Lucas do R. B. Brasilino da Silva

Aceleração HTTP: Um comparativo de performance entre as soluções Squid e Varnish

Monografia de Pós-Graduação "*Lato Sensu*" apresentada ao Departamento de Ciência da Computação para obtenção do título de Especialista em "Administração em Redes Linux"

Orientador
Prof. MSc. Joaquim Quinteiro Uchôa

Lavras
Minas Gerais - Brasil
2009

Lucas do R. B. Brasilino da Silva

Aceleração HTTP: Um comparativo de performance entre as soluções Squid e Varnish

Monografia de Pós-Graduação "*Lato Sensu*" apresentada ao Departamento de Ciência da Computação para obtenção do título de Especialista em "Administração em Redes Linux"

Aprovada em *Novembro de 2009*

Prof. Marluce Rodrigues Pereira

Prof. Sandro Melo

Prof. MSc. Joaquim Quinteiro Uchôa
(Orientador)

Lavras
Minas Gerais - Brasil
2009

Esse trabalho é dedicado a todos que contribuíram e contribuem para que a World Wide Web seja um vetor democrático de desenvolvimento social, econômico e cultural. É especialmente dedicado ao físico inglês Tim Berners-Lee, que há décadas vislumbrou um mundo sem barreiras geográficas.

Agradecimentos

Agradeço ao Professor Joaquim Quinteiro Uchôa por ter acolhido com entusiasmo essa monografia e por sua dedicação em orientá-la.

Agradeço à minha esposa Simone pelo apoio e incentivo constantes, pela preocupação no andamento dessa monografia, pelas revisões gramaticais e pelos mimos diários.

Agradeço aos colegas da Procuradoria Geral da República e da 4Linux pelas oportunidades de desenvolver trabalhos em áreas as quais me permitem crescer como profissonal.

Sumário

1 Introdução **1**

2 Aceleração HTTP **3**

2.1 Servidor *proxy/cache* . . . 3

2.2 Servidor *proxy/cache* reverso . . . 4

2.3 Hyper Text Transfer Protocol (HTTP) . . . 6

2.3.1 *Uniform Resource Locator* - URL . . . 6

2.3.2 Requisição e resposta . . . 8

2.4 *Headers* HTTP que implementam *caching* . . . 11

2.4.1 *Header* `Expires` . . . 11

2.4.2 *Header* `Cache-Control` . . . 12

2.4.3 *Header* `Last-Modified` . . . 13

2.4.4 *Header* `If-Modified-Since` . . . 13

2.5 Mecanismos de *caching* do HTTP . . . 14

2.5.1 Modelo baseado em expiração . . . 15

2.5.2 Modelo baseado em validação . . . 15

2.5.3 Heurística de *caching* . . . 16

2.6 Mecanismos de `caching` aplicados ao acelerador HTTP . . . 17

3 Principais Aplicativos para Aceleração HTTP **19**

3.1 Squid 19

3.2 Varnish 21

3.3 Outros aplicativos 23

3.3.1 Apache 23

3.3.2 Cherokee 23

3.3.3 Lighttpd 24

4 Metodologia Utilizada **25**

4.1 *Hardware* 26

4.1.1 *Hardware* cliente 26

4.1.2 *Hardware* servidor 27

4.2 Rede de interconexão 27

4.3 Sistema Operacional 28

4.3.1 Parametrizações 28

4.3.1.1 Fila de conexões pendentes 29

4.3.1.2 Tempo de espera para fechamento de conexão . 29

4.3.1.3 Portas TCP locais disponíveis 30

4.3.1.4 Máximo de descritores de arquivos por processo 30

4.3.1.5 Sumário de parametrizações 31

4.4 Ferramenta de medição de performance 31

4.4.1 Medição baseada em sessões 35

4.5 Preparação do ambiente de ensaios de carga 37

4.5.1 Servidor *Web* 37

4.5.1.1 Instalação do Apache 37

4.5.1.2 Página *Web* de prova 38

4.5.1.3 Configuração do Apache 39
4.5.2 Acelerador HTTP 40
4.5.2.1 Instalação do Varnish 40
4.5.2.2 Configuração do Varnish 40
4.5.2.3 Instalação do Squid 41
4.5.2.4 Configuração do Squid 42
4.5.3 Ferramenta de medição 43
4.5.3.1 Arquivo de *workload* de sessões 43
4.6 Sistemática dos ensaios de carga 44
4.6.1 Modos de operação do Apache 45

5 Resultados e Discussão 47
5.1 Pré-testes 47
5.1.1 Largura de banda da rede de interconexão 47
5.1.2 Performance do Apache sem acelerador 48
5.2 Resultados dos ensaios do Squid e Varnish 49
5.2.1 Respostas HTTP por segundo 49
5.2.2 Tempo médio de conclusão de respostas 50
5.2.2.1 Modo **CACHE_TOTAL** 50
5.2.2.2 Modo **CACHE_PARCIAL** 51
5.2.2.3 Modo **SEM_CACHE** 52
5.3 Discussão 53

6 Conclusão 55
6.1 Outros critérios de escolha 56
6.2 Trabalhos Futuros 56

A Arquivos de configuração **61**

A.1 Arquivo `/etc/sysctl.conf` 61

A.2 Arquivo `httpd.conf` 61

A.3 Arquivo `varnish.vcl` 63

A.4 Arquivo `squid.conf` 64

Lista de Figuras

2.1 Disposição de um *proxy/cache* 4

2.2 Disposição de um *proxy/cache* reverso 5

2.3 Disposição de um *acelerador HTTP* 6

2.4 Exemplo de URL 7

2.5 Formato de URL para *schema* `http` 7

2.6 Exemplo de requisição HTTP 8

2.7 Formato de uma requisição HTTP 9

2.8 Exemplo de resposta HTTP 10

2.9 Header `Expires` 12

2.10 Header `Cache-Control` 12

2.11 Header `If-Modified-Since` 14

4.1 Conversão de respostas por segundo para largura de banda 25

4.2 Arranjo para ensaios de carga 26

4.3 Saída do comando `mii-tool` 28

4.4 Consulta ao número máximo de descritores 31

4.5 Exemplo de saída do `httperf` 33

4.6 Exemplo de arquivo de sessão 36

4.7 Página *Web* de prova 39

4.8 Arquivo de *workload* utilizado nos ensaios 43

4.9 Inicialização do Apache no modo **CACHE_TOTAL** 46

5.1 Respostas por segundo do Apache 49

5.2 Respostas por segundo: Squid e Varnish 50

5.3 Tempo médio no modo **CACHE_TOTAL** 51

5.4 Tempo médio no modo **CACHE_PARCIAL** 52

5.5 Tempo médio no modo **SEM_CACHE** 53

Lista de Tabelas

2.1 Categorias de código de resposta 10
2.2 Código de repostas mais comuns 11
2.3 Diretivas do `Cache-Control` 13
3.1 Versões do Squid 20
4.1 Características do kernel quanto à operação em rede 28
4.2 Sumário de parametrizações 31
4.3 Arquivos que compõem a página *Web* de prova 38

Resumo

A utilização da *World Wide Web* cresce a cada dia. Com isso, a demanda de acesso a servidores de aplicação *Web* cresce em uma razão muito maior. Uma das formas de amenizar a carga sobre tais servidores é incluir uma camada de *caching*. Esse trabalho discute inicialmente como o protocolo HTTP suporta *Web caching* e como se comportam, em termos de performance, duas das soluções mais usadas de aceleração HTTP cujas licenças são aprovadas pela *Open Source Initiative*: Squid e Varnish.

Palavras-Chave: Hyper Text Transfer Protocol; HTTP; World Wide Web; Web; Performance; Proxy; Cache; Caching; Surrogate Server

Capítulo 1

Introdução

A *World Wide Web* desempenha um papel central na Internet. Esse é um fato natural pois ela impulsionou e difundiu o uso da *grande rede* entre os usuários não-técnicos.

Estudos recentes demonstram que os protocolos de rede que dão sustentação à *World Wide Web*, ou *Web* como é mais conhecida, são responsáveis por mais de 70% do tráfego global da Internet (RABINOVICH; SPATSCHEK, 2002). Esse percentual certamente aumenta diariamente ao se firmar a tendência do uso de aplicativos e sistemas baseados na *Web*, bem como no aparecimento de novos serviços neste ambiente.

A versão 2.0 da *Web* consolidou a Internet como uma plataforma, não mais apenas como um meio de comunicação. Desta forma, tornou-se base para novos tipos de interação entre pessoas: redes sociais, compartilhamento de áudio e vídeo, *weblogs* (*blogs*), *wikis* e *tagging* colaborativo. Estabeleceu-se um nível de interatividade nunca antes experimentado: *Web sites* como Orkut, Facebook, Del.icio.us, Twitter, Youtube e Last.fm, entre outros, demonstram esta tendência.

Um ponto crucial quanto à usabilidade da *Web* é a **latência**, que pode ser definida como o tempo entre a solicitação de uma página e sua total renderização no navegador e apresentação ao usuário (SIEMINSKI, 2005). Nesse tempo estão incluídas a busca e a transferência de objetos (páginas HTML, figuras, folhas de estilo, etc) através do protocolo HTTP - *Hyper Text Transfer Protocol*. Para minimizar esse tempo, o protocolo HTTP provê mecanismos de *caching*.

Estudos sobre o comportamento humano indicam que o homem executa, em média, 6 operações elementares por minuto. Pode-se inferir que uma latência

acima de 10 segundos rompe com essa média e pode causar sensação de desconforto e desinteresse (SIEMINSKI, 2005). Outros estudos, focados no comportamento durante o uso da *Web*, indicam que a latência máxima tolerável é de 8 segundos (ZONA RESEARCH, 1999) e quanto mais experiente menos paciente é o usuário.

Existem várias tecnologias envolvidas em minimizar a latência experimentada pelo usuário. Dentre elas, podem-se destacar servidores *proxy/cache* e servidores *surrogate* (ou *proxy/cache* reverso) (RABINOVICH; SPATSCHEK, 2002). Um acelerador HTTP é um caso particular desse último.

Dentre as soluções de aceleração HTTP distribuídas sob uma licença aprovada pela *Open Source Initiative*, certamente a mais conhecida e utilizada é o Squid. Por estar sendo desenvolvido há mais de uma década, ele detém uma grande base de usuários, além de ser disponível virtualmente em todos os sistemas operacionais *Unix-like* (Linux, Solaris, BSD's, etc) e foi, há alguns anos, portado para o kernel do Windows NT. Grandes *sites*, como Wikipedia, implementam-o como acelerador HTTP com sucesso.

Em 2006 foi lançada uma nova solução chamada Varnish. Afirmando que o Squid tem um *design* antigo e que foi *adaptado* para ser um *proxy/cache* reverso, o Varnish foi pensado desde o início para ser um acelerador HTTP e para usar ao máximo recursos otimizados presentes nos S.O's *Unix-like* modernos. Portanto, os desenvolvedores do Varnish alegam que sua performance é muito superior à do Squid.

Este trabalho tem como objetivo o estudo comparativo em termos de performance entre o Squid e o Varnish como aceleradores HTTP. O Capítulo 2 é dedicado a uma revisão do protocolo HTTP e suas funcionalidades de *caching*. O Capítulo 3 apresenta os principais aplicativos que provêem funcionalidades de aceleração HTTP, com ênfase no Squid e Varnish. O Capítulo 4 discorre sobre a metodologia e ferramentas utilizadas na produção dos dados comparativos entre as soluções que são tema deste trabalho. O Capítulo 5 demonstra os resultados experimentais. O Capítulo 6 apresenta as conclusões do trabalho e propõe trabalhos futuros.

Capítulo 2

Aceleração HTTP

A **aceleração HTTP** é provida por um ou vários servidores *proxy/cache* implementados como *proxy/cache* reverso (servidor *surrogate*) com a característica de estar no mesmo *hardware* servidor ou na mesma LAN que o servidor *Web* de origem[1].

2.1 Servidor *proxy/cache*

Proxy/cache é o servidor que busca nos servidores *Web* de origem (ou em outros *proxy/caches*) os objetos como se fossem requisitados pelos clientes. Tipicamente está "próximo", na mesma LAN ou a poucos *hop*'s, dos requisitantes. É também conhecido como *forward proxy/cache* (NAGARAJ, 2004). A Figura 2.1 ilustra sua implementação.

Normalmente, os clientes HTTP necessitam ser configurados para utilizarem o servidor *proxy/cache*. Alternativamente há a possibilidade da implantação no modo *interception proxy* (RABINOVICH; SPATSCHEK, 2002). Nessa modalidade, um equipamento de rede (roteador, *firewall*, etc) intercepta os pacotes TCP cuja porta de destino seja 80 e os divergem para o *proxy/cache*.

Em linhas gerais, se adequadamente configurado, o *proxy/cache* melhora a performance:

[1] *Servidor de origem* é o termo aplicado aos servidores que mantém a cópia original do objeto *Web*.

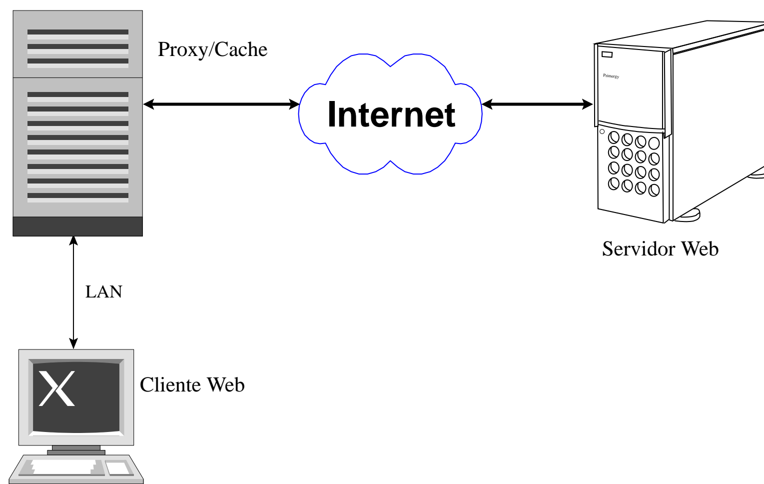

Figura 2.1: Disposição de um *proxy/cache*

- Reduzindo a latência percebida pelo usuário ao acessar conteúdos *Web*. Esta redução se dá proporcionalmente ao *hit rate*[2];

- Racionalizando o uso da banda passante na rede, pois os objetos servidos pelo *proxy/cache* atravessa uma pequena parte da rede;

- Minimizando a demanda de serviço ao servidor *Web* de origem, diminuindo sua carga de processamento;

- Funcionando como um *cache* de conexões TCP.

