Otimizando Servidores Web de Alta Demanda

Renato Hirata

Dissertação de Mestrado

Otimizando Servidores Web de Alta Demanda

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Renato Hirata e aprovada pela Banca Examinadora.

Campinas, 11 de Abril de 2002

Paulo Lício de Geus (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Otimizando Servidores Web de Alta Demanda

Renato Hirata

Abril de 2002

Banca Examinadora:

- Prof. Dr. Paulo Lício de Geus (Orientador)
  Instituto de Computação, UNICAMP
- Prof. Dr. Adriano Mauro Cansian
  IBILCE, UNESP
- Prof. Dr. Rogério Drummond
  Instituto de Computação, UNICAMP
- Prof. Dr. Célio Cardoso Guimarães (Suplente)
  Instituto de Computação, UNICAMP

Resumo

Com o explosivo crescimento da Web, sua popularização e sua grande importância como meio de comunicação global, cada vez mais a atenção se volta para questões de desempenho. Pelo fato de ser um serviço relativamente recente, são poucos os estudos que abordam de modo geral todos os fatores que influenciam no seu desempenho e como tais fatores estão inter-relacionados. Assim, o objetivo deste trabalho é abordar o problema de desempenho na Web como um todo, levantando e analisando todos os componentes relacionados, fornecendo embasamento teórico para o entendimento desses tópicos e apresentando suas possíveis soluções, de forma a cobrir esta lacuna na bibliografia existente. Em especial, se concentra em tópicos de software na porção servidora de sistemas de grande demanda. Como resultado, o leitor encontrará um texto único na otimização servidores de grande porte, que é um trabalho de referência e ao mesmo tempo um guia de otimização[1].

1. Este trabalho foi desenvolvido junto à Comissão de Vestibulares da Universidade Estadual de Campinas e seus resultados utilizados para a implementação do *site* de alta demanda desta instituição.

Abstract

With the Web explosive growth, its popularization and its great importance as a way of global communication , more and more attention is devoted to its performance issues. Due to the Web being a relatively new service, very few works approach the issue of performance affecting factors in a broad sense, as well as their inter-relationship. The goal of this work is to fill that gap by enumerating and analysing every component that affects Web performance. This is done by covering theoretical issues for a clear understanding of all topics involved and by presenting possible solutions and suggestions. The main focus of this work is on the server portion of high performance systems. As a result, the reader will find a unique text on optimizing performance of very large Web sites, that is both a reference work and a tuning guide[1].

1. This work was developed with the support of the "Vestibular" Comission of the State University of Campinas, and its results were used to implement the high demanding site of this institution.

Dedico aos meus pais, e minha avó em memória, as pessoas que mais me ajudaram em todos os sentidos, fornecendo algo mais importante que o conhecimento técnico, mas o conhecimento da vida, representado em atos concretos de amor, humildade, caráter e respeito ao próximo.

Agradecimentos

Ao professor Paulo Lício de Geus, por toda orientação para o desenvolvimento deste trabalho.

À minha avó, Joana, que não pôde estar presente "fisicamente" agora, mas sempre esteve e estará em memória e espírito.

Aos meus pais, Francisco e Hideko, por tudo.

Aos parentes próximos, em especial minhas tias, Luzia, Rogéria, Ana e Cida, e meu tio Gabriel, por todo o apoio em minha estadia em Campinas.

Aos meus irmãos, José Carlos, Margarete, Flávio e Caio, distantes, mas presentes.

À Liliane, pelo apoio e compreensão.

Ao pequeno Sávio, pela inspiração.

Aos meus amigos, que sempre me apoiaram e acreditaram na realização deste trabalho.

À Comissão Permanente para os Vestibulares da Unicamp, por todo suporte e amizade.

Ao Centro de Computação da Unicamp, em especial ao Fábio Mengue, pelo apoio.

A Deus, pela oportunidade de realização de mais este importante passo em minha vida.

Conteúdo

1 Introdução e Motivações **17**

2 O Sistema Operacional **21**
2.1 O Unix . . . 22
2.1.1 Histórico . . . 22
2.1.2 Sabores de Unix . . . 23
2.1.2.1 Linux . . . 24
2.1.2.2 BSD . . . 24
2.1.2.3 Solaris . . . 24
2.1.2.4 Digital Unix . . . 25
2.1.2.5 Irix . . . 25
2.1.2.6 Mach OS . . . 25
2.1.2.7 HP-UX . . . 25
2.1.3 O UNIX como servidor TCP/IP . . . 25
2.2 O Linux . . . 26
2.2.1 Características Gerais . . . 26
2.2.2 Controle de versões do Linux . . . 27
2.2.3 Por que o Linux? . . . 27
2.3 O Kernel . . . 28
2.3.1 Definição . . . 28
2.3.2 Modos de execução e as System Calls . . . 28
2.3.3 Arquitetura de kernels . . . 29
2.4 Otimizando o kernel do Linux . . . 30
2.4.1 Compilação otimizada . . . 31
2.4.1.1 Configurando o novo kernel . . . 31
2.4.1.2 A Compilação . . . 34
2.4.1.3 Configurando o LILO . . . 34
2.4.2 Configurando parâmetros do kernel . . . 37
2.4.2.1 Estrutura do /proc . . . 37
2.4.2.2 Alterando parâmetros: /proc/sys . . . 41
2.4.2.3 Estrutura do /proc/sys . . . 42
2.5 . . . Processos e Threads 42
2.5.1 Conceitos . . . 43
2.5.1.1 Escalonamento . . . 43
2.5.1.2 Preemptividade . . . 43
2.5.1.3 Criação de Processos . . . 43

2.5.2 Threads . . . 44
2.5.2.1 Definição . . . 44
2.5.2.2 Forma de implementação do pacote de threads . . . 45
2.5.2.3 Suporte a threads no Linux . . . 46
2.5.2.4 Implementação do kernel via threads . . . 47
2.5.3 Otimizações possíveis . . . 48
2.6 Sistema de Arquivos . . . 49
2.6.1 Implementação de Sistema de Arquivos . . . 50
2.6.1.1 Bloco . . . 50
2.6.1.2 Superbloco . . . 52
2.6.1.3 Inode . . . 52
2.6.1.4 Diretórios . . . 53
2.6.1.5 Descritor de Arquivo . . . 55
2.6.2 Manutenção de Sistema de Arquivos . . . 56
2.6.2.1 Desfragmentação de Discos . . . 56
2.6.2.2 Particionamento Adequado . . . 57
2.6.2.3 Evitando o registro de último acesso de leitura . . . 58
2.6.2.4 Evitar o uso de links simbólicos . . . 59
2.6.2.5 Otimizar as variáveis de PATH . . . 59
2.6.2.6 Evitar "~user" . . . 60
2.6.3 Cache de Disco . . . 60
2.6.3.1 Tamanho do Buffer . . . 60
2.6.3.2 Atualização em Disco . . . 61
2.6.3.3 Flushing de buffers no Linux (kernel 2.4) . . . 62
2.6.3.4 Evitando o cache: Raw Devices . . . 64
2.6.4 Otimizando acesso a discos IDE no Linux via hdparm . . . 64
2.6.4.1 Exibindo configuraçoes e desempenho atuais . . . 65
2.6.4.2 Habilitando o DMA . . . 66
2.6.4.3 Habilitando transferências de 32 bits . . . 66
2.6.4.4 Habilitando a transferência de múltiplos setores por interrupção . . . 67
2.7 Gerenciamento de memória . . . 67
2.7.1 Funções do Gerenciador de memória . . . 67
2.7.2 Como a memória é utilizada . . . 68
2.7.3 Memória Virtual . . . 69
2.7.3.1 Segmentação da memória em páginas . . . 69
2.7.3.2 Swapping . . . 70
2.7.4 Mapeando arquivos em memória . . . 72
2.8 Implementação da pilha TCP/IP no Kernel . . . 73
2.8.1 Estrutura . . . 73
2.8.2 Organização do código . . . 74
2.8.3 Estruturas de dados envolvidas . . . 76
2.8.4 mbuf (Memory Buffer) . . . 76
2.9 Ferramentas de monitoramento . . . 78
2.9.1 ps (Process Status) . . . 78
2.9.2 top . . . 79
2.9.3 vmstat . . . 79
2.9.3.1 Processos . . . 79

2.9.3.2 Memória . . . 79
2.9.3.3 Swap . . . 80
2.9.3.4 I/O . . . 80
2.9.3.5 Sistema . . . 80
2.9.3.6 CPU . . . 80
2.9.4 sar (System activity reporter) . . . 80
2.9.5 netstat . . . 81
2.9.5.1 Conexões Ativas . . . 81
2.9.5.2 Exibindo portas abertas e identificando os programas servidores . . . 82
2.9.5.3 Informações de Interfaces . . . 82
2.9.5.4 Contadores de Protocolos . . . 82
2.9.6 Ferramentas de Monitoramento de Tráfego . . . 83

3 O TCP 85

3.1 Interface com a camada de Aplicação . . . 85
3.1.1 Sockets . . . 86
3.1.2 Primitivas . . . 86
3.1.3 Ilustrando conexão cliente servidor via sockets . . . 89
3.2 Segmentos TCP . . . 90
3.3 Estabelecimento e Fechamento de Conexão . . . 93
3.3.1 O three-way-handshake . . . 93
3.3.1.1 Implementação . . . 94
3.3.1.2 Tamanho das filas de requisições . . . 95
3.3.2 Atendimento da conexão pela aplicação . . . 97
3.3.3 Ataques de negação de serviços e os “syn_cookies” do Linux . . . 97
3.3.4 Tamanho e Fragmentação de Segmentos . . . 98
3.3.5 Path Maximum Transmission Unit Discovery . . . 100
3.4 Fechamento de conexão . . . 101
3.5 Controle do Fluxo de Dados . . . 102
3.5.1 Fluxo de bytes e as flags Push e Urgent . . . 102
3.5.2 Buffers e Janelas de envio e recepção . . . 103
3.5.3 Tamanho da Janela e o bandwidth-delay product . . . 104
3.5.4 Slow – Start . . . 106
3.5.5 Algoritmo de prevenção de congestionamento . . . 106
3.6 Controle de Timers . . . 110
3.6.1 Timer de Retransmissão . . . 110
3.6.2 Timer de Persistência . . . 111
3.6.3 Keepalive timer . . . 111
3.6.4 Close-Wait Interval Timer . . . 112
3.7 Extensões do TCP para Redes rápidas . . . 112
3.7.1 Opção Window-scale . . . 113
3.7.2 Opção Timestamp . . . 113
3.7.3 PAWS (Protection Against Wrapped Sequence Numbers) . . . 114
3.8 T/TCP - Transaction TCP . . . 114

4 O HTTP 117

4.1 Histórico . . . 117

4.2 O protocolo HTTP . . . 117
4.3 Características . . . 118
4.3.1 Sem Estados . . . 118
4.3.2 Assimétrico . . . 119
4.3.3 Baseado em Texto . . . 119
4.4 Formato das mensagens HTTP . . . 119
4.4.1 Mensagens de Requisição HTTP . . . 120
4.4.1.1 Linha de Requisição . . . 120
4.4.1.2 Cabeçalho . . . 120
4.4.2 Mensagens de Resposta HTTP . . . 121
4.4.2.1 Linha de Status . . . 122
4.4.2.2 Cabeçalho . . . 123
4.5 Interações com o TCP . . . 124
4.5.1 Conexões não Persistentes . . . 124
4.5.2 Conexões Persistentes . . . 126
4.5.3 Pipelining . . . 127
4.6 Controle de Cache e Otimização de Banda . . . 127
4.6.1 O GET condicional . . . 127
4.6.2 Range Requests . . . 128
4.6.3 Compressão de Dados . . . 128

5 Software Servidor Web 129
5.1 Arquitetura de Servidores Web . . . 129
5.1.1 Questões de Desempenho na Implementação de Servidores Web . . . 130
5.1.1.1 Cópia e Leitura de Dados . . . 130
5.1.1.2 Notificação de Eventos . . . 130
5.1.1.3 Caminho de comunicação dos dados através das camadas de rede 130
5.1.2 Implementação do Servidor Web no espaço do usuário . . . 131
5.1.2.1 Invocados pelo inetd . . . 131
5.1.2.2 Servidores Web Multi-Processos (MP) . . . 132
5.1.2.3 Servidores Web Multi-Threads (MT) . . . 133
5.1.2.4 Single Process Event Driven (SPED) . . . 134
5.1.2.5 Modelos Híbridos . . . 136
5.1.3 Implementação do Servidor Web no espaço do kernel . . . 137
5.2 Exemplos de Servidores Web . . . 138
5.2.1 Apache . . . 138
5.2.2 IIS . . . 138
5.2.3 NCSA . . . 139
5.2.4 iPlanet (Netscape) . . . 139
5.2.5 Zeus . . . 139
5.2.6 kHTTPd . . . 139
5.2.7 TUX . . . 140
5.3 Apache . . . 141
5.3.1 Histórico . . . 141
5.3.2 Os Módulos do Apache . . . 142
5.3.3 Arquitetura para Atendimento de Conexões Simultâneas . . . 142
5.3.3.1 Apache 1.3 para Unix . . . 142

5.3.3.2 Apache 2.0 . . . 144
5.3.4 Otimizando o Apache . . . 145
5.3.4.1 Compilação Otimizada . . . 145
5.3.4.2 Configurando parâmetros de execução . . . 145
5.4 Geração de Páginas Dinâmicas . . . 152
5.4.1 CGI (Common Gateway Interface) . . . 152
5.4.1.1 Problemas de Desempenho do CGI . . . 152
5.4.2 FastCGI . . . 154
5.4.3 APIs . . . 154
5.4.4 Scripts embutidos . . . 156
5.4.4.1 SSI . . . 156
5.4.4.2 PHP . . . 156
5.4.4.3 ASP . . . 157
5.4.5 Mod_Perl . . . 157
5.4.6 Java . . . 157
5.4.6.1 Porção Cliente - Applets . . . 157
5.4.6.2 Porção servidora – Servlets . . . 158

6 Distribuição de carga 161
6.1 Servidores Web Distribuídos Localmente . . . 162
6.1.1 Arquitetura baseada em cluster . . . 162
6.1.1.1 Mecanismos de roteamento . . . 163
6.1.1.2 Classificação quanto ao fluxo de saída . . . 164
6.1.2 Arquitetura Web Distribuída . . . 167
6.1.2.1 Distribuição selecionada pelo cliente . . . 167
6.1.2.2 Mecanismo de roteamento por DNS puro . . . 167
6.1.2.3 Mecanismo de roteamento no servidor Web . . . 169
6.1.3 Algoritmos de Escalonamento . . . 170
6.1.4 NAT (Network Address Translation) . . . 171
6.1.4.1 Segurança (Masquerading) . . . 172
6.1.4.2 Distribuição de Carga (Servidores Virtuais) . . . 173
6.1.5 Proxy Reverso . . . 173
6.2 Servidores Web Distribuídos Geograficamente . . . 174
6.2.1 Arquiteturas . . . 174
6.2.1.1 Servidores Espelhados com seleção pelo cliente . . . 174
6.2.1.2 Servidores Web Distribuídos Geograficamente . . . 174
6.2.1.3 Cluster de Servidores Distribuídos Geograficamente . . . 175
6.2.2 Questões sobre Escalonamento Global . . . 176
6.3 Outras Considerações . . . 176
6.3.1 HTTP é escalável . . . 176
6.3.2 Replicação e Sincronização . . . 177
6.4 Implementações Comerciais e protótipos para balanceamento de carga . . . 177
6.4.1 CISCO . . . 178
6.4.1.1 Cisco LocalDirector . . . 178
6.4.1.2 Distributed Director . . . 179
6.4.2 F5 Networks . . . 179
6.4.2.1 F5 – Big IP Controller . . . 179

6.4.2.2 F5 – 3DNS Controller 180
6.4.3 Classificação de Produtos e protótipos 180
6.4.3.1 Distribuídores de nível 4 181
6.4.3.2 Distribuídores de nível 7 181

7 Conclusão **183**
7.1 Resultados práticos obtidos 188
7.2 Trabalhos Futuros 191

Bibliografia **193**

Índice de Figuras

2.1 Relação entre aproveitamento de espaço e taxa de transferência 51
2.2 Representação de um inode 53
2.3 Dois processos lendo a mesma página no processo tradicional 72
2.4 Dois processos mapeando a mesma página em seus respectivos espaços de endereçamento. 73
2.5 Organização do código TCP/IP. Ao lado são referenciadas as camadas correspondentes no protocolo OSI. 74
2.6 Comunicação entre as camadas. 75
2.7 Relações fundamentais entre estruturas de dados do kernel para implementação de sockets. 76
2.8 Principais estruturas de dados envolvidas na implementação do TCP/IP. 77
3.1 Uso das primitivas. 89
3.2 Cabeçalho de um segmento TCP. 91
3.3 Segmentos trocados no three-way handshake 94
3.4 Filas (so_q0 e so_q) mantidas para o atendimento de conexões. 95
3.5 Relação entre MTU e MSS 99
3.6 Troca de segmentos durante o fechamento de conexão. 101
3.7 Duplicando-se o RTT, duplica-se a capacidade do duto. 105
3.8 Duplicando-se a largura de banda, duplica-se a capacidade do duto. 105
3.9 Visualização do slow-start e prevenção a congestionamento. 108
4.1 Formato Geral de uma mensagem de requisição HTTP. 121
4.2 Formato geral de uma mensagem de resposta HTTP. 123
4.3 Troca de pacotes e RTTs para uma conexão TCP. Linhas claras mostram pacotes requeridos pelo TCP que não afetam a latência da comunicação 125
5.1 No modelo MP cada servidor manipula uma requisição por vez. Processos executam estágios seqüencialmente. 133
5.2 O modelo MT utiliza-se de um único espaço de endereçamento e múltiplas threads concorrentes de execução, cada uma manipulando uma requisição distinta. 134
5.3 O modelo SPED se utiliza de um único processo para executar todo o processamento e atividade de disco baseado em eventos. Múltiplas requisições são manipuladas por um único processo. 135
5.4 O modelo AMPED utiliza-se de um único processo para o processamento orientado à evento, mas vários auxiliares para manipular operações de disco. 137
5.5 Modelo de processos do Apache 1.3. 143
5.6 Servidor Web e o processo CGI 152

Índice de Tabelas

2.1 Entradas específicas por processo em /proc. 38
2.2 Informações sobre o kernel em /proc 38
2.3 Informações "IDE" em /proc/ide 40
2.4 Informaçoes de rede em /proc/net 40
2.5 Informações sobre terminais em /proc/net 41
3.1 Primitivas TCP 89
7.1 Principais parâmetros customizados relacionados ao Sistema Operacional e ao TCP 190
7.2 Principais parâmetros customizados relacionados ao Apache. 190

Ficha Catalográfica elaborada pela Biblioteca do IMECC da UNICAMP

Hirata, Renato
H613o Otimizando Servidores Web de alta demanda / Renato Hirata. – Campinas, SP: [s.n.], 2002.

