start_time = time.time()
11         result = sock.connect_ex((address, port))
12         elapsed_time = time.time() - start_time
13     except Exception, e:
14         result = -1
15         logger.error('Problem␣checking␣service␣('+ address + ':' + str(
               port) +'):␣' + str(e))
16     finally:
17         sock.close()
18
19     # Em caso de sucesso retorna (True, 'latencia do servico')
20     if result == 0:
21         return (True, elapsed_time)
22     # Em caso de erro retorna (False, 0)
23     else:
24         return (False, 0)
```

Como pode ser verificado no quadro 2 este tipo de checagem só abrange serviços TCP, ou seja, no atual estado do sistema não é possível verificar a latência de um serviço UDP, ficando como uma tarefa futura de implementação.

[9] socket — Low-level networking interface: <https://docs.python.org/2/library/socket.html>

5.5 Processo de *failover*

Em um cenário ativo-*standby*, caso o sistema ativo venha a falhar, algumas providências serão tomadas reativamente a situação (logo após a falha) e algumas providências já devem estar sendo tomadas preventivamente (anteriormente a falha). Aqui será explicado sucintamente esse processo.

O o módulo Python `config_synchronizer.py` (subcomponente do slbd) atua preventivamente, pois sincroniza continuamente a configuração do balanceador, para que o nó em *standby* tenha o mesmo comportamento do ativo quando ocorrer um *failover*. A seguir as tarefas realizadas pelo componente:

1. O arquivo de configuração da solução de balanceamento de carga ficará localizado em uma pasta específica do sistema (e.g. `/etc/slb/slb_conf.json`).

2. Periodicamente (e.g. de 10 em 10 segundos) essa arquivo será transferido via o protocolo SCP (*Secure Copy Protocol*), que é um meio de transferência de arquivos entre dois hosts remotos de forma segura, fazendo uso de criptografia, este protocolo é baseado em outro, o SSH. Para realizar essa tarefa será necessária uma biblioteca Python (linguagem do componente em questão) para comunicação via o protocolo SCP. O código no Quadro 3 exemplifica realização da transferência usando as bibliotecas paramiko e scp[10] para Python.

Quadro 3: Código fonte Python: Envio de arquivo via SCP

```
1  import scp
2  import paramiko
3
4  # Cria o cliente SSH
5  def createSSHClient(server, port, user, password):
6      client = paramiko.SSHClient()
7      client.load_system_host_keys()
8      client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
9      client.connect(server, port, user, password)
10     return client
11
12 # Conecta com o servidor
13 ssh_cliente = createSSHClient("10.10.10.2", 22, "root", "passwd")
14 scp_cliente = scp.SCPClient(ssh_cliente.get_transport())
15 # Transfere o arquivo
16 scp_cliente.put(files=["/etc/slb/slb\_conf.json"],remote_path="/etc/slb/
       slb\_conf.json",recursive=False, preserve_times=False)
```

3. Quando o *failover* ocorrer e for iniciado o slbd no novo balanceador ativo, o arquivo de configuração será o mais recente possível e o serviço continuará com as mesmas características do antigo balanceador ativo.

[10] <https://pypi.python.org/pypi/scp>

As duas instâncias do Application Manager no balanceador ativo e *standby* atuam de forma reativa, estão em constante troca de *heatrbeats* via um cabo de rede exclusivo, mas além disso também podem ser trocados comandos administrativos (e.g. efetuar um uma troca de estados, primário vira em *standby* e secundário vira ativo). Os *heartbeats* são trocados via uma comunicação UDP. Caso seja percebida uma falha no servidor primário, de forma reativa serão realizadas as seguintes tarefas:

1. A instância do componente heartbeat no servidor secundário ao perceber que o seu par não está mais se comunicando, se comunica com a instância do Application Manager informando a falha do outro servidor.

2. O Application Manager se comunica via seu *socket* TCP com a interface da aplicação que ele está gerenciando, no caso o slbd, enviando um comando para que ele se torne a instância ativa.

3. O slbd ao receber o comando, configura duas interfaces de rede (e.g. eth0 e eth1), uma com o VIP do serviço de balanceamento de carga e outra com o IP para comunicação interna com os servidores reais e para servir como *gateway* da rede interna. Logo após o slbd carrega o módulo do *kernel* (slbcore) e o configura de acordo com o conteúdo do arquivo de configuração slb_conf.json.

É importante ressaltar que esse processo de *failover* descrito é considerado *stateless* e não *stateful*. Um *failover stateless* não define que a tabela de conexões que regia na instância ativa (agora com falha) será passada e usada pelo servidor em *standby* (agora se tornando ativo), ou seja, todas as conexões TCP ativas que existiam entre clientes e servidores reais serão perdidas, já em um processo de *failover stateful* essa tabela de conexões também será compartilhada (KOPPARAPU, 2002, p. 96). A sincronização da configuração que foi projetada e implementada garante apenas que o novo balanceador de carga ativo saberá quais são os servidores reais que ele redistribui a carga, quais portas poderá atender, qual algoritmo será utilizado, etc.

5.6 Comunicação entre slbd e slbcore

Em alguns momentos na descrição deste trabalho foi falada sobre o modo de configuração do slbcore, foi dito que ela seria feita através de uma interação via Netlink com o slbd, este *daemon* iria realizar a leitura da configuração ou iria receber um comando do slbcli, e logo após iria se comunicar com o módulo do *kernel* para configurá-lo. Mas ainda não foi explicado o motivo dessa decisão arquitetural, qualquer um poderia se questionar do porquê do slbcore não ler diretamente o arquivo slb_conf.json, ou do porquê que o slbcli não se comunica diretamente com ele. A seguir as explicações para estes fatos.

De acordo com Kroah-Hartman (2005), a leitura de arquivos dentro de módulos do *kernel* é uma má ideia pois os problemas que isso pode causar são devastadores dentro de um módulo. Em primeiro lugar existe o problema da interpretação dos dados (*parsing*[11]), uma simples interpretação errada pode abrir espaços para *buffer overflows*, que significa ultrapassar os limites de escrita de um *buffer*[12], acessando memória indevida e oferecendo uma abertura para execução de instruções mal intencionadas (LEVY, 1996). Ou seja, usuários mal intencionados podem inserir instruções que fazem acesso ou manipulação de arquivos protegidos. Outro problema é a padronização da estrutura de arquivos da distribuição Linux em questão, dependendo da distribuição em que se encontra o módulo do *kernel* o local de arquivos de configuração pode ser diferente, oferecendo uma complexidade em se encontrar o arquivo e abrindo espaço para mais problemas. Deve se lembrar que erros dentro de módulos do *kernel* são devastadores, pois nesse modo de CPU o sistema fica quase irrestrito a nível de acesso de instruções e áreas de memória.

Para evitar esse problema de acesso direto a arquivos de configuração o ideal é a configuração via uma aplicação no modo usuário. Para estabelecer essa comunicação existem diversas formas, podem ser utilizados os sistemas de arquivo padrão Linux para informações de processos ou para informações de configuração, /proc e /sys respectivamente. Também pode ser usado o IOCTL, uma função que define uma forma de entrada e saída de dados destinados a um *device file*[13], onde poderão ser enviados qualquer tipo de comando com parâmetros associados (SALZMAN; BURIAN; POMERANTZ, 2007, p. 2). Outra forma pode ser a utilização da família de *sockets* Netlink, especialmente desenvolvida para comunicação entre processos nos dois modos de CPU, estritamente para uso interno (no mesmo *host*) . O Netlink foi escolhido pelo fato de ter sido desenvolvido especificamente para este objetivo, provendo mecanismos otimizados para esta função de comunicação entre processos nos dois modos. O Quadro 4 contém partes do código da implementação da comunicação no slbcore. Já o Quadro 5 contém partes de código da implementação do slbd.

A função contida no quadro 4 foi previamente registrada no momento em que o módulo do *kernel* é carregado (via a função `netlink_kernel_create(...)`), sendo uma função de *callback* que é chamada toda vez em que é recebida uma mensagem. Essa função é chamada via uma funcionalidade chamada de interrupção (BOVET; CESATI, 2000, p. 96):

> Uma interrupção é normalmente definida como um evento que altera a sequência de instruções executada por um processador. Estes eventos

[11] Análise sintática: <http://pt.wikipedia.org/wiki/An%C3%A1lise_sint%C3%A1tica_(computa%C3%A7%C3%A3o)>

[12] <http://pt.wikipedia.org/wiki/Buffer_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o)>

[13] Um arquivo que normalmente representa um dispositivo físico, mas qualquer módulo do *kernel* pode registrar um.