2.2 Servidor *proxy/cache* reverso

O servidor *proxy/cache* reverso pode ser entendido como uma variante da implementação descrita na Seção 2.1. Nessa modalidade o servidor está próximo à origem do conteúdo e não à sua destinação, conforme Figura 2.2.

Um acelerador HTTP é um caso particular de *proxy/cache* reverso, onde se encontra diretamente conectado através de uma rede de banda idealmente infinita ao servidor *Web* de origem. A Figura 2.3 ilustra-o.

[2]Taxa definida pela quantidade de objetos servidos pelo *proxy/cache* sobre o total requisitado

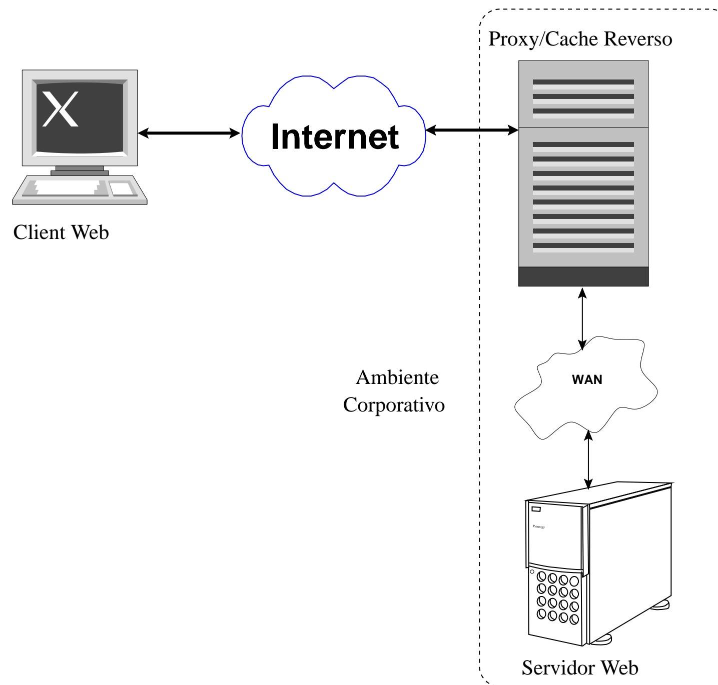

Figura 2.2: Disposição de um *proxy/cache* reverso

As vantagens do uso de um acelerador HTTP são as mesmas já citadas na seção anterior, acrescentando-se:

- Mimetiza o comportamento da origem do conteúdo;
- Configura os *headers* HTTP de forma conveniente nas respostas dos objetos servidos a partir do *cache*;
- Mantém uma forte consistência em seu *cache* local;
- Pode ser usado como um *gateway* de conexões não-persistentes para persistentes, diminuindo o número de conexões abertas ao servidor *Web* de origem;
- Pode ser usado como um "tradutor" de versões de protocolos. (Ex: HTTP/1.0 → HTTP/1.1).

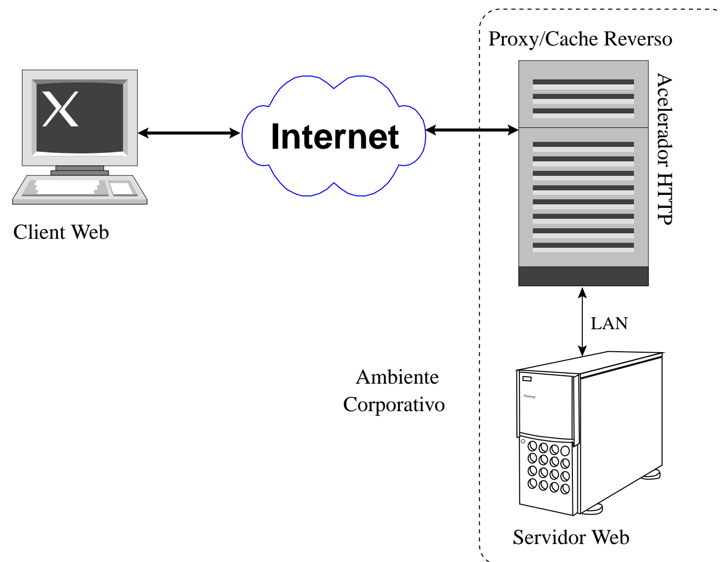

Figura 2.3: Disposição de um *acelerador HTTP*

2.3 Hyper Text Transfer Protocol (HTTP)

O HTTP é um protocolo baseado em requisição e resposta. Foi inicialmente definido como HTTP/0.9 (BERNERS-LEE, 1994). A versão HTTP/1.0 foi então definida pela RFC 1945 (BERNERS-LEE; FIELDING; FRYSTYK, 1996) para expandir suas possibilidades. Nessa versão foram introduzidas informações que permitiam *caching* de conteúdo *Web*. Com o passar do tempo e com o uso, tais informações mostraram-se limitadas.

Em junho de 1999, foi publicada a RFC 2616 (FIELDING *et al.*, 1999) que define o HTTP/1.1. Essa nova versão em muito ampliou as funcionalidades do protocolo. Um capítulo foi dedicado a *web caching*, introduzindo a idéia de mecanismos baseados em **expiração** e em **validação**.

A partir deste ponto do texto será abordado a versão 1.1 do protocolo HTTP.

2.3.1 *Uniform Resource Locator* - URL

Uniform Resource Locator (BERNERS-LEE; MASINTER; MCCAHILL, 1994) é um conceito fundamental na *Web*. Sua função é endereçar cada objeto disponível

na Internet em uma relação de N para 1: o mesmo objeto pode ser localizado por URL's diferentes (RABINOVICH; SPATSCHEK, 2002). Na Figura 2.4 tem-se um exemplo.

Basicamente uma URL é definida na forma:

```
<scheme>:<scheme-specific-part>
```

O *scheme* que se aplica a este texto é o `http` (Figura 2.5), que é dividido em:

- `host`: Nome do servidor ou endereço IP onde encontra-se o objeto;
- `port`: Porta TCP utilizada para conexão. Se omitida deve-se utilizar as portas-padrão 80 para *schema* `http` e 443 para a variante segura `https`;
- `path`: Caminho no servidor, a partir de um raiz, que o objeto deve ser buscado;
- `searchpart`: Uma sequencia pré-determinada de valores para serem referenciadas como argumentos. Também conhecida como *query string*. É opcional.

Normalmente, o servidor *proxy/cache* utiliza a URL como argumento principal de consulta para determinar se o objeto já foi armazenado localmente em sua área de *caching*. Muitos utilizam uma técnica de *hashing* para que a localização seja eficiente. Pode-se deduzir que, como um objeto por ser referenciado por várias URL's, o *proxy/cache* armazene cópias idênticas desperdiçando espaço. Porém, como o tamanho médio de um objeto é de 4Kb, o armazenamento de algumas cópias não chega a ser preocupante, dado que é comum áreas de armazenamento de centenas de Megabytes a alguns Gigabytes.

```
http://arl.ginux.ufla.br/files/arl_logo.gif
```

Figura 2.4: Exemplo de URL

```
http://<host>:<port>/<path>?<searchpart>
```

Figura 2.5: Formato de URL para *schema* `http`

```
GET /files/arl_logo.gif HTTP/1.1
Host: arl.ginux.ufla.br
User-Agent: BrasilinoBrowser/1.0
Accept: text/html, image/*
Connection: close
```

Figura 2.6: Exemplo de requisição HTTP

2.3.2 Requisição e resposta

Requisição é a ação promovida por um cliente *Web* para buscar um conteúdo (objeto) disponível em um servidor. Como seu nome sugere, o protocolo HTTP preocupa-se com **transferência** dos dados que compõem o objeto, tanto na direção cliente → servidor quanto na direção servidor → cliente. Uma categoria de clientes conhecida como navegadores gráficos também acumulam a função de renderizar o objeto ao usuário.

Um cliente, ao solicitar a busca do objeto identificado pela URL ilustrada na Figura 2.4, efetuará a resolução DNS para determinar o IP do servidor `arl.ginux.ufla.br` (a parte `host` da URL), se conectará à porta 80 desse e efetuará a requisição conforme Figura 2.6.

Essa requisição segue o formato padrão ilustrado na Figura 2.7. A primeira linha é a *request line* (linha de requisição), que contém o método, a parte *path* da URL e a versão do protocolo. Os métodos são GET, HEAD, OPTIONS, POST, PUT, DELETE e TRACE.

O método GET é o método mais comum e responsável por mais de 97% das requisições. É usado para buscar o objeto especificado na parte *path*. Na linha de requisição `GET /files/arl_logo.gif HTTP/1.1` o arquivo GIF `arl_logo.gif` será buscado.

As linhas subsequentes são os *headers*, ou cabeçalhos. Eles qualificam a requisição e a resposta provendo informações necessárias como qual codificação e/ou língua é aceita pelo cliente, quais objetos são aceitos para transferência, dados sobre autenticação, sobre *caching*, entre outros.

Os *headers* são divididos em:

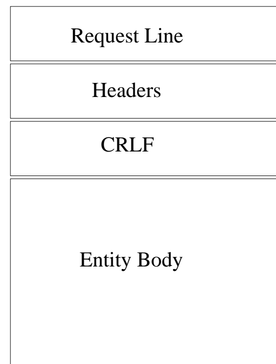

Figura 2.7: Formato de uma requisição HTTP

- *General headers*: São aplicáveis tanto em requisições quanto em respostas;
- *Request headers*: São aplicáveis apenas em requisições;
- *Response headers*: São aplicáveis apenas em respostas;
- *Entity headers*: Descreve os dados contidos na requisição ou na resposta.

Na requisição exemplificada na Figura 2.6, o `Host`, `User-Agent` e `Accept` são *request headers*. O `Connection` é um *general header*. Para efeito de simplificação, todos os *headers* de uma requisição são chamados de *request headers*.

Após os *headers*, seguem-se dois caracteres que sinalizam o fim da parte inicial da requisição: *carriage return* (CR) e *line feed* (LF), cujos códigos ASCII são respectivamente 13 e 10.

Entity body compreende os dados do objeto que está sendo transferido. O método POST é a única requisição que envia dados ao servidor. Nesse caso, o *entity body* é preenchido com os dados a serem enviados. Na requisição exemplificada não há *entity headers*, uma vez que não há *entity body*.

A resposta HTTP tem um formato similar à requisição. A diferença básica é que na primeira linha é colocada uma *status line*, ou linha de *status*. Na Figura 2.8 demonstra a resposta do servidor *Web* `arl.ginux.ufla.br` quando da requisição do arquivo `arl_logo.gif` conforme Figura 2.6.

A *status line* é formada pela versão do protocolo utilizado na resposta, um código numérico do *status* (*status code*) e sua descrição textual. O código numérico é

```
HTTP/1.1 200 OK
Date: Sat, 14 Feb 2009 11:35:05 GMT
Server: Apache
Last-Modified: Tue, 27 Nov 2007 20:35:25 GMT
ETag: "40d56-129f-43fef05453d40"
Accept-Ranges: bytes
Content-Length: 4767
Cache-Control: max-age=1209600
Expires: Sat, 28 Feb 2009 11:35:05 GMT
Connection: close
Content-Type: image/gif

GIF89a ^@Z^@ç^@^@^B^B^B^B<82>î<99>h...
```

Figura 2.8: Exemplo de resposta HTTP

agrupado em 5 categorias separadas pelas centenas conforme Tabela 2.1, enquanto os códigos mais usuais estão sumarizados na Tabela 2.2.

Nessa resposta, os *headers* `Date`, `Cache-Control` e `Connection` são *general headers*, os `Server` e `ETag` são *response headers* e por fim `Last-Modified`, `Content-Length`, `Expires` e `Content-Type` são *entity headers*. Coloquialmente todos os *headers* contidos em uma resposta são chamados de *response headers*.

A partir da linha em branco gerada pelos dois caracteres CR e LF, inicia-se a transferência dos *bytes* referente ao arquivo solicitado que, por economia de espaço, apenas os primeiros estão demonstrados na Figura 2.8.

Tabela 2.1: Categorias de código de resposta

Código do *status*	Categoria de resposta
1xx	Informação
2xx	Sucesso
3xx	Redirecionamento
4xx	Erro do cliente
5xx	Erro do servidor

Tabela 2.2: Código de repostas mais comuns

Código de *status*	Descrição textual
200	OK
206	Partial Content
301	Moved Permanently
302	Found
304	Not Modified
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
503	Service Unaviable

Um cliente HTTP de posse destas informações salva o arquivo `arl_logo.gif` em um diretório local. Caso seja gráfico, renderiza-o ao cliente.

Esse texto foca no uso dos *headers* `Expires`, `Cache-Control`, `Last-Modified` e `If-Modified-Since`. Os dois primeiros são os mais importantes pois implementam os mecanismos de *caching* do HTTP.

2.4 *Headers* HTTP que implementam *caching*

2.4.1 *Header* `Expires`

`Expires` é um *response header* que foi introduzido na versão 1.0 do protocolo HTTP (BERNERS-LEE; FIELDING; FRYSTYK, 1996) como um primeiro esforço para estabelecer a funcionalidade de *caching*. Seu valor deve ser a data de expiração do objeto, ou seja, até quando ele é válido. Portanto, `Expires` define o *tempo absoluto* da expiração.