Orientador: Paulo Lício de Geus.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Redes de computação - Administração. 2. Desempenho 3. Internet (Redes de computadores). 4. Redes de computação - Protocolos. 5. UNIX (Sistema operacional de computador) - Otimização. 6. TCP/IP (Protocolos de redes de computação) 7. World Wide Web (Sistema de Recuperação de Informação). I. Geus, Paulo Lício de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título

Capítulo 1

Introdução e Motivações

A Internet é uma das grandes revoluções tecnológicas da comunicação dos últimos anos. Grande parte desta popularidade no meio comercial deveu-se à tecnologia Web, que introduziu uma interface gráfica e intuitiva baseada em *hiperlinks*, o que possibilitou a sua fácil utilização por pessoas que não apenas do meio computacional. Este crescimento exponencial da rede e sua utilização em massa, têm alterado profundamente o modo como as corporações e entidades operam seus negócios, de modo a não se tornarem obsoletas e conseqüentemente mais competitivas, utilizando-se das facilidades introduzidas por esta nova tecnologia.

Cada vez mais surgem aplicações baseadas na Web, podendo-se se citar como as mais importantes: o ensino a distância, bibliotecas digitais, aplicações Intranet, e principalmente o comércio eletrônico.

À medida que cresce o número de entidades que confiam na Web como ferramenta em seus ramos de atuação, aumenta-se a atenção para questões de desempenho, uma vez que a imagem da entidade perante o cliente, passa a ser moldada a partir da imagem que os clientes têm do *site*. Além disso, em *sites* comerciais (que empregam o comércio eletrônico) a degradação de desempenho pode representar prejuízo direto para a entidade, pela diminuição do número de vendas e pela perda de um cliente para um concorrente que possui um *site* mais eficiente.

Um outro fator a se considerar é a utilização da tecnologia Web para a implantação de sistemas internos, por meio de uma Intranet. Esta solução tem sido vastamente adotada principalmente pela possibilidade da criação de uma aplicação extremamente portável, pois funcionará sobre um *browser*, que é bastante comum em qualquer plataforma. Assim, problemas de desempenho na tecnologia Web que foi usada como suporte, podem refletir também na

produtividade interna da entidade, e não somente na imagem da entidade perante o público externo.

Em comparação a uma demanda tão grande para se atender todos os requisitos de desempenho na Web, nota-se um relativo atraso nas soluções que satisfaçam tais requisitos, principalmente pelo fato desta demanda ter surgido de forma tão repentina.

Atualmente pode-se encontrar na literatura textos que abordam um ou outro tópico isolado relativo ao desempenho em servidores Web. Todavia, observa-se uma lacuna relativa a trabalhos que abordem de maneira geral todos os tópicos, como eles estão inter-relacionados, além de um embasamento teórico que justifique as propostas de otimizações e que contribua para o melhor entendimento do leitor.

É importante notar que a natureza da degradação de desempenho do servidor pode estar presente em um grande número de componentes, e um simples *upgrade* de hardware, em geral não é suficientemente aceitável nem tampouco possui uma relação custo-benefício satisfatória. Latência pode surgir não apenas pela impotencialidade do hardware, mas também através de inúmeros outros fatores como protocolos utilizados, configurações de sistema operacional, do software servidor, da utilização de métodos inadequados para se construir páginas dinâmicas, dentre outras possibilidades.

A solução de problemas de desempenho pode ainda ser obtida pelo emprego de métodos mais complexos como a distribuição de carga entre servidores, que possivelmente podem estar em locais geograficamente distantes, e que necessitam de algum mecanismo para prover uma distribuição adequada da carga. O hardware é de fato um fator importante, mas é apenas um componente dentre um emaranhado de outros fatores.

Assim, o objetivo deste trabalho é levantar precisamente todos esses fatores inerentes ao desempenho na Web sob o ponto de vista de software do servidor, analisá-los isoladamente e em suas interações, apontando possíveis soluções, além de fornecer um completo embasamento teórico de forma a se tornar uma importante referência para administradores de *sites* de alta demanda.

O leitor irá perceber que esta dissertação não segue uma estrutura convencional. O assunto e o objetivo do trabalho sugeriram uma estrutura mais plana, sem uma separação clara entre introdução, ambientação do problema, trabalhos correlatos, tópico específico de pesquisa, resultados e conclusões. O texto é dividido em tópicos de interesse não hierárquicos, e para

cada tema específico de estudo soluções e sugestões de melhoria são apresentadas junto à discussão do tema propriamente dito.

Este trabalho é dividido em 6 partes, cada uma cobrindo um tópico considerado chave no estudo de desempenho na Web. O capítulo 2 apresenta o Sistema Operacional, sua estrutura, funcionamento e pontos específicos em cada subsistema que podem ser otimizados de forma a prover uma melhor utilização do hardware. O capítulo 3 apresenta o TCP, a camada mais complexa e com a maior quantidade de pontos de estudos quanto ao desempenho, de toda a hierarquia TCP/IP. No capítulo seguinte se aborda o HTTP e suas interações com o TCP/IP.

O capítulo 5 descreve o software servidor Web quanto às suas possíveis arquiteturas, exemplos comerciais ou livres, otimizações e interações com o Sistema Operacional. O capítulo 6 descreve mecanismos de distribuição de carga que são aplicados quando um único servidor ou a rede de acesso não está sendo capaz de atender à demanda de requisições.

O capítulo 7 encerra o trabalho, apresentando as principais conclusões obtidas e sugestões de trabalhos futuros.

Em cada capítulo são apresentados aspectos relacionados ao desempenho Web, acompanhado de um embasamento teórico ao mesmo tempo que se propõe sugestões de otimizações quando pertinente. Esta estrutura em formato de manual, visa facilitar a assimilação por parte do leitor.

As informações e conclusões foram obtidas a partir de experimentos práticos e principalmente das referências contidas na Bibliografia, construída para servir de guia para o leitor interessado em se aprofundar em um tópico em específico.

Os conhecimentos adquiridos neste trabalho serviram de base para a implementação do *site* da Comissão de Vestibulares da Unicamp, orgão que contribuiu para este estudo fornecendo a infra-estrutura necessária e um ambiente de aplicação real importantissímo, já que sofre cargas altíssimas quando da divulgação dos resultados dos vestibulares.

Capítulo 2

O Sistema Operacional

Considerando o estudo do desempenho Web, pode-se classificar o Sistema Operacional como a unidade software presente entre o hardware e o software servidor Web, cuja função é servir de mediador entre essas duas entidades, traduzindo requisições do servidor Web em ações de hardware.

O hardware sempre terá seu desempenho máximo limitado por especificações físicas, mas muitas vezes diversos motivos impedem que tais limites sejam alcançados. Para se obter o máximo que o hardware é capaz, é necessária uma correta configuração do Sistema Operacional.

De forma resumida, as ações a serem executadas pelo Sistema Operacional quando atuando em um servidor Web são: obter requisições da rede, encontrar o arquivo ou executar um programa correto e retornar o resultado através da rede de maneira mais eficiente possível [74].

Diferente da aparente simplicidade dos resultados são os processos envolvidos para a sua obtenção que demandam o estudo detalhado do Sistema Operacional.

Assim sendo, este capítulo visa descrever conceitualmente a arquitetura e o funcionamento de subsistemas específicos do sistema operacional, que mais se relacionam com questões de desempenho em servidores Web, oferecendo para cada subsistema, propostas de otimizações. A base de estudo são sistemas Unix.

Este capítulo abrange:

- Sistemas Operacionais Unix
- O Linux
- Otimização do *kernel*
- Processos e *Threads*
- Sistemas de Arquivos

- Gerenciamento de Memória
- Ferramentas de monitoramento

2.1 O Unix

2.1.1 Histórico

A primeira versão do Unix foi desenvolvida por Ken Thompson em *Bell Labs* (AT&T) em meados dos anos 70, baseando-se nas idéias de multiusuário e multitarefa do projeto de pesquisa do governo chamado *MULTICS*.

O *UNICS* (*UNiplexed Information and Computing Service* – nome original do sistema) impressionou por ser compacto e eficiente. Essas qualidades incentivaram outros cientistas da Bell Labs, dentre eles Dennis Ritchie que sugeriu que o Unics fosse reescrito utilizando a linguagem C, recém criada por ele, e assim foi feito. A partir de então, o sistema agora batizado como Unix tinha uma vantagem a mais sobre outros sistemas: ser escrito em uma linguagem de alto nível, tendo assim seu código fonte inteligível. Somente uma pequena porção do seu *kernel* era escrito na linguagem Assembly, o que significava também que portar o sistema para plataformas de hardware variadas era relativamente fácil [61].

Além de um Sistema Operacional que se tornaria dominante no mundo da computação, de quebra os cientista da Bell Labs desenvolviam a linguagem C, que se tornaria também um marco, uma das linguagens mais difundidas desde então.

Em 1974, Thompson e Ritchie publicaram um paper histórico [109] na ACM sobre o Unix que despertou o interesse de muitas universidades em obter o sistema. Uma vez que a AT&T era um monopólio de telefonia dos Estados Unidos e não poderia comercializar software, o código fonte do Unix poderia ser licenciado pelas universidades por uma taxa irrisória.

A Universidade da Califórnia em Berkeley em particular, continuou o desenvolvimento do Unix, acrescentando a ele a implementação do TCP/IP, financiado pelo DARPA (*Defense Advanced Research Projects Agency*), um conjunto de protocolos de rede que também viria a se torna um padrão de fato. À implementação do Unix em Berkeley foi dada o nome de BSD (*Berkeley Software Distribution*).

Além da implementação TCP/IP, outras melhorias significativas foram adicionadas à distribuição BSD do Unix, tais como o aperfeiçoamento do gerenciamento de memória, sistema de arquivos e adição de utilitários como editor de texto (*vi*), *shell* (csh), compiladores (Pascal e

Lisp). Todas essas melhorias levaram empresas como a Sun e DEC dentre outras a basear suas próprias versões do Unix no BSD ao invés da versão oficial (já batizada como *System V*) que também teve seu desenvolvimento continuado na AT&T. Como conseqüência, o BSD tornou-se bastante reconhecido nos meios acadêmicos, de pesquisa e de defesa, criando a divisão do Unix em dois campos distintos: o de Berkeley (BSD) e o da AT&T (System V).

Ao final dos anos 80, duas versões diferentes e bastante incompatíveis do Unix estavam amplamente em uso: o 4.3 BSD e o System V Release 3. Soma-se a isso várias outras versões comerciais que implementavam outras facilidades fora do padrão.

Essa fragmentação inibiu de maneira significativa as ambições comerciais do Unix, uma vez que era impossível para os desenvolvedores de software criarem pacotes únicos de programas que pudessem ser executados em qualquer versão do Unix (como acontece com o Microsoft Windows, por exemplo).

Assim, várias tentativas de padronização foram feitas, destacando-se o projeto POSIX (*Portable Operating System unIX*), mediado pelo IEEE Standard Board, que parcialmente cumpriu o seu papel de unir o BSD e o System V. Parcialmente pois, mesmo após o projeto POSIX, a IBM, DEC e HP uniram-se no consórcio "Open Software Foundation" (OSF) no desenvolvimento de mais um derivado do BSD, o OSF/1. A resposta da AT&T foi a união com a Sun na criação do projeto "Unix International" que continuava o desenvolvimento do System V.

O que se observa hoje no mundo Unix é o surgimento de diferentes variantes desse sistema, sendo que cada um deles se desenvolve em diferentes direções [65], e uma total padronização é ainda inexistente.

Nesse sentido, a linguagem Java surge como uma esperança, pois à medida que mais aplicações são desenvolvidas nessa linguagem, pequenas diferenças entre Sistemas Operacionais tendem a se tornarem irrelevantes, e estes passem a ser competitivos por fatores como desempenho e custo, não pela compatibilidade [74].

2.1.2 Sabores de Unix

Nesta seção são descritos alguns dos mais importantes sistemas operacionais derivados do Unix empregados como servidores Web.

2.1.2.1 Linux

O Linux é um *kernel* de sistema operacional inicialmente desenvolvido pelo finlandês Linus Torvalds em 1991. Sua plataforma inicial era a Intel, mas atualmente suporta outras arquitetura como Alpha, PowerPC e mesmo SPARC.

O Linux foi o Sistema Operacional escolhido como principal fonte de estudo deste trabalho e será mais detalhado adiante.

2.1.2.2 BSD

O BSD é atualmente similar e o principal "rival" do Linux no sentido de atuar em plataforma Intel e possuir o código fonte aberto. Devido a conflitos entre os grupos de desenvolvimento, existem três projetos distintos de BSD (FreeBSD, NetBSD e OpenBSD).

Ao contrário do Linux, seu desenvolvimento inclui não apenas o *kernel*, mas também utilitários e documentação. Uma outra diferença marcante é o fato de que cada BSD possui um projeto bastante controlado e coeso, enquanto o Linux possui o formato mais livre. Assim, o BSD é mais indicado para tarefas onde a confiabilidade seja um fator crítico [99].

2.1.2.3 Solaris

Solaris é a versão do Unix desenvolvida pela Sun Microsystems. Inicialmente derivado de Berkeley (SunOS) tornou-se compatível com System V Release 4 quando do advento do projeto "Unix International" passando a se chamar Solaris. É obviamente o Sistema Operacional dominante na plataforma SPARC, alcançando junto com esta o status de Unix mais utilizado no mundo comercial.

Tem-se como pontos fortes do Solaris a eficiência na alocação de memória e na implementação da pilha TCP/IP [74]. Embora a aquisição do código fonte ainda exija uma demasiada burocracia, a sua falta não compromete a possibilidade de *tuning* deste sistema, uma vez que possui uma grande quantidade de parâmetros de *kernel* alteráveis. Todavia, instalações padrões do Solaris já vêm preparadas para atuar como servidores Web de grande porte, demandando assim poucas configurações adicionais.

2.1.2.4 Digital Unix

O Digital Unix foi um dos primeiros sistemas operacionais de 64 bits. É bastante conhecido por seu bom desempenho de I/O, obtido através de *drivers* de discos eficientes e boas implementações TCP/IP. É bastante escalável em sistemas multiprocessados, tendo como principal vitrine o mecanismo de busca *Altavista* (www.altavista.com).

2.1.2.5 Irix

Versão do Unix desenvolvido pela Silicon Graphics para rodar exclusivamente em suas estações gráficas de alto desempenho. Embora otimizado para aplicações gráficas, características como rápido acesso a memória e disco fazem do conjunto da Silicon Graphics também bastante utilizado como servidores Web de grande demanda. *http://www.sgi.com* possui maiores informações sobre o *tuning* do Irix para serviços Web.

2.1.2.6 Mach OS

O Mach OS foi desenvolvido em Carnegie-Mellon e proveu a base para o desenvolvimento do Sistema Operacional NextStep, onde a implementação original do primeiro *browser* HTTP foi feita por Tim Berners-Lee [16] no CERN na Suíça. O Mach OS ficou conhecido também pela sua arquitetura de *microkernels*. Atualmente não há mais desenvolvimento do Mach OS original em Carnegie-Mellon, porém outras entidades como a OSF e a Universidade Utah ainda continuam o projeto [29]. A Apple através do MacOS X usa FreeBSD sobre Mach [10].

2.1.2.7 HP-UX

Criado pela Hewlett Packard, HP-UX é projetado para sua linha de servidores. É uma versão comercial de Unix conhecida pela sua utilização em aplicações comerciais corporativas [131]. Dentre suas principais características inclui-se o sistema de arquivos Veritas. Informações sobre otimização do HP-UX pode ser encontrado em [35].

2.1.3 O UNIX como servidor TCP/IP

O Unix tem como características principais a sua longa história e o fato de ter surgido como um Sistema Operacional de código fonte aberto, o que possibilitou que seu desenvolvimento fosse baseado no trabalho de toda a comunidade acadêmica e da Internet. Nenhum outro Sistema

Operacional além do Unix, foi tão extensivamente depurado por pessoas de todo o mundo, resultando nas suas principais qualidades: estabilidade, desempenho e elegância.

Vale ainda lembrar que a primeira implementação do TCP/IP que se expandiu com sucesso foi sob um Unix, o de Berkeley (BSD). Assim, as origens da Internet são claramente relacionadas com a popularização do Unix.

O protocolo HTTP e o primeiro software Servidor Web (o HTTPd da Universidade de Illinois) foram desenvolvidos em plataforma Unix. O HTTP herdou muitas características do FTP e nasceu como um clássico *daemon* Unix. Assim, graças ao Unix, o salto tecnológico existente do protocolo FTP para o HTTP foi muito menor do que o subseqüente impacto que este protocolo causaria no mundo inteiro [74].

2.2 O Linux

2.2.1 Características Gerais

O Linux foi inicialmente desenvolvido por Linus Torvalds em 1991, baseando-se no Sistema Operacional de fins educativos criado por Tanenbaum, o Minix [123]. Apesar de ser bastante jovem, o Linux tem vivenciado uma popularidade espetacular desde sua criação, e hoje é um dos Sistemas Operacionais mais utilizados nos mais variados meios.

Linus continua bastante envolvido no desenvolvimento do Linux, mantendo-o atualizado de acordo com o desenvolvimento de novos hardwares e coordenando a atividade de centenas de desenvolvedores ao redor do mundo. Embora, o projeto inicial tenha sido desenvolvido em plataforma Intel, com o tempo o Linux tornou-se disponível para outras arquiteturas incluindo Sparc, Motorola, Power PC, IBM System/390 e mais recentemente para plataformas 64 bits como Alpha, Sparc64 e Itanium.

Um dos benefícios mais atraentes do Linux é o fato de ser um Sistema Operacional não comercial, tendo seu código fonte sob a licença pública GNU[1] estando assim disponível gratuitamente para quem desejar utilizá-lo.

Como já exposto, o Linux é apenas o *kernel* e não é um Sistema Operacional completo. Ele não engloba por exemplo, utilitários de sistema de arquivos, sistemas de janelas ou editor de

1. O projeto GNU é coordenado pela *Free Software Foundation, Inc*. (http://www.gnu.org). GNU é uma abreviação recursiva para “GNU is Not Unix”.

textos. Todavia, esses programas podem ser obtidos também gratuitamente sob a licença GNU, e instalados no sistema de arquivos suportado pelo Linux sem maiores problemas. Os usuários podem optar também por distribuições comerciais, que em geral abrigam todos os utilitários em um único conjunto de CD*s*, juntamente com um método de instalação próprio. Dentre essas distribuições destacam-se a Red Hat, Slackware, SuSE e a brasileira Conectiva.

A versão 2.4 do *kernel* do Linux, assegura ser compatível com o padrão IEEE POSIX, facilitando assim a portabilidade de programas originários de outros sistemas Unix. Além disso, o Linux não segue rigorosamente nenhuma variante em particular do Unix (BSD ou System V). Ao invés disso, tenta adotar as boas características e padrões de projetos de diferentes *kernels* do Unix [21].

2.2.2 Controle de versões do Linux

O Linux diferencia versões em desenvolvimento de versões estáveis usando um simples esquema de numeração. Cada versão é descrita por 3 números separados por pontos, onde os 2 primeiros indicam a versão estável e o terceiro indica o *release*.

Quando o segundo número é par ele indica uma versão estável. Caso seja ímpar indica uma versão em desenvolvimento.