> correspondem a sinais elétricos gerados por circuitos de hardware de dentro e de fora do chip da CPU.

Além da interrupção de hardware (que foi explicada acima), também existe uma interrupção por software, onde a fonte da interrupção é originada via software (como o próprio nome diz), o que é o caso do Netlink.

A estrutura recebida como parâmetro "`sk_buff`" é uma das mais importantes estruturas de dados quando se trabalha com redes no *kernel* do Linux, a seguir a explicação desta de acordo com Benvenuti (2005):

> `sk_buff`: Aqui é onde o pacote de rede é armazenado. Esta estrutura é usada em todas as camadas de rede para armazenar seus cabeçalhos, informações sobre os dados de usuário (o *payload*), e outras informações internamente necessárias para coordenar seus trabalhos.

Nesta função C é feito um *parsing* da mensagem enviada pelo slbd, onde são extraídos o nome do comando a ser executado e seus parâmetros (dados que estavam contidos no `sk_buff`).

Já no método Python do slbd foram reutilizadas as classes encontradas no pacote pynl80211, pois neste já estavam implementadas as funcionalidades de construção e envio de mensagens Netlink, onde detalhes técnicos desta comunicação já haviam sido abstraídos em uma API.

Outra questão que deve ser esclarecida é o porquê de não existir uma comunicação direta entre o slbcli e o slbcore. O slbcli é somente um programa utilitário, quando um comando é enviado a este utilitário ele somente executa a sua função e depois o seu processo deixa de existir, ou seja, ele não é um processo em *background*. Isso já é um dos pontos a favor da existência do slbd como um componente intermediário, ele possibilita que exista um processo que represente o serviço de balanceamento de carga enquanto ele estiver ativo na máquina. Outra questão que favorece a existência do slbd é a sua responsabilidade de implementar a interface de comunicação com o Application Manager, fazendo a escuta da conexão TCP para recepção de comandos de troca de estado. Sem o slbd esta conexão não poderia ser estabelecida continuamente, já que o slbcli é somente um programa utilitário e não um processo em *background*.

5.7 Estruturas de dados utilizadas

Antes de iniciar a explicação das estruturas de dados utilizadas no módulo slbcore, é necessária uma pequena explicação sobre como foi dado o gerenciamento de memória para alocação dessas e estruturas. No âmbito dos LKM's Linux, o pedido de memória para alocar estruturas de dados de forma dinâmica pode ser feita da forma convencional que é

feita com a `libc` [14] via um `malloc/calloc`, ou então pode ser utilizada as funções especiais para LKM's de requisição de páginas de memória. A função convencional `malloc` realiza uma *system call* para o `kernel` alocar uma área de memória segmentada, já a função `kmalloc` (utilizada em modo de *kernel* de CPU) requisita uma área de memória contígua, além do que ser possível explicitar como as páginas de memória devem ser coletadas (BOVET; CESATI, 2000, p. 152). Esta forma de explicitação pode ser dada a partir dos parâmetros a seguir a função `kmalloc`:

GFP_USER: Modo de alocação normal de memória com baixa prioridade. Pode dormir;

GFP_KERNEL: Modo de alocação normal de memória com média prioridade. Pode dormir;

GFP_ATOMIC: Modo de alocação com alta prioridade. Nunca dorme;

Entre outros .

Os dois primeiros modos possibilitam que a execução do bloco de código seja parada por conta da leiura/escrita de memória, fazendo com que outras partes de código assumam a execução durante a operação de I/O. O modo `GFP_ATOMIC` faz com que a requisição de memória seja atômica, ou seja, seja executada ininterruptamente. Isso é essencial quando se trabalha com funções de tratamento de interrupção (pois devem ser operações com tratamentos extremamente velozes), o que é o caso tanto das funções de *callback* tanto do Netlink quanto do Netfilter (este será explicado posteriormente). Como todo o slbcore trabalha em torno dessas duas interrupções, então foi somente utilizado este parâmetro de alocação de memória (`GFP_ATOMIC`).

Para facilitar o desenvolvimento do slbcore foi utilizado a coleção de estruturas e macros do "uthash". Inicialmente este projeto provia a estrutura de dados *hash* (uma estrutura otimizada para buscas, funcionando basicamente na associação de uma chave de pesquisa a valores), mas depois foram adicionados pelo desenvolvedor listas e *arrays*, todas para a linguagem C utilizando o poder das macros-substituições. Primeiramente os códigos dessa biblioteca foram modificados para comportar o seu uso me módulos do *kernel*, já que a forma de alocação de memória utilizada foi feita de forma diferente (como foi dito anteriormente).

A modelagem das estruturas ficou da seguinte maneira: A estrutura principal seria um *hash* para representar os serviços que o balanceador estava configurado, onde a chave de busca seria exatamente a porta do serviço (um inteiro), informação que é extraída dos segmentos TCP/UDP. Esse serviço contém uma lista circular de servidores reais (o

[14] Biblioteca padrão C: <http://www.gnu.org/software/libc/>

que facilita no momento do escalonamento dos mesmos). O servidor real além de possuir a informação de IP, também possui um *array* de clientes que estão sendo atendidos no momento. A decisão de utilização de cada estrutura está totalmente ligado aos benefícios trazidos por cada tipo de estrutura de dados, onde tais elementos foram levados em consideração no momento da modelagem do sistema, já que cada elemento (serviço, servidor e cliente) possui diferentes requisitos e comportamentos no sistema. No quadro 6 está o código de definição destes elementos.

Exemplos de macros utilizadas para manipular as estruturas são estas a seguir:

`HASH_ADD_INT16(services, port, add_service);` Macro para adicionar um novo serviço no *hash*, onde a sua chave é um inteiro de 16 bits;

`HASH_FIND_INT16(services, &nport, service_found);` Macro para encontrar um serviço a part do número da porta (`nport`);

`CDL_APPEND(service_found->servers, server_itr);` Macro para inserir um servidor a uma lista circular de servidores;

`CDL_FOREACH(service_itr->servers, server_itr)` Macro para iterar sobre os servidores em uma lista circular;

`utarray_eltptr(server_itr->clients, i);` Macro que retorna o ponteiro para um cliente do *array* a partir de seu índice.

`utarray_insert(clients, &new_client, new_index);` Macro que insere um cliente no *array* em um determinado índice.

5.8 Persistência de sessão

Um dos maiores problemas que os balanceadores de carga devem superar é o fato das sessões de clientes. Existem diversas aplicações que atualmente são dependentes de uma sessão, estas são denominadas aplicações *stateful*, um exemplo é uma loja *online* onde o usuário possui um carrinho de compras, as informações sobre este carrinho de compras são mantidas em memória RAM pelo servidor (memória volátil), ou seja, não são inseridas em uma base de dados persistente. Caso este serviço esteja replicado em vários servidores reais da solução de balanceamento de carga, quando um usuário se realizar uma requisição (e.g. uma página *web*) ele será destinado a um servidor real, mas quando ele efetuar outra requisição (e.g. outra página *web*) ele corre o risco de ser destinado a outro servidor real (dependendo da decisão do algoritmo de escalonamento), caso o usuário tenha solicitado a compra de um produto e este foi adicionado ao seu carrinho de compras virtual, ao ser enviado para outro servidor ele perderia essa informação. Então o ideal é que um usuário mantenha a sua conexão com um único servidor para evitar problemas de perda da sessão.

Um fato que deve ser levado em consideração é que o TCP é um protocolo nativamente orientando a conexão, já o UDP não. Mas mesmo o UDP não sendo orientado a conexão, os protocolos da camada de aplicação que fazem uso do UDP podem determinar mecanismos de estabelecimento de uma sessão, fazendo que até mesmo segmentos UDP vindos de um mesmo usuário, tenham a necessidade de serem enviados para um único servidor.

De acordo com Kopparapu (2002, p. 53), existem meios de contornar este problema, um deles é a persistência de sessão baseada em IP, que foi o método implementado no slbd para amenizar o problema. Este método é relativamente simples, ele determina que ao chegar uma nova requisição de um cliente, é verificado se ele já possui uma conexão recente estabelecida com algum servidor (acessado via a estrutura de dados dos serviços/servidores/clientes), caso ele possua alguma a requisição será enviada ao servidor em questão, caso não possua a requisição será enviada a algum servidor selecionado pelo algoritmo de escalonamento.