No exemplo ilustrado na Figura 2.9 o objeto foi buscado no dia 27 de fevereiro de 2009, uma sexta feira, às 21:13 no horário de *Greenwich* (*header* `Date`) porém será válido até 17 de janeiro de 2038 às 19:14 também no horário de *Greenwich*. Até essa data, o servidor *proxy/cache* servirá o objeto diretamente ao cliente, não buscando-o no servidor *Web* de origem. Um tempo no passado informa que o objeto já está expirado, necessitando que seja requisitado novamente em um novo acesso. Essa é uma forma que os servidores *Web* de origem usam para forçar que um objeto não seja cacheado.

```
HTTP/1.0 200 OK
Server: Apache
Content-Type: image/gif
Date: Fri, 27 Feb 2009 21:13:45 GMT
Last-Modified: Wed, 07 Jun 2006 19:50:30 GMT
Expires: Sun, 17 Jan 2038 19:14:07 GMT
Connection: Keep-Alive
```

Figura 2.9: Header `Expires`

```
HTTP/1.1 200 OK
Server: Apache
Content-Type: image/gif
Date: Fri, 27 Feb 2009 21:13:45 GMT
Cache-Control: max-age=3600, public
Last-Modified: Wed, 07 Jun 2006 19:50:30 GMT
Connection: Keep-Alive
```

Figura 2.10: Header `Cache-Control`

2.4.2 *Header* `Cache-Control`

Para aumentar a flexibilidade nos mecanismos de *caching* foi introduzido, no HTTP versão 1.1, o *header* `Cache-Control`. Esse *header* é mais completo e mais complexo. Por ser um *general header*, ele pode ser usado tanto em uma requisição quanto em uma resposta. Porém este texto trata-o apenas quanto ao uso em uma resposta. O valor pode ser uma ou mais **diretivas** separadas por vírgula, como exemplificado na Figura 2.10. A Tabela 2.3 sumariza as diretivas desse *header*.

A diretiva `max-age` define o *tempo relativo* em segundos da expiração tendo como base o momento (data e hora) da requisição. A resposta presente na Figura 2.10 está definindo que a expiração do objeto acontecerá em 3600 segundos.

Em uma resposta que estejam presentes tanto o `Expires` quanto o `Cache-Control`, esse último prevalecerá.

Tabela 2.3: Diretivas do `Cache-Control`

Diretiva	Valor	Significado
no-cache	nenhum	A resposta não pode ser cacheada
no-store	nenhum	A resposta não pode ser armazenada no cliente (*proxy/cache* ou navegador)
private	nenhum	A resposta só poderá ser reutilizada pelo cliente que o requisitou inicialmente
public	nenhum	A resposta poderá ser cacheada e compartilhada entre clientes diferentes
must-revalidade	nenhum	A resposta deverá sempre ser validada, geralmente através de uma requisição condicional
proxy-revalidation	nenhum	O mesmo que `must-revalidade` porém aplica-se apenas a *proxy/caches*
max-age	segundos	A resposta será válida durante este tempo
s-maxage	segundos	O mesmo que `max-age` porém aplica-se apenas a *proxy/caches*
no-transform	nenhum	A resposta deverá ser idêntica à fornecida pelo servidor de origem

2.4.3 *Header* `Last-Modified`

`Last-Modified` é um *entity header* que indica a data da última modificação do objeto.

Os objetos cujas repostas estão ilustradas nas Figuras 2.9 e 2.10 tiveram sua última modificação (de conteúdo) datada de 7 de junho de 2006 às 19:50, horário de *Greenwich*

2.4.4 *Header* `If-Modified-Since`

`If-Modified-Since` é um *request header* utilizado para questionar ao servidor, tanto *Web* de origem quanto *proxy/cache*, se o objeto foi modificado desde uma data específica. Essa data é explicitada no valor do *header*.

A Figura 2.11 é igual à requisição da Figura 2.6, porém adicionando-se o questionamento através do `If-Modified-Since`.

```
GET /files/arl_logo.gif HTTP/1.1
Host: arl.ginux.ufla.br
User-Agent: BrasilinoBrowser/1.0
If-Modified-Since: Fri, 27 Feb 2009 10:00:00 GMT
Accept: text/html, image/*
Connection: close
```

Figura 2.11: Header `If-Modified-Since`

2.5 Mecanismos de *caching* do HTTP

Caching não teria utilidade se não melhorasse de forma significativa a performance minimizando a latência experimentada pelo usuário. Seu objetivo é, em alguns casos, eliminar a necessidade do envio de requisições e, em outros, eliminar a necessidade do objeto *Web* ser transferido por completo (NAGARAJ, 2004).

O protocolo HTTP baseia-se no conceito de *time to live* (TTL). *Time to live*, ou *tempo de vida*, indica a duração tempo que um objeto é **válido**. Quando um objeto está cacheado por um tempo maior que seu TTL ele é dito **inválido**. O termo **não-expirado** também é empregado como sinônimo de *válido*, enquanto o termo **expirado** é empregado como sinônimo de *inválido*. Esse conceito também é utilizado em outras tecnologias: DNS por exemplo.

As requisições a um objeto válido são satisfeitas pelo *proxy/cache*. Quando o objeto tornar-se inválido, o *proxy/cache* deverá ou buscá-lo novamente ou revalidá-lo através de uma *requisição condicional* (RABINOVICH; SPATSCHEK, 2002).

Em uma requisição condicional, o cliente HTTP especifica certas condições através de seus *request headers*. O servidor, tanto um *proxy/server* quanto um *Web* de origem, avalia a condição e se ela for verdadeira responderá com "`200 OK`" acrescido do conteúdo do objeto no corpo da resposta, ou seja, uma resposta como se a requisição não fosse condicional. Caso a condição seja falsa, o servidor responde com uma *status line* "`304 Not Modified`".

O comportamento de um *proxy/cache*, quanto ao objeto inválido, depende do modelo de *caching* aplicado pelo servidor *Web* de origem. Esses modelos são definidos na RFC2616 (FIELDING *et al.*, 1999) como *modelo baseado em expiração* e *modelo baseado em validação*.

2.5.1 Modelo baseado em expiração

O modelo de *caching* baseado em expiração tem como objetivo evitar requisições ao servidor *Web* de origem, diminuindo sua carga de processamento e a latência da rede (FIELDING *et al.*, 1999).

Utilizando-se o *header* `Expires` com uma data no futuro ou o `Cache-Control` com a diretiva `max-age`, o servidor *Web* de origem sinaliza ao *proxy/cache* que um dado objeto é válido por um determinado período de tempo, ou seja, informa seu TTL.

Esse modelo é muito aplicado em conteúdos estáticos como figuras, vídeos, arquivos em geral (PDF, ODP, ODT, etc), folhas de estilos (CSS), código javascripts, páginas HTML estáticas, etc. Pode ser também aplicado a conteúdos que tendam a ser dinâmicos, como uma página com sumário de notícias, desde que o tempo de expiração seja curto, tipicamente na ordem de minutos.

2.5.2 Modelo baseado em validação

Esse modelo baseia-se na possibilidade de se efetuar uma requisição condicional ao servidor e tem como objetivo minimizar a largura de banda de rede a ser utilizada na busca de um objeto (FIELDING *et al.*, 1999).

Quando o *proxy/cache* tem armazenado um objeto inválido ele realiza uma requisição condicional adicionando o *header* `If-Modified-Since` com a data recebida através do *header* `Last-Modified` de quando o objeto foi buscado inicialmente. Existem duas possibilidades de resposta:

1. Se a condição for verdadeira, ou seja, o objeto foi modificado desde a data e hora indicadas no `If-Modified-Since`, o servidor *Web* de origem:
   (a) responde com a linha de status “200 OK”;
   (b) acrescenta o *header* `Last-Modified` atualizando no *proxy/cache* o tempo da última modificação;

   (c) possivelmente informa uma nova data de expiração (ou seja, um novo TTL), através do *header* `Expires` ou `Cache-Control`.

2. Se a condição for falsa, ou seja, o objeto **não** foi modificado desde a data e hora indicadas no `If-Modified-Since`, o servidor *Web* de origem:

   (a) responde com a linha de *status* “304 Not Modified”;

   (b) possivelmente adiciona uma nova data de expiração.

Se ocorrer a segunda possibilidade, o objeto será revalidado pelo *proxy/cache*. Como não haverá a transferência integral do objeto, consequentemente haverá uma economia significativa na largura de banda da rede.

É possível forçar que o objeto **sempre** seja revalidado. Basta o servidor *Web* de origem adicionar `Cache-Control: max-age=0, must-revalidade` em sua resposta.

O modelo de validação é muito utilizado em informações onde há a necessidade tempestiva de se acessar a última versão do conteúdo disponível, como por exemplo: notícias de última hora, valores de ações, placares de jogos, páginas HTML dinâmicas, etc.

2.5.3 Heurística de *caching*

Como os servidores *Web* de origem nem sempre provêem informações que possam sugerir o uso de algum modelo de *caching*, os *proxy/caches* aplicam algoritmos para determinar se um objeto é válido ou não. Tais algoritmos não são padronizados pela RFC2616 e dependem da implementação de cada *proxy/cache*.

Geralmente objetos serão considerados não-cacheáveis se em suas respectivas URL's (RABINOVICH; SPATSCHEK, 2002):

- Contiverem o caracter `?`, onde presume-se que também contenham uma `searchpart`;
- Contiverem `cgi-bin` como parte de seu `path`;
- Não utilizarem os métodos `GET` e `HEAD`;
- Contiverem alguns *headers* como o `Authorization`;

- Contiverem o *header* `Cookie` em sua requisição ou *Set-Cookie* em sua resposta.

Se a resposta de um objeto contiver apenas o *header* `Last-Modified`, o *proxy/cache* calcula a idade do objeto pela equação:

```
age = Date - (Last-Modified)
```

E considera o objeto válido por uma fração do valor calculado em `age`. No Squid (THE SQUID TEAM, 2009), por exemplo, esta fração por padrão é de 20%.

2.6 Mecanismos de `caching` aplicados ao acelerador HTTP

Conforme visto na Figura 2.3 e definido na Seção 2.2, o acelerador HTTP é um servidor *proxy/cache* reverso que está próximo ao servidor *Web* de origem, geralmente conectado por uma LAN ou sendo executado no mesmo *hardware*.

É importante frisar que, normalmente, os servidores *Web* de origem são servidores de aplicação em geral conectados a servidores de banco de dados e outros recursos. Tais servidores, ao serem muito requisitados, necessitam de grande poder computacional.

A economia de largura de banda propiciada pelo modelo baseado em validação não é impactante nem deve ser levada em consideração[3]. Por outro lado, a diminuição da carga e o número de requisições propiciados pelo modelo baseado em expiração é considerável.

Portanto, uma política plausível de implantação de um acelerador HTTP é a de escolher o mecanismo de *caching* baseado em expiração onde conteúdos estáticos podem ser cacheados por dias, enquanto os que mudam com frequência podem expirar em poucos segundos. Com isso, o acelerador terá uma rápida inconsistência em seu *cache*.

Outra política muito comum é a de se usar o modelo baseado em validação apenas nos conteúdos dinâmicos para que, ao mudarem, sejam imediatamente buscados, enquanto aos conteúdos estáticos é aplicado o modelo baseado em expiração. Dessa forma, o acelerador terá uma forte consistência em seu *cache*, visto que fará mais requisições condicionais ao servidor *Web* de origem.

[3] Atualmente as LAN's são implantadas com no mínimo 100Mb/s de largura de banda

Capítulo 3

Principais Aplicativos para Aceleração HTTP

3.1 Squid

Em 1994, o *Internet Research Task Force on Resource Discovery* (IRTF-RD) iniciou um projeto para o desenvolvimento de uma ferramenta integrada para buscar, extrair, organizar, pesquisar, cachear e replicar informações baseadas em tecnologia *Web*. Esse projeto foi batizado de Harvest (WESSELS, 2004). Seu módulo de *caching* tornou-se bastante popular.

No início de 1996, após vários pesquisadores saírem do IRTF-RD, um deles, Duane Wessels, integrou o *National Laboratory for Applied Network Research* (NLANR). Tendo em mãos o código do módulo de *caching* do Harvest, ele o renomeou para Squid e o disponibilizou sob licença *GNU General Public License* (GPL). Em 2000 o NLANR deixou de custear seu desenvolvimento e, desde então, um grupo de voluntários continua a liberar novas versões sempre adicionando funcionalidades (WESSELS, 2004).

O Squid (THE SQUID TEAM, 2009) é um servidor *proxy/cache* estável, rápido e flexível. Possui grande número de usuários ao redor do mundo. Conta com vasta documentação disponível na Internet formada de tutoriais nas mais diversas línguas e um *site* bem estruturado contendo, entre vários recursos, um *wiki*[1] muito detalhado.

[1] `http://wiki.squid-cache.org`

Além de ser um *forward proxy/cache*, topologia discutida na Seção 2.1, o Squid foi adaptado e melhorado para ser um *proxy/cache* reverso, topologia discutida na Seção 2.2.

Existem duas famílias de versões paralelas do Squid: versão 2 e versão 3. A versão 2 (Squid-2) é a continuação do desenvolvimento da versão 1. Já a versão 3 (Squid-3) é uma reorganização e reescrita total do código-fonte, além da adição de suporte a alguns protocolos como o *Edge Side Includes* (ESI) (TSIMELZON, 2001) e o *Internet Content Adaptation Protocol* (ICAP) (ELSON; CERPA, 2003).

Na Tabela 3.1 estão as versões disponíveis no tempo da elaboração deste trabalho.

Tabela 3.1: Versões do Squid

Família de versões	Última versão disponível
Squid-2	2.7.STABLE7
Squid-3	3.0.STABLE18

Este trabalho utiliza a versão 2.7.STABLE7.

O Squid suporta algumas funcionalidades que podem ser utilizadas quando está operando como um acelerador HTTP a fim de ampliar as possibilidades de sua implantação. Destacam-se:

- **Internet Cache Protocol (ICP):**
  Alguns servidores *Web* de origem, por exemplo o Zope, suportam ICP. Apesar de ser um protocolo simples e não confiável, caso seja implementado com cuidado pode ser útil.

- **Hyper Text Caching Protocol (HTCP):**
  É uma versão melhorada do ICP.

- **Balanceamento de carga:**
  Distribui a carga de requisições não satisfeitas pelo *proxy/cache* a um conjunto (*pool*) de servidores *Web*. A escolha de um servidor entre o conjunto é feita através de *round-robin* com possibilidade de definição de servidor preferencial.