Novos *releases* de versões estáveis tem apenas a função de corrigir *bugs* informados por usuários. Não são feitas grandes alterações como a modificação de estrutura de dados e algoritmos principais. Esse tipo de alterações profundas são livremente feitas em *kernels* em desenvolvimento.

Para nosso estudo foram utilizadas versões estáveis 2.2 e 2.4 do *kernel* do Linux.

2.2.3 Por que o Linux?

Para o melhor entendimento do funcionamento e possíveis otimizações a serem feitas no sistema operacional, testes práticos foram feitos. Foi escolhido o Linux como plataforma de estudo pelos seguintes motivos:

- Possui o código fonte aberto, sendo o representante mais expressivo do movimento de software livre.
- Suporta plataformas Intel, portanto permite o uso de hardware extremamente barato.

- Bastante portável para outras arquiteturas.
- Suporte gratuito, amplo e eficaz através de grupos de discussão.
- Implementação que bem representa a filosofia Unix. Assim, os conceitos de otimização abordados são facilmente aplicáveis a outros sistemas.
- Abriga requisitos de Sistemas Operacionais modernos como memória virtual, sistema de arquivo virtual, *threads*, suporte a SMP (*Symetric MultiProcessor*).
- Sistema Operacional que mais cresce em popularidade tanto no meio acadêmico como no comercial.

Alguns indícios fortes da popularidade crescente do Linux são: o lançamento do SGBD Oracle para plataforma Linux e o plano de compatibilidade entre o AIX e o Linux lançado pela IBM [68].

Este texto buscará ser genérico nos conceitos aplicados à organização e *tuning* do Sistema Operacional, utilizando o Linux como exemplo quando necessário.

2.3 O *Kernel*

2.3.1 Definição

O *kernel* é o programa mais importante do conjunto de programas que compõem o Sistema Operacional. É inteiramente carregado em memória (exceção feita aos módulos e *microkernels*) quando o sistema é inicializado e contém os procedimentos críticos necessários para o funcionamento do sistema. O *kernel* determina as capacidades de hardware servindo como um gerenciador de recursos [133].

2.3.2 Modos de execução e as *System Calls*

Um Sistema Operacional deve prover um ambiente de execução para as aplicações de usuários além de interagir com componentes de hardware. Alguns Sistemas Operacionais permitem que programas de usuários interajam diretamente com o hardware (por exemplo o MS-DOS). Em oposição, os Sistemas Operacionais Unix ocultam dos programas de usuários todos os detalhes de baixo nível relacionados ao hardware. Quando um programa de usuário deseja fazer uso de algum componente de hardware, o deve fazer através de uma requisição ao Sistema Operacional

que possui um interface bem definida para isso: as *system calls* (chamadas de sistema). O Sistema Operacional avalia então o pedido e caso conceda o acesso, serve de mediador entre o hardware e o programa de usuário.

Com o objetivo de reforçar este mecanismo, Sistemas Operacionais modernos utilizam-se de certas funções de hardware que proibem que programas de usuário interajam diretamente com os componentes de hardware ou acessem regiões de memórias arbitrárias. Em geral, o hardware provê pelo menos dois modos de execução para a CPU: o modo "não privilegiado", para programas de usuários, e o modo "privilegiado" para o *kernel*. Esses modos são freqüentemente chamados de "Modo Usuário" e "Modo *Kernel*". Freqüentes chaveamentos entre esses dois modos de execução podem penalizar consideravelmente o desempenho geral do sistema uma vez que dados devem ser copiados, primeiro a partir do *buffer* do dispositivo para o *kernel* e depois do *buffer* do *kernel* para o processo do usuário (ou na outra direção). Por esse motivo, implementações do Software Servidor Web diretamente no *kernel* têm surgido para servir como sistemas servidores Web de alto desempenho. Exemplos dessas implementações são o *khttpd e o Tux* que serão abordadas em 5.1.3 "Implementação do Servidor Web no espaço do kernel", página 137.

As *system calls* possibilitam ainda que códigos fonte de programas de usuário sejam bastante portáveis. Na realidade, padronizações como a POSIX baseiam-se em especificações de *system calls*.

2.3.3 Arquitetura de *kernels*

Na arquitetura monolítica cada camada do *kernel* é integrada em um único programa que executa no modo *kernel* por detrás do processo corrente. Em oposição, a arquitetura de *microkernels* demanda apenas um pequeno conjunto de funções do *kernel*, em geral mecanismos de escalonamento, comunicação entre processos e sincronização. Vários processos de sistema que rodam sobre o *microkernel* implementam outras funções do Sistema Operacional como alocação de memória, manipuladores de interrupções de sistema e *drivers* dos dispositivos.

A grande maioria de *kernels* de Unix são monolíticos, exceção feita ao Mach OS originário de Carnegie-Mellon e seus derivados que seguem a filosofia de *microkernels*.

A principal desvantagem da filosofia microkenels é que ela tende a ser mais lenta, uma vez que a passagem explícita de mensagens entre as camadas do Sistema Operacional pode causar atraso.

As vantagens dessa arquitetura se resumem em três: *microkernels* podem ser facilmente portáveis para outras arquiteturas, visto que todos os componentes dependentes de hardware residem no código do *microkernel*. Em segundo lugar, *microkernels* fazem melhor uso de memória, pois processos de sistema que não estão sendo utilizados não necessitam permanecer em memória. Por último, força os programadores a adotarem uma programação modularizada, pois os programas devem interagir com independentes camadas do SO [21].

O uso de módulos em *kernels* monolíticos é outra técnica que contempla muitas vantagens de *microkernels* além de interferir muito pouco no desempenho. Módulos são arquivos no formato objeto que podem ser ligados ao *kernel* em tempo de execução. Ao contrário da implementação de *microkernels*, os módulos não executam em um processo de sistema separado, mas funcionam em modo *kernel* como parte dele, assim como qualquer outra função ligada estaticamente.

As principais vantagens do uso de módulos são: por ser modularizado permite uma programação mais estruturada e uma melhor utilização da memória; é transparente ao usuário pois a ligação é feita automaticamente pelo *kernel*; dispensa a necessidade da recompilação do *kernel* a cada novo hardware adicionado ao sistema, pois cada *driver* pode ser construído em formato de módulo, sendo carregado sob demanda [112].

Uma pequena penalização no desempenho ocorre apenas quando do ligamento e desligamento do módulo. Todavia, Este processo pode ser comparado à criação e remoção de processos de sistema na filosofia *microkernel*. Não há passagem explícita de mensagens entre processos, como ocorre com *microkernels*.

O Linux é composto de um *kernel* monolítico com suporte a módulos.

2.4 Otimizando o *kernel* do Linux

Na maioria dos casos, as distribuições padrões do Linux são genéricas, ou seja, criadas para funcionar razoavelmente bem em uma grande variedade de ambientes e configurações de hardware. Assim, existe um grande potencial de melhoria no desempenho no sistema, considerando-se ambiente e hardware específicos.

Essa melhoria de desempenho pode ser alcançada de duas formas: 1) Fazendo uma compilação otimizada do *kernel*, eliminando código desnecessário. 2) Refinando as configurações do *kernel* pós-otimizado.

2.4.1 Compilação otimizada

Como exposto, o *kernel* do Linux é monolítico com suporte a módulos. A parte monolítica principal do *kernel* é na verdade um binário que necessita ser inteiramente carregado em memória para ser executado. Assim, quanto mais enxuto for o *kernel*, mais ágil será seu funcionamento e menos memória irá consumir.

2.4.1.1 Configurando o novo *kernel*

Para a compilação do *kernel* do Linux é necessário que se tenha presente no sistema o código fonte, que se encontra em `/usr/src/linux` (ou `/usr/src/linux-2.4` para o *kernel* 2.4). Esse código pode ser instalado no momento da instalação do Sistema Operacional ou posteriormente via pacotes RPM (em CD ou diretamente de http://www.kernel.org, *site* oficial para distribuição do *kernel* do Linux). No caso da instalação via pacotes são necessários os pacotes *kernel-headers* e *kernel-sources*.

O primeiro passo em se otimizar o *kernel* é a eliminação de código desnecessário que em geral concentra-se em *drivers* de dispositivos que sequer são utilizados pelo sistema. Embora os desenvolvedores do Linux se esforcem na criação de *drivers* mais genéricos possíveis, independentes de arquitetura (como o *driver* IDE), a maioria dos dispositivos vêm com características próprias do fabricante e uma generalização ocasionaria uma subutilização dos recursos [105].

A eliminação do código excedente é feita através da configuração do *kernel* a ser compilado. Para esta configuração utiliza-se os utilitários: `config` (modo texto), `menuconfig` (modo texto com menus) ou `xconfig` (modo gráfico). A partir do diretório raiz (`/usr/src/linux`) executa-se o comando: `make[config|menuconfig|xconfig]`

Os componentes para a instalação são exibidos em grupos. Em geral para cada item é possível selecionar uma das opções: "Yes" (incorpora o item selecionado no binário do *kernel*); "No" (não incorpora nem implementa a funcionalidade) ou "Módulo" (implementa como um módulo do *kernel*).

Abaixo uma lista das configurações principais a serem feitas no *kernel*, baseando-se em uma máquina servidora Web, proposta pelo autor:

1. Setar *Processor Family* de acordo com o tipo de processador presente no sistema.

Essa funcionalidade é importante pois otimizações tem sido adicionadas ao Linux para que seja mais rápido nos processadores mais recentes (tais como Pentium III ou IV), aproveitando melhor os novos recursos oferecidos por esses chips [105].

2. Habilitar MTRR (*Memory Type Range Register Support*) para Pentium III e IV.

Este item habilita o uso de registradores específicos pelo processador para que acessos a intervalos contíguos de memória sejam feitos mais eficientemente.

3. Somente habilitar *Symetric multiprocessing* em sistemas multiprocessados.

4. Desabilitar *Advanced Power Management* (APM).

APM é uma especificação da BIOS que visa a economia de energia usando diferentes técnicas. É mais útil para lap-tops que utilizam-se de baterias como fonte de energia. Além disso, não existe compatibilidade entre este recurso em sistemas multiprocessados, ocasionando inclusive a não inicialização da máquina [89].

5. Desabilitar *chipset/bugfix support* para chips não utilizados pelo sistema.

6. Habilitar *PCI bus-master DMA support* e *Use DMA by default*, caso utilize-se discos com este recurso nesta arquitetura. Não se aplica a discos SCSI.

7. Habilitar *custom support* para *chipsets* presentes no sistema.

8. Desabilitar *Network Firewalls*.

Esta funcionalidade refere-se a filtros de pacotes executados por um roteador.

9. Desabilitar *IP: multicasting*

10. Desabilitar *IP: advanced router*

11. Desabilitar *IP: optimize as router not host*

12. Habilitar *IP: TCP Syncookie Support*.

Esta opção é util para se prevenir de ataques de negação de serviço do tipo "*Syn flooding*" em servidores. Utiliza-se de um protocolo criptográfico de desafios que permite que usuários legítimos continuem se conectando mesmo sob um ataque. Mais informações na Seção 3.3.3.

13. Habilitar: *IP: Allow large Windows*

Apenas sob determinadas situações, onde a maioria dos clientes conecta-se através de uma rede que possui grande latência. Pode ocasionar desgaste desnecessário de memória. Mais informações Seção 3.7.1.

14. Habilitar somente tipos de dispositivos SCSI realmente existentes no sistema. Por exemplo disco rígido, mas não unidade de fita ou CD-ROM.

15. Desabilitar *Probe all LUNs*.

Existem dispositivos SCSI, como um CD *Jukebox*, que podem conter múltiplos LUNs (*Logical Unit Number*). Estes atuam logicamente como múltiplos dispositivos. Como dispositivos que possuem essa funcionalidade são raros, em geral não se incorpora esta funcionalidade ao *kernel*.

16. Desabilitar *SCSI error reporting*.

Como é útil apenas em situações esporádicas de problemas e dispositivos e contribui com aproximadamente 12KB a mais no *kernel*, não é vantajoso possuir este recurso.

17. Desabilitar *SCSI logging facility*.

18. Habilitar somente controladoras SCSI específicas do sistema.

19. Desabilitar todos em *Network device support* não requeridos pelo sistema. Por exemplo, HIPPI, SLIP, PPP, FDDI, etc.

20. Instalar somente *drivers* Ethernet específicos do sistema.

21. Habilitar dispositivos *Token-Ring* somente se necessário.

22. Habilitar dispositivos WAN somente se necessário.

23. Desabilitar dispositivos para *Amateur Radio Support*

24. Desabilitar dispositivos para IrDA (Rede sem fio por Infravermelho).

25. Habilitar ISDN somente se necessário.

26. Desabilitar *Old CD-ROM drivers* (CD-ROMs que não são SCSI/IDE/ATAPI).

27. Setar *Maximum number of Unix98 PTYs* para 64.

Este número deve ser alto apenas para servidores telnet, rlogin ou ssh. Para servidores exclusivamente Web, onde não há login remoto para usuários, um valor alto apenas consumiria mais memória.

28. Desabilitar todos *Joysticks*.

29. Desabilitar todos *Video for Linux* (Multimídia).

30. Desabilitar todos *Ftape*.

31. Setar suporte a USB de acordo com dispositivos existentes.

32. Desabilitar todos *Filesystems* exceto ISSO 9660 CD-ROM, Second Extend fs, /dev/pts e proc.

33. Desabilitar todos *Network file systems*.

34. Desabilitar todos *Partition Types*.

35. Desabilitar todos *Native Language Support*.

36. Desabilitar todos *Sound cards*.

Após a seleção dos itens grava-se a configuração no arquivo `.config` que servirá de *script* para a compilação do *kernel*. Diferentes configurações podem ser gravadas em diferentes arquivos o que facilita o controle de versões.

2.4.1.2 A Compilação

Após a configuração do *kernel* a ser gerado vem o processo de compilação em si, que é composto pelos seguintes passos:

- `make dep`: seta as dependências.
- `make clean`: prepara a árvore de diretórios para a construção eliminando versões anteriores.
- `make bzImage`: cria o *kernel* que se encontrará em `/usr/src/linux/<arch>/boot/bzImage`.
- `make modules`: constrói possíveis módulos selecionados.
- `make modules_install`: copia os módulos do *kernel* para o diretório apropriado (`/lib/modules`).
- copiar o arquivo `/usr/src/linux/arch/i386/boot/bzImage` para `/boot/NewvmLinuz`.

2.4.1.3 Configurando o LILO

No momento da inicialização do sistema, um programa designado *boot loader* é invocado pela BIOS para carregar o restante da imagem do *kernel*. Este programa localiza-se no primeiro setor do disco conhecido como *Master Boot Record* (MBR), juntamente com a tabela de partições.

Cada entrada da tabela de partições contém os setores iniciais e finais de partição e o tipo de sistema operacional que o manipula.

Quem designa qual das partições será a ativa e qual imagem do *kernel* a ser carregada é o *boot loader* , sendo que no Linux o mais popular é o LILO (*LInux LOader*).

Assim, através do LILO é possível manter diferentes versões do *kernel* selecionáveis no momento do *boot*. Isso é útil quando deseja-se comparar diferentes versões de *kernel* compilados, além de prover um mecanismo de proteção contra problemas no novo *kernel*, mantendo disponível o *kernel* original.

Desta forma, para colocar em funcionamento o *kernel* recém compilado, deve-se configurar o LILO, editando-se o arquivo `/etc/lilo.conf`. Este arquivo tem conteúdo similar a:

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
image=/boot/vmlinuz-2.2.14-14cl
       label=linux
       root=/dev/hda6
       read-only
```

Para se adicionar o novo *kernel* ao LILO deve-se acrescentar uma seção semelhante à existente modificando "image". Na imagem de *kernel* antiga, alterar "label":

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
default = linux

image=/boot/vmlinuz-2.2.14-14cl
      label=linux-old
      root=/dev/hda6
      read-only

image=/boot/Newvmlinuz
      label=linux
      root=/dev/hda6
      read-only
```

b) Executar `/sbin/lilo` para ativar as mudanças. Se tudo correr bem, a saída do comando deverá ser:

```
Added linux *
Added linux-old
```

O asterisco depois de "linux" indica a imagem de *kernel default*. Na próxima reinicialização do sistema haverá duas opções de boot: *linux* e *linux-old*, sendo que a primeira será a utilizada caso o usuário não faça a seleção dentro do tempo estipulado em "timeout" (em ms).

Este formato de arquivo do Linux aplica-se à arquitetura de discos IDE. No caso da configuração de discos SCSI deve-se utilizar o initrd.

2.4.2 Configurando parâmetros do *kernel*

Uma vez estando com o *kernel* compilado de acordo com a situação particular do hardware do sistema, o próximo passo é fazer um ajuste mais fino do *kernel* de acordo com o ambiente do sistema.

A maioria das variantes de Unix atuais possuem a capacidade de se auto-ajustarem. Isso é feito no processo de inicialização, onde o *kernel* vasculha o sistema para descobrir o quanto de memória RAM há disponível, qual o tamanho da área de *swap* e qual a velocidade da CPU. De posse dessas informações o *kernel* faz ajustes do tamanho do cache nos parâmetros de memória virtual e em outros subsistemas [11].

É aconselhável que o administrador do sistema refine esses ajustes iniciais uma vez que ele possui o conhecimento da carga que o sistema sofrerá no decorrer do dia. Além disso, o *tuning* automático feito pelo sistema pode não contemplar todos os requisitos. Todavia esse ajuste manual deve ser feito com cautela, uma vez que alterações equivocadas podem levar a situações indesejáveis, incluindo-se queda do sistema. Assim, é importante que parâmetros do *kernel* não sejam modificados em uma máquina em produção.

O Linux permite que o sistema seja otimizado em tempo de execução através do sistema de arquivos virtual "`/proc`". Esse sistema possui arquivos que podem ser alterados diretamente ou via interface "`sysctl`".

No Solaris o correspondente ao `/proc` é o `/etc/system` e ao sysctl é o ndd. Maiores informações sobre a forma de configuração de parâmetros do *kernel* do Solaris podem ser obtidas em [129]. No Windows, alterações no modo de execução do *kernel,* podem ser feitas via "registry".

2.4.2.1 Estrutura do /proc

O `/proc` é um sistema de arquivos virtual que provê uma interface para as estruturas de dados do *kernel*, sendo inclusive referenciada como uma “janela” do *kernel*. Ele pode ser usado tanto para obter informações sobre o sistema quanto para se alterar certos parâmetros do *kernel* em tempo de execução. É "virtual" pois não ocupa espaço em disco. Os arquivos são criados "*on the fly*" pelo *kernel* no momento em que são acessados. É por isso que o tamanho dos arquivos aparecem zerados quando este diretório é listado.

Para se ter suporte a este sistema de arquivos é necessário que se habilite esse mecanismo nas configurações da compilação do *kernel* (2.4.1 "Compilação otimizada", página 31). Embora contribua com aproximadamente 18 KB a mais no binário final do *kernel* [80], é uma funcionalidade essencial. Utilitários como `ps` e `top` utilizam-se desse diretório para geração de seus resultados.

O `/proc` é composto basicamente pelos seguintes arquivos e subdiretórios:

a) um subdiretório para cada processo ativo nomeado pelo PID do processo, sendo que cada subdiretório contém a seguinte estrutura os arquivos e subdiretórios listados na tabela 2.1.