Infelizmente o método de persistência baseado em IP não é infalível, Kopparapu (2002, p. 58) descreve o problema deste método com *proxies* de rede. Um *proxy* é um servidor que atua como intermediário de um grupo de usuários a uma rede externa para melhorar a segurança (*firewall*) e desempenho (*caching*). Este grupo de usuários ao fazerem requisições a uma rede externa são representados externamente pelo *proxy*, consequentemente este irá fazer um processo de NAT para que as requisições tenham como remetente o seu endereço. Caso usuários de um mesmo *proxy* façam requisições ao serviço de balanceamento de carga, todos os n usuários terão o mesmo IP de remetente (o IP do *proxy* que os representa), o que poderá sobrecarregar um servidor real, que receberá um grande número de requisições de diferentes usuários por causa da persistência de sessão. Existem alguns meios de contornar parte deste problema, mas devido as suas complexidades de implementação eles não foram trabalhados neste projeto.

5.9 Interceptação e manipulação de pacotes

A partir do da versão 2.3.X do *kernel* do Linux foi inserida uns dos mais importantes componentes do *kernel* se tratando de redes. O projeto desenvolvido por Russel e Welte (2002), teve a intenção de definir novas formas de filtragem e manipulação de pacotes de rede em sistemas Linux. Foi desenvolvido um grande *framework* para manipulação de pacotes chamado de Netfilter, totalmente integrado a implementação da pilha de protocolos TCP/IP do sistema, e junto a ele foram desenvolvidos módulos do *kernel* e aplicativos utilitários para exercer a funcionalidade de *firewall*, como o iptables e o arptables.

Dentro deste *framework* foram definidos vários *hooks* (ganchos), que são pontos bem definidos na travessia de pacotes da pilha de protocolos do sistema. Cada protocolo

define os seus *hooks*, e possibilitam que funções de *callback* sejam registradas a eles, possibilitando que um módulo do *kernel* tenha acesso aos pacotes que atravessam a pilha de protocolos no *hook* registrado em questão. Ao ter acesso aos pacotes de rede, podem ser efetuadas alterações e ainda podem ser dadas instruções ao netfilter sobre o que deve ser feito com o pacote a partir do retorno da função de *callback* (RUSSEL; WELTE, 2002). A seguir as cinco constantes que podem servir como retorno de função, seguido dos seus respectivos efeitos:

NF_ACCEPT: O pacote continua a sua travessia normalmente;

NF_DROP: O pacote será descartado;

NF_STOLEN: O pacote deverá ser esquecido pelo netfilter, o responsável pelo seu destino agora será do próprio módulo.;

NF_QUEUE: O pacote deve ser enfileirado para uso de aplicativos no modo usuário de CPU;

NF_REPEAT: Esse *hook* com este mesmo pacote em questão deverá ser chamado novamente.

Para registrar um *hook* é necessário informar alguns dados ao *framework* netfilter, estes dados são passados via a *struct* `nf_hook_ops`, que possui o formato do código no Quadro 7.

Cada um dos campos serão explicados para melhor entendimento do funcionamento do netfilter. O primeiro campo `hook` é um ponteiro para uma função implementada no próprio módulo do `kernel`, ela será a função de *callback* que irá receber os pacotes de rede. O netfilter define um protótipo para essa função com o formato do código no Quadro 8.

A seguir a descrição de cada parâmetro:

hooknum: Tipo de *hook* (será melhor explicado adiante);

skb: Estrutura que contém o pacote, ou conjunto de pacotes a serem manipulados. Seu conteúdo é mutável de acordo com a camada de rede em que se encontra, por conta de estruturas `union` que ele contém;

in: Ponteiro para a estrutura que representa a interface de rede em que o pacote chegou ao *host* (e.g eth0). Válido para pacotes que acabaram de chegar, senão fará referência a `NULLl`;

out: Ponteiro para a estrutura que representa a interface de rede em que o pacote será destinado para sair do *host* (e.g eth0), o que foi definido após roteamento do mesmo. Válido para pacotes que estão para sair, senão fará referência a `NULL`;

okfn: Ponteiro para uma função que será invocada caso todas as funções de *callback* registradas ao *hook* em questão retornem `NF_ACCEPT`.

Dentro da função `hook_function_name` que poderá ser feita a manipulação e filtragem dos pacotes de rede. O próximo campo da estrutura principal é o `pf`, que faz uma simples referência a família do protocolo que está sendo trabalhado, `PF_INET` para IPv4 e `PF_INET6` para IPv6, no caso do slbcore o `pf` será a princípio `PF_INET`. O próximo campo é o `hooknum`, este é um dos mais importantes pois define qual tipo de *hook* será trabalhada na função de *callback*, definindo em que ponto da travessia de rede o pacote ficará acessível. A Figura 15 define os cinco *hooks* acessíveis da travessia. A descrição de cada *hook* e cada evento da travessia virá a seguir.

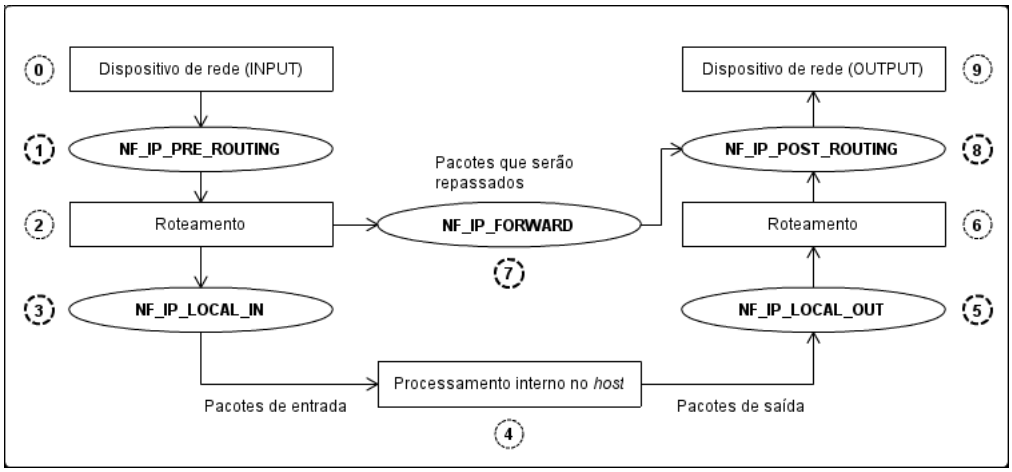

Figura 15: Tipos de *hook* com possibilidade de registro

Fonte: Wehrle et al. (2004). Adaptado pelo autor.

0. O pacote chega em uma interface de rede de entrada do *host*;
1. O **primeiro** *hook*, antes do roteamento inicial é possível acessar o pacote;
2. Ocorre o roteamento do pacote;
3. O **segundo** *hook*, é analisado que o pacote se destina ao *host* em questão e antes dele ser processado é possível acessá-lo;
4. O pacote é processado pelas aplicações do *host* a quais ele era destinado;
5. O **terceiro** *hook*, após o processamento do pacote no *host* ele pode ser acessado;
6. Ocorre o roteamento de saída do pacote processado no *host*;

7. O **quarto** *hook*, é analisado que o pacote não se destina ao *host* em questão, e antes dele ser repassado (i.e. *forwarded*) é possível acessá-lo;

8. O **quinto** *hook*, o pacote está de saída do *host* e antes disso é possível acessá-lo;

9. O pacote chega a interface de rede de saída do *host*.

No caso do slbcore o que se deseja realizar é um DNAT (*Destination Network Adress Translation*), ou seja, o endereço de destino do pacote de rede será alterado e repassado para o roteamento (onde será enviado ao servidor real escolhido). A realização do DNAT sempre deverá ocorrer na chegada do pacote a rede, para isso o seu acesso via o netfilter deverá ser feito pelos *hooks* `NF_IP_PRE_ROUTING` ou `NF_IP_LOCAL_IN`. Como não é desejado que o pacote seja processado por nenhuma aplicação dentro do servidor de balanceamento de carga, é necessário que ele seja acessado antes do roteamento ocorrer, pois ele será destinado a este fim já que o IP de destinatário é o VIP. Então será utilizado o *hook* `NF_IP_PRE_ROUTING`.