- **Cache Array Routing Protocol (CARP):**
  É outra forma de distribuição de carga. Baseia-se na escolha de um servidor

Web dentre os vários através de um *hash* da URL requisitada. Assim, a requisição de um objeto sempre será repassada ao mesmo servidor *Web* de origem.

- **Redirecionadores externos:**
  Suportam o uso de programa externo para reescrita de URL.

- **Estatísticas em tempo real:**
  Através do utilitário de linha de comando `squidclient` é possível obter diversas informações em tempo real.

A implementação na prática de ICP e HTCP tem-se mostrado ineficiente. É quase nulo o número de servidores *Web* de origem que suportam esses protocolos. Por serem baseados em *pergunta e resposta* podem inserir tempos de espera da resposta HTTP, o que certamente acarretará aumento da latência.

3.2 Varnish

O Varnish (KAMP, 2009) é um *daemon* servidor rico em funcionalidades e que foi desenvolvido a partir do zero com o objetivo claro de ser um poderoso *proxy/cache* reverso. A primeira versão estável foi liberada em 2006 e nomeada de 1.0.

Seu arquiteto e principal desenvolvedor, Poul-Henning Kamp, é há muito tempo parte integrante do grupo de desenvolvedores do *kernel* do FreeBSD. De posse de seus conhecimentos intrísecos ao núcleo de sistemas operacionais *Unix-like* modernos, ele identificou que o *design* do Squid é antigo. Ao iniciar a escrita do Varnish, preocupou-se em utilizar de forma otimizada os recursos dos S.O's modernos, como *memory mapped file* e comunicação entre processos *shared memory*.

O *Website* do projeto é organizado na forma de um *site trac*[2], contendo alguns tutoriais, documentações, lista de perguntas frequêntes, um *timeline* das modificações realizadas no código-fonte e uma interface *Web* para o Subversion, sendo possível o acesso ao código-fonte em desenvolvimento.

Em 15 de outubro de 2008, foi lançada a versão 2.0 incorporando várias modificações e aperfeiçoamentos, principalmente quanto à estabilidade. No tempo da escrita deste trabalho a versão disponível é a 2.0.4.

[2]http://trac.edgewall.org/

Entre as funcionalidades disponíveis no Varnish, destacam-se:

- **Varnish Configuration Language (VCL):**
  É uma linguagem de domínio específico (DSL - *Domain Specific Language*) utilizada para configurar o Varnish. Tem a sintaxe similar ao Perl ou C e é extremamente flexível. Permite a definição do comportamento desde a requisição do objeto até sua entrega ao cliente.
  A VCL permite uma sub-funcionalidade: a introdução de códigos em C para uma customização granular.

- **Edge Side Includes (ESI):**
  O protocolo ESI permite a separação de uma página *Web* em pequenos pedaços, que podem ser armazenados em *cache* individualmente.

- **Reescrita e redirecionamento de URL:**
  Ao contrário do Squid que precisa de um código externo para a reescrita da URL, o próprio Varnish efetua este procedimento através de funções disponíveis na VCL.

- **Gerência em tempo real:**
  Através de uma porta TCP é possível a realização de gerência em tempo real. A conexão à essa porta é via utilitário `telnet`.

- **Estatísticas e registros em tempo real:**
  Através de uma comunicação entre processos *shared memory*, os utilitários `varnishlog` e `varnishhist` registram os *logs* de acesso e geram estatísticas, respectivamente.

- **Balanceamento de carga:**
  Atualmente são suportados dois algoritmos: *round-robin* e *random*.

- **Utilitário *Web* para administração:**
  Está em desenvolvimento o suporte a um utilitário baseado na *Web* para administração e configuração em tempo real do *daemon*.

Contudo, o Varnish implementa poucos aspectos da RFC 2616. O próprio autor afirma que o Varnish não é um *cache* nos termos dessa RFC. Ele prefere pensá-lo como um "*servidor web que obtém seus conteúdos através de HTTP*"[3]. Porém, as funcionalidades não implementadas por padrão podem ser incluídas através de um arquivo VCL.

[3] `http://projects.linpro.no/pipermail/varnish-misc/2007-November/001173.html`

3.3 Outros aplicativos

Existem algumas outras soluções que podem ser utilizadas com o objetivo de acelerar o acesso a um *site* ou aplicação *Web*, porém, nem todos são classificados como aceleradores HTTP.

3.3.1 Apache

Apache (THE APACHE SOFTWARE FOUNDATION, 2009) é um servidor *Web* bastante popular. Segundo o site Netcraft (NETCRAFT, 2009), de acordo com a pesquisa realizada em fevereiro de 2009, ele é utilizado cerca de 50% dos *sites* ao redor do mundo.

Um dos fatores que promoveram o uso do Apache foi sua estrutura modularizada: vários módulos com os mais diversos propósitos foram escritos. Dentre eles, existem três que combinados podem agregar a funcionalidade de *proxy/cache* reverso: `mod_proxy`, `mod_cache` e `mod_disk_cache`.

Os módulos de *caching* foram declarados estáveis e funcionais com o advento da versão 2.2 do Apache. Porém, o autor desse trabalho teve experiências anteriores negativas na implementação dos módulos acima listados em ambiente de produção. Pode ser citado, a fim de exemplificação, a impossibilidade de ser invalidado (excluído) um objeto em *cache*.

3.3.2 Cherokee

Cherokee (ORTEGA, 2009) é um servidor com um *design* moderno e eficiente. Seu objetivo é obter o máximo em performance do *hardware* através de *multithreading*. *Benchmarks* iniciais pontam que ele chega a ser 20 vezes mais rápido que o Apache. Segundo avaliações realizadas por seu autor [4], o Cherokee é o servidor *Web* mais rápido disponível sob uma licença aprovada pela *Open Source Initiative*.

Um dos grandes atrativos deste servidor é sua forma de configuração. O *cherokee-admin* é um aplicativo *Web* que configura todo e qualquer funcionamento do Cherokee, de forma simples e centralizada.

[4] `http://www.alobbs.com/1345/Cherokee_0_8_1_benchmark.html`

O *handler* `HTTP reverse proxy` faz com que o Cherokee atue como um *proxy* reverso com suporte a balanceamento de carga. Porém o mesmo não atua como *cache*.

3.3.3 Lighttpd

O Lighttpd (THE LIGHTTPD TEAM, 2009) disputa junto ao Cherokee o título de servidor *Web* mais rápido do mundo. É mais estável que seu competidor e portanto pode ser usado em ambiente de produção. A página *Web* do projeto afirma que ele é utilizado por vários *sites* como Youtube, Wikipedia e Meebo.

Também dispõe de suporte a *proxy* reverso e a *caching*, ambos através de módulos. Porém, conforme a própria página de documentação do módulo de *caching*, essa implementação visa armazenar em *cache* os objetos que seriam servidos pelo próprio Lighttpd e não os provenientes de um outro servidor *Web*.

Capítulo 4

Metodologia Utilizada

Existem algumas métricas que possibilitam investigar se um *software* utiliza recursos, funcionalidades e algoritmos de forma otimizada. Por exemplo, o número necessário de pulsos de *clock* para o processador finalizar uma tarefa pode ser usado como uma métrica. Outro exemplo: o número de trocas de modo entre *user mode* e *kernel mode* pode também ser utilizado como métrica.

Nesse trabalho, as métricas aplicadas aos aceleradores HTTP são o número de respostas HTTP por segundo e o tempo médio de conclusão de respostas.

A primeira métrica é importante para avaliar a taxa de respostas HTTP que o acelerador consegue manter. Essa taxa é normalmente igual ao número de requisições por segundo suportada pelo acelerador, definindo o *throughput* do mesmo (WESSELS, 2001).

Normalmente, o *throughput* é apresentado como largura de banda, geralmente em megabits por segundo (Mb/s). Para realizar a conversão de respostas HTTP por segundo para largura de banda aplica-se a equação da Figura 4.1, onde *tamanho medio* é o tamanho médio das requisições em kilobytes (KB).

$$largura_de_banda = \frac{respostas_por_segundo \cdot tamanho_medio}{1024}\ [em\ Mb/s]$$

Figura 4.1: Conversão de respostas por segundo para largura de banda

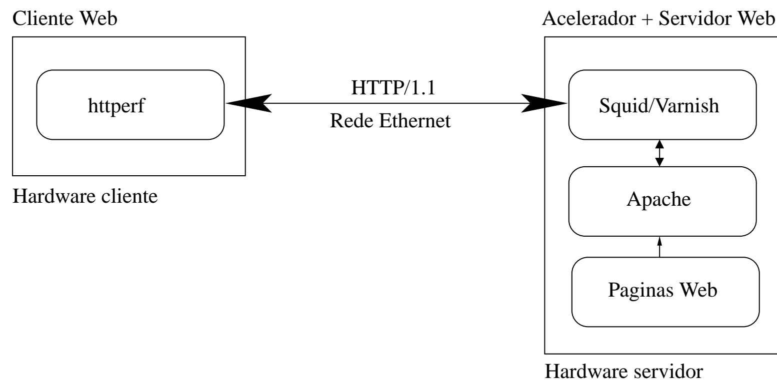

Figura 4.2: Arranjo para ensaios de carga

A segunda métrica - tempo de conclusão de respostas - sugere a capacidade de redução da latência experimentada pelo usuário. Esse tempo tende a aumentar com o aumento do número de requisições HTTP por segundo sobre o acelerador (WESSELS, 2001).

O método utilizado nesse trabalho foi idealizado de forma a mimetizar um ambiente *Web* real. Uma página *Web* de prova é servida pelo Servidor Web Apache. Entre a ferramenta de medição de performance e o Apache está o acelerador HTTP, em um arranjo ilustrado na Figura 4.2.

O *hardware* cliente executa a ferramenta de medição de performance *Web* `httpef` (Seção 4.4), responsável por gerar o *workload* e coletar dados estatísticos. O *hardware* servidor executa o acelerador HTTP, atendendo às requisições do cliente, ao mesmo tempo que executa o Apache como servidor *Web* de origem e armazena a página *Web* de prova (Seção 4.5).

4.1 *Hardware*

4.1.1 *Hardware* cliente

Por ser responsável apenas pela execução da ferramenta de medição de performance *Web*, o *hardware* cliente não necessita ser uma máquina de última geração. Foi adotado um computador com as seguintes características:

- Processador Intel Celeron M 1,50Ghz
- Memória RAM de 2GB
- Disco rígido IDE de 120Gb
- Placa de rede fast ethernet Broadcom BCM4401

4.1.2 *Hardware* servidor

Como processa toda a carga de requisições proveniente do cliente, o *hardware* servidor é mais robusto. Esse computador tem as seguintes características:

- Processador Intel Core 2 Duo de 1,80GHz
- Memória RAM de 4Gb
- Disco rígido SATA de 250Gb
- Placa de rede gigabit ethernet Realtek RTL8168B

4.2 Rede de interconexão

A interconexão entre o *hardware* cliente e servidor é realizada através de uma rede ethernet. É natural a utilização desse protocolo uma vez que é *de-facto* padrão universal.

Ao invés de ser disposto um *hardware* de rede[1] entre os computadores envolvidos no ensaio, foi instalado um *cross-cable* conectando diretamente às placas de rede do *hardware* cliente e servidor. Evitam-se, assim, possíveis latências e minimiza-se a possibilidade de colisão de *frames* ethernet durante as simulações.

A Figura 4.3 demonstra que o *link* ethernet foi negociado entre os computadores a 100 Mb/s *full-duplex*.

Por fim, o *hardware* servidor é configurado com o IP 192.168.1.1 e o cliente com o IP 192.168.1.2, ambos com máscara 255.255.255.0.

[1] switches, hubs, roteadores, etc.

```
# mii-tool eth0
eth0: negotiated 100baseTx-FD flow-control, link ok
```

Figura 4.3: Saída do comando `mii-tool`

Tabela 4.1: Características do kernel quanto à operação em rede

Chave do `sysctl`	Valor padrão	Descrição
`fs.file-max`	358898	Máximo descritores em todo S.O.
`fs.nr_open`	1048576	Máximo de descritores por processo
`net.ipv4.tcp_max_syn_backlog`	1024	Fila de conexões pendentes
`net.ipv4.ip_local_port_range`	32768 a 61000	Portas TCP locais disponíveis
`net.ipv4.tcp_fin_timeout`	60	Tempo de espera para fechamento de conexão

4.3 Sistema Operacional

O sistema operacional presente em ambos os *hardwares*, cliente e servidor, é o Debian GNU/Linux 5.0 Lenny cujo *kernel* instalado é o `2.6.26-2-686-bigmem`. Esse *kernel* é relativamente novo e de estabilidade e performance comprovadas. As principais características quanto ao seu comportamento em rede também foram pesquisadas através do utilitário `sysctl`, conforme Tabela 4.1. Fizeram-se necessárias modificações de certos parâmetros para garantir que as aplicações envolvidas (ferramenta de medição, acelerador HTTP e servidor *Web*) não sofressem com a exaustão de alguns recursos. Tais modificações são detalhadas na Seção 4.3.1.

Para evitar uso desnecessário de recursos dos computadores envolvidos no ensaio, vários outros processos foram finalizados: interface gráfica, *daemons* servidores diversos, etc. O único serviço em execução é o `ntpd`.

4.3.1 Parametrizações

O processo de inicialização e estabelecimento de um grande número de conexões TCP consome recursos como portas locais e descritores de arquivos em um grau mais elevado que uma utilização corriqueira do sistema operacional.

Os *kernels* empacotados na maioria das distribuições GNU/Linux têm valores modestos na configuração de sua pilha TCP/IP. Dessa forma, alguns valores *default* precisam ser parametrizados visando prover os recursos necessários para as ferramentas de medição.

4.3.1.1 Fila de conexões pendentes

O *kernel* mantém uma fila de conexões pendentes onde pacotes TCP com a *flag* SYN habilitada foram recebidos, porém, pacotes correspondentes com as *flags* SYN e ACK ainda não foram enviados. Em outras palavras, o *three-way handshake* ainda não foi estabelecido (STEVENS, 1994).