Componente	Conteúdo
cmdline	Argumentos da linha de comando
environ	Valores das variáveis de ambiente
fd	Diretório contendo todos os Descritores de Arquivos
mem	Memória utilizada pelo processo
stat	Status do Processo
status	Status do Processo em formato inteligível
cwd	Link para diretório de trabalho corrente
exe	Link para o arquivo executável do processo
maps	Mapas de memória
root	Link para o diretório root do processo
statm	Informações do Estado de memória do processo

Tabela 2.1: Entradas específicas por processo em /proc.

b) arquivos contendo informações sobre o *kernel*:

Componente	Conteúdo
apm	Informações sobre Gerenciamento de Energia
cmdline	linha de comando do Kernel
cpuinfo	Informações sobre a CPU
devices	Dispositivos disponíveis (Bloco ou caracter)
dma	Canais DMA utilizados
filesystems	Sistemas de Arquivos utilizados
interrupts	Interrupções utilizadas
ioports	Portas I/O utilizadas
kcore	Imagem do kernel
kmsg	Mensagens do kernel
ksyms	Tabela de símbolos do kernel

Tabela 2.2: Informações sobre o *kernel* em /proc

Componente	Conteúdo
loadavg	Carga média
locks	Kernel locks
meminfo	Informações de memória
misc	Miscellaneous
modules	Lista de módulos carregados
mounts	Sistemas de Arquivos montados
partitions	Tabela de partições conhecidas no sistema
rtc	Relógio de tempo real
slabinfo	Slab pool info
stat	Estatísticas Gerais.
swaps	Utilização de área de swap
uptime	Tempo de atividade contínua do sistema
version	Versão do kernel

Tabela 2.2: Informações sobre o *kernel* em /proc

Neste subdiretório é possível saber, por exemplo, informações sobre a utilização de memória:

```
# cat /proc/meminfo
total:    used:    free:  shared: buffers:  cached:
Mem:  31432704 30621696   811008 46055424  1236992 14012416
Swap: 106889216   765952 106123264
MemTotal:     30696 kB
MemFree:        792 kB
MemShared:    44976 kB
Buffers:       1208 kB
Cached:       13684 kB
SwapTotal:   104384 kB
SwapFree:    103636 kB
```

c) subdiretório "ide" que dá informações sobre dispositivos ide.

Componente	Conteúdo
Cache	Cache do dispositivo
capacity	Capacidade
driver	Driver e versão
geometry	Disposição física e lógica
identify	Bloco de identificação do dispositivo
media	Tipo de mídia
model	Modelo do dispositivo
settings	Configuração do dispositvo
smart_thresholds	Gerenciamento de disco: thresholds
smart_values	Gerenciamento de disco: valores

Tabela 2.3: Informações "IDE" em /proc/ide

d) subdiretório "net" que abriga informações gerais sobre a rede incluindo configurações e tráfego.

Componente	Conteúdo
arp	Tabela ARP do kernel
dev	Dispositivos de rede com estatísticas
dev_mcast	Grupos de multicast
dev_stat	Estados dos dispositivos
ip_masq	Diretório contendo tabela de mascaramento
ip_masquerade	Tabela principal de mascaramento
netstat	Estatísticas de rede
raw	Estatísticas de "raw devices"
route	Tabela de roteamento do kernel
rpc	Diretório contendo informações sobre RPC
rt_cache	Cache de Roteamento
snmp	Dados sobre SNMP(Simple Network Management Protocol)
sockstat	Estatísticas de sockets
tcp	Sockets TCP
tr_rif	Tabela de Roteamento Token ring RIF
udp	Sockets UDP
unix	Sockets UNIX
wireless	Interface de dados para Wireless
igmp	Endereços IP para multicast
psched	Parâmetros globais para escalonamento de pacotes

Tabela 2.4: Informaçoes de rede em /proc/net

Componente	Conteúdo
netlink	Lista de sockets PF_NETLINK
ip_mr_vifs	Lista de Interfaces virtuais multicast
ip_mr_cache	Lista de cache de roteamento multicast
udp6	sockets UDP(IPv6)
tcp6	sockets TCP(IPv6)
raw6	Estatísticas de "Raw device" (IPv6)
igmp6	Endereços IP de multicast (IPv6)
if_inet6	Lista de endereços de interface (Ipv6)
ipv6_route	Tabela de roteamento do kernel (IPv6)
rt6_stats	Estatísticas globais de roteamento (Ipv6)
sockstat6	Estatísticas de socket(IPv6)
snmp6	Dados SNMP(IPv6)

Tabela 2.4: Informaçoes de rede em /proc/net

e) subdiretório "scsi" (caso exista uma controlada SCSI em funcionamento no sistema). Provê informações sobre todos os dispositivos reconhecidos incluindo tipo e fabricante.

f) subdiretório "tty"que dá informações sobre terminais em uso.

Componente	Conteúdo
drivers	Lista de drivers
ldiscs	Disciplinas de linhas registradas
driver/serial	Estatística de uso estado de linhas tty

Tabela 2.5: Informações sobre terminais em /proc/net

g) subdiretório "sys", que é o mais importante de todos, pois é nele que se concentram os parâmetros passíveis de modificação no *kernel*. Assim, será explorado com mais detalhes a seguir.

2.4.2.2 Alterando parâmetros: /proc/sys

Como já exposto, certos arquivos neste diretório servem como forma de se controlar o comportamento de determinadas partes do *kernel* em tempo de execução. Por exemplo, para se controlar a freqüência de *flushing* de caches de disco, pode se utilizar de dois métodos:

1) Diretamente, utilizando o camando "echo", redirecionando o valor para o arquivo correspondente ao parâmetro do *kernel* a ser modificado:

```
Ex: #echo "70 1200 64 0 20000 60000 90 0 0" > /proc/sys/vm/bdflush
```

Adicionando-se esta linha de comando no arquivo *script* `/etc/rc.d/rc.local`, ativa-se esta funcionalidade a cada reinicialização do sistema.

2) Através da interface `sysctl`, editando o arquivo de configuração `/etc/sysctl.conf` com a adição do seguinte trecho:

```
# melhorar desempenho da memória virtual
vm.buffercache = "70 1200 64 0 20000 60000 90 0 0"
```

2.4.2.3 Estrutura do /proc/sys

O diretório `/proc/sys` é composto pelos seguintes subdiretórios:

- `fs`: Contém informações específicas sobre sistema de arquivos, manipuladores de arquivos, inodes, *dentries* (manipulação de diretórios), informações sobre quotas e suporte a formatos binários.
- `kernel`: Contem informações gerais sobre o *kernel* em execução.
- `vm`: Abriga parâmetros sobre o subsistema de memória virtual.
- `net`: Onde ficam os parâmetros de rede.

Os parâmetros alteráveis mais relevantes desses diretórios, serão abordados em conjunto com a explicação de funcionamento de cada subsistema relacionado, nas próximas seções.

2.5 Processos e *Threads*

A principal entidade em qualquer Sistema Operacional é o processo. Processo pode ser entendido como uma instância de um programa em execução. Cada processo possui um único identificador (PID), um dono, prioridade, espaço em memória além de diversos outros atributos.

Unix é um Sistema Operacional multiusuário e multitarefa. Assim, múltiplos processos pertencendo a múltiplos usuários podem ser executados concorrentemente.

2.5.1 Conceitos

2.5.1.1 Escalonamento

Em um sistema monoprocessado apenas um processo pode ser executado exatamente em um dado instante. A impressão de que processos estão sendo executados em paralelo deve-se ao fato de que o *kernel* executa cada processo por um instante, depois interrompe-o e executa o próximo num círculo repetitivo. A eleição do próximo processo a ser executado, baseia-se em um *ranking* que é recalculado dinamicamente, e leva em conta prioridades definidas pelo sistema ou pelo usuário, quanto tempo o processo está inativo, interrupções de hardware e outros fatores. A entidade que controla o escolamento reside no *kernel* e é chamada "escalonador".

2.5.1.2 Preemptividade

Os Sistemas Unix em geral são Sistemas Operacionais multitarefa com processos de usuário preemptivos. Em outras palavras, quem decide quanto tempo o processo será executado é o Sistema Operacional via escalonador que interrompe a execução do processo invocando o escalonador. Em contrapartida, em Sistemas Operacionais não preemptivos, cada processo decide voluntariamente quando liberar a CPU. No caso do Linux, embora os processos sejam preemptivos, o *kernel* não é. Assim quando executado em modo *kernel*, há certas partes do código que assumem que nunca serão interrompidas, sendo executadas de modo atômico. Desta forma, *kernels* não preemptivos não podem ser usados como Sistemas Operacionais de tempo real. Atualmente, entre sistemas Unix convencionais, apenas o Solaris e o Mach OS possuem o *kernel* totalmente preemptivo [21]. Mesmo assim, programação em tempo real sob o Unix ainda envolve uma série de incertezas [74].

2.5.1.3 Criação de Processos

A criação de processos no Unix é feita através da chamada de sistema `fork()`, que faz uma cópia do processo pai para o novo processo filho que terá um novo PID. Cada novo processo

terá suas próprias estruturas de execução e região de memória. Como a criação de processos não era considerada uma tarefa muito freqüente quando da concepção do fork, este não é eficiente.

Em servidores Web a ineficiência na criação de processos interfere em muito no desempenho, particularmente em 2 casos: na forma como o software servidor Web cria processos para atender requisições simultâneas, e na forma como os CGIs para criação de páginas dinâmicas são executados (para cada requisição um novo processo é criado e posteriormente destruído). Para o primeiro problema sugere-se a utilização de *threads* (discutida a seguir). Para o segundo, sugere-se o uso de FastCGI que cria o processo uma única vez mantendo-o ativo em *background* de forma a atender múltiplos pedidos, agindo como um *daemon* (mais detalhes, além de outras alternativas para a criação de páginas dinâmicas são apresentadas na Seção 5.4).

Forking Web Servers e *Threading Web Servers* são vistos em 5.1.2 "Implementação do Servidor Web no espaço do usuário", página 131.

2.5.2 Threads

2.5.2.1 Definição

Thread, também conhecido como processo leve, é um fluxo seqüencial de execução dentro de um processo. Programação *multithreaded* é portanto uma forma de programação paralela onde várias *threads* são executadas concorrentemente dentro de um mesmo processo. Todas as *threads* são executadas em um mesmo espaço de memória e podem desta forma trabalhar concorrentemente em dados compartilhados [80].

Programação *multithreaded* difere de programação utilizando multiprocessos uma vez que as *threads* dividem o mesmo espaço de memória e outros recursos como descritores de arquivos, ao invés de executarem cada uma no seu próprio espaço de memória. Através deste recurso, o escalonamento entre *threads* é mais rápido pois a quantidade de informações a ser resgatada e carregada no processador é menor. Pelo mesmo motivo, a criação de uma nova *thread* é uma tarefa mais ágil (cerca de 100 vezes mais rápida do que a criação de um processo [74][1]).

1. Em geral o "quão pesado" o processo é em relação a uma *thread* depende do Sistema Operacional. Por exemplo, no AIX essa diferença é bastante marcante [62]. Já no Linux, internamente as implementações de ambos são bastante semelhantes.

Threads também são úteis por outras razões. Primeiramente, elas permitem que um sistema multiprocessado seja melhor explorado pelo programa, uma vez que elas podem ser executadas em paralelo em diferentes processadores, funcionando assim mais rápido do que um programa com apenas uma linha de execução, que pode executar em apenas um processador em um dado instante. Em segundo lugar, mesmo em sistemas monoprocessados, o uso de *threads* permite que o processamento não seja bloqueado por operações de I/O, atribuindo-se a essa tarefa a uma única *thread* e liberando-se as outras para diferentes operações. Por último, há tarefas que são melhores expressadas como várias *threads* que se comunicam entre si e trabalham cooperativamente, ao invés de um único monolítico programa seqüencial. Como exemplos desses programas incluem-se servidores e interfaces gráficas.

2.5.2.2 Forma de implementação do pacote de *threads*

Existem duas formas de se implementar um pacote de *threads*: no espaço de usuário e no espaço de *kernel*, que serão descritas a seguir.

Implementação no Espaço de Usuário

A implementação de *threads* no espaço de usuário evita o *kernel* e controla todas as tabelas por si própria, incluindo o próprio escalonamento entre as *threads*. Para o *kernel*, um processo *multithreaded* é implementado como um processo convencional, com apenas um fluxo de execução.

O escalonamento entre as *threads* não pode ser preemptiva, uma vez que dentro de um único processo não há interrupções de *clock*, fato que obrigaria o escalonador a agir. A não ser que a *thread* entre em um sistema de controle de tempo por sua própria vontade, não há forma de se eleger uma outra *thread* para ser executada [123].

Como vantagens, cita-se em primeiro lugar, a sua rapidez de escalonamento, pois não é necessário se mudar para o modo *kernel*. Em segundo lugar, teoricamente esta implementação pode escalar melhor, pelo fato de não requerer espaço adicional em tabelas ou pilha do *kernel*, o que poderia impor um limite no caso de um grande número de *threads*. Por último, esse tipo de pacote pode ser implementado em *kernels* que não possuem suporte a *threads* (como as primeiras implementações do Unix).

Como desvantagens, tem-se a possibilidade da monopolização do tempo de execução por uma única *thread*. Outro problema grave é o fato de não fazer uso de múltiplos processadores em sistemas multiprocessados. Por último, vem o fato de que se uma *thread* fizer uma chamada de sistema bloqueante de I/O, todas as outras *threads* do processo são bloqueadas. Uma solução implementada por alguns pacotes de *threads* que funcionam em modo usuário é o uso de *wrappers* sobre as chamadas de sistemas, de forma a torná-las não bloqueantes.

Implementação em Modo *Kernel*

Nesta técnica, *threads* são implementadas diretamente no *kernel*, usando-se, em geral, várias tabelas internas (cada tarefa possuindo uma tabela de *threads*). O escalonamento é preemptivo e feito pelo escalonador do *kernel*.

Essa implementação elimina os principais problemas da implementação em espaço de usuário: não há possibilidade de existir uma *thread* monopolizante, não há problemas com I/O bloqueante e faz uso adequado de sistemas SMP.

2.5.2.3 Suporte a *threads* no Linux

O Linux oferece suporte a aplicações *multithreaded* através da chamada de sistema não padrão `clone()`. A implementação é bastante diferente das adotadas em sistemas como SVR4 e Solaris, uma vez que internamente existe apenas uma única tabela de tarefas e um único mecanismo de escalonamento tanto para processos quanto para *threads*. Em outras palavras, *threads* são tratadas da mesma forma que processos: ambos são tarefas.

A chamada de sistema `clone()` realiza as mesmas operações que o `fork()` tradicional, com o adicional de permitir, através de *flags*, que partes específicas do processo, como memória, tabela de parâmetros do sistema de arquivos, arquivos abertos e manipuladores de sinal, sejam compatilhadas. Na realidade o `fork()` é implementado via `clone()` passando todos as *flags* de compartilhamento como "0". (`fork ( ) = clone (0)`).

Existem muitos argumentos contra essa implementação do Linux. Primeiramente, o processo de escalonamento pode tornar-se mais lento, uma vez que a *thread* torna-se um pouco mais "pesada". Além disso, programadores não podem se utilizar explicitamente da presença de múltiplos processadores para fazer com que suas *threads* necessariamente executem em

paralelo, pois o *kernel* não faz distinção de que a *thread* N e a *thread* M fazem parte da mesma aplicação [12].

Já o processo de criação de uma nova *thread*, continua sendo visivelmente mais rápida do que a criação de um novo processo, pois o conteúdo de memória em tabelas de arquivos não são duplicadas.

Como a chamada de sistema `clone()` não faz parte de um padrão, para que seja possível a criação de programas portáveis para outros sistemas, deve-se fazer uso de bibliotecas. Em particular, a biblioteca "LinuxThreads" [80] é a mais comum e tornou-se padrão nas distribuições do Linux, estando integrada à glibc2. Ela é compatível com o padrão POSIX 1003.1c e utiliza-se das chamadas de sistemas `clone()` produzindo assim, suporte à criação de *threads* no modo *kernel*.

2.5.2.4 Implementação do *kernel* via *threads*

Tudo o que foi exposto até o momento sobre *threads* relaciona-se ao uso deste recurso para a implementação de tarefas executadas por aplicativos de usuários. Da mesma forma, é possível implementar o *kernel* utilizando-se do conceito de *threads*. A essas *threads* dá-se o nome "*kernel threads*".

Como exemplos de tarefas que são delegadas às *kernel threads* incluem-se: atualização (*flushing*) de caches de discos, paginação de *frames* de memória, atendimento às conexões de rede, dentre outras. De fato, não é eficiente executar tais tarefas de modo linear. Neste contexto, o uso de *kernel threads* encaixa-se perfeitamente. Uma outra função bastante importante, é a associação de *kernel threads* a cada processo de usuário de forma a melhorar o desempenho quando da mudança de contexto entre modo usuário e modo *kernel* [74]. Esta prática é adotada pelo *kernel* do Solaris, que cria um *pool* de *kernel threads* [87].

Kernel threads diferenciam-se de processos regulares nos seguintes aspectos:

- Cada *kernel thread* executa uma função específica do *kernel*, enquanto processos regulares podem executar funções de sistema somente via chamada de sistema.
- *Kernel threads* executam somente em modo *kernel*, ao passo que processos regulares funcionam tanto em modo usuário quanto em modo *kernel*.

Os benefícios trazidos pelo uso de *threads* na implementação do *kernel* são os mesmos daqueles trazidos quando da sua implementação em processos de usuários: melhor exploração

da estrutura não seqüencial, permitindo que as tarefas sejam melhor organizadas, além de fazer melhor uso da arquitetura SMP.

Como um exemplo, uma das importantes melhorias no *kernel* do Linux 2.3/2.4 em relação ao 2.2 é a utilização mais eficiente de *threads* na implementação do *kernel* principalmente na pilha de protocolos TCP/IP [21]. Experimentos realizados por Brant [24] mostram uma grande melhoria na escalabilidade do *kernel* 2.3/2.4 em relação ao 2.2 quando utilizado em SMP.

2.5.3 Otimizações possíveis

No contexto de processos e *threads*, dois limites fundamentais a serem analisados são: o número máximo de processos do sistema e o número máximo de processos por usuário. Esses limites podem ser um gargalo em servidores Web que fazem uso da criação de uma grande quantidade de processos/*threads* filhos para o atendimento de requisições simultâneas de páginas estáticas ou páginas dinâmicas via interface CGI.

Experimentos por nós realizados demonstram que nas distribuições padrões do Linux estas são as principais causas de problemas de desempenho. O primeiro limite em *kernels* 2.2, é somente alterável diretamente no código fonte do *kernel* no arquivo `/usr/src/linux/include/linux/tasks.h`, constante `NR_TASKS`, necessitando-se assim de uma recompilação do *kernel* para que as mudanças tenham efeito. Em *kernels* 2.4 este limite não é mais *hardcoded*. O segundo limite, o número máximo de processos por usuário, pode ser alterado através da interface `ulimit`.