Mas somente esse DNAT não resolve todos os problemas, caso o pacote já tenha sido processado pelos servidores reais e tenha sido enviado de volta ao balanceador, então este pretende despachá-lo de volta ao cliente, mas neste caso o IP de remetente irá como o do servidor real e não o VIP do balanceador de carga. Para resolver esse problema foi necessário fazer uso de outro *hook*, um `NF_IP_POST_ROUTING` para troca do IP de remetente do servidor real pelo VIP.

O próximo campo da estrutura de configuração `nf_hook_ops` é o `priority`, esse campo indica qual a prioridade de acesso do *hook* ao pacote em questão, pois em um mesmo host podem existir diversos módulos do *kernel* com *hooks* definidos. Então quanto maior a prioridade de acesso prévio a um pacote deverá ser indicado pela constante NF_IP_PRI_FIRST, e no caso de existir uma prioridade mínima de acesso poderá ser indicado pela constante NF_IP_PRI_LAST. Entre essas duas constantes existem outras que definem prioridades intermediárias.

A seguir no quadro 9 está o código do *hook* de pré-roteamento do slbcore, exatamente o código que é executado todas as vezes em que se chega uma nova requisição no balanceador de carga (um exemplo a ser dado).

No quadro 9 os tipos de dados `__be32` representam espaços de memória inteiros, não sinalizados, de 32 bits e com a ordenação dos bits na forma *big-endian* (bit mais significativo vem em primeiro), os outros tipos de dados seguem a mesma linha de pensamento. A ordenação *big-endian* é extremamente comum em protocolos de rede, o que é o caso do IP, TCP e UDP que são os principais protocolos trabalhados em todo esse projeto.

Quadro 4: Código fonte C: Função de recebimento de mensagem Netlink no slbcore

```
1  static void slb_netlink_recv(struct sk_buff *skb) {
2      // Header da mensagem
3      struct nlmsghdr *nlh = NULL;
4      // Buffer para o conteudo da mensagem
5      char *slbd_msg = NULL;
6      __u16 i = 0;
7      // Nome do comando extraido da mensagem
8      char *command_name = NULL;
9      // String auxiliar
10     char *param = NULL;
11     // Ponteiro para string auxiliar
12     char ** str_ref;
13     // Array de parametros
14     UT_array *params = NULL;
15     // Array auxiliar
16     UT_array *split_msg = NULL;
17
18     // Extrai o header e o conteudo da mensagem
19     nlh = (struct nlmsghdr *)skb->data;
20     slbd_msg = (char *) nlmsg_data(nlh);
21
22     // Preenche o array de tokens que vieram da mensagem
23     utarray_new(split_msg, &ut_str_icd);
24     if (slb_str_split(slbd_msg, CHUNK_SEPARATOR_TOKEN, split_msg) != 0)
           {
25         logger_error("Invalid message: %s", slbd_msg);
26         goto clean_up;
27     }
28
29     // Atribui o nome do comando
30     str_ref = (char **) utarray_eltptr(split_msg, 0);
31     if(str_ref) {
32         command_name = *str_ref;
33     }
34
35     // Atribui os parametros do comando
36     utarray_new(params, &ut_str_icd);
37     for (i = 1; i < utarray_len(split_msg); i++) {
38         str_ref = (char **) utarray_eltptr(split_msg, i);
39         if(str_ref) {
40             param = strdup(*str_ref);
41         }
42
43         utarray_push_back(params, &param);
44     }
45
46     // Executa o comando
47     slb_execute_command(command_name, params);
48
49     // Limpa o array
50     clean_up:
51         utarray_free(split_msg);
52 }
```

Quadro 5: Código fonte Python: Método de envio de mensagem Netlink no slbd

```
1  def _send_slbcore_command(self, command_name, msg):
2      # Verifica se o comando faz parte dos comandos habilitados e se o
           daemon nao foi parado
3      if (command_name in self._COMMANDS_SLBCORE) and SlbDaemon().
           check_if_running() is True:
4          try:
5              # Cria a conexao e o payload da mensagem
6              netlink_conn = netlink.Connection(netlink.NETLINK_USERSOCK)
7              netlink_msg = netlink.Message(netlink.NLMSG_DONE, flags=
                   netlink.NLM_F_REQUEST, payload=bytes(msg) + b'\x00')
8              # Envia a mensagem
9              netlink_msg.send(netlink_conn)
10         except socket.error, e:
11             logger.error('Problem␣connecting␣with␣slbcore:␣' + str(e))
```

Quadro 6: Código fonte C: Estruturas de dados do slbcore

```
1
2  struct slb_client;
3
4  // Servidor
5  typedef struct slb_server {
6      __be32 addr; // Endereco do servidor
7      UT_array *clients; // Array de servidores
8      struct slb_server *prev, *next; // Estrutuas do utlist para
           funcionalmente da lista circular
9  } slb_server;
10
11 // Cliente
12 typedef struct slb_client {
13     __be32 addr; // Endereco do cliente
14     slb_server *server; // Servidor que atende o cliente
15     unsigned long long sec_last_conn; // Tempo em que foi feita a ultima
           requisicao
16 } slb_client;
17
18 // Servico
19 typedef struct slb_service {
20     __be16 port; // Chave do servico
21     slb_server *servers; // Lista de servidores
22     slb_server *actual_server; // Servidor atual que atende o servico
23     UT_hash_handle hh; // Estrutura auxiliar do uthash
24 } slb_service;
```

Quadro 7: Código fonte C: *Struct* de cofiguração de um *hook*

```
1 struct nf_hook_ops
2 {
3     struct list_head list; // Nao deve ser preenchido
4     nf_hookfn *hook; // Ponteiro para a funcao que ira registrar o hook
5     int pf; // A familia do protocolo de rede
6     int hooknum; // O identificador do tipo de hook
7     int priority; // Prioridade do hook
8 };
```

Quadro 8: Código fonte C: Protótipo de função para registrar um *hook*

```
1 unsigned int hook_function_name(unsigned int hooknum,
2                   struct sk_buff **skb,
3                   const struct net_device *in,
4                   const struct net_device *out,
5                   int (*okfn)(struct sk_buff*));
```

Quadro 9: Código fonte C: Hook de pré-roteamento do slbcore

```
1  static unsigned int slb_pre_routing_hook(unsigned int hooknum, struct sk_buff *skb, const
       struct net_device *in_device, const struct net_device *out_device, int (*okfn)(
       struct sk_buff *)) {
2
3      // Eh recuperado o cabecalho IP do sk_buff
4      struct iphdr *ip_head = (struct iphdr *)skb_network_header(skb);
5      struct tcphdr *tcp_head = NULL;
6      struct udphdr *udp_head = NULL;
7
8      // A partir do cabecalho IP eh armazenado um ponteiro para o destinatario
9      //  (para uma futura substituicao), eh extraido o IP do device de entrada e o IP
10     //  do remetente
11     __be32 *dest_addr = &ip_head->daddr;
12     __be32 src_addr = ip_head->saddr;
13     __be32 in_dev_addr = slb_ip_from_device(in_device);
14     __be16 dest_port;
15
16     slb_server *server;
17
18     // Se o pacote eh interno o aceita
19     if(src_addr == LOCALHOST_IP)
20         return NF_ACCEPT;
21     // Se o protocolo for TCP extrai o seu cabecalho e porta de destino do servico
22     if(ip_head->protocol == IPPROTO_TCP) {
23         tcp_head = (struct tcphdr *)((__u32 *)ip_head + ip_head->ihl);
24         dest_port = tcp_head->dest;
25     // Se o protocolo for UDP extrai o seu cabecalho e porta de destino do servico
26     } else if (ip_head->protocol == IPPROTO_UDP) {
27         udp_head = (struct udphdr *)((__u32 *)ip_head + ip_head->ihl);
28         dest_port = udp_head->dest;
29     // Se o protocolo for qualquer outro o aceita
30     } else {
31         return NF_ACCEPT;
32     }
33
34     // Se o IP de entrada for diferente do VIP, entao o aceita pois veio de outro lugar
35     if(in_dev_addr != slb_conf.slb_address)
36         return NF_ACCEPT;
37
38     // Eh feito todo o processo de balanceamento de carga a partir do IP do
39     //  remetente e a porta de destino, sendo retornado o servidor responsavel
40     //  pelo atendimento a partir do algoritmo de escalonamento ou da tabela de
           persistencia
41     server = slb_load_balance(src_addr, dest_port);
42
43     // Eh trocado o destinatario da requisicao (atualmente como o VIP) pelo IP do
44     //  servidor real escolhido. Depois sao recalculados os checksums e corrigidos
45     //  nos cabecalhos do pacote IP e segmento TCP ou UDP
46     *dest_addr = server->addr;
47     slb_recalculate_checksums(ip_head, tcp_head, udp_head);
48
49     // Por fim o pacote eh aceito, ele sera roteado para o seu destino posteriormente
50     //  na fase de roteamente.