O *kernel* inicia o *three-way handshake* assim que a chamada de sistema `accept()`[2] é executada pela aplicação[3]. Se o número de pacotes na fila de conexões pendentes for maior que seu comprimento, conexões serão perdidas. Portanto, é indicado aumentar seu tamanho.

A chave `net.ipv4.tcp_max_syn_backlog`[4] define o comprimento da fila de conexões pendentes. Nesse trabalho seu valor é configurado para 2048.

4.3.1.2 Tempo de espera para fechamento de conexão

Sempre que uma aplicação fecha uma conexão, através da chamada de sistema `close()`[5], seu estado é mudado para `TIME_WAIT` durante um tempo de 2 vezes o *maximum segment lifetime* (MSL) antes de ser realmente fechada. MSL é o tempo que um pacote é válido em um segmento de rede. Essa forma de término de conexão é denominada *active close* (STEVENS, 1994).

Como efeito colateral, a porta local associada à conexão não pode ser reutilizada durante o tempo em que seu estado é `TIME_WAIT`. O *kernel*, por padrão, mantém esse tempo de espera em 60 segundos, ou seja, o MSL é de 30 segundos.

Conforme a Tabela 4.1, a faixa de portas locais que podem ser utilizadas para iniciar uma conexão é de 32768 a 61000. Essa faixa resulta em 28232 portas disponíveis. Se o tempo em `TIME_WAIT` é de 60 segundos, então é possível manter no máximo uma taxa de 470 conexões por segundo.

---

[2] `http://linux.die.net/man/2/accept`

[3] Após o recebimento de um pacote TCP de início de conexão (SYN)

[4] Do utilitário `sysctl`

[5] `http://linux.die.net/man/2/close`

Frente a este fato limitador, nesse trabalho o tempo de espera para fechamento de conexão é definido em 10 segundos. Dessa forma, o MSL é de 5 segundos: tempo suficiente em uma rede com baixa taxa de colisão. A chave `net.ipv4.tcp_fin_timeout` configura o tempo em questão.

Se a faixa de portas locais disponíveis não fosse alterada, então, ao diminuir o tempo em `TIME_WAIT` para 10 segundos, conclui-se que é possível manter uma taxa de 2823 conexões por segundo: um ganho expressivo. Na Seção 4.3.1.3 é discutido o aumento na faixa de portas locais.

4.3.1.3 Portas TCP locais disponíveis

Cada conexão TCP é definida por um par de *sockets* compreendendo um endereço IP e uma porta do cliente e um endereço IP e uma porta do servidor. Considerando cliente e servidor dotados de apenas um endereço IP cada um e que a porta [6] do servidor é fixa, o único recurso que muda entre várias conexões cliente ⇔ servidor é a porta do cliente.

Ao serem geradas várias conexões simultâneas, portas locais no cliente são alocadas. Durante o ensaio de carga existe a possibilidade do número de portas disponíveis no cliente se esgotar conforme discutido na Seção 4.3.1.2.

Nesse trabalho, a faixa de portas locais disponíveis, configurada através da chave `net.ipv4.ip_local_port_range`, é definida de 1025 a 65535, o que disponibiliza 64510 portas locais. Considerando o tempo de espera de fechamento de conexões de 10 segundos (Seção 4.3.1.2), há a possibilidade teórica de se manter uma taxa de 6451 conexões por segundo.

4.3.1.4 Máximo de descritores de arquivos por processo

A Tabela 4.1 demonstra que na configuração do *kernel* o limite de descritores de arquivos por processo é suficiente para se atingir as 6451 conexões por segundo. Porém, o Linux implementa *POSIX Capabilities*, que limita o número de descritores em 1024. A Figura 4.4 ilustra a consulta a este valor.

Nesse trabalho aplica-se o valor 8192 para o número máximo de descritores por processo, seja ele cliente ou servidor, através do comando `ulimit`.

[6] A porta do servidor está em estado `LISTEN`

```
# ulimit -n
1024
```

Figura 4.4: Consulta ao número máximo de descritores

Tabela 4.2: Sumário de parametrizações

Chave do `sysctl`	Valor padrão	Descrição
`net.ipv4.tcp_max_syn_backlog`	2048	Fila de conexões pendentes
`net.ipv4.ip_local_port_range`	1025 a 65535	Portas TCP locais disponíveis
`net.ipv4.tcp_fin_timeout`	10	Tempo de espera para fechamento de conexão

4.3.1.5 Sumário de parametrizações

A Tabela 4.2 sumariza as parametrizações consideradas nesse trabalho. Um exemplo do conteúdo a ser adicionado ao arquivo `/etc/sysctl.conf` encontra-se no Apêndice A.1, para que os valores sejam aplicados ao *kernel* em sua inicialização.

4.4 Ferramenta de medição de performance

A ferramenta de medição de performance do acelerador HTTP utilizado nesse estudo é o `httperf` (MOSBERGER, 2009). Os principais fatores de sua escolha foram a facilidade de uso, a capacidade de gerar vários *workloads* e o seu *design* interno moderno.

Desenvolvido e mantido por David Mosberger, pesquisador no HP Research Labs, a última versão (0.9.0) teve uma forte contribuição do *Student Software Engineering Team* da Universidade de Calgary, Canadá. Várias empresas o utilizam como ferramenta de medição de performance, entre elas o Yahoo!. É distribuído sob a licença *General Public License* (GPL).

O `httperf` foi projetado para atingir boa performance, predição de seu comportamento e extensibilidade. A boa performance é conseguida com a escrita da ferramenta na linguagem C com cuidados redobrados com certas operações críticas. A predição de seu comportamento é conseguida ao se depender o mínimo

possível do sistema operacional. A extensibilidade é conseguida com os 5 tipos de geradores de *workloads* totalmente configuráveis (MOSBERGER; JIN, 1998).

Os geradores de *workloads* são selecionados por opções de linha de comando. São eles:

- Padrão (*default*)
  Gera um número fixo de requisições HTTP de uma determinada URL.
- `--wlog`
  Gera requisições baseado em uma lista de URL's. Utilizado para regerar requisições registradas em um *log*.
- `--wsess`
  Efetua medições de sessão, que são picos de requisições separados por um atraso que simula o tempo de ação do usuário;
- `--wesslog`
  Similar ao `--wsess`, porém as URL's estão definidas em um arquivo.
- `--wset`
  Gera uma determinada taxa de requisições baseado em uma lista de URL's.

Nesse trabalho é utilizado o gerador `--wesslog` utilizando um arquivo de entrada. A Seção 4.4.1 aborda esse arquivo.

Exemplo de ensaio utilizando `httperf`:

```
$ httperf --hog --server=192.168.1.1 --num-conns=10000 --num-calls=1 --rate=20 \
          --timeout=10
```

Esse exemplo informa ao `httperf` que utilize a URL `http://192.168.1.1/` e efetue 10000 conexões (`--num-cons`) a uma taxa de 20 conexões por segundo (`--rate`). Para cada conexão deve ser realizada uma requisição HTTP (`--num-calls`) com um tempo de expiração de 10 segundos (`--timeout`).

Após a realização do ensaio é gerada uma saída conforme ilustrada na Figura 4.5. As estatísticas são agrupadas em até 7 seções de resultados. As seções são separadas entre si por um espaço.

```
Total: connections 10000 requests 10000 replies 10000 test-duration 499.950 s

Connection rate: 20.0 conn/s (50.0 ms/conn, <=1 concurrent connections)
Connection time [ms]: min 0.5 avg 0.6 max 5.4 median 0.5 stddev 0.1
Connection time [ms]: connect 0.1
Connection length [replies/conn]: 1.000

Request rate: 20.0 req/s (50.0 ms/req)
Request size [B]: 64.0

Reply rate [replies/s]: min 20.0 avg 20.0 max 20.0 stddev 0.0 (99 samples)
Reply time [ms]: response 0.5 transfer 0.0
Reply size [B]: header 256.0 content 45.0 footer 0.0 (total 301.0)
Reply status: 1xx=0 2xx=10000 3xx=0 4xx=0 5xx=0

CPU time [s]: user 57.98 system 441.46 (user 11.6% system 88.3% total 99.9%)
Net I/O: 7.1 KB/s (0.1*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

Figura 4.5: Exemplo de saída do `httperf`

A primeira seção é a `Total`. Nessa seção são sumarizados o número de conexões iniciadas, o número de requisições, o número de respostas e o tempo de duração do ensaio.

```
Total: connections 10000 requests 10000 replies 10000 test-duration 499.950 s
```

A segunda seção é a `Connection`. Essa seção reporta a performance das conexões *iniciadas* durante o teste, o tempo mínimo, médio e máximo de vida da conexão, sua mediana e desvio-padrão. Explicita também o tempo médio para que a conexão seja estabelecida e a média de respostas por conexão.

Tempo de vida da conexão é o tempo entre se estabelecer a conexão e terminá-la com sucesso.

```
Connection rate: 20.0 conn/s (50.0 ms/conn, <=1 concurrent connections)
Connection time [ms]: min 0.5 avg 0.6 max 5.4 median 0.5 stddev 0.1
```

```
Connection time [ms]: connect 0.1
Connection length [replies/conn]: 1.000
```

A terceira seção é a `Request`. Essa seção reporta a taxa de *requisições* HTTP efetuadas e o período de cada requisição. Se não forem simuladas conexões persistentes, ou seja, for realizada uma requisição por conexão esses resultados tendem a ser similares aos apresentados na seção `Connection`.

```
Request rate: 20.0 req/s (50.0 ms/req)
Request size [B]: 64.0
```

A quarta seção é a `Reply`. Essa seção reporta a taxa mínima, média, máxima de *respostas* e seu desvio-padrão. Reporta também o tempo médio de resposta, que é o tempo entre o primeiro *byte* da requisição ser enviado até o primeiro *byte* da resposta ser recebido, além do tempo gasto para transferir (ler) a resposta completa.

```
Reply rate [replies/s]: min 20.0 avg 20.0 max 20.0 stddev 0.0 (99 samples)
Reply time [ms]: response 0.5 transfer 0.0
Reply size [B]: header 256.0 content 45.0 footer 0.0 (total 301.0)
Reply status: 1xx=0 2xx=10000 3xx=0 4xx=0 5xx=0
```

A quinta seção é a `Miscellaneus`, sumarizando o uso de CPU e largura de banda da rede durante o ensaio.

```
CPU time [s]: user 57.98 system 441.46 (user 11.6% system 88.3% total 99.9%)
Net I/O: 7.1 KB/s (0.1*10^6 bps)
```

A sexta seção é a `Errors` que sumariza o total de erros. O `client-timo` informa o número de vezes que uma conexão, requisição ou sessão falhou devido a um *timeout* enquanto `fd-unavail` contabiliza quantos descritores de arquivos o `httperf` tentou usar que não estavam disponíveis.

```
Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

Caso sejam utilizados os *workloads* `--wsess` ou `--wsesslog` uma sétima seção é apresentada, chamada `Session`. Ela é similar à seção `Reply`. A diferença é que os resultados são agrupados por sessão e não por requisição.

4.4.1 Medição baseada em sessões

Uma característica muito importante do `httperf` é a capacidade de gerar um *workload* baseado em sessões. Sessão é um conjunto de URL's requisitadas simultaneamente, simulando o comportamento de um navegador ao acessar um determinado conteúdo. Há também a possibilidade de incluir um tempo de espera entre uma sessão e outra. Esse tempo de espera simula o tempo que um usuário leva para tomar uma decisão ao receber o conteúdo. Por exemplo, essa decisão pode ser a de acessar um outro *hyperlink*. O tempo de espera é chamado de *tempo de pensamento do usuário* (*user think time*).

Dos dois tipos de geradores de *workload* baseados em sessão, o `--wsesslog` é o utilizado nesse trabalho. Sua sintaxe é:

```
--wssesslog=N,X,F
```

Onde `N` é o número total de sessões a serem geradas, `X` é o tempo padrão de pensamento do usuário e `F` é um arquivo de entrada que especifica a sessão. Nesse arquivo estão descritos as URL's, o método HTTP e, opcionalmente, o tempo de pensamento do usuário[7]. Várias sessões podem ser definidas se as URL's correspondentes forem separadas por uma linha em branco.

No exemplo da Figura 4.6 são criadas duas sessões. A primeira requisita `/index.html`, `/cabecalho.gif` e `/figura.gif` simultaneamente. Após um tempo de pensamento do usuário de 2 segundos, `/compra.html`, `/compra.css` e `/objeto.gif` são requisitados.

Na segunda sessão é realizado um `POST` em `/reclamacao.php` e logo após `/saida.html` é requisitado.

[7] Que irá sobrepor ao tempo padrão definido por `X`

```
/index.html think=2.0
      /cabecalho.gif
      /figura.gif
/compra.html
      /compra.css
      /objeto.gif

/reclamacao.php method=POST contents=''texto=O produto nao presta''
/saida.html
```

Figura 4.6: Exemplo de arquivo de sessão

Ao efetuar um teste com o `httperf` utilizando o arquivo da Figura 4.6 gravado com o nome `exemplo.ld`:

```
$ httperf --hog --wsesslog=20000,1,exemplo.ld --server=localhost --rate=100 \
         --timeout=5
```

Uma saída é produzida com as seis seções abordadas mais a seção `Session` que reporta a taxa mínima, média e máxima das seções finalizadas bem como o desvio-padrão. Informa também a média de conexões por sessão, o tempo médio para se completar com sucesso uma sessão e o tempo médio até haver uma falha em sessões. Por fim, reporta um histograma, informando quantas sessões receberam zero, uma, duas, três, quatro, cinco, seis e sete respostas.

```
Session rate [sess/s]: min 80.00 avg 99.51 max 100.01 stddev 3.16 (20000/20000)
Session: avg 1.00 connections/session
Session lifetime [s]: 1.0
Session failtime [s]: 0.0
Session length histogram: 0 0 10000 0 0 0 10000
```

4.5 Preparação do ambiente de ensaios de carga

Como ilustrado na Figura 4.2, o ambiente de ensaio de carga consiste em 2 computadores instalados com Sistema Operacional Debian 5.0 Lenny parametrizado (Seção 4.3). Essa seção descreve quais e como os *softwares* são instalados.