Em relação ao atraso causado pelas freqüentes criações e destruições de processos para a criação de páginas dinâmicas via CGI, caso seja o ponto crítico, aconselha-se a utilização da plataforma FastCGI ou APIs em servidores Web. Ambas alternativas, para a criação de páginas dinâmicas, implicam na alteração da estrutura de implementação da aplicação e serão abordadas na seção 5.4 "Geração de Páginas Dinâmicas", página 152.

Sobre a forma como os servidores Web criam filhos para servir páginas estáticas, os servidores que se utilizam do conceito de *multithreads* (Netscape/Apache 2.0) em geral possuem um desempenho melhor do que os baseados em multiprocessos, pelo fato do segundo grupo em geral consumir mais memória, ter um escalonamento mais lento e possuir uma criação mais ineficiente. Processos/*threads* filhos também podem ser configurados de tal forma a terem um tempo de vida maior, atendendo assim a múltiplas requisições, não necessitando-se desta forma,

que um novo filho seja criado para cada nova requisição. Tal funcionalidade pode ser configurada no software servidor Web. As possíveis configurações no servidor serão abordadas mais aprofundadamente em 5.3.4.2 "Configurando parâmetros de execução", página 145.

Como o processo servidor Web e seus filhos são processos de usuário, tais compartilham os recursos de CPU com outros processos presentes no sistema. Assim, quanto menor o número de processos concorrentes, melhor será o desempenho. Desta forma é aconselhável desativar todos os outros processos não essenciais e executar o servidor Web em modo texto, como uma máquina dedicada para este serviço. Esta atitude garante também uma melhor segurança no sistema, pois quanto menor o número de serviços disponíveis, menor a probabilidade de *bugs* que ocasionam falhas de segurança.

Uma outra estratégia possível é incrementar a prioridade de execução do processo servidor Web, que pode ser feito com permissões de root através da chamada de sistema `nice()`. Todavia, isso pode ser perigoso em servidores Web muito carregados, uma vez que funções essenciais de sistema podem ser ignoradas, ocasionando-se até a queda do sistema [74].

Configurar o escalonador aumentando o *timeslice* para cada processo pode contribuir para a melhoria de desempenho, pois reduz-se o tempo gasto com escalonamento. Além disso, servidores Web dedicados não possuem serviços interativos que poderiam ser afetados por essa medida [74].

Por último, como já citado, pode se optar por executar o software servidor Web não como um processo, mas como uma parte integrante do próprio *kernel*. Isso evita o desperdício de tempo com mudanças de contexto, além de permitir um acesso direto a certas estruturas do *kernel*, como *buffers* de rede, o que permite grandes possibilidades de melhoria de desempenho. Maiores informações sobre esse tipo de servidores são tratadas em 5.1.3 "Implementação do Servidor Web no espaço do kernel", página 137.

As ferramentas ps, top e vmstat são úteis para se detectar anormalidades no gerenciamento de processos e são descritas em 2.9 "Ferramentas de monitoramento", página 78.

2.6 Sistema de Arquivos

A maioria das aplicações lidam com informações que devem ser persistentes e armazenadas em grande quantidade, qualidades que não podem ser obtidas através do armazenamento de informações em RAM, no espaço de memória de cada processo. Para isso, informações são

gravadas em mídias externas (discos rígidos sendo os mais importantes) em unidades chamadas arquivos. Os arquivos são gerenciados pelo Sistema Operacional, mais especificamente pelo Sistema de Arquivos, cuja função é implementar operações sobre os arquivos como: estruturar, criar, nomear, ler, etc.

Pelo acesso a disco ser ordens de grandeza mais lento do que o acesso à memória, a correta utilização do sistema de arquivos pode proporcionar uma grande melhoria no desempenho geral do sistema.

Cada derivado do Sistema Operacional Unix faz uso do seu próprio sistema de arquivos. Embora todos sigam a interface padrão POSIX, cada um é implementado de uma forma diferente, buscando garantir o melhor desempenho e confiabilidade possíveis, as duas principais características de um sistema de arquivos. Entre os principais sistemas de arquivos para o ambiente Unix, destacam-se: o UFS (*Unix File System*, derivado de Berkeley e padrão da maioria dos Sistemas Unix), o EXT2 (*Second Extended File System*, padrão do Linux), o XFS (desenvolvido pela Silicon Graphics) e o VxFS (Veritas File System, comercial que implementa inúmeras funcionalidades). Entre outros sistemas de arquivos baseados em discos remotos, destacam-se o NFS e o SMB.

Alguns sistemas operacionais Unix como o Linux e Solaris, suportam ainda o conceito de *Virtual File Systems (VFS)* que provê uma interface comum para vários tipos de sistemas de arquivos reais. Isso permite que o sistema monte discos ou partições que abriguem arquivos em diferentes formatos.

Esta seção tratará do conceito Sistema de Arquivos hospedado em discos locais.

2.6.1 Implementação de Sistema de Arquivos

No Unix, discos são implementados como dispositivos de bloco, que na visão de sistemas de arquivos, são apenas uma série de blocos de tamanho fixo que podem ser lidos e escritos de maneira não seqüencial. O sistema de arquivos não precisa se preocupar com onde na mídia física do disco o bloco deve ser gravado. Essa tarefa é delegada ao *driver* do dispositivo.

2.6.1.1 Bloco

É a unidade de acesso e armazenamento de informações no disco ou qualquer outro dispositivo de bloco, sob a visão do Sistema Operacional. O tamanho do bloco é uma questão importante.

Devido à forma como discos são organizados, em geral o tamanho do bloco é múltiplo do tamanho do setor e potência de 2 (a maioria dos discos têm como tamanho padrão do setor 512 bytes). Se o bloco for de 1 Kbyte em um disco cujo o tamanho do setor seja 512 bytes, o sistema de arquivos e irá gravar e ler sempre dois setores consecutivos, tratando-os como uma unidade indivisível.

Possuir uma unidade de alocação muito grande, implica em um acesso mais rápido, pois um arquivo será composto de poucos blocos e menos tempo será gasto com *seeks e delays* de rotação. Por outro lado, mais espaço em disco é gasto. Citando um exemplo, caso o tamanho médio dos arquivos do sistema fosse 6KB e o tamanho do bloco fosse por exemplo 8Kbytes, teríamos um desperdício médio de espaço de 25%.

Como exemplo, supondo um disco com as seguintes características:

- 32 Kbytes por trilha
- tempo de rotação: 16.67 ms
- tempo de seek: 30 ms

O tempo para se ler um bloco de Z bytes será a soma dos tempos de seek, rotação e de tranferência: 30 + 8,3 + (Z/32768) . 16,67. O gráfico da figura 2.1 mostra através da linha sólida (escala da esquerda), a taxa de transferência desse disco em função do tamanho do bloco. Fazendo uma suposição de que todos os arquivos tenham 1 Kbyte, a linha pontilhada (escala da direita) informa o aproveitamento de espaço em disco. Nota-se que uma boa utilização do disco (bloco < 2 Kbytes) implica em baixas taxas de transferência.

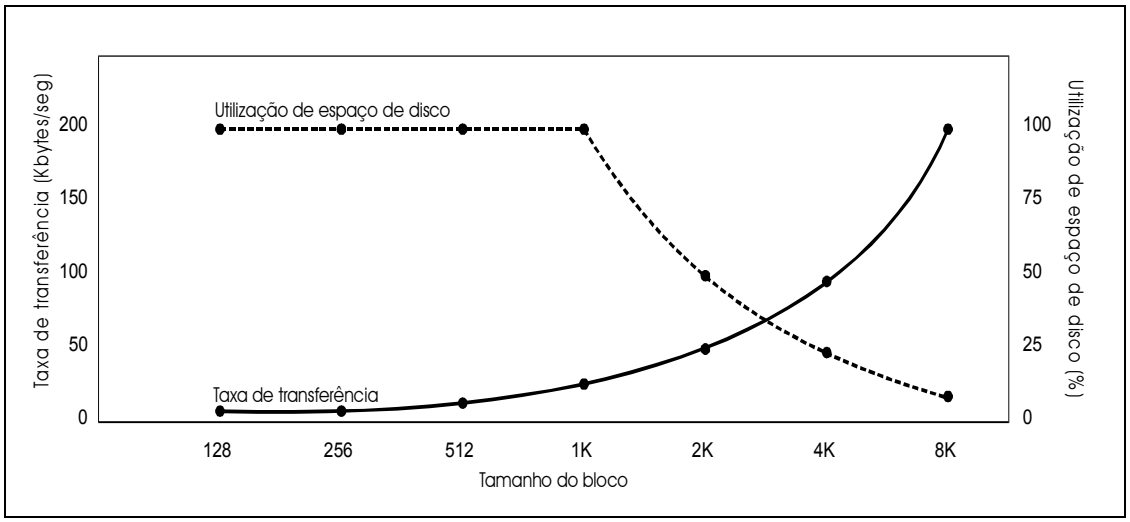

Figura 2.1: Relação entre aproveitamento de espaço e taxa de transferência

Um bom parâmetro para a escolha do tamanho do bloco é o tamanho médio dos arquivos. Para arquivos grandes, é aconselhável um tamanho de bloco também maior. Em geral, o tamanho médio para conteúdos estáticos da Web é de 10 a 20 KB [74]. Caso o conteúdo a ser servido seja de imagens de alta resolução, vídeos ou *download* de softwares este tamanho médio pode facilmente ultrapassar 1MB. O que aconselha-se neste caso é uma distribuição bimodal dos arquivos, ou seja, abrigar os arquivos grandes em um sistema de arquivos configurado para possuir bloco maiores, separados dos arquivos de tamanho menor abrigados em um outro sistema de arquivos.

2.6.1.2 Superbloco

O superbloco contém informações sobre o tamanho e formato do sistema de arquivos. Entre outras informações destacam-se o tamanho do bloco, o número de blocos livres, o número de inodes livres e o primeiro inode do sistema, que descreve o diretório root "/".

2.6.1.3 Inode

Cada arquivo no sistema é representado por um único inode (index-node) que contém informações básicas sobre o arquivo, como nome, permissões, datas (último acesso e modificação de conteúdo e atributos) e referências para os blocos de dados. Um nome associado a um arquivo pode sofrer mudanças. Todavia, um inode é único para um arquivo e continua o mesmo durante toda a existência deste arquivo. Cada inode recebe uma identificação única no sistema.

Para arquivos grandes, o número máximo de referências para blocos de dados comportados pelo inode pode não ser suficiente para representar todo o arquivo. Surge assim o conceito de bloco de indireção, que armazena endereços de outros blocos e não dados propriamente ditos. Faz-se então, a partir do inode, referência a este bloco. Essas indireções podem ainda ser duplas ou triplas, como representado na figura 2.2.

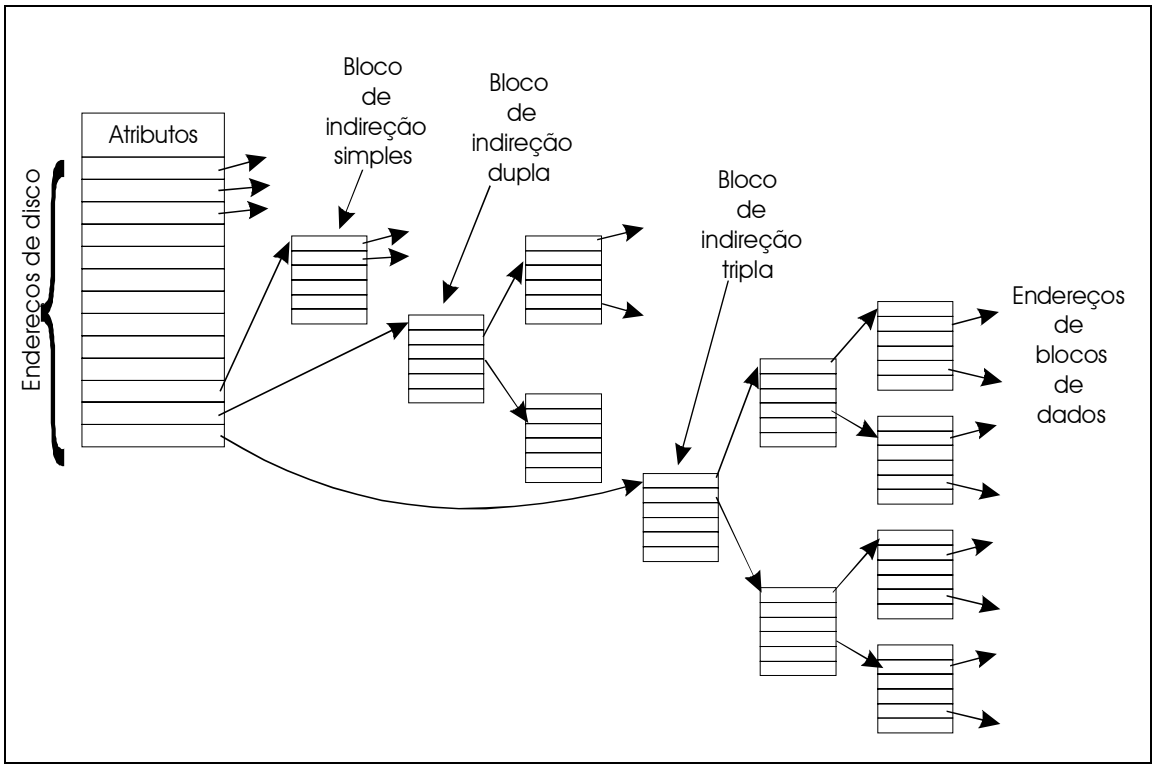

Figura 2.2: Representação de um inode

O sistema de arquivos representa em memória os inodes utilizados no momento através de uma tabela que serve também de cache de inodes. Assim, em servidores de grande demanda o valor *default* limite desta tabela pode não comportar a carga real. Desta forma, aconselha-se alterar esse limite `/proc/sys/fs/inode-nr` (como especificado em 2.4.2.2 "Alterando parâmetros: /proc/sys", página 41), em geral para 3 ou 4 vezes o valor máximo de arquivos abertos [94] especificado em `/proc/sys/fs/file-max` (ver 2.6.1.5 "Descritor de Arquivo", página 55).

2.6.1.4 Diretórios

Diretórios em sistemas Unix são implementados como se fossem arquivos especiais, que contêm uma lista de nome de arquivos ou outros diretórios, mapeados para número de inodes.

A função principal do diretório é prover informações necessárias para se localizar arquivos de dados.

Uma questão que surge na implementação de sistema de arquivos é onde se armazenar os atributos de arquivos (proprietário, permissões, datas): diretamente na entrada do arquivo no diretório ou no inode do arquivo. No primeiro caso, o acesso pode ser sutilmente mais rápido, visto que não se necessitaria acessar inclusive o inode de cada arquivo para se obter informações como permissão de acesso. Todavia, isso dificulta a implementação do conceito de compartilhamento de arquivos, via *hard-links*, uma vez que seria necessário duplicar e gerenciar informações sobre o arquivo em diferentes diretórios. Assim, sistemas de arquivos suportados por Sistemas Unix, em geral adotam a segunda prática.

Resolução de caminho de diretórios

O processo de resolução de caminho de diretórios é bastante custoso. Por exemplo, para se obter o inode do arquivo referenciado por `/var/logs/www/access_log`, são necessários os seguintes passos:

- Localizar e carregar o diretório raiz "`/`"cujo número de inode encontra-se no superbloco.
- Buscar neste diretório a entrada correspondente à "`var`" e obter seu número de inode.
- Carregar o inode referente ao diretório `/var` e localizar a entrada correspondente à "logs".
- Carregar o inode referente ao diretório `/var/logs` e localizar a entrada correspondente à "www".
- Carregar o inode referente ao diretório `/var/logs/www` e finalmente localizar o número de inode para o arquivo `access_log`.

Acrescente-se, a cada acesso a inode, uma validação do controle de acesso. Além disso, cada acesso a inode implica em uma operação de leitura de disco que envolve tempo de *seek* e de rotação.

No caso de caminhos relativos o processo é o mesmo, começando-se pelo diretório atual de trabalho, ao invés da raiz, evitando-se buscas iniciais.

Manter a hierarquia de diretório com poucos níveis, usar nomes curtos e diferenciá-los preferencialmente pelas primeiras letras do nome e utilizar caminhos relativos são alternativas para se reduzir o tempo com a resolução de nomes [74].

Outra alternativa implementada pelo *kernel* é a utilização do *Cache de Nome de Diretórios* que guarda em memória, geralmente utilizando-se de tabelas *hash*, os inodes correspondentes aos diretórios mais comumente utilizados.

No Solaris o tamanho máximo dessa tabela é baseado em `maxusers`. No Linux não há parâmetros passíveis de alteração para a configuração desta funcionalidade. Em `/proc/sys/fs/dentry` pode-se apenas visualizar algumas informações de cache de diretórios.

2.6.1.5 Descritor de Arquivo

Um descritor de arquivos representa a interação entre um processo e um arquivo aberto, servindo como uma interface entre as aplicações e o sistema de arquivos.

Para a aplicação, nada mais é que um inteiro retornado pela chamada de sistema `open()`, que internamente serve como um índice dentro de uma lista de possíveis arquivos abertos pelo processo. Em geral, os índices 0, 1 e 2 referem-se respectivamente ao *stdin* (entrada padrão), *stdout* (saída padrão) e *stderr* (saída de erro padrão) [8].

Cada nó desta lista, cujo descritor de arquivos é um índice, contém informações sobre a interação, sendo a mais importante o *file pointer* que indica a posição corrente dentro do arquivo sobre onde a próxima operação (leitura/escrita) surtirá efeito.[1]

Existe um limite quanto ao tamanho desta lista, ou seja, um número máximo de arquivos que um processo pode abrir simultaneamente. No Linux este limite é de 1024 (*hard limit*) e só pode ser alterado diretamente no código fonte do *kernel*. Para isso é necessário se editar os arquivos `limits.h` e `fs.h` no diretório `/usr/src/linux/include/linux` e alterar a definição de `NR_OPEN`. Abaixo do *hard limit*, outros valores limites podem ser estipulados (*soft limits*) através da interface ulimit, sem a compilação do *kernel*.

Outro parâmetro relacionado ao limite de descritores de arquivos é o número máximo que o *kernel* pode alocar. É razoável se ter 256 para cada 4MB de RAM [82] [94]. Assim, por exemplo, caso o sistema abrigue 512 MB, pode-se configurar o valor limite para 32.768. No Linux esse valor limite *default* é 4.096 em *kernels* 2.2 e 8192 em *kernels* 2.4, e pode ser alterado diretamente em `/proc/sys/fs/file-max` (ver 2.4.2.2 "Alterando parâmetros: /proc/sys", página 41) não necessitando-se recompilar o *kernel*.

1. As chamadas de sistema padrão relativas aos processos de leitura, escrita e movimentação do *file pointer* são respectivamente: `read(), write() e seek()`.

A configuração correta desses parâmetros é de fundamental importância em servidores Web de grande porte, uma vez que contam como descritores, além dos arquivos abertos (do sistema de arquivos) também os *sockets* de conexão de rede. No caso de servidores *proxy*, esse limite é ainda mais crítico, já que contam-se 2 descritores para cada conexão cliente.