51     return NF_ACCEPT;
52 }
```

6 Resultados

O capítulo de resultados tem o intuito de demonstrar como foram realizados os testes para validação da real utilidade do sistema proposto. Mas antes de demonstrar tais resultados é importante demonstrar um pouco o funcionamento do sistema como um todo.

O mesmo ambiente utilizado para desenvolvimento foi o ambiente utilizado para a realização dos testes (2 Macbkooks, 1 HP, 1 roteador e 1 *switch*). Quando se pensou em aluma estratégia de testes para conseguir validar a solução de balanceamento de carga logo se pensou em algum tipo de teste de desempenho. Seguindo esta linha de pensamento também foi analisada alguma ferramenta que simulasse um grande volume de requisições ao balanceador de carga, de forma totalmente automatizada. A ferramenta escolhida para realizar tal operação foi o JMeter, mantida pela Apache Software Foundation. De acordo com os mantenedores da ferramenta (APACHE SOFTWARE FOUNDATION, 2015):

> O Apache JMeter pode ser usado para testar o desempenho tanto em recursos estáticos quanto recursos dinâmicos (...). Ele pode ser usado par simular uma alta carga em um servidor, grupo de servidores, rede ou objeto para testar o seu poder ou para analizar de forma geral o desempenho em diferentes tipos de carga (...).

O JMeter trabalha com o conceito de plano de testes, onde em um plano podem ser realizados diferentes tipos de testes que simulam diferentes grupos de usuários. Para simular um serviço em todos os servidores reais foi utilizado o servidor web Apache, também mantido pela Apache Software Foundation, dentro dele foi instalado uma simples aplicação em PHP para calcular os 1200 primeiros números primos existentes (o que gera um custo computacional de cerca de 1 segundo para as VM's criadas). A automação de instalação do Apache, seus plugins e também a instalação do *script* PHP foi totalmente realizada pelo Vagrant.

Para o teste no slb foi criado o seguinte plano de testes no JMeter:

Grupo de usuários: 100 usuários (*threads*)

Tempo entre o lançamento de cada usuário: 600 milisegundos

Requsições por usuário: Cada usuário executauma requisição HTTP ao script ao Apache (porta 80), o contexto das requisição é exatamenente o script PHP que executa o cálculo dos números primos.

Duração do teste: 120 segundos

Com esse plano de testes então foram geradas 200 requisições para o slb em um espaço de tempo de 120 segundos. Para simular diferentes clientes reais, ou seja, com diferente IP's, foi utilizada a funcionalidade do JMeter de mutação do IP a partir de uma lista de IP's válidos a serem utilizados dentro das disponíveis na máquina. Como os testes no JMeter foram realizados utilizando o sistema operacional Fedora 20, foi utilizada a funcionalidade do Linux de adição de múltiplos IP's a uma interface de rede (e.g. eth0). Dessa forma foram gerados 50 diferentes IP's para a interface de saída da requisição, gerando um total de 4 requisições por IP diferente.

O JMeter valida o resultado da resposta do Apache pelo código de resultado da requisição HTTP, onde o código 200 indica sucesso. Além da validação do sucesso no atendimento das requisições pelo sistema de balanceamento (onde todas obtiveram êxito), foram utilizados relatórios de agregação que a ferramenta JMeter gera, mais especificamente um relatório que gera o tempo médio de resposta para todos os clientes e outro relatório que gera medidas estatística sobre as requisições, como o desvio padrão, vazão (*throughput*), média e mediana, todas relacionadas ao tempo de resposta do serviço.

Este mesmo plano de testes foi executado três vezes, um para cada algoritmo de balanceamento de carga implementado ("Round-robin", "Menor Latência" e "Menos conexões"). A seguir nas figuras 16, 17, 18, 19, 20 e 21 os resultados para cada um (um gráfico de tempo de resposta e um gráfico de medidas estatísticas para cada).

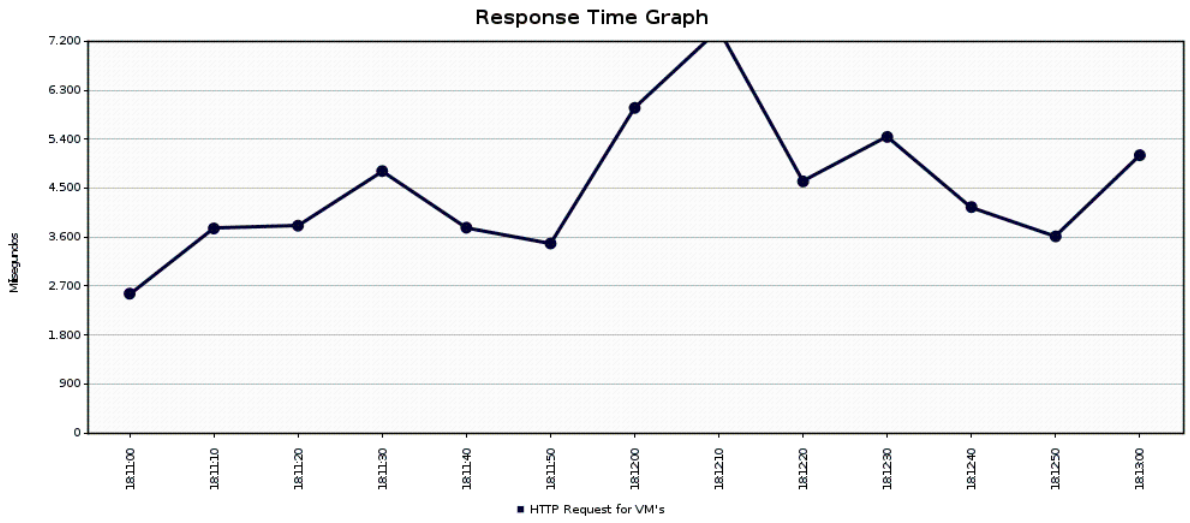

Figura 16: Gráfico de tempo de resposta do algoritmo Round-robin

Como pode se observar nos resultados o algoritmo Round-robin se demonstrou o com o melhor desempenho entre os três em relação ao tempo médio de resposta, também se demonstrou o com maior constância de valores por conta do seu baixo desvio padrão. Tudo isso pode se justificar exatamente pelas características do *cluster* utilizado em questão, neste cenário todos os servidores reais possuíam as mesmas configurações, formando um *cluster* homogêneo, que é o cenário ideal para utilização do algoritmo Round-robin. A

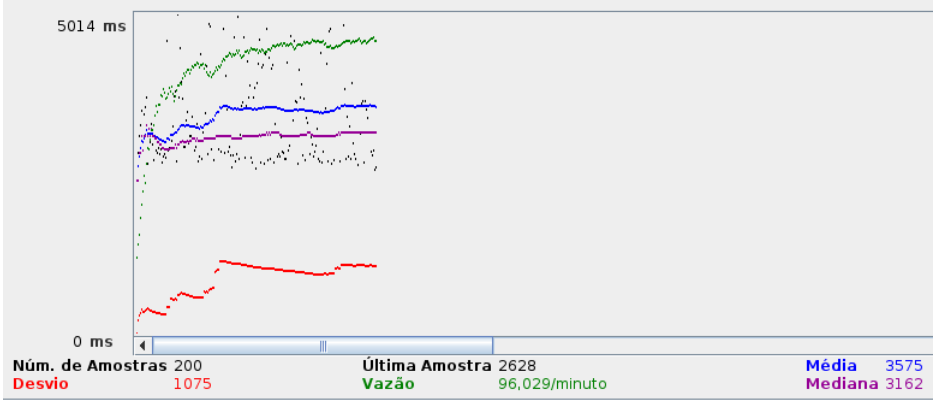

Figura 17: Gráfico de medidas estatísticas do algoritmo Round-robin

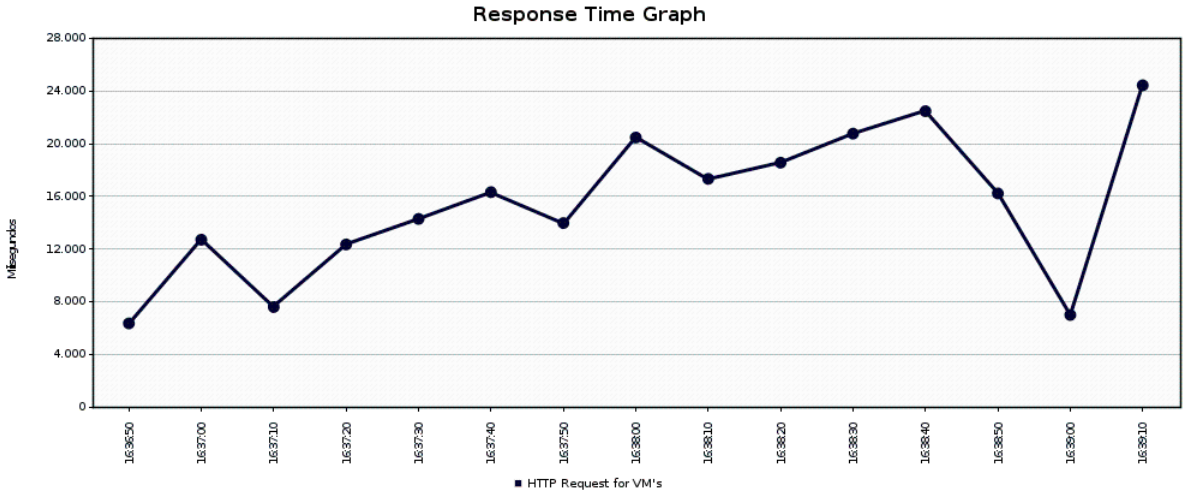

Figura 18: Gráfico de tempo de resposta do algoritmo “Menor Latência”

sobrecarga que é gerada pela complexidade do monitoramento e reordenamento dos servidores reais no “menor latência” faz com que se perca tempo quando o *cluster* é homogêneo, sendo então recomendado o seu uso quando os servidores reais tem diferentes configurações, objetivando a diminuição da sobrecarga nos servidores com uma configuração menos potente e concentrando-a nos servidores com maior poder computacional.