4.5.1 Servidor *Web*

Os *daemons* que fazem parte do provedor de conteúdo *Web* e que são executados no *hardware* servidor são compilados e instalados a partir do código-fonte. Assim, é possível avaliá-los em sua versão original sem a aplicação de *patches* e/ou modificações porventura efetuadas pelos desenvolvedores da distribuição.

Objetivando uma melhor organização, o diretório `/usr/local/acelhttp` é utilizado como raiz de toda a instalação. Desse modo, o Apache será instalado em `/usr/local/acelhttp/apache2`, o Squid em `/usr/local/acelhttp/squid` e o Varnish em `/usr/local/acelhttp/varnish`. Assim, isolam-se estes *daemons* dos que podem ser instalados pela distribuição. Facilita-se também a desinstalação de todo o ambiente, simplesmente comandando:

```
# rm -rf /usr/local/acelhttp
```

Os processos de instalação pressupõem que no *hardware* servidor estejam disponíveis o compilador GCC, o interpretador PERL e suas respectivas dependências.

4.5.1.1 Instalação do Apache

O Apache instalado nesse estudo é o de versão 2.2.14, o último em tempo de escrita desse trabalho. Após o *tarball* ser baixado, é executado:

```
# tar -jxvf httpd-2.2.14.tar.bz2 -C /usr/local/src
# cd /usr/local/src/httpd-2.2.14
# ./configure --prefix=/usr/local/acelhttp/apache2 \
```

```
  --with-mpm=worker --enable-headers
# make
# make install
```

Com estas opções, o Apache será instalado com o módulo `Headers` compilado *built-in*. Para iniciá-lo, comanda-se:

```
# cd /usr/local/acelhttp
# ulimit -HSn 8192
# ./apache2/bin/apachectl start
```

A inclusão do módulo `Headers` permite a configuração do Apache para que o mesmo gere os *headers* HTTP informando ao acelerador o que deve ou não ser armazenado em *cache*.

4.5.1.2 Página *Web* de prova

Uma página *Web*, ilustrada na Figura 4.7, foi criada para ser usada como corpo de prova no acesso ao acelerador HTTP, permitindo a medição de seu comportamento.

A Tabela 4.3 sumariza os arquivos que compõem a página *Web* de prova, com os respectivos tamanhos individuais e total.

Tabela 4.3: Arquivos que compõem a página *Web* de prova

Arquivo	Tamanho	Descrição
`pagina.html`	5KB	Página HTML principal
`estilo.css`	0,85KB	Folha de estilo CSS
`timbernersless.jpg`	121KB	Foto de Tim Berners-Lee
`valid-xhtml10.png`	2KB	Selo W3C de conformidade XHTML
`vcss.gif`	2KB	Selo W3C de conformidade CSS
`gnu.png`	22KB	Mascote da FSF
`tux.png`	33KB	Mascote do Linux
Total:	185,85KB	

Figura 4.7: Página *Web* de prova

Como o tamanho médio das páginas disponíveis na Internet é da ordem de poucas centenas de KB's, essa página *Web* de prova imprimirá uma carga aos aceleradores HTTP similar ao que são submetidos em um ambiente real de produção.

Para que o Apache sirva esses conteúdos, seus arquivos devem ser copiados para o diretório `htdocs` presente no diretório de instalação:

```
# cp pagina-carga/* /usr/local/acelhttpd/apache2/htdocs
```

4.5.1.3 Configuração do Apache

Após a instalação do Apache, um arquivo de configuração `httpd.conf` é disponibilizado no diretório `/usr/local/acelhttp/apache2`. Tendo-o como ponto de partida e com poucas modificações chega-se à configuração utilizada nesse trabalho.

As modificações são:

- parametrizações do *MPM worker* para permitir o atendimento a um grande número de requisições por segundo;

- fazer que o *daemon* escute conexões pela porta TCP 1024;
- definir o usuário e grupo do *daemon* para *nobody* e *nogroup*, respectivamente;
- definir 3 modos de geração de *headers* HTTP para que o acelerador armazene ou não objetos em seu *cache*.

A listagem do arquivo está disponível no Apêndice A.2.

4.5.2 Acelerador HTTP

4.5.2.1 Instalação do Varnish

A instalação do Varnish é uma tarefa simples, uma vez que não possui muitas opções. Para compilação e instalação utilizam-se os comandos a seguir:

```
# tar zxvf varnish-2.0.4.tar.gz -C /usr/local/src
# cd /usr/local/src/varnish-2.0.4
# ./configure --prefix=/usr/local/acelhttp/varnish
# make
# make install
```

4.5.2.2 Configuração do Varnish

Por ter sido desenvolvido desde o início para ser um acelerador HTTP de alta performance, o Varnish não necessita que muitos parâmetros sejam modificados para atender ao ensaio de carga.

O Varnish não segue completamente a RFC 2616 (KAMP, 2009). Entre as funcionalidades que não estão implementadas figura a de não seguir a diretiva `no-cache` no *header* `Cache-Control`. Isto significa que se uma resposta contiver o *header* `Cache-Control: no-cache` o conteúdo será *cacheado*.

Nesse trabalho deseja-se que o acelerador HTTP siga os *headers* gerados pelo servidor *Web* Apache, logo, uma configuração VCL precisa ser implementada de forma a:

- definir o servidor *Web* de origem. Nesse contexto também chamado de servidor *backend*;
- não armazenar em *cache* os objetos cujas respostas contenham o *header* `Cache-Control: no-cache`;
- retirar o *header* `Age` de respostas não *cacheadas*.

No Apêndice A.3 está listado o arquivo de configuração `varnish.vcl` que atende aos requisitos acima citados. Essa configuração deve ser disponibilizada no diretório `/usr/local/acelhttp/varnish/etc`.

Após sua instalação, inicia-se o varnish da seguinte maneira:

```
# cd /usr/local/acelhttp/varnish
# chown -R nobody.nogroup var
# ulimit -HSn 8192
# ./sbin/varnishd -u nobody -a 0.0.0:80 \
                  -f etc/varnish.vcl \
                  -s file,var/varnish.cache,2047M
```

4.5.2.3 Instalação do Squid

O Squid, por ser um *forward proxy/cache* posteriormente adaptado para ser um *proxy/cache* reverso, mantém suporte a várias funcionalidades que não são interessantes quando opera como um acelerador HTTP. Portanto, várias opções são desabilitadas para que se obtenha um programa executável mais otimizado.

O processo de compilação e instalação é descrito abaixo:

```
# tar jxvf squid-2.7.STABLE7.tar.bz2 -C /usr/local/src
# cd /usr/local/src/squid-2.7.STABLE7
# ./configure --prefix=/usr/local/acelhttp/squid \
              --enable-storeio="aufs" \
              --enable-removal-policies="heap" \
              --disable-icmp \
```

```
                --enable-err-languages="Portuguese" \
                --enable-default-err-language="Portuguese" \
                --enable-epoll --disable-linux-netfilter \
                --disable-linux-tproxy \
                --disable-ident-lookups \
                --with-pthreads --with-aio --with-maxfd=8192
# make
# make install
```

4.5.2.4 Configuração do Squid

A configuração implementada para o Squid visa fazê-lo funcionar como um acelerador HTTP suportando um grande número de requisições por segundo.

O Squid utiliza sua infraestrutura de *cache* hierárquico para implementar a funcionalidade de um *proxy/cache* reverso da seguinte forma:

- define-se o servidor *Web* de origem como sendo um servidor HTTP **pai** (em termos hierárquicos);
- adiciona-se a opção `originserver` da diretiva `cache_peer`;
- informa-se que as requisições devem ser atendidas pela porta 80 através da diretiva `http_port`;
- adicionam-se as opções `vhost`, `act-as-origin` e `http11` à diretiva `http_port`, afim de indicar respectivamente que *hosts* virtuais são suportados. Os *headers* `Date` e `Expires` são gerados pelo Squid a cada resposta HTTP e utiliza-se HTTP 1.1 na comunicação com o servidor *Web* de origem.

As configurações referentes à quantidade de memória alocada, tamanho máximo de um objeto em memória, tamanho do *cache* em disco, políticas de substituição de objetos em disco e memória foram escolhidas de forma a maximizar a performance.

No Apêndice A.4 encontra-se a listagem completa do arquivo de configuração `squid.conf`.

```
/pagina.html
        /estilo.css
        /timbernerslee.jpg
        /valid-xhtml10.png
        /vcss.gif
        /gnu.png
        /tux.png
```

Figura 4.8: Arquivo de *workload* utilizado nos ensaios

4.5.3 Ferramenta de medição

No *hardware* designado para ser o cliente *Web*, descrito na Seção 4.1.1, é necessário instalar o `httperf`. A distribuição Debian disponibiliza esse *software* em seus repositórios. Portanto, ele pode ser instalado através do comando `aptitude`:

```
# aptitude install httperf
```

Outra forma de instalação é através do comando `apt-get`, que não é mais indicado pela comunidade Debian, ou baixando o pacote e instalando-o com o comando `dpkg`.

4.5.3.1 Arquivo de *workload* de sessões

Para que seja efetuada uma medição baseada em sessões, conforme Seção 4.4.1, é necessária a definição de um arquivo de entrada a ser usado pelo `httperf` para a geração do *workload*. Este arquivo está listado na Figura 4.8.

A característica principal do arquivo de *workload* é a requisição sequencial dos objetos HTTP que compõem a página *Web* de carga. Esse comportamento é similar aos *browsers* mais utilizados pelos usuários.

4.6 Sistemática dos ensaios de carga

Todo processo de análise estatística deve seguir condições pré-determinadas para colhimento dos resultados experimentais. Portanto, uma sistemática para execução dos ensaios foi desenhada a partir de algumas definições.

A primeira definição é que cada ensaio deve ser iniciado a partir de um mesmo estado do sistema operacional. Isto garante que mecanismos como *buffers* de disco, *caches* de memória RAM, quantidade de portas TCP disponíveis, entre outros, sejam inicialmente iguais e não contribuam para desviar os resultados de um valor real. É improvável garantir um estado estritamente igual do sistema operacional, mas certamente os estados de início do ensaio serão semelhantes. Assim, cada ensaio é executado logo após uma inicialização (*boot*).

A segunda definição é que cada dado coletado, que são pontos nos gráficos estatísticos, seja uma média aritmética de cinco ensaios. Exemplificando: o resultado do comportamento de um acelerador HTTP a uma carga de 70 requisições por segundo será a média entre cinco ensaios executados com o `httperf`.

A terceira definição é que cada ensaio tenha uma duração de 30 minutos. *Daemons* servidores de rede levam um bom tempo para atingirem um estado uniforme de comportamento quanto à resposta (WESSELS, 2001). Em alguns ensaios é sugerido que o colhimento dos dados estatísticos ocorra depois de várias horas de submissão de carga sobre o servidor.

A quarta definição é que o ensaio ocorrerá depois de satisfeita uma sessão (Seção 4.5.3.1) onde o acelerador HTTP poderá popular seu *cache*.

Como o *hardware* servidor é dotado de 2 *cores* em sua CPU, a quinta definição é que o servidor *Web* Apache é executado no *core* 0, quanto o acelerador HTTP é executado no *core* 1. Através do utilitário `taskset` é possível informar qual processo rodará em um *core*. Dessa forma, os resultados obtidos são menos dependentes do escalonador de processos do sistema operacional.

A sexta definição é que todos os sistemas envolvidos estejam com horários sincronizados pelo serviço `ntpd`, a fim de que seja mantida coerência nas informações geradas nos *headers* `Date` (WESSELS, 2001).

A sétima definição é que os ensaios utilizem o mecanismo de *caching* baseado em expiração (Seção 2.5.1).

Por fim, a última definição é que todos os serviços, *daemons*, interface gráfica entre outros, excluíndo-se o `ntpd`, não estejam em execução.

Seguindo-se essas oito definições é possível o levantamento uniforme dos dados estatísticos.

4.6.1 Modos de operação do Apache

Para que sejam reproduzidas situações onde o acelerador HTTP é submetido em ambiente de produção, foram definidos três modos de operação do Apache. Cada modo basicamente gera um conjunto de *headers* `Cache-Control` com uma certa diretiva (Tabela 2.3) informando ao acelerador como tratar um dado objeto.

Os três modos são:

- Modo **SEM_CACHE**:
  Nesse modo toda e qualquer resposta do Apache contém o *header* `Cache-Control` com a diretiva `no-cache`. Nenhum objeto é *cacheado* pelo acelerador.
  O objetivo desse modo é avaliar a latência introduzida pelo acelerador HTTP quando se opera apenas como um *proxy*, uma vez que nenhum objeto será por ele servido.

- Modo **CACHE_PARCIAL**:
  Nesse modo a resposta referente ao objeto `pagina.html` contém a diretiva `no-cache` em `Cache-Control`. Todas as demais respostas contém a diretiva `max-age` com o valor `84600`, indicando que o acelerador armazene estes objetos por um dia.
  O objetivo desse modo é avaliar uma ponderação entre performance conseguida pelo *caching* de objetos com a latência introduzida na requisição do objeto que porventura poderia ser dinâmico[8].

- Modo **CACHE_TOTAL**:
  Nesse modo todas as respostas contêm a diretiva `max-age` com o valor `84600`.
  O objetivo desse modo é avaliar a performance do acelerador HTTP na forma ideal, que é conseguida quando todos os objetos requisitados são satisfeitos a partir do *cache*, desonerando totalmente o servidor *Web* de origem.

O Apache é iniciado em um dos três modos descritos a partir do uso da opção `-D` seguido de uma *definição* (também chamado de *label*) de linha de comando. Essa opção instrui ao Apache a avaliar e considerar as diretivas entre o

[8]Conteúdo HTML

bloco `<IfDefine definição> ... </IfDefine>`, conforme fim do arquivo de configuração `httpd.conf` listado no Apêndice A.2. Diretivas do módulo `Headers`, que está disponível no Apache utilizado nos ensaios (Seção 4.5.1.1), permitem a customização de *headers* HTTP. Com tais diretivas, os três modos de operação são implementados.