Foi observado em nossos experimentos que, a partir de cargas médias, invariavelmente ocorrerá problemas de desempenho em um servidor Web Linux 2.2 que não tenha 2º parâmetro (limite do *kernel*) configurado corretamente. Na prática, a chamada de sistema `accept()` retornará um erro. Com relação ao 1º parâmetro (limite por processo), caso o servidor Web seja multiprocessado, este não será o problema, já que para cada requisição um novo processo será criado com uma única conexão ativa. Todavia, caso o servidor Web implemente o conceito de *threads* compartilhando também a tabela de arquivos abertos, esse limite torna-se igualmente crítico.

2.6.2 Manutenção de Sistema de Arquivos

A seguir, são relacionadas algumas tarefas que contribuem para uma melhor velocidade de acesso a sistemas de arquivos.

2.6.2.1 Desfragmentação de Discos

Após o uso constante do sistema de arquivos através de processos como criação, remoção e escrita em arquivos em geral, o sistema de arquivos tende a ficar desorganizado, visto que novos arquivos ou acréscimos aos já existentes podem ocasionar a alocação de blocos não contíguos, obtidos através de buracos não utilizados no disco. Isso degrada o desempenho, já que acessar arquivos fragmentados gera uma maior movimentação das cabeças dos discos, aumentando o tempo de acesso.

Essa situação torna-se crítica principalmente se o disco estiver muito cheio, pois será ainda mais difícil a alocação de blocos contíguos para a gravação de arquivos. Para se amenizar este problema, aconselha-se uma taxa máxima de ocupação de disco de 90% [74]. Acima desta taxa, um disco pode levar 50 % a mais de tempo para efetuar uma gravação, se comparado ao mesmo, caso estivesse vazio [133].

Para se evitar a fragmentação aconselha-se a manutenção do sistema de arquivos com espaço excedente, além de temporariamente utilizar-se de ferramentas de desfragmentação. Pode-se

também, simplesmente fazer um *dump* do sistema de arquivos para uma unidade de fita, formatar o disco e regravar os arquivos.

2.6.2.2 Particionamento Adequado

Uma dúvida freqüente que ocorre durante a instalação do Sistema Operacional, relaciona-se sobre qual é a melhor forma de se particionar o disco em sistema de arquivos.

Existem duas práticas comumente adotadas no Linux. A primeira delas é criar-se apenas 3 partições: "`/boot`", "`swap`" e a raiz "/".

O "`/boot`" torna-se necessário em alguns sistemas devido o limite da BIOS para se reconhecer os primeiros 1024 cilindros.

O "`swap`" é o espaço reservado para o sistema de memória virtual, e não é utilizado pelo sistema de arquivos, sendo assim criado em uma partição separada.

Por último, cria-se um único sistema de arquivos, a raiz "/", que abriga todo o restante do sistema.

A segunda prática, não cria uma raiz "monolítica", mas sim vários sistemas de arquivos separados sob ela como o "`/var`", "`/home`", "`/usr`". É o método mais aconselhável pelos seguintes motivos:

- Permite que o sistema seja distribuído entre vários discos.
- Permite montar sistemas de arquivos separados que possuam apenas permissão para leitura.
- Permite setar limites rígidos para certos tipos de arquivos, como por exemplo o `/var/spool/mail`, que poderia consumir espaço excessivo que deveria ser utilizado para outros fins.
- No caso de uma falha no sistema de arquivos, o escopo do dano é menor.

No contexto de servidores Web, os primeiros dois motivos são os mais importantes. Para se evitar a carga de acesso em único disco, pode-se distribuir o conteúdo entre um ou mais discos. Por exemplo, é possível isolar a gravação de *logs* do software servidor Web em um sistema de arquivos localizado em disco e controladora diferentes de onde se encontra o conteúdo estático. Da mesma forma, pode-se criar mais de um sistema de arquivo para *swapping,* localizados em diferentes discos, de forma a permitir que o processo de *swapping* (*ver* "Swapping temporário",

página 71) distribua a carga entre eles. Por esses motivos, aconselha-se o uso de vários discos de menor capacidade ao invés de um disco único de grande capacidade.

Colocar partições de *swapping* próximo ao início do disco pode trazer também certas melhorias de desempenho. O começo do disco é fisicamente localizado na posição mais externa do cilindro, permitindo leituras e escritas mais densas por rotação. Experimentos mostram que partições localizadas no final do disco podem ser até 3MB/s mais lentas [94].

Montar um sistema de arquivos *read-only* pode ser útil para um conjunto de páginas estáticas que sofre poucas alterações, no sentido de coibir a ação de *hackers* pela "pichação" de páginas.

2.6.2.3 Evitando o registro de último acesso de leitura

Os sistemas de arquivos Unix em geral, registram informações temporais sobre a criação, alteração e último acesso de leitura. Existe um custo associado em se registrar a data de último acesso, pois isso implica em uma escrita em disco. Esse processo penaliza de modo significativo servidores Web onde o acesso a disco concentra-se basicamente em leitura repetitiva de arquivos estáticos e o registro de último acesso de leitura raramente possui alguma utilidade.

Sendo assim, é importante que se desative esse processo, que pode ser feito em diferentes níveis: do sistema de arquivos completo, de diretórios ou arquivos.

Sistema de Arquivos

Para o sistema de arquivos, no Linux, deve-se montá-lo com a opção "noatime". Isso é feito alterando-se o arquivo `/etc/fstab`. Por exemplo, para se configurar um sistema de arquivos que abrigue apenas páginas estáticas cujo o ponto de montagem seja "`/www/docs`", deve acrescentar na linha correspondente a esse sistema de arquivos, o atributo "noatime":

```
/dev/hda1   /www/docs   ext2  default, noatime, 0, 0
```

Para que a alteração surta efeito é necessário se reinicializar o sistema. Listando-se o arquivo `/proc/mounts` pode-se verificar se sistema de arquivos foi montado com o atributo desejado.

Diretórios

A desativação para diretórios inteiros é feita, no EXT2 do Linux, através do comando:

```
# chattr -R +A /www/w3docs
```

que irá desativar recursivamente (-R), o processo de gravação de *timestamp* de último acesso, para todos os componentes do diretório especificado.

Arquivos

Para arquivos individuais, também no EXT2:

```
# chattr +A filename
```

2.6.2.4 Evitar o uso de *links* simbólicos

Aconselha-se a não utilização de *links* simbólicos nos diretórios de conteúdo pois eles são não apenas mais um nível de indireção, mas também implicam em mais dois acessos a discos: o primeiro para ler o inode do *link* (verificando permissões de acesso como qualquer outro arquivo) e o segundo para ler o conteúdo do *link* que na verdade é um texto que contém o caminho para o arquivo alvo.

Hard links todavia, podem ser usados sem penalização do desempenho, pois eles mapeiam o nome do *link* diretamente para o inode do arquivo destino.

2.6.2.5 Otimizar as variáveis de PATH

Deve-se manter as variáveis `PATH` e `LD_LIBRARY_PATH` curtas e apontando primeiro para as bibliotecas que o servidor web irá realmente utilizar. Caso contrário, vários acessos desnecessários a diretórios serão feitos quando da execução de CGIs ou outros novos processos. É aconselhável a manutenção de `/lib` com o primeiro na lista de `LD_LIBRARY_PATH`.

2.6.2.6 Evitar "~user"

A maioria de softwares servidores web entendem URLs da forma http://servidor/~usuario/arquivo.html, resolvendo o "~user" para o diretório *home* daquele usuário. Essa resolução implica em uma leitura de `/etc/passwd` e uma busca seqüencial neste arquivo, para a obtenção do caminho do diretório *home* do usuário. Posteriormente o caminho desse diretório deverá ser resolvido para o acesso do arquivo desejado. Todo esse processo é dispendioso, principalmente em servidores de grande demanda.

2.6.3 Cache de Disco

Como já citado, acesso a disco é em geral ordens de grandeza mais lento do que acesso à memória. Por este motivo, sistemas de arquivos modernos são projetados para limitar ao máximo o número de acessos ao disco.

A técnica mais comum para se atingir este objetivo, é o uso do *buffer cache*, que é uma coleção de blocos que logicamente pertencem ao disco, mas fisicamente estão mapeados em memória.

A cada leitura a disco, é checado primeiro se o bloco requerido já se encontra no cache. Em caso afirmativo, a requisição é atendida sem acessar o disco. Caso contrário, o bloco é trazido para o cache (ficando assim disponível para posteriores requisições) e repassado para o processo. Esse mecanismo beneficia em muito servidores HTTP, porque nesses, os mesmos arquivos tendem a ser requisitados repetidamente.

O *buffer cache* contem não somente dados de arquivos, mas também metadados como superblocos, inodes de arquivos abertos, etc.

2.6.3.1 Tamanho do *Buffer*

O tamanho do *buffer* pode ser estático ou dinâmico. A definição estática geralmente leva em conta o total de memória RAM existente no sistema, e este tamanho permanece constante. Na definição dinâmica, o tamanho do *buffer* considera o total de memória livre num dado instante, sendo possível ainda, explicitar limites mínimo e máximo de uso para *buffers* em relação ao total de memória. É importante lembrar que uma grande quantidade de memória para *buffer* implica numa menor quantidade de memória para processos, o que pode aumentar a taxa de paginação

e de *swapping*. Em geral, sistemas Unix utilizam toda a memória não utilizada para *buffers*, liberando dinamicamente esta memória quando requerido por processos.

2.6.3.2 Atualização em Disco

Para *buffers* alterados e que necessitam ser atualizados em disco, dá-se o nome de *buffers* "sujos".

O processo de atualização de *buffers* sujos (*flushing*) é feita de modo assíncrono, com intuito de melhorar o desempenho, visto que várias operações de escrita em um *buffer* podem ser satisfeitas por uma única escrita física em disco. Além disso, operações de escrita são menos críticas que operações de leitura, já que um processo em geral não é bloqueado aguardando uma rotina de escrita, mas é bloqueado aguardando uma rotina de leitura [21].

Um *buffer* sujo, poderia permanecer em memória por um tempo indeterminado, até o *shutdown* do sistema. Todavia, a estratégia de se atrasar ao máximo a gravação do *buffer* possui duas desvantagens:

- no caso de uma queda de energia e perda do conteúdo da memória RAM, uma grande quantidade de atualizações são perdidas, gerando-se inconsistências no sistema de arquivos.
- O tamanho do *buffer* e conseqüentemente da RAM requerida poderia tornar-se muito grande.

Assim, *buffers* sujos são gravados em disco sob as seguintes condições:

- quando esgosta-se a capacidade do *buffer* e mais *buffers* são requeridos, ou a porcentagem de *buffers* sujos é muito grande. Nessas condições a *kernel-thread bdflush* (próxima seção) é ativada.
- quando muito tempo se passou desde que o *buffer* esteja sujo. Nessa condição ativa-se o *kupdate*, que atua de tempos em tempos.
- Um processo requer que todos os *buffers* do dispositivo ou de um arquivo em particular sejam atualizados em disco invocando a chamada de sistema `sync()`.

Quando um bloco é lido do disco para ocupar uma posição do *buffer* e este está cheio, torna-se necessário a remoção de um bloco do cache e posterior gravação no disco (caso o tenha sido modificado). A eleição desse bloco a ser retirado pode ser feita utilizando-se de diversos algoritmos como o LRU (*Least Recently Used*) ou FIFO.

O uso de caches de discos, embora contribua para a melhoria desempenho, pode levar à inconsistência do sistema de arquivos ou na perda de dados de usuários. Suponha-se por exemplo, que um inode foi modificado e encontra-se em cache, no momento em que o sistema cai. Referências inválidas a blocos devido à versão desatualizada do inode podem ocasionar a inconsistência no sistema de arquivos, pela referência inválida a blocos. Como metadados são considerados mais críticos, o tempo de atualização para este tipo de informação pode ser diferenciada em alguns sistemas.

2.6.3.3 *Flushing* de *buffers* no Linux (*kernel* 2.4)

No Linux as informações de *buffer* cache são configuráveis em `/proc/sys/vm/bdflush`, onde é possível se determinar a freqüência de *flushing* de *buffers*.

bdflush

O arquivo `/proc/sys/vm/bdflush` controla o funcionamento das *kernel-thread* "bdflush" e "kupdate". Alterando alguns valores desse arquivo pode-se atrasar a freqüência de ativação dessas *threads,* diminuindo-se assim o número de acessos a discos e melhorando o desempenho geral.

Os parâmetros contidos neste arquivo são na ordem:

```
1) nfract
2) ndirty
3) nrefill
4) não utilizado
5) interval
6) age_buffer
7) nfrac_sync
8) não utilizado
9) não utilizado
```

Os parâmetros `nfract` e `nfract_sync` (1 e 7), indicam a porcentagem máxima de *buffers* sujos que podem se encontrar em cache. Setar estes parâmetros para um valor alto significa que o Linux irá atrasar a chamada do bdflush por um grande período de tempo, até que a porcentagem de *buffers* sujos atinja a especificada. Opta-se assim por uma escrita em "rajadas" que diminui o tempo geral de acesso a disco. Quando a porcentagem de *buffers* sujos supera `nfract`, o bdflush passará a gravar os dados assincronamente. Quando essa porcentagem

superar `nfrac_sync`, os programas passam a gravar os dados de modo síncrono, com o objetivo de evitar o preenchimento de memória com *buffers* sujos. O *default* do Linux é respectivamente 30 e 60% . Aconselha-se alterar estes valores para 70 e 90% [94].

O segundo parâmetro `ndirty` indica o número máximo de *buffers* sujos que o bdflush pode escrever em disco de uma só vez. Novamente, um número alto implica em uma escrita em rajadas. O *default* do Linux é 500, e aconselha-se alterar este valor para 1200 [94].

O terceiro parâmetro, em *kernels* 2.2, indica o número de *buffers* que o bdflush irá adicionar à lista de *buffers* livres. Nesta situação o bdflush é invocado quando a lista está vazia e não se é possível alocar mais espaço para *buffers* por escassez de memória. Assim, o *bdflush* deverá liberar o número de *buffers* indicado neste parâmetro, fazendo atualizações em disco quando necessário. O fato de se possuir *buffers* alocados de antemão agiliza o processo de alocação. Todavia, nesta situação o sistema já sofre de falta de memória. Assim um número alto para este parâmetro pode ser contraprodutivo. Aconselha-se assim manter o padrão do Linux que é de 64. Em *kernels* 2.4 este parâmetro não é mais utilizado.

O parâmetro `interval` controla a freqüência de ativação do *kupdate* para o *flushing* de *buffers* que já envelheceram, ou seja, estejam sujos por um tempo maior que `age_buffer` (*buffers* "velhos"). O valor default para `interval` é de 5 segundos. É prática comum entre usuários de laptops elevar este limite para alguns minutos de forma a permitir que o disco rígido pare de rotacionar e com isso se economize energia (*Advanced Power Management*). Vale lembrar, que o *flushing* ainda irá acontecer, pela ativação do *bdflush*, caso a porcentagem de *buffers* sujos atinja um valor alto (> *ndirty*). Para servidores Web, aconselha-se a alteração deste valor para no mínimo 20 segundos [74].

O parâmetro `age_buffer,` determina quando um *buffer* é considerado "velho" e deverá ser atualizado na próxima ação de *kupdate*. O valor default é 30 segundos. Baseado em `interval` aconselha-se a alteração deste parâmetro para 60 segundos [74].

É importante lembrar que esses parâmetros possuem limites *hardcoded* no fonte do Linux. Caso deseje-se alterar esses limites é necessária a edição do arquivo `/usr/src/linux-2.4/fs/buffer.c` e posterior recompilação do *kernel*.

As mudanças propostas baseiam-se no caso de servidores Web onde não há processos de gravação a não ser de *logs*, cuja a perda de dados não seria desastrosa no caso de uma queda do sistema. Desta forma, propõem-se atrasar ao máximo o processo de checagem e gravação dos

buffers, evitando-se assim o atraso tanto de CPU pela ativação das *threads* quanto de acesso a disco.

O ponto negativo quanto ao acúmulo de dados a serem gravados é o fato de que quando a gravação ocorre, ela será mais demorada podendo causar um atraso na execução dos outros processos. Uma forma de contornar esse problema é o uso de *Direct Access Memory* (DMA) (ver Seção 2.6.4.2), mecanismo que controla a transferência de dados entre a memória e o disco sem o uso da CPU. Uma vez que a transferência é iniciada a CPU pode retornar sua atenção para outros processos, não afetando assim o tempo de resposta.

2.6.3.4 Evitando o cache: *Raw Devices*

Raw devices são dispositivos cujo o acesso é feito sem a utilização do subsistema de arquivos, incluindo os caches de discos (*buffers*, inode ou diretórios) com o objetivo de se melhorar o desempenho em determinadas situações.

A princípio isso parece ser um contra-senso pelo fato de cache ser um mecanismo criado para a melhoria de desempenho.

Raw devices são úteis quando uma grande quantidade de dados, que excede o tamanho da memória do sistema é lido, ou os dados já encontram-se em *cache* em um outro nível.

Um bom exemplo disso é a implementação do banco de dados para suporte a decisões do SGBD Oracle, que utiliza-se de um grande segmento de memória, onde já implementa o cache por si próprio. Assim, utilizar mais um nível de cache, no caso o do Sistema Operacional, somente aumentaria o *overhead* [120].

2.6.4 Otimizando acesso a discos IDE no Linux via hdparm

Significativas melhorias de desempenho são observadas quando se otimiza o acesso a discos IDE no Linux. O *kernel* usa parâmetros conservadores como padrão que podem ser alterados através do comando hdparm.

O comando hdparm tem o formato:

```
#hdparm <opções> <dispositivo>
```

2.6.4.1 Exibindo configuraçoes e desempenho atuais

O uso do hdparm sem parâmetros, exibe as configurações atuais do dispositivo:

```
# /sbin/hdparm /dev/hda
```

Para se listar informações físicas do disco, incluindo limites de configurações, utiliza-se o parâmetro "-i":

```
# /sbin/hdparm -i /dev/hda
```

Uma funcionalidade bastante interessante do hdparm é a possibilidade de uma simulação de desempenho do disco baseado na configuração atual. A opção "-t" invoca o teste para o disco físico, ignorando-se o subsistema de caches de discos (ver 2.6.3 "Cache de Disco", página 60), e opção "-T" invoca o teste com leituras apenas sobre o subsistema de cache (não faz acesso a discos).

```
# hdparm -tT /dev/hda

/dev/hda:

 Timing buffer-cache reads:   128 MB in  1.32 seconds =
96.97 MB/sec

 Timing buffered disk reads:  64 MB in  6.29 seconds = 10.17
MB/sec
```

"-t" é útil para se testar o efeito das alterações feitas.

2.6.4.2 Habilitando o DMA

Um primeiro parâmetro a ser observado é o uso de DMA (*Direct Memory Access*) que permite que acessos entre o disco e memória sejam feitos sem a interrupção do processador, melhorando significativamente as operações de I/O conforme testes realizados. Caso na compilação do *kernel* não tenha sido habilitado o parâmetro *"use DMA by default" (ver 2.4.1 "Compilação otimizada", página 31),* o uso DMA deverá ser explicitado pelo comando :

```
#/sbin/hdparm -d1 /dev/hda
```

Este comando deve ser adicionado nos *scripts* de inicialização. Distribuições comerciais recentes do Linux vêm com este parâmetro setado como padrão no *kernel*.