O algoritmo “menos conexões” neste caso teve um desempenho similar ao Round-robin, pois pelo curto período de tempo da execução dos testes o algoritmo detectava que todos os servidores continham números similares de conexões, tendo o seu comportalmento similar ao do Round-robin, com a diferença da sobrecarga que a contagem de conexões oferece.

Para melhor ilustrar como ficou a distribuição da quantidade de clientes por servidor real nos testes, a seguir uma lista de IP’s dos servidores reais e entre parênteses a quantidade de clientes simulados que foram atendidos:

Round-robin: 172.16.0.1 (3), 172.16.0.2 (3), 172.16.0.3 (3), 172.16.0.4 (3), 172.16.0.5

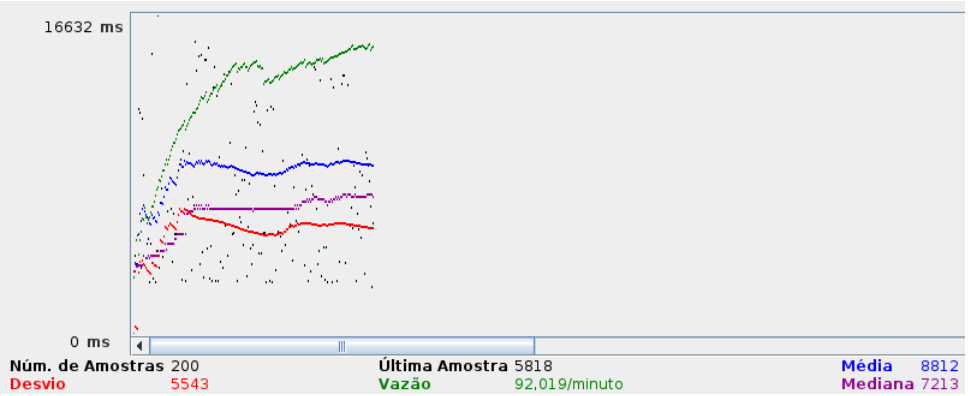

Figura 19: Gráfico de medidas estatísticas do algoritmo “Menor Latência”

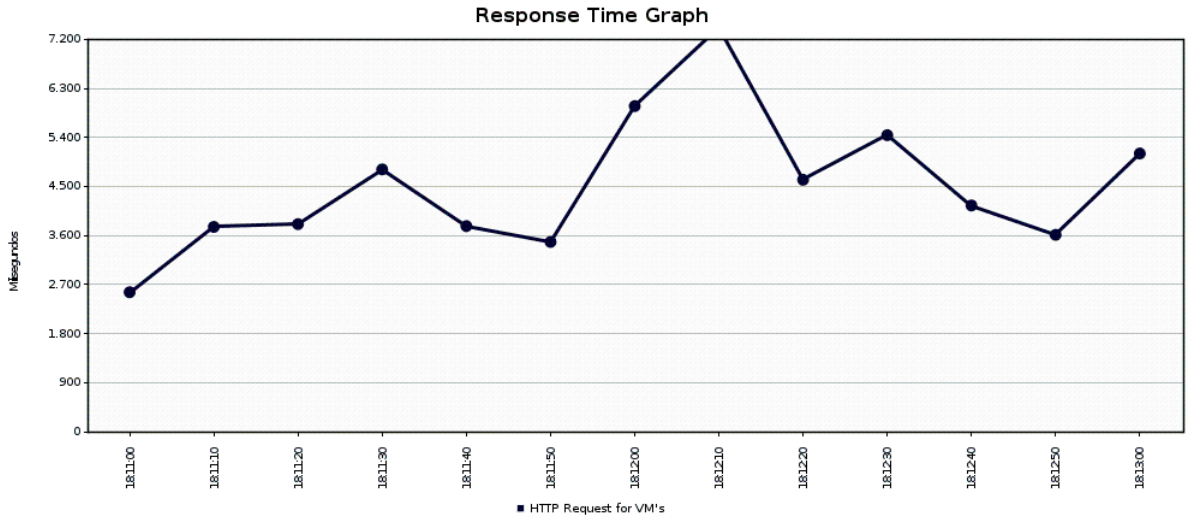

Figura 20: Gráfico de tempo de resposta do algoritmo “Menos Conexões”

(3), 172.16.0.6 (3), 172.16.0.21 (4), 172.16.0.22 (4), 172.16.0.23 (4), 172.16.0.24 (4), 172.16.0.25 (4), 172.16.0.26 (4), 172.16.0.27 (4), 172.16.0.28 (4)

Menor Latência: 172.16.0.1 (1), 172.16.0.2 (3), 172.16.0.3 (0), 172.16.0.4 (7), 172.16.0.5 (1), 172.16.0.6 (7), 172.16.0.21 (1), 172.16.0.22 (1), 172.16.0.23 (1), 172.16.0.24 (7), 172.16.0.25 (8), 172.16.0.26 (3), 172.16.0.27 (8), 172.16.0.28 (2)

Menos Conexões: 172.16.0.1 (3), 172.16.0.2 (3), 172.16.0.3 (3), 172.16.0.4 (3), 172.16.0.5 (4), 172.16.0.6 (3), 172.16.0.21 (4), 172.16.0.22 (4), 172.16.0.23 (3), 172.16.0.24 (4), 172.16.0.25 (4), 172.16.0.26 (4), 172.16.0.27 (4), 172.16.0.28 (4)

Como foi dito anteriormente, a distribuição no Round-robin ficou praticamente idêntica ao “menos conexões”, já o “menor latência” teve uma distribuição extremamente disforme, enquanto um servidor atendeu 8 clientes, outro servidor atendeu nenhum cliente.