Por exemplo, a inicialização do Apache no modo **CACHE_TOTAL** é demonstrada na Figura 4.9. Para inicializar os demais modos, basta substituir *definição* pelo nome do modo desejado.

```
# cd /usr/local/acelhttp/apache2/
# ./bin/apachectl -DCACHE_TOTAL -k start
```

Figura 4.9: Inicialização do Apache no modo **CACHE_TOTAL**

Capítulo 5

Resultados e Discussão

Esse capítulo apresenta os resultados obtidos com os ensaios seguindo a metodologia descrita no Capítulo 4.

5.1 Pré-testes

Antes dos ensaios de carga serem iniciados, dois testes prévios serão necessários. Tais pré-testes definirão uma base de comparação para os ensaios propriamente ditos.

5.1.1 Largura de banda da rede de interconexão

A rede de interconexão utilizada nos ensaios de carga é uma ethernet a 100 Mb/s *full-duplex* (Seção 4.2). Na prática, a largura de banda é menor que a nominal. Para que seja determinada a largura de banda **real** da rede, é necessário utilizar uma ferramenta de *benchmarking*. A ferramenta `iperf` (IPERF, 2009) foi escolhida para esse fim.

Tendo-se conhecimento da largura de banda real da rede é possível ponderar se durante os ensaios a rede satura. Uma rede saturada certamente causaria uma latência artificial na taxa de resposta do acelerador, o que levaria a uma avaliação errônea de sua performance.

O tráfego HTTP pode ser considerado unidirecional. Geralmente, uma requisição é enviada em apenas um pacote TCP, enquanto a resposta normalmente necessita de vários pacotes para ser transmitida (WESSELS, 2001). Devido a esse comportamento, o pré-teste se dá através do *downstream* a partir do servidor em direção ao cliente.

Utilizando-se a ferramenta `iperf` para gerar e contabilizar o tráfego TCP na direção do servidor para o cliente durante 30 minutos, chegou-se ao resultado que a largura de banda real da rede é de 11476 KB/s, equivalente a 89,7 Mb/s.

5.1.2 Performance do Apache sem acelerador

Semelhantemente ao pré-teste envolvendo a rede de interconexão, é necessário um pré-teste de performance do servidor *Web* Apache. A realização desse teste prévio permitirá avaliar o método de ensaio como um todo, encontrando possíveis gargalos e problemas diversos. Também possibilitará verificar se os dados levantados serão válidos.

Os ensaios realizados seguiram a sistemática descrita na Seção 4.6 e foram gerados em uma escala de 10 a 50 sessões por segundo, com intervalos de 10 em 10 sessões por segundo. Como cada sessão efetua 7 requisições (Seção 4.5.3.1), os dados foram coletados a 70, 140, 210, 280 e 350 requisições por segundo.

A Figura 5.1 ilustra a curva de resposta do Apache. Ela demonstra que, dentro dos limites estabelecidos, o número de respostas por segundo é igual ao número de requisições por segundo, ou seja, seu comportamento é linear. Assim, comprova-se que não houve *timeouts* nem conexões recusadas por impossibilidade de atendimento.

É interessante denotar que no limite de 350 requisições por segundo a largura de banda utilizada é de 9448,5 KB/s, ou 73,8 Mb/s. Esse valor equivale a 82% da banda real. É uma carga alta para qualquer servidor *Web*.

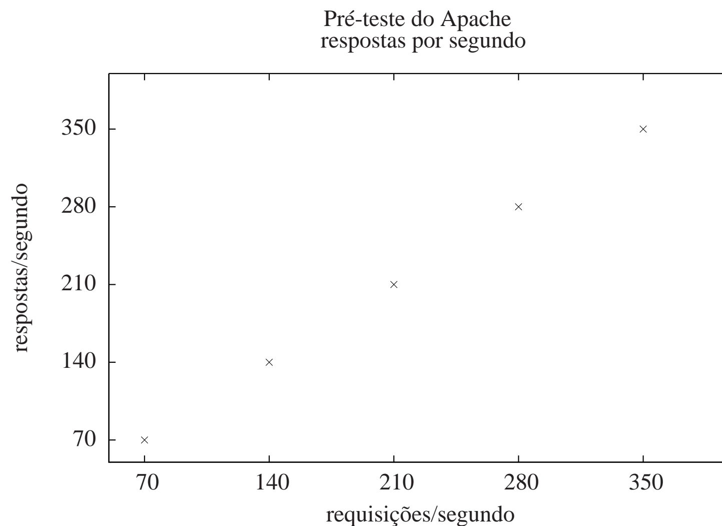

Figura 5.1: Respostas por segundo do Apache

5.2 Resultados dos ensaios do Squid e Varnish

5.2.1 Respostas HTTP por segundo

A primeira métrica avaliada nesse trabalho é o número de respostas HTTP por segundo. Essa métrica é importante pois define o *throughput* que o acelerador consegue manter.

O comportamento de ambos os aceleradores HTTP nos três modos de operação do servidor *Web* de origem Apache, discutido na Seção 4.6.1, foram iguais e está ilustrado na Figura 5.2. Esse gráfico demonstra que o Squid e o Varnish têm respostas idênticas e lineares até o limite de 350 requisições por segundo. É provável que, a partir das 400 requisições por segundo, essa resposta deixe de ser linear, pois este é um número aproximado onde a rede deve começar a saturar.

Portanto, em termos de *respostas por segundo*, as duas soluções estudadas tiveram, dentro dos limites estabelecidos, a mesma performance.

5.2.2 Tempo médio de conclusão de respostas

O tempo médio de conclusão de respostas foi computado através da soma dos campos `response` e `transfer` apresentado na seção `Reply` da saída do `httperf`. A Figura 4.5 ilustra um exemplo. O valor `response` informa a diferença de tempo entre o envio do primeiro *byte* da requisição e a chegada do primeiro *byte* da resposta. Por sua vez, o `transfer` informa o tempo de recebimento de toda a resposta (*status line* + *headers* + *entity body*).

Cada modo de operação do Apache impõe resultados diferentes que são demonstrados nas seções subsequentes.

5.2.2.1 Modo CACHE_TOTAL

A Figura 5.3 explicita os resultados obtidos por ambos os aceleradores.

No modo **CACHE_TOTAL** *todas* as requisições são satisfeitas pelos aceleradores.

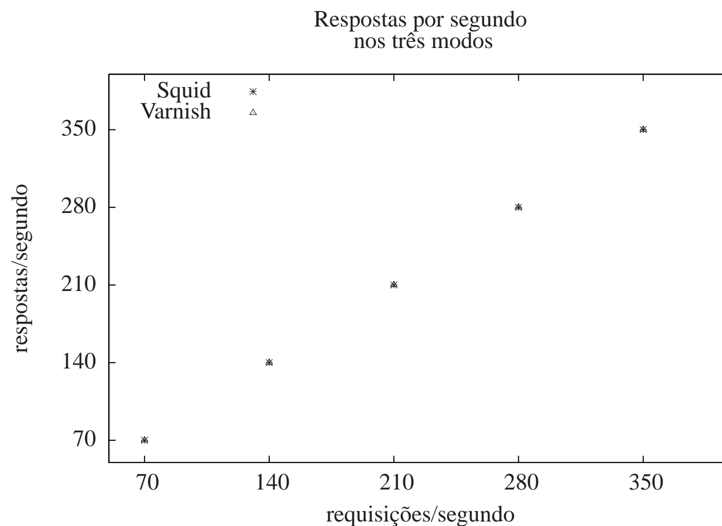

Figura 5.2: Respostas por segundo: Squid e Varnish

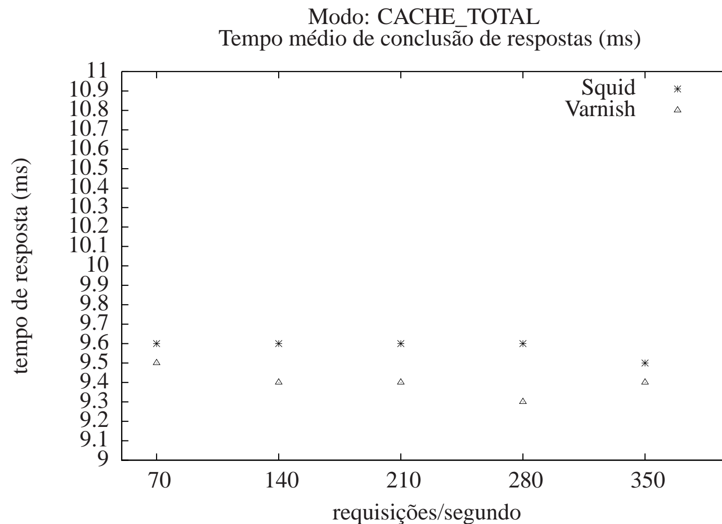

Figura 5.3: Tempo médio no modo **CACHE_TOTAL**

O Varnish obteve tempos médios de conclusão de respostas de 9,3 a 9,5 ms. Por sua vez o Squid obteve 9,5 e 9,6 ms. Percebeu-se uma tendência da diferença dos tempos diminuir com o aumento do número de requisições por segundo.

Calculando-se a média geral, ou seja, a média dos tempos médios de 70 a 350 requisições por segundo, chegou-se ao valor de 9,40 ms para o Varnish e 9,58 ms para o Squid. Em termos percentuais, o Varnish foi 1,88% mais rápido que o Squid.

5.2.2.2 Modo CACHE_PARCIAL

A Figura 5.4 ilustra os resultados obtidos por ambos os aceleradores.

No modo **CACHE_PARCIAL**, a requisição à página HTML *pagina.html* não é satisfeita pelos aceleradores, sendo assim repassada ao Apache. Porém todos os outros objetos são respondidos pelos aceleradores.

O Varnish obteve tempos de 9,5 e 9,7 ms. Por sua vez, o Squid obteve 9,6 e 9,7 ms. Novamente observou-se uma tendência da diferença entre os tempos diminuir com o aumento do número de requisições por segundo. Em 350 reqs/seg essa diferença foi zero.

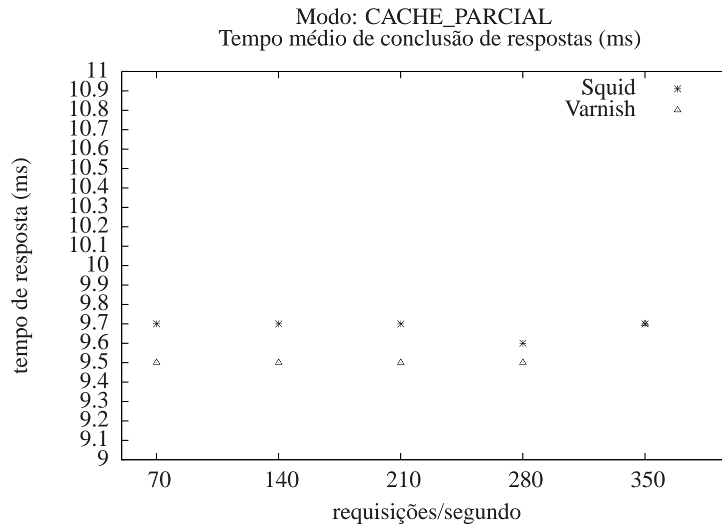

Figura 5.4: Tempo médio no modo **CACHE_PARCIAL**

Obtendo-se as médias gerais, o Varnish obteve 9,54 ms e o Squid 9,68 ms. A diferença, em termos percentuais, foi de 1,44%.

5.2.2.3 Modo SEM_CACHE

A Figura 5.5 demonstra os resultados obtidos por ambos os aceleradores.

No modo **SEM_CACHE**, a requisição de todos os objetos não são satisfeitas pelo acelerador, assim são repassadas para o Apache. Desta forma, há uma latência introduzida pelo acelerador.

O Varnish obteve tempos de 10,0 a 10,2 ms. Por sua vez o Squid obteve 10,4 e 10,5 ms. Aos 350 reqs/seg houve um aumento da diferença do tempo médio de conclusão de respostas.

Em termos de médias gerais, o Varnish obteve 10,12 ms e o Squid 10,44 ms, uma diferença de 3,07%: A maior diferença dentre os 3 modos utilizados no ensaio.

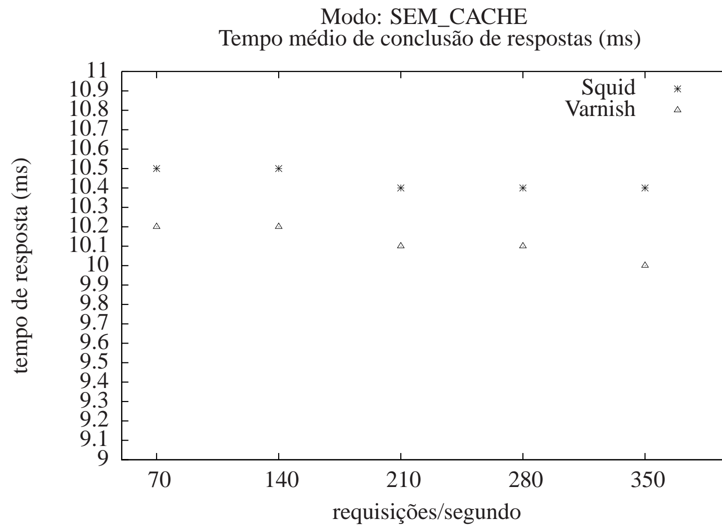

Figura 5.5: Tempo médio no modo **SEM_CACHE**

5.3 Discussão

Para a realização dos ensaios de carga, uma atenção especial foi dispensada na configuração do Squid. A maior preocupação foi o de aplicar uma parametrização compatível com a do Varnish.

O tamanho do *cache* de ambas as soluções foi limitado a 2GB. Esse limite é imposto pelo Varnish, por usar *memory mapped file* como *cache* quando compilado em uma plataforma de 32 bits.

O Squid armazena seus *hot objects*[1] em memória RAM. Um aspecto essencial é parametrizar esse espaço de memória de forma correta, de forma a **nunca** forçar o sistema operacional a realizar paginação. Muitas implementações do Squid apresentam baixa performance apenas porque o sistema operacional está paginando. A diretiva `cache_mem` define esse tamanho de memória.