O funcionamento do DMA dependerá também do suporte ao *chip-set* da placa-mãe compilada no *kernel*, especificada pelo parâmetro de compilação "*PCI bus-master DMA Support"*, além da tecnologia do disco utilizado.

O modo de DMA em geral é obtido automaticamente pelo *driver*. Porém, pode ser usado explicitamente pelo parâmetro "-X".

Da mesma forma, a utilização desses modos depende do suporte do disco. Habilitar o modo "Ultra DMA 2" para um disco que suporta apenas o modo DMA padrão, ocasiona a degradação do desempenho.

```
/sbin/hdparm -d1 -X34 /dev/hda - Explicita o uso do modo DMA2

/sbin/hdparm -d1 -X66 /dev/hda - Explicita o uso do modo
"Ultra DMA 2"
```

2.6.4.3 Habilitando transferências de 32 bits

O modo de transferência padrão entre o barramento PCI e a controladora de disco é de 16 bits. A maioria dos dispositivos atuais suportam a transferência em 32 bits ou mesmo 32 bits assíncrono. Esse parâmetro pode ser alterado pela *flag* "-c". "-c 1" seta a transferência para o modo 32 bits, e "-c 3" para o modo 32 bits assíncrono.

```
#/sbin/hdparm -c 3 /dev/hda
```

2.6.4.4 Habilitando a transferência de múltiplos setores por interrupção

O campo "multcount" exibido pelo hdparm sem parâmetros, controla quantos setores são trazidos do disco em uma única interrupção de I/O. É importante notar que o disco possui um limite para esse parâmetro, explicitado "MaxMultSect" quando da execução de "hdparm -i".

Esse parâmetro pode ser alterado pela *flag* "-m XX", onde XX especifica o número de setrores:

```
#/sbin/hdparm -m 16 /dev/hda
```

2.7 Gerenciamento de memória

Memória é um dos recursos mais importantes no sistema no contexto de desempenho. Tanto é, que a primeira atitude quase inconsciente do administrador de sistemas quando se deparando com um problema de desempenho, é o acréscimo de memória. E de fato, a falta de memória degrada consideravelmente o desempenho, principalmente pela necessidade do sistema utilizar-se do disco, como visto muito mais lento, como expansão de memória. Mais que isso, memória nunca será demais; o excedente, não usado por processos ou *kernel*, é empregado como *buffers* de disco, que possibilitam uma forma de acesso assíncrono e um mecanismo de cache eficiente como visto em *2.6.3 "Cache de Disco", página 60*.

O gerenciador de memória é o subsistema do *kernel* que cuida do uso eficiente desse precioso recurso e possui um relacionamento bastante estreito com o subsistema de arquivos pelos conceitos de memória virtual e cache de discos.

2.7.1 Funções do Gerenciador de memória

O subsistema de gerenciamento de memória deve prover:

- Aumento do espaço de endereçamento, de forma que programas possam referenciar mais memória do que fisicamente existe. Isso possibilita que inúmeros processos sejam executados concorrentemente (memória virtual).
- Alocação eficiente de memória entre o *kernel* e processos de forma a evitar a fragmentação.
- Possibilidade de vários programas residirem em memória ao mesmo tempo, melhorando a taxa de utilização da CPU.
- Proteção de dados entre processos, atribuindo espaços de memória privados para processos, além de evitar que processos sobrescrevam a área de código.
- Mapeamento de arquivos em memória, possibilitando que processos mapeiem um arquivo para uma área de memória virtual e o acesse diretamente na memória.
- Acesso justo à memória física entre os processos envolvidos, levando em conta o desempenho do sistema.
- Memória compartilhada, permitindo que processos compartilhem regiões de memória, como por exemplo código executável.

2.7.2 Como a memória é utilizada

A maioria dos sistemas operacionais Unix distinguem claramente duas porções da memória. Um espaço é destinado a abrigar a imagem *kernel*, incluindo o código e estrutura de dados estáticas. Em geral essa região nunca é liberada, e portanto indisponível para outros propósitos. O espaço restante é usualmente manipulado pelo subsistema de memória virtual e é usado de três formas:

- Para abrigar estruturas de dados dinâmicas do *kernel* como descritores de arquivos, *buffers* de rede ou estruturas `/proc` [120]. Em geral para servidores Web de grande demanda onde múltiplas conexões são tratadas simultaneamente, um montante considerável de memória é consumido neste quesito.
- Para satisfazer requisições de processos para áreas de memórias genéricas ou para mapeamento de arquivos em memória.
- Para se conseguir um melhor desempenho de discos e outros dispositivos pela utilização de caches.

Em geral, caches são construídos para ocupar todo o espaço de memória não utilizado por processos ou pelo *kernel*. Esse é o motivo pelo qual ferramentas que medem a taxa de utilização de memória apontam um valor baixo para a memória livre, mesmo em situações onde existe bastante memória disponível, alocáveis através da desapropriação de *buffers* de disco.

2.7.3 Memória Virtual

Memória virtual é uma técnica que permite que processos enderecem de forma transparente mais memória RAM que fisicamente existe no sistema através do uso do disco como extensão. Isso permite por exemplo, que um programa cujo tamanho exceda a quantidade de memória disponível consiga ser executado, uma vez que apenas partes do programa utilizadas do momento são mantidas em memória. O restante que não coube em memória permanece em disco e será resgatado para a memória quando necessário, num processo conhecido como *swapping*.

A memória virtual também é fundamental para a implementação de múltiplos processos, pois permite que estes executem concorrentemente mesmo com uma quantidade de memória limitada.

Todavia, toda essa funcionalidade não vem sem um preço: a degradação do desempenho, pelo acesso ao disco. O processo de *swapping* contrasta com o conceito de cache de discos, pois este último melhora o desempenho com o uso de memória RAM não utilizada, enquanto o primeiro estende o conteúdo de memória endereçável com a penalização do desempenho. Além disso, ocorre um gasto adicional de memória pela alocação de estrutura de dados de controle, o que diminui o espaço de memória física.

2.7.3.1 Segmentação da memória em páginas

A maioria dos sistemas de gerenciamento utilizam-se da segmentação da memória em páginas para a implementação da memória virtual.

Um espaço de endereçamento virtual maior que o espaço de endereçamento físico é criado e dividido em unidades de tamanho fixo (em geral 4 ou 8 KB) chamadas páginas, que possuem unidades correspondentes no espaço de endereçamento físico denominadas *frames* de página.

Processos só reconhecem endereços virtuais, sendo que esses são convertidos em endereços físicos por um componente de hardware integrante da CPU chamado MMU (*Memory*

Management Unit). O MMU utiliza-se de tabelas de páginas que contém o mapeamento entre os endereços lógicos e físicos. Dependendo da arquitetura da CPU, a implementação desta tabela pode se dar através de diferentes níveis.

A existência da memória virtual permite a alocação de trechos de memória contíguos através do uso de várias páginas não contíguas no espaço físico.

2.7.3.2 Swapping

Execução

Como o espaço de endereçamento físico é menor que o espaço de endereçamento virtual (essa é a função da memória virtual), é claro que nem todas as páginas virtuais podem estar representadas em páginas físicas. Quando ocorre um acesso a um endereço de memória pertencente a uma página virtual que não possui a correspondente em memória física, dá-se a chamada "page fault". A "page fault" é então sinalizada pela CPU ao gerenciador de memória que escolhe então uma página física, que caso esteja ocupada, tem seu conteúdo gravado em disco (*swap-out*), no sistema de arquivos designado para isso. Em seguida, o gerenciador carrega do disco para a memória física, a página correspodente à referência feita (*swap-in*).

Este processo de troca de informações entre a memória e o disco é designado *swapping*.

No exemplo citado, a granularidade de *swapping* foi a página. Uma outra técnica é utilizar como granularidade o processo inteiro. Autores costumam referenciar a primeira como "paginação" e a segunda como "*swapping* de processos".

A implementação pura de *swapping* de processos é a mais simples, porém degrada consideravelmente o desempenho, pois o processo não poderá ser executado (ou ter a execução continuada) enquanto todas as suas páginas não forem trazidas para a memória. Em outras palavras, o tempo gasto para se resgatar o processo inteiro do disco (*swap-in*) é maior do que o tempo gasto com escalonamento. Esta técnica era utilizada nas primeiras implementações do Unix [123].

As duas técnicas podem ainda ser combinadas, como é feita no Solaris, onde a paginação é feita em circunstâncias normais e o *swapping* de processos é somente usado em situações críticas de falta de memória [120]. Neste, apenas o *swap-out* é feito para o processo inteiro, enquanto o *swap-in* e execuçao pode ser feito por páginas. Ainda assim, o processo de *swap-out* será lento, principalmente na presença de processos que ocupam grande espaço em memória. Esse é um

dos argumentos desfavoráveis à implementação de servidores Web *multithreaded*, em sistemas que consideram *threads* como parte do processo (ver 2.5.2 "Threads", página 44), já que o processo servidor tornar-se-á cada vez maior na presença de múltiplas requisições (ver Seção 5.1). O Linux utiliza apenas a paginação no processo de *swapping*.

Em geral o processo de paginação indica que o sistema está carente de memória, caso contrário o processo inteiro estaria presente em RAM e "page faults" não iriam ocorrer. Exceções são feitas quando um programa é inicializado (uma vez que ele precisa ser lido do disco) ou quando um processo acessa arquivos de dados.

A escolha de qual página será substituída geralmente baseia-se em algoritmos como o FIFO ou LRU, sendo este o último o mais comum (Solaris e Linux o utilizam).

Área de *Swapping*

O espaço de disco reservado para o armazenamento de páginas temporárias, conhecido como "área de *swap*", é composta de uma ou mais partições definidas no momento da instalação do sistema (ver 2.6.2.2 "Particionamento Adequado", página 57).

Partições desse tipo, são mantidas sem formatação e não são usadas pelo sistema de arquivos.

Caso existam duas ou mais partições, o processo de *swapping* será balanceado entre elas, de forma a paralelizar o processo de I/O, melhorando o desempenho.

Swapping temporário

O processo de *swapping* é ativado sob demanda quando do advento de uma *page-fault*, ou temporariamente quando a quantidade de memória livre torna-se crítica. O Linux utiliza-se da *kernel-thread* "kswapd" que é ativada a cada 10 segundos toda vez que o número de páginas livres atinge um valor abaixo de um limiar pré-definido.

O arquivo de configuração `/proc/sys/vm/kswapd` provê 3 parâmetros sendo que apenas o terceiro (*swap-cluster*) é utilizado no *kernel* 2.4. Este parâmetro indica o número de grupos de páginas que serão atualizados em discos a cada ativação do *kswapd*. É aconselhável um valor alto, para que seja feita grande quantidade de atualizações de uma única vez e o sistema de disco não tenha de fazer operações de *seek* muito freqüentemente.

Neste sentido, o Solaris possui um mecanismo mais aprimorado que o Linux, onde existem vários limiares que controlam a ativação de diferentes taxas de *swapping* a se utilizar. [120] apresenta mais detalhes.

2.7.4 Mapeando arquivos em memória

A forma tradicional de acesso a arquivos no Unix dá-se através de chamadas de sistemas. Isso torna-se ineficiente uma vez que é necessário uma ou duas (para acessos não seqüenciais) chamadas de sistemas para cada operação de I/O. Além disso, caso mais de um processo esteja acessando o mesmo arquivo, cada um manterá cópias possivelmente iguais em suas áreas privadas de memória. Deve-se considerar ainda mais uma cópia dessa informação nos caches de *buffer*. A figura 2.3 ilustra a situação.

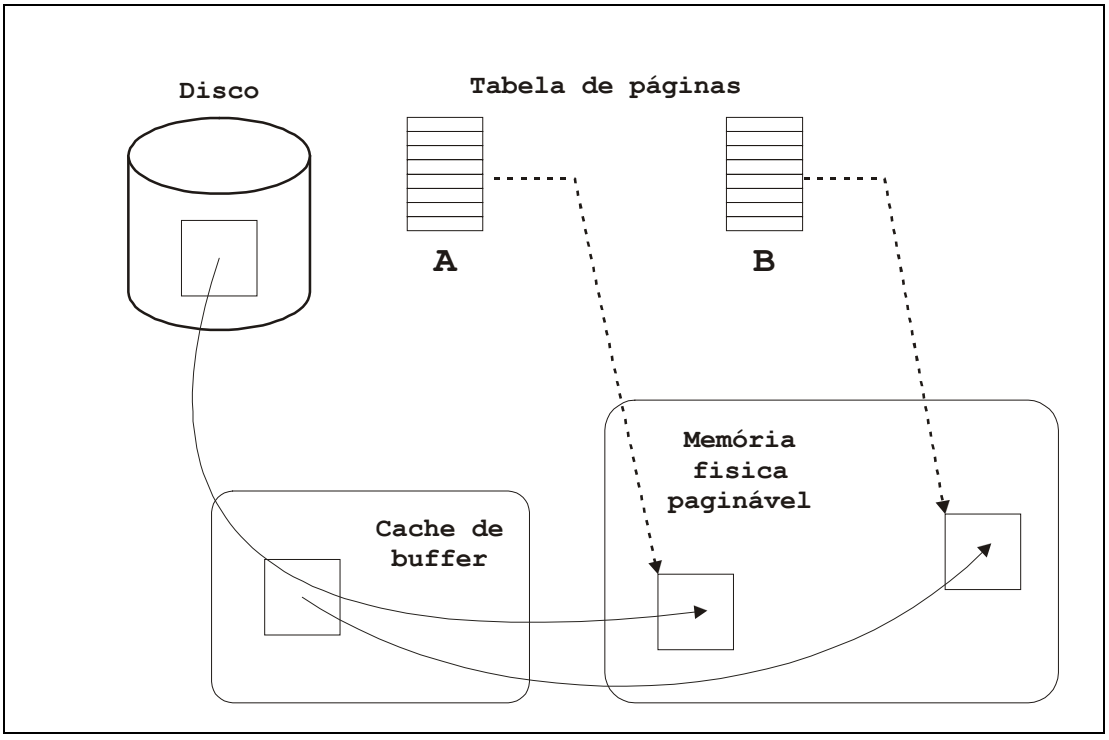

Figura 2.3: Dois processos lendo a mesma página no processo tradicional

Uma solução alternativa é o mapeamento de arquivos em memória. Nela o arquivo é trazido para a memória pelo *kernel* de modo que os processos que façam uso dele o compartilhem em memória. O processo de mapeamento é ilustrado pela figura 2.4.

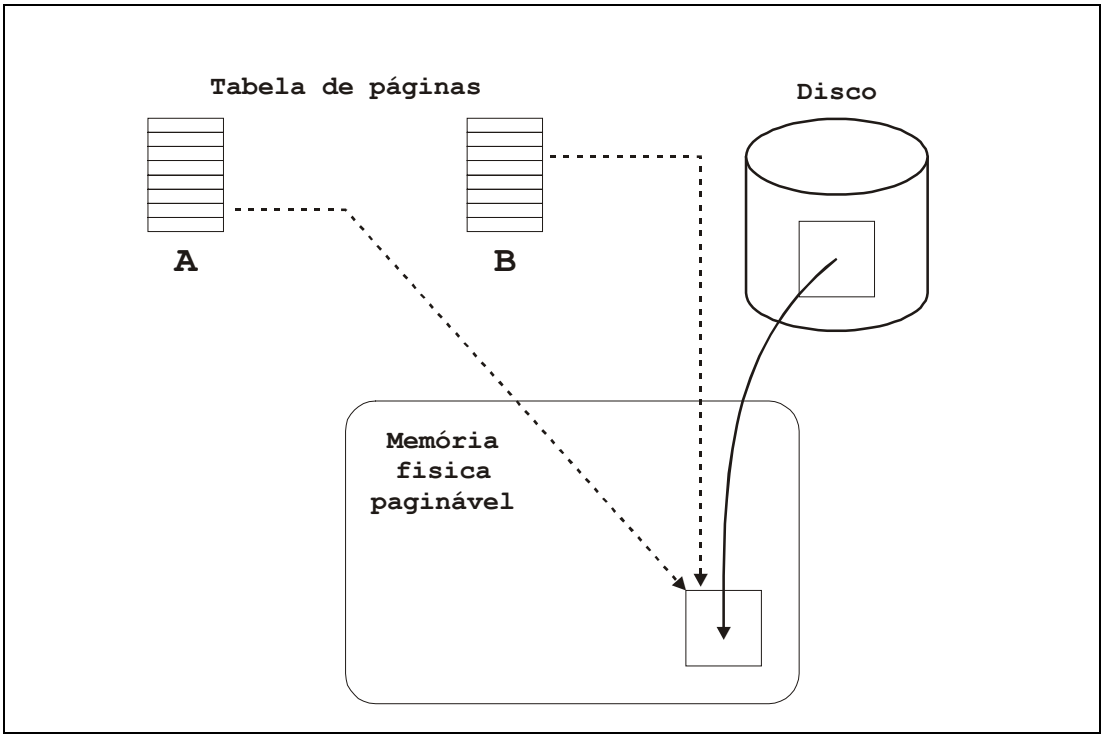

Figura 2.4: Dois processos mapeando a mesma página em seus respectivos espaços de endereçamento.

Isso evita as freqüentes mudanças de contexto entre modo usuário e modo *kernel*, uma vez que após o mapeamento do arquivo, chamadas de sistemas não são mais executadas. Além disso, apenas uma cópia da informação é mantida em memória.

Servidores Web como o Apache se beneficiam com esta funcionalidade, uma vez que é comum múltiplos processos referenciarem o mesmo arquivo. Os sistemas Unix disponibilizam o mapeamento de arquivo em memória através da chamada de sistema `mmap()`.

2.8 Implementação da pilha TCP/IP no *Kernel*

2.8.1 Estrutura

A implementação do software TCP/IP reside no *kernel* do Sistema Operacional de forma a poder ser compartilhado por todos os aplicativos do sistema.

Em geral a pilha de protocolos é implementada baseando-se no conceito de *multithreads* ou multiprocessos, onde tarefas são implementadas por diferentes fluxos de execução, facilitando a

implementação e gerenciamento dos protocolos de rede. Por exemplo, o controle dos múltiplos *timers* utilizados pelos protocolos é mais facilmente implementado a partir de fluxos de execução independentes [44].

Prioridades de processos são igualmente importantes na implementação dos protocolos, uma vez que funções críticas podem possuir prioridade maior de execução. Citando mais um exemplo, a porção de software da pilha de protocolos, que precisa aceitar pacotes do hardware à medida que eles chegam, deve ter prioridade de execução sobre um programa de usuário comum.

A implementação do software de rede no *kernel* reside entre o hardware (interfaces de rede) e os processos de usuário. A interface com os processos de usuários é feita através das *system calls* correspondentes às tarefas de redes (*socket layer*) e a interface com os dispositivos de rede é feita através dos *drivers* que abstraem funções de baixo nível, como é explicado na próxima seção.

2.8.2 Organização do código

O código de rede é organizado em 3 camadas no *kernel* como demonstrada na figura 2.5 (referente à implementação BSD padrão) [117].