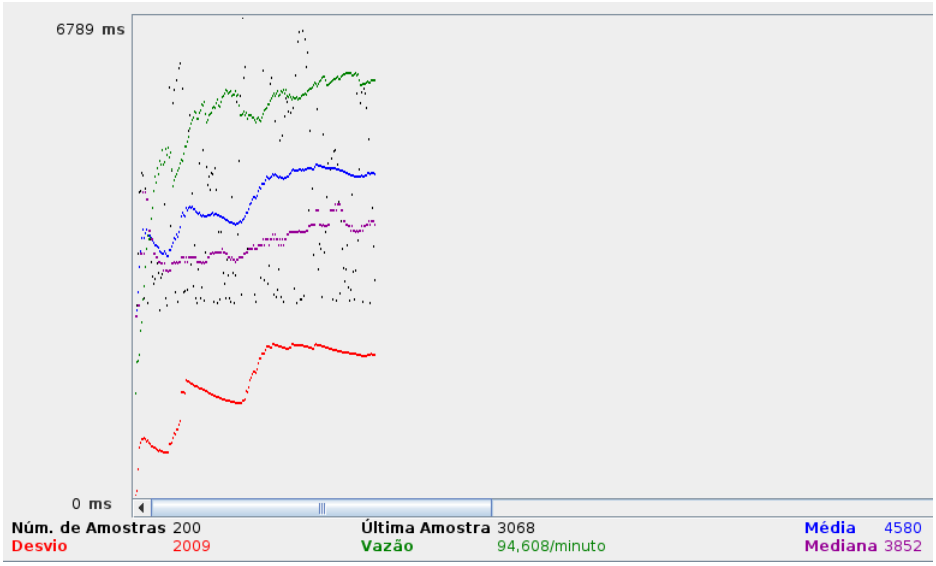

Figura 21: Gráfico de medidas estatísticas do algoritmo “Menos Conexões”

7 Conclusão

Por intermédio deste trabalho de conclusão de curso foi possível realizar um estudo sobre diversas áreas da computação que tangem desde redes de computadores, sistemas operacionais, engenharia de software e outras. Todo este estudo foi necessário para se obter maior entendimento sobre ferramentas de balanceamento de carga de servidores, deste o seu histórico de uso, suas principais funcionalidades, benefícios e sua arquitetura geral.

Na introdução deste trabalho foi indicada a motivação real da implementação desta ferramenta, e na conclusão deve ser feita uma reafirmação da importância que este tipo de produto tem no mundo hoje. O uso de balanceadores de carga é uma das formas de garantia de desempenho, disponibilidade e confiabilidade de serviços, e juntamente com outros tipos de soluções forma uma estrutura básica de uma infraestrutura de TI que visa oferecer estes serviços a um grande número de usuários sem pecar com a sua qualidade a um nível técnico.

Para fins de pesquisa e desenvolvimento foi feito o projeto e toda a implementação de uma solução original de balanceamento de carga, tendo como participantes os presentes autor e orientador deste trabalho. A solução proposta conseguiu atingir resultados tangíveis, sendo projetada a desenvolvida uma solução de balanceamento de carga confiável para sistemas Linux, com todo o ambiente automatizado via ferramentas de gerência de configuração e virtualização capazes de replicar o trabalho e testes em outras ocasiões. O produto final em si ainda não pode ser considerado um produto de mercado, no sentido de utilização real por empresas, isso pois ainda carecem algumas funcionalidades que outras soluções de mercado provêem e também carece de uma real suíte de testes para garantia da sua qualidade (testes unitários, testes de integração, testes de sistema, etc).

7.1 Possibilidades de trabalhos futuros

O contexto desse projeto foi a disciplina de Trabalho de Conclusão de Curso 2, do curso de Engenharia de Software da Universidade de Brasília, realizado em um período de aproximadamente 1 ano. O projeto tem capacidade de evoluir com futuras melhorias a serem realizadas pelo autor, e como o código fonte foi disponibilizado sob licença livre no GitHub outros interessados podem contribuir para a sua evolução. A seguir alguns dos diversos tópicos de melhoria que poderão ser investidos:

1. Desacoplamento do slb com o Application Manager, gerênciando o sistema de *failover* como um plugin. Isso possibilitaria que outras soluções pudessem ser utilizadas

em seu lugar caso seja o desejo do usuário.

2. Geração de um processo de empacotamento e instalação da solução, podendo gerar um pacote .deb (Para sistemas baseados em Debian), um pacote .rpm (Para sistemas baseados em Red Hat), entre outros formatos. A solução atualmente carece dessa funcionalidade.

3. Melhor utilização das ferramentas de depuração para desenvolvimento do módulo do *kernel* slbcore. Dessa forma seriam previstos erros, otimizados procedimentos, entre outros, já que este módulo é crítico no sistema como foi dito no decorrer do trabalho.

4. Criação de melhores políticas de segurança. Atualmente a única política de segurança tomada para o sistema foi o uso do protocolo SSL/TLS para comunicação via socket entre os processos no modo usuário (slbd, slbcli e Application Manager).

5. Melhorias na automação de configuração do ambiente de desenvolvimento, facilitando a mudança das configurações das máquinas virtuais, entre outros.

6. Entre outras infindáveis possibilidades.

Referências

APACHE SOFTWARE FOUNDATION. *Apache JMeter*. 2015. Disponível em: <http://jmeter.apache.org>. Citado na página 87.

ATWOOD, J. *Understanding User and Kernel Mode*. 2008. Blog Coding Horror. Disponível em: <http://blog.codinghorror.com/understanding-user-and-kernel-mode/>. Citado na página 32.

BENVENUTI, C. *Understanding Linux Network Internals*. 1. ed. EUA: O'Reilly, 2005. Citado 2 vezes nas páginas 71 e 76.

BOURKE, T. *Server Load Balancing*. 1. ed. EUA: O'Reilly, 2001. Citado 5 vezes nas páginas 27, 37, 39, 40 e 52.

BOVET, D. P.; CESATI, M. *Understanding the Linux Kernel*. EUA: O'Reilly, 2000. Citado 4 vezes nas páginas 27, 33, 75 e 77.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions*, v. 14, n. 2, 1988. Citado na página 46.

CISCO SYSTEMS. *Cisco ASA 5500 Series Configuration Guide using the CLI*. 8.2. ed. [S.l.], 2010. Cap. 33. Disponível em: <http://www.cisco.com/c/en/us/td/docs/security/asa/asa82/configuration/guide/config.html>. Citado na página 49.

COMER, D. E. *Internetworking with TCP/IP: Principles, Protocol, and Architecture*. 6. ed. New Jersey, EUA: Pearson, 2013. Citado 2 vezes nas páginas 30 e 53.

FREE SOFTWARE FOUNDATION. *What is free software?* 2014. Versão 1.135. Disponível em: <https://www.gnu.org/philosophy/free-sw.en.html>. Citado na página 31.

GARLAN, D.; SHAW, M. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, v. 1, 1994. Citado na página 51.

HORMAN, S. *Linux Virtual Server Tutorial*. 2003. Ultra Monkey Project. Disponível em: <http://www.ultramonkey.org/papers/lvs_tutorial/html/>. Citado na página 54.

IBM RATIONAL SOFTWARE. *Rational Unified Process (RUP): Diretrizes de um Documento de Arquitetura de Software*. 2001. Disponível em: <http://www.wthreex.com/rup/portugues/process/modguide/md_sad.htm>. Citado 3 vezes nas páginas 51, 57 e 62.

JAYASWAL, K. *Administering Data Centers: Servers, Storage, and Voice over IP*. Indianapolis, EUA: Wiley, 2006. Citado 4 vezes nas páginas 27, 47, 48 e 49.

KARIMI, A. et al. A new fuzzy approach for dynamic load balancing algorithm. *International Journal of Computer Science and Information Security (IJCSIS)*, v. 6, n. 1, 2009. Citado na página 34.

KOPPARAPU, C. *Load Balancing Servers, Firewalls, and Caches*. New York, EUA: Wiley, 2002. Citado 5 vezes nas páginas 48, 68, 72, 74 e 79.

KROAH-HARTMAN, G. Things you never should do in the kernel. *Linux Journal*, n. 133, 2005. Citado na página 75.

LARANJEIRA, L. A. F. *Manutenção e Evolução de Software: Aula 3A, Apresentação do Trabalho*. 2011. Universidade de Brasília, Faculdade UnB Gama. 21 slides. Citado 4 vezes nas páginas 56, 60, 61 e 103.

LEVY, E. Smashing the stack for fun and profit. *Phrack Magazine*, v. 7, n. 49, 1996. Citado na página 75.

MADHURAVANI, P.; SUMALATHA, G.; SHANMUKHI, M. An emperical study of static and dynamic load balancing algorithms in parallel and distributed computing environment. *International Journal of Emerging trends in Engineering and Development*, v. 3, n. 2, 2012. Citado 3 vezes nas páginas 46, 47 e 56.