Alguns tipos de *access control list*'s (ACL) do Squid são notáveis consumidores de CPU. Dentre eles podem ser destacados os tipos `url_regex`, `urlpath_regex`, `srcdom_regex` e `dstdom_regex`. Um mínimo de ACL's deve ser configurado em implementações do Squid como aceleradores HTTP em *Web sites* muito requisita-

[1]Objetos recentemente acessados

dos. O arquivo `squid.conf`, listado no Apêndice A.4, exemplifica o uso mínimo de ACL's.

Percebe-se, dos resultados obtidos, que a performance do Squid é pouco inferior a do Varnish. No modo **CACHE_TOTAL** os tempos de conclusão de respostas registrados pelo Squid foram 0,1 a 0,3 ms mais lentos que os do Varnish. É interessante frisar que a diferença dos tempos diminuiu com o aumento no número de requisições por segundo. Tal fato certamente advém dos algoritmos de seleção de objetos em *cache* do Squid.

Um comportamento similar é observado no modo **CACHE_PARCIAL**. Nesse modo, os tempos de conclusão de respostas do Squid foram 0,1 a 0,2 ms mais lentos que os do Varnish. Novamente constata-se que o Squid aproxima-se, e iguala-se, do Varnish quando submetido a um alto número de requisições por segundo.

As maiores diferenças foram registradas no modo **SEM_CACHE**. O Squid foi 0,3 a 0,5 ms mais lento que o Varnish. Uma explicação pode ser a forma que o Varnish abre novas conexões com o servidor *Web* de origem: a utilização de *threads* permite a execução concorrente e extremamente rápida de várias requisições ao *backend*.

Por fim, observa-se que ambas soluções conseguem manter tempos de conclusão de resposta praticamente constantes independente do número de requisições por segundo.

Capítulo 6

Conclusão

O estudo empírico realizado nesse trabalho comprova que ambas as soluções podem ser implementadas com sucesso como aceleradores HTTP por terem suportado uma taxa muito alta de requisições por segundo enquanto mantinham tempos de conclusão de respostas abaixo dos 10 milisegundos[1].

Ser submetido a um número de 350 requisições por segundo significa entregar conteúdo ao cliente *Web* a uma largura de banda de 9489,6 KB/s, ou 74,14 Mb/s. Com esses números chega-se à conclusão que em 24 horas de operação seriam atendidas 30 milhões e 240 mil requisições e seriam transferidos 781,92 GB de dados.

O Squid é um *daemon single-threaded* que utiliza I/O assíncrono em rede e chamadas de sistema tradicionais `open()` / `read()` / `write` / `close()` quando lida com o *cache* em sistema de arquivos. Como essas chamadas são síncronas, deve ser utilizada a modalidade de I/O AUFS por utilizar *multithreading*. Há também a possiblidade de se usar um *daemon* externo (`diskd`).

O Varnish é um *daemon multithreaded* que utiliza I/O síncrono em rede e um arquivo mapeado em memória[2] como *cache*.

Apesar do fato de que cada solução implementa um *design* diferente, a performance de ambas são tecnicamente iguais.

[1] Quando a resposta é satisfeita pelo *cache*
[2] *memory mapped file*

6.1 Outros critérios de escolha

Como a performance de ambas as soluções foi similar, outros critérios podem indicar quais dos dois *softwares* analisados pode melhor se adequar em uma implementação.

Se o total de objetos a ser armazenado em *cache* for superior a 2GB, em uma arquitetura 32 bits, ou 20 GB, em uma arquitetura 64 bits, a escolha recai sobre o Squid. Por utilizar um *memory mapped file* como *cache*, o Varnish limita o tamanho do *cache* pelo espaço máximo de endereçamento de memória de seu processo.

Em alguns casos é desejável não depender do servidor *Web* de origem para se implementar *caching*. Certos servidores de aplicação e *Content Management Systems* (CMS) não foram desenvolvidos para informar ao *proxy/cache* o que deve ou não ser armazenado em *cache*. Portanto, se um dos objetivos for não seguir os *headers* HTTP de *caching* gerados pelo servidor *Web* de origem, a melhor solução é a utilização do Varnish. Motivos: a facilidade na inclusão, remoção e modificação de *headers* de resposta de um servidor *Web* de origem e por não seguir totalmente a RFC 2616 (Seção 4.5.2.2).

Das duas soluções, apenas o Squid implementa suporte nativo a HTTPS. Para que seja possível a utilização de HTTPS com o Varnish, deve-se utilizar outros *daemons* para decifrar e cifrar o tráfego HTTP, como o Pound, o que aumenta a complexidade na administração da infraestrutura.

6.2 Trabalhos Futuros

Esse trabalho avaliou a performance do Squid e Varnish utilizando o mecanismo de *caching* baseado em expiração para que poucas requisições fossem repassadas ao Apache. Uma proposta para um trabalho futuro é o levantamento dos mesmos dados e com a mesma metodologia utilizando o mecanismo de *caching* baseado em validação.

Outros trabalhos podem ser desenvolvidos utilizando diferentes ferramentas de medição de performance, como o Apache JMeter[3] e o Web Polygraph[4] para a comparação e a comprovação dos resultados obtidos.

[3] `http://jakarta.apache.org/jmeter/`
[4] `http://www.web-polygraph.org/`

Por fim, pode ser realizado uma comparação em sistemas multiprocessados executando uma instância do Varnish contra várias instâncias do Squid, uma por *core*. Assim, este estudo contemplaria um paralelo entre um *daemon multithreaded* e vários *daemons single-threaded*.

Referências Bibliográficas

BERNERS-LEE, T. *The Original HTTP as defined in 1991*. 1994. http://www.w3.org/Protocols/HTTP/AsImplemented.html. Acessado em: 6 de fevereiro de 2009.

BERNERS-LEE, T.; FIELDING, R.; FRYSTYK, H. *Hypertext Transfer Protocol – HTTP/1.0*. IETF, maio 1996. RFC 1945 (Informational). (Request for Comments, 1945). Disponível em: <http://www.ietf.org/rfc/rfc1945.txt>.

BERNERS-LEE, T.; MASINTER, L.; MCCAHILL, M. *Uniform Resource Locators (URL)*. IETF, dez. 1994. RFC 1738 (Proposed Standard). (Request for Comments, 1738). Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986. Disponível em: <http://www.ietf.org/rfc/rfc1738.txt>.

ELSON, J.; CERPA, A. *Internet Content Adaptation Protocol (ICAP)*. IETF, abr. 2003. RFC 3507 (Informational). (Request for Comments, 3507). Disponível em: <http://www.ietf.org/rfc/rfc3507.txt>.

FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T. *Hypertext Transfer Protocol – HTTP/1.1*. IETF, jun. 1999. RFC 2616 (Draft Standard). (Request for Comments, 2616). Updated by RFC 2817. Disponível em: <http://www.ietf.org/rfc/rfc2616.txt>.

IPERF. 2009. http://iperf.sourceforge.net. Acessado em: 03 de março de 2009.

KAMP, P.-H. *The Varnish Project*. 2009. http://varnish.projects.linpro.no. Acessado em: 03 de março de 2009.

MOSBERGER, D. *Httperf*. 2009. http://www.hpl.hp.com/research/linux/httperf/. Acessado em: 03 de março de 2009.

MOSBERGER, D.; JIN, T. httperf - a tool for measuring web server performance. In: *In First Workshop on Internet Server Performance*. [S.l.]: ACM, 1998. p. 59–67.

NAGARAJ, S. V. *Web Caching and Its Applications*. [S.l.]: Kluwer Academic Publishers, 2004.

NETCRAFT. *February 2009 Web Server Survey*. [S.l.], 2009. Acessado em: 03 de março de 2009.

ORTEGA, A. Lopez. *Cherokee Web Server*. 2009. http://www.cherokee-project.com. Acessado em: 03 de março de 2009.

RABINOVICH, M.; SPATSCHEK, O. *Web caching and replication*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0-201-61570-3.

SIEMINSKI, A. The impact of Proxy caches on Browser Latency. *International Journal of Computer Science & Applications*, II, n. II, p. 5 – 21, 2005.

STEVENS, W. R. *TCP/IP Illustrated, Volume 1: The Protocols*. [S.l.]: Addison-Wesley, 1994. ISBN 0-201-63346-9.

THE APACHE SOFTWARE FOUNDATION. *Apache HTTP Server Project*. [S.l.], 2009. Acessado em: 03 de março de 2009.

THE LIGHTTPD TEAM. *Lighttpd fly light*. [S.l.], 2009. Acessado em: 03 de março de 2009.

THE SQUID TEAM. *The Proxy/Cache Squid*. [S.l.], 2009. Acessado em: 03 de março de 2009.

TSIMELZON Mark. *ESI Language Specification 1.0*. [S.l.]: World Wide Web Consortium, August 2001. http://www.w3.org/TR/esi-lang.

WESSELS, D. *Web Caching*. First edition. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001. ISBN 1-56592-536-X.

WESSELS, D. *Squid: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2004. ISBN 0596001622.

ZONA RESEARCH. *The Economic Impacts of Unacceptable Web Site Download Speeds*. [S.l.], 1999. Acessado em: 11 de fevereiro de 2009.

Apêndice A

Arquivos de configuração

Esse apêndice apresenta vários arquivos de configuração utilizados no trabalho.

A.1 Arquivo /etc/sysctl.conf

A listagem a seguir deve ser **adicionado** ao arquivo /etc/sysctl.conf para efetivar a cada inicialização do sistema operacional as parametrizações discutidas na Seção 4.3.1 e suas subseções.

```
net.ipv4.tcp_max_syn_backlog = 2048
net.ipv4.ip_local_port_range = 1025 65535
net.ipv4.tcp_fin_timeout = 10
```

A.2 Arquivo httpd.conf

A seguir está listado o arquivo de configuração do Apache utilizado nos ensaios de carga.

```
# Configuração do Apache 2.2 para ensaios de carga
```

```
ServerRoot "/usr/local/acelhttp/apache2"

Listen 1024

User nobody
Group nogroup

ServerAdmin lucas.brasilino@gmail.com
ServerName localhost

DocumentRoot "/usr/local/acelhttp/apache2/htdocs"

EnableSendfile on
ServerLimit 25
StartServers 5
ThreadLimit 256
MaxClients 6400
ThreadsPerChild 256
MaxRequestsPerChild 0

<Directory />
    Options FollowSymLinks
    AllowOverride None
    Order deny,allow
    Deny from all
</Directory>

<Directory "/usr/local/acelhttp/apache2/htdocs">
    Options Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>

<IfModule dir_module>
    DirectoryIndex index.html
</IfModule>

<FilesMatch "^\.ht">
    Order allow,deny
    Deny from all
    Satisfy All
</FilesMatch>

ErrorLog "logs/error_log"
LogLevel warn
<IfModule log_config_module>
    LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
               %\"%{User-Agent}i\"" combined
    LogFormat "%h %l %u %t \"%r\" %>s %b" common
    <IfModule logio_module>
      LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
                \"%{User-Agent}i\" %I %O" combinedio
    </IfModule>
    CustomLog "logs/access_log" common
</IfModule>
```

```
<IfModule alias_module>
    ScriptAlias /cgi-bin/ "/usr/local/acelhttp/apache2/cgi-bin/"
</IfModule>

<IfModule cgid_module>
</IfModule>
<Directory "/usr/local/acelhttp/apache2/cgi-bin">
    AllowOverride None
    Options None
    Order allow,deny
    Allow from all
</Directory>

DefaultType text/plain
<IfModule mime_module>
    TypesConfig conf/mime.types
    AddType application/x-compress .Z
    AddType application/x-gzip .gz .tgz
</IfModule>

<IfModule ssl_module>
SSLRandomSeed startup builtin
SSLRandomSeed connect builtin
</IfModule>

<IfDefine SEM_CACHE>
        Header set Cache-Control no-cache
</IfDefine>

<IfDefine CACHE_PARCIAL>
        Header set Cache-Control max-age=84600
        SetEnvIf Request_URI "fws\.jpg$" NAO_CACHEIA=1
        Header set Cache-Control no-cache env=NAO_CACHEIA
</IfDefine>

<IfDefine CACHE_TOTAL>
        Header set Cache-Control max-age=86400
</IfDefine>
```

A.3 Arquivo `varnish.vcl`

A seguir encontra-se listado o arquivo `varnish.vcl` citado na Seção 4.5.2.2.

```
backend default {
.host = "127.0.0.1";
.port = "1024";
}
```

```
sub vcl_fetch {
        if (obj.http.Cache-Control ~ "no-cache") {
                return (pass);
        }
    return (deliver);
}
sub vcl_deliver {
        if (resp.http.Cache-Control ~ "no-cache") {
                remove resp.http.Age;
        }
     return (deliver);
}
```

A.4 Arquivo `squid.conf`

A seguir tem-se o arquivo `squid.conf` discutido na Seção 4.5.2.4.

```
cache_peer localhost parent 1024 0 no-query no-digest no-netdb-exchange originserver

http_port 80 vhost act-as-origin http11
icp_port 0

cache_effective_user nobody
cache_effective_group nogroup

cache_dir aufs /usr/local/acelhttp/squid/var/cache 2048 16 256
cache_replacement_policy heap GDSF

cache_mem 1536 MB
maximum_object_size_in_memory 256 KB
memory_replacement_policy heap GDSF


acl all src all
acl manager proto cache_object
acl localhost src 127.0.0.1/32
acl to_localhost dst 127.0.0.1/32
acl web dst 192.168.15.102/32

acl localnet src 10.0.0.0/8        # RFC1918 possible internal network
acl localnet src 172.16.0.0/12        # RFC1918 possible internal network
acl localnet src 192.168.0.0/16        # RFC1918 possible internal network

acl Safe_ports port 80                # http

http_access allow manager localhost
http_access allow manager localnet
http_access deny manager
```

```
http_access deny !Safe_ports

http_access allow to_localhost
http_access allow web
http_access deny all

hierarchy_stoplist cgi-bin ?
access_log none

acl apache rep_header Server ^Apache
broken_vary_encoding allow apache
coredump_dir /usr/local/acelhttp/squid/var/cache
```