NATáRIO, R. *Load Balancing III*. 2011. Blog Network and Servers. Disponível em: <http://networksandservers.blogspot.com.br/2011/03/balancing-iii.html>. Citado na página 40.

RUSSEL, R.; WELTE, H. *Linux netfilter Hacking HOWTO*. 2002. Rev. 521. Disponível em: <http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html>. Citado 3 vezes nas páginas 34, 79 e 80.

SALZMAN, P. J.; BURIAN, M.; POMERANTZ, O. *The Linux Kernel Module Programming Guide*. 2007. Versão 2.6.4. Linux Documentation Project. Citado 2 vezes nas páginas 33 e 75.

SHARMA, S.; SINGH, S.; SHARMA, M. Performance analysis of load balancing algorithms. *World Academy of Science, Engineering and Technology*, n. 38, 2008. Citado na página 36.

STATISTIC BRAIN. *Google Annual Search Statistics*. 2014. Disponível em: <http://www.statisticbrain.com/google-searches/>. Citado na página 23.

TANENBAUM, A. S. *Redes de Computadores*. Trad. 4. ed. São Paulo: Campus Elsevier, 2003. Citado 2 vezes nas páginas 28 e 29.

TIOBE SOFTWARE. *TIOBE Programming Community Index*. 2014. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Citado na página 55.

TORVALDS, L. *Re: Compiling C++ kernel module + Makefile*. 2004. Mensagem enviada a lista de e-mails em 19 janeiro 2004. Disponível em: <https://groups.google.com/forum/\#!forum/fa.linux.kernel>. Citado na página 54.

W3SCHOOLS. *OS Platform Statistics*. 2014. Disponível em: <http://www.w3schools.com/browsers/browsers_os.asp>. Citado na página 31.

WEHRLE, K. et al. *The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel*. EUA: Prentice Hall, 2004. Citado na página 81.

WIKIPEDIA. *Internet protocol suite — Wikipedia, The Free Encyclopedia*. 2014. Disponível em: <http://en.wikipedia.org/wiki/Internet_protocol_suite>. Citado na página 30.

ZHANG, W. Linux virtual server for scalable network services. *National Key Laboratory of Parallel and Distributed Processing*, 2000. The Linux Virtual Server Project. Citado 6 vezes nas páginas 40, 41, 42, 43, 44 e 45.

Apêndices

APÊNDICE A – Lista de comandos do slbcli.py

A seguir a lista de comandos que o utilitário slbcli mostra ao executar um `--help`:

```
root@slb-primary-vm-debian-7:/vagrant/slb# python slbcli.py --help
usage: slbcli.py [-h] [--status] [--clear-config] [--address ADDRESS]
                 [--address-dev ADDRESS_DEV] [--gateway GATEWAY]
                 [--gateway-dev GATEWAY_DEV]
                 [--cluster-network ADDRESS/CIDR_CLASS] [--ssh-port SSH_PORT]
                 [--credentials USER,PASS]
                 [--backup-slb ADDRESS,USER,PASS,PORT]
                 [--algorithm {ROUND_ROBIN,LOWER_LATENCY,LEAST_CONNECTIONS}]
                 [--add-server ADDRESS:PORT] [--save-config]
                 [--rm-credentials] [--rm-ssh-port] [--rm-server RM_SERVER]
                 [--rm-server-port ADDRESS:PORT] [--show-config] [--version]
                 [--load | --unload | --reload] [--with-redundancy]

slbcli - A command-line interface for the slb system (Server Load Balancing)

optional arguments:
  -h, --help            show this help message and exit

management:
  --load                Load the slb module and initializes the slb functions
  --unload              Unload the slb module and stop the slb functions
  --reload              Reload the slb module and restart the slb functions
  --with-redundancy     Set the slb to work in a redundant system with primary
                        and backup nodes, initially at active and standby
                        modes, respectively (Actually this system is only
                        integrated with Application Manager software)

show:
  --show-config         Show the actual slb configuration in JSON format
  --version             show program's version number and exit

execute:
  --status              Check the status of slb
  --clear-config        Clear the actual in-use configuration of slb
  --address ADDRESS     Set the slb address for incoming clients requests
  --address-dev ADDRESS_DEV
                        Set the slb device for incoming clients requests
  --gateway GATEWAY     Set the slb gateway address for delivery client
                        requests to real servers
  --gateway-dev GATEWAY_DEV
                        Set the slb device for delivery client requests to
                        real servers
  --cluster-network ADDRESS/CIDR_CLASS
                        Set the slb cluster network address in CIDR notation
                        to perform NAT in real servers
  --ssh-port SSH_PORT   Set the slb SSH port
  --credentials USER,PASS
                        Set the slb OS host credentials for the redundant
                        system (Actually this system is only integrated with
                        Application Manager software). The arguments are user
                        and password, separated by a comma ','
  --backup-slb ADDRESS,USER,PASS,PORT
                        Set the slb backup information for the redundant
                        system (Actually this system is only integrated with
                        Application Manager software). The arguments are
                        address, user, password and SSH port, separated by a
                        comma ','
  --algorithm {ROUND_ROBIN,LOWER_LATENCY,LEAST_CONNECTIONS}
                        Set a scheduling algorithm
  --add-server ADDRESS:PORT
                        Add a real server for slb
  --save-config         Save in the default configuration file the actual slb
                        configuration
  --rm-credentials      Remove the slb OS host credentials for the redundant
                        system (Actually this system is only integrated with
                        Application Manager software)
  --rm-ssh-port         Remove the slb SSH port
  --rm-server RM_SERVER
                        Remove a real server from slb
  --rm-server-port ADDRESS:PORT
                        Remove a real server port from slb
```

Figura 22: Comandos do slbcli.py

APÊNDICE B – Lista de comandos do amcli

A seguir uma lista comandos do componente amcli do Application Manager e seus respectivos efeitos (LARANJEIRA, 2011):

START: Inicia a aplicação em um ou dois hosts.

RESTART: Fecha e recomeça a aplicação (exceto se ela for ACTIVE e não houver uma versão STANDBY)

SHUTDOWN: Fecha a aplicação (exceto se ela for ACTIVE e não houver uma versão STANDBY)

SWITCH: Reverte o estado da aplicação (ACTIVE e STANDBY)

RELOAD: Recarrega o arquivo de configuração estática da aplicação.

TEST: Faz com que a instância STANDBY da aplicação mude para MAINTENANCE.

UNTEST: Faz com que a instância MAINTENANCE da aplicação mude para STANDBY.

PRIMARY: Muda a instância da aplicação que não é PRIMARY para PRIMARY (no arquivo de configuração).

ENABLE: Muda a instância da aplicação que é MAINTENANCE para STANDBY e a libera para se tornar ACTIVE caso a dual falhe (mudando o valor no arquivo de configuração dinâmica UNINHIBITED)

DISABLE: Muda a instância da aplicação que é STANDBY para MAINTENANCE e a restringe para não se tornar ACTIVE caso a dual falhe (mudando o valor no arquivo de configuração dinâmica para INHIBITED)

INHIBIT: Restringe a instância da aplic. a não se tornar ACTIVE (caso seja STANDBY e a dual falhe) mudando o valor no arquivo de configuração dinâmica para INHIBITED.

UNINHIBIT: Libera a instância da aplic. para se tornar ACTIVE (caso seja STANDBY e a dual falhe) mudando o valor no arquivo de configuração dinâmica para UNINHIBITED.

GETPID: Solicita o PID de um processo específico da instância local da aplicação.

KILLPROC: Executa o comando Unix/Linux kill -0 sobre um processo específico de uma aplicação.

KILLRM: Executa o comando Unix/Linux kill -0 sobre o AppMgr que roda em um dado host.

QUERY: Solicita o estado da aplicação (das duas instâncias)

REBOOT: Fecha e recomeça a aplicação (exceto se ela for ACTIVE e não houver uma versão STANDBY)

SHUTDOWN-F: Fecha a instância da aplicação mesmo se ela for ACTIVE e a outra não for STANDBY.

DISABLE-F: Muda a instância da aplicação para o estado MAINTENANCE mesmo se ela for ACTIVE.

RESTART-F: Fecha e recomeça a instância da aplic. mesmo se ela for ACTIVE e não houver STANDBY.