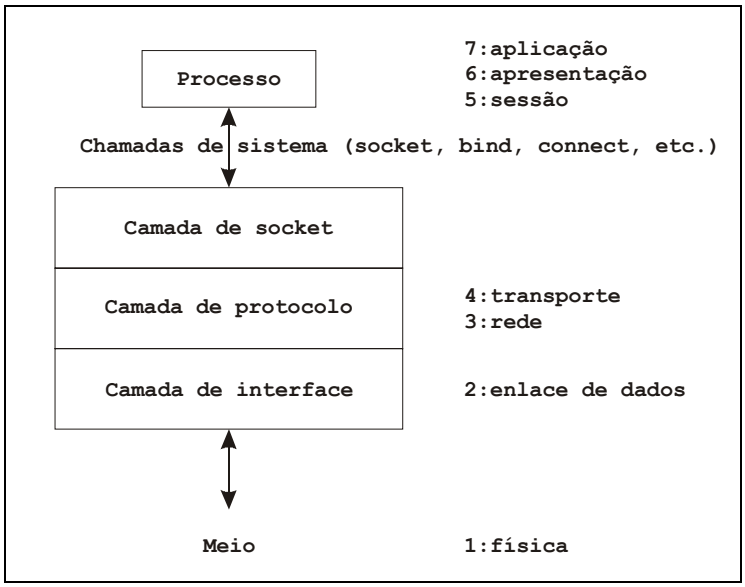

Figura 2.5: Organização do código TCP/IP. Ao lado são referenciadas as camadas correspondentes no protocolo OSI.

1. **A Camada de *socket*** é uma interface (independente de protocolo) entre os processos de usuários e a camada dependente de protocolo especificada a seguir. A camada de *socket* é relativamente simples e tem como função verificações básicas de passagem de parâmetros e a invocação de funções da camada inferior.

2. **A camada de protocolo** abriga a implementação de famílias de protocolos. No BSD são 4: TCP/IP, XNS, OSI e Unix *domain*. Cada família de protocolo tem sua estrutura própria, sendo que este texto aborda a primeira família.

3. **A Camada de interface (*drivers*)** abriga os *drivers* que se comunicam com dispositivos de rede.

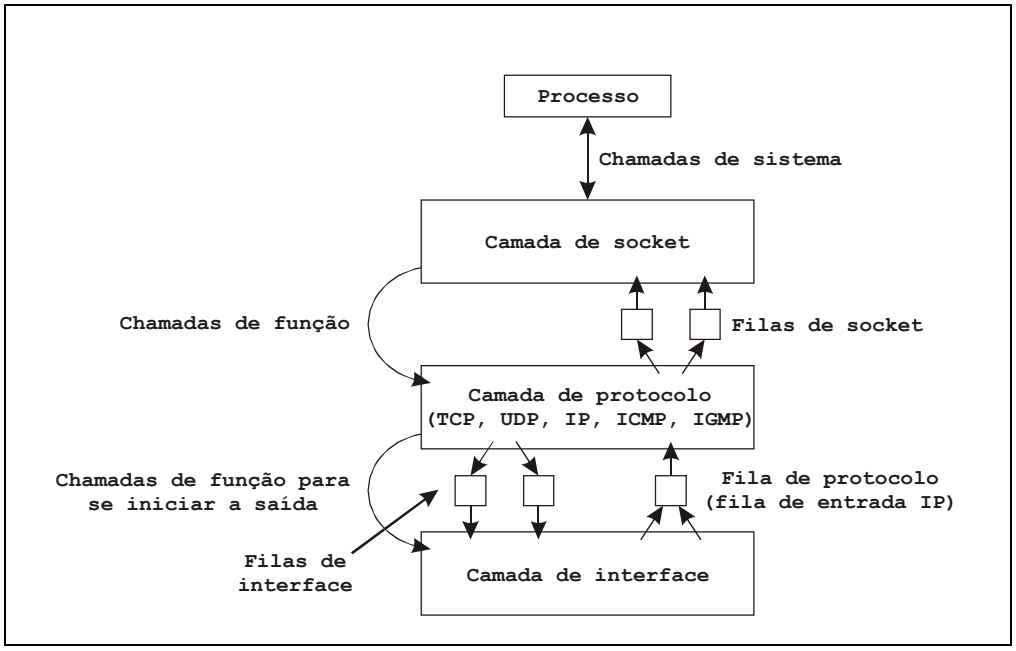

Figura 2.6: Comunicação entre as camadas.

A figura 2.6 ilustra a comunicação existente entre as camadas para o tráfego de entrada e saída de dados considerando apenas protocolos Internet.

Observa-se o uso de filas: uma para cada interface de rede, uma para o IP e uma para cada *socket*.

2.8.3 Estruturas de dados envolvidas

A manipulação de conexões de rede por um processo é feita através de descritores, da mesma forma que arquivos convencionais o são. Assim compartilham as mesmas posições na limitada tabela de arquivos abertos por um processo. A figura 2.7 ilustra as estruturas de dados desse relacionamento.

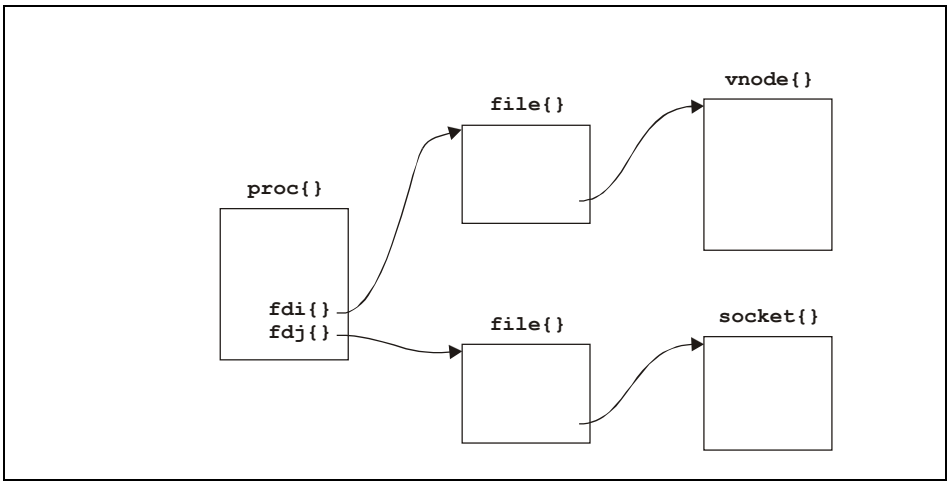

Figura 2.7: Relações fundamentais entre estruturas de dados do *kernel* para implementação de *sockets*.

Como definido em 2.6.1.5 "Descritor de Arquivo", página 55, o descritor de arquivo/*socket* nada mais é que o índice para uma entrada na tabela. Esta por sua vez aponta para a estrutura `file{}` que, dependendo do tipo, pode referenciar uma estrutura de *inode* ou de *socket*. Este último serve como raiz para outras informações relativas ao *socket* como conexões existentes (endereços de origem e destino, tipo de conexão (TCP ou UDP), ponteiros para *buffers,* dentre outras.

2.8.4 mbuf (*Memory Buffer*)

Protocolos de rede demandam facilidades providas pelo gerenciador de memória como a manipulação de *buffers* de tamanho variado, concatenação de dados a esses *buffers* (à medida que dados vão sendo passados às camadas inferiores do protocolo) e a otimização da quantidade de dados a serem copiados nessas operações. O desempenho de protocolos de rede está

diretamente relacionado ao esquema de gerenciamento de memória empregado pelo *kernel* [117].

A estrutura de dados principal utilizada nas implementações BSD e derivados, incluindo a do Linux, é a *mbuf*, cuja função é abrigar os dados que trafegam do processo do usuário à interface de rede e vice-versa.

Esta estrutura, de tamanho pequeno (128 bytes), é utilizada em cadeias. São de 3 tipos:

1. O que transporta apenas dados.

2. O que transporta dados e informações de cabeçalho. É o primeiro *mbuf* que descreve um pacote de dados.

3. O que é um ponteiro para um cluster de 1024 ou 2048 bytes que busca diminuir o tamanho da lista de *buffers*.

Como mostra a figura 2.8 o encadeamento de mbufs segue em duas direções para os dados de entrada e de saída.

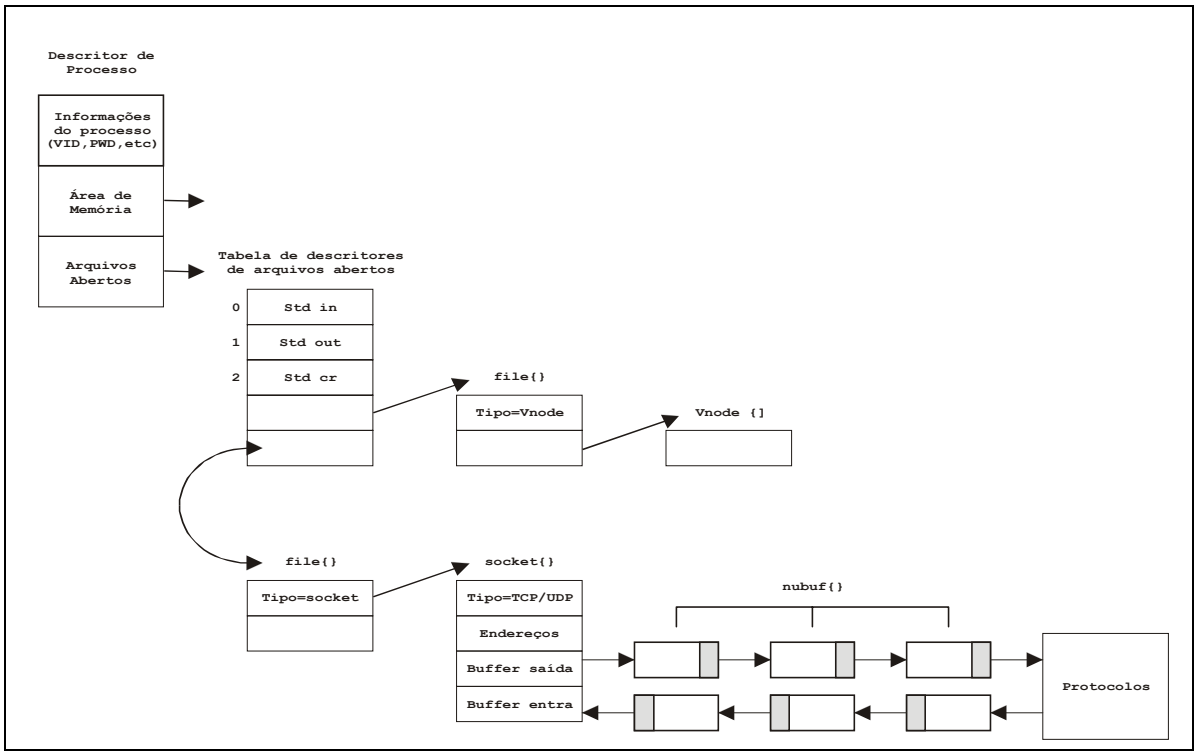

Figura 2.8: Principais estruturas de dados envolvidas na implementação do TCP/IP.

Mais detalhes sobre os protocolos de redes são abordados nos próximos capítulos.

2.9 Ferramentas de monitoramento

A maioria das versões de Unix possui ferramentas de monitoramento de subsistemas do Sistema Operacional. Nesta seção são apresentadas as principais e como elas podem ser utilizadas para se medir a atividade do sistema, possibilitando a detecção de pontos de latência.

Dependendo das versões de Unix utilizadas a sintaxe pode variar demandando consultas às *man pages* correspondentes. Os exemplos utilizados referem-se ao Linux Red Hat 7.1.

2.9.1 ps (Process Status)

ps é uma das ferramentas básicas, que lista todos os processos presentes no Sistema Operacional com informações sobre CPU e o uso de memória.

Através desse comando é possível verificar, por exemplo, o número de processos filhos criados pelo servidor Web para se atender múltiplas requisições simultâneas:

```
#ps -ef | grep -c httpd
750
```

Isto é útil para se determinar se o número de processos filhos demandado pela carga atual do sistema está próximo do limite máximo configurado no servidor Web. Caso esteja, é necessário a reconfiguração do servidor, aumentando esse limite. A Seção 5.3.4.2 mostra mais detalhes. Pelo fato do Linux não diferenciar internamente *threads* de processos (como exibido em 2.5.2.3 "Suporte a threads no Linux", página 46) ao se executar servidores *multithreaded* como o Apache 2.0 cada *thread* é exibida como se fosse um processo na listagem do ps, diferentemente do que acontece no Solaris, onde as *threads* são ocultadas no processo pai.

Através do ps pode-se detectar também serviços desnecessários que estejam sendo executados pelo sistema e que devem ser desabilitados.

2.9.2 top

O top é uma ferramenta utilizada para se determinar em tempo real processos que estão consumindo mais recursos no momento. Assemelha-se bastante a um ps "auto-refreshing".

Uma informação bastante interessante fornecida pelo top é o *load average* (carga média) que indica a média de processos prontos aguardando para serem executados. Um número alto indica deficiência de capacidade da CPU. Geralmente servidores que necessitam executar uma grande quantidade de CGIs, demandam um poder de processamento maior do que servidores de páginas estáticas somente.

2.9.3 vmstat

vmstat provê um resumo das atividades de várias funções do sistema incluindo utilização de memória, paginação, atividade de disco, chamadas de sistema e utilização da CPU.

As informações baseiam-se na coleta de informações dentro de um intervalo fixo fornecido pelo usuário. A primeira linha de estatística refere-se a informações do sistema desde a última reinicialização.

A seguir um resumo das informações disponibilizadas pelo vmstat.

2.9.3.1 Processos

Existem 3 campos: *r,b,w*, que na melhor das hipóteses devem se manter zerados.

O primeiro *r* indica o número de processos que estão aguardando para ser executados. O segundo indica o número de processos aguardando operações de I/O. O terceiro indica o número de processos que poderiam ser executados caso não estivessem na área de *swap*.

Valores maiores que zero no primeiro parâmetro indicam problemas no poder de processamento. No terceiro indica falta de memória.

2.9.3.2 Memória

Especifica a quantidade de memória utilizada para diferentes funções:

- ***Swap***: montante de memória virtual utilizada pelo sistema em *kilobytes*. Um valor maior que zero indica que a memória física já não comporta a carga atual.

- ***Free***: quantidade de memória não utilizada. Mesmo em casos de falta de memória e necessidade da utilização da área de *swapping* este valor tende a manter-se num limite mínimo maior que zero, reservado ao *kernel* para ser utilizado em situações críticas.
- ***Buffer***: montante de memória usada como *buffer* de dispositivos. Um valor alto indica grande quantidade de dados em cache dispensando acessos a discos. Valores baixos para sistemas com carga alta indicam falta de memória, uma vez que espaço de memória que poderia ser utilizada para *buffer* está alocada para estruturas do *kernel* ou processos de usuários.

2.9.3.3 Swap

São dois campos, si: *swapped in* e so: *swapped out*, que indicam a freqüência de *swap*. Igualmente, valores maiores que zero para esses campos indicam a falta de memória.

2.9.3.4 I/O

São dois parâmetros bi: blocks in e bo: blocks out. Em blocos por segundo indicam a freqüência de gravação e leitura de blocos em dispositivos. Valores maiores que zero nestes parâmetros não necessariamente indicam a falta de memória, uma vez que acessos a discos podem ser feitos por I/O de programas e não *swapping* somente.

2.9.3.5 Sistema

Parâmetros relacionados à atividade do escalonador. O campo *in* indica a quantidade de interrupções (oriundas de hardware incluindo o *clock*) ocorridas por segundo. *CS* indica o número de mudanças de contexto executadas por segundo pelo escalonador.

2.9.3.6 CPU

Mostra a porcentagem de uso de CPU por processos de usuário (*user time*), por tarefas do sistema (*system time*) além da porcentagem de tempo que a CPU ficou ociosa (*idle time*).

2.9.4 sar (System activity reporter)

O sar provê informações semelhantes ao vmstat, embora seja mais completo e flexível. Dados podem ser armazenados num formato binário compacto, para a coleta durante um longo período. Entre as informações mais importantes fornecidas pelo sar destacam-se:

- **Dados de paginação**: (sar -B) exibe freqüência de paginação em KB/s de dados lidos e gravados.
- **Uso de CPU**: (sar -v) exibe informações sobre uso da CPU (idle, user, system).
- **Uso de memória**: (sar -r) exibe como a memória está sendo utilizada.
- **Interface de rede**: (sar -n) exibe informações sobre as interfaces de rede como tráfego de entrada/saída em pacotes ou bytes, além da freqüência de erros.

2.9.5 netstat

netstat exibe várias informações relacionadas ao ambiente de rede. Na realidade não existe um tema que unifique as diferentes informações providas, exceto o fato de que todas são relacionadas a informações da rede [97].

Os usos mais comuns do netstat são:

- Observar quais as conexões atuais e qual o status dessas.
- Inspecionar informações de configuração e tráfego de interfaces de rede.
- Obter informações estatísticas de vários protocolos de rede.
- Exibir tabela de rotas (útil apenas para roteadores).

2.9.5.1 Conexões Ativas

```
[root@orion /root]# netstat -at
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:pop3s                 *:*                     LISTEN
tcp        0      0 *:http                  *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:smtp                  *:*                     LISTEN
tcp        0      0 *:https                 *:*                     LISTEN
tcp        1  21901 orion:http              200.144.39.35:36837     LAST_ACK
tcp        0      0 localhost:35452         localhost:telnet        ESTABLISHED
tcp        1  21901 orion:http              200.144.39.35:37191     LAST_ACK
tcp        0      0 orion:pop3s             mozart.convest.uni:1450 TIME_WAIT
tcp        1  21901 orion:http              200.144.39.50:43334     LAST_ACK
tcp        1  21901 orion:http              200.144.39.35:37338     LAST_ACK
tcp        0    128 orion:ssh               mozart.convest.uni:1086 ESTABLISHED
```

Através desta opção é possível identificar situações importantes em um servidor Web, como por exemplo, o número de conexões ativas, número de conexões perdidas e que permanecem ocupando memória através de *buffers*, além de se identificar possíveis ataques de negação de

serviço através da grande quantidade de conexões no estado SYN_RECEIVED (ver 3.3.3 "Ataques de negação de serviços e os “syn_cookies” do Linux", página 97).

Através da opção –c, que atualiza as informações a cada segundo, é possível também observar o ciclo de vida de uma conexão obtendo-se mais informações de suporte na configuração e refinamento dos diversos timers da pilha de protocolos TCP/IP (ver 3.6 "Controle de Timers", página 110).

2.9.5.2 Exibindo portas abertas e identificando os programas servidores

Através da opção –nap é possível se identificar quais portas estão recebendo conexões, além de quais programas (incluindo o *path* e PID) estão ouvindo essas portas. netstat -l exibe somente as portas em LISTEN.

É importante que um servidor Web seja uma máquina exclusiva a este fim, tanto por questões de desempenho quanto de segurança, e através deste comando é possível identificar outros serviços que devem ser desabilitados.

2.9.5.3 Informações de Interfaces

Através da opção -i é possível se obter o *status* das interfaces de rede.

```
[root@orion /root]# netstat -i
Kernel Interface table
Iface   MTU Met    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0   1500   0   276549      0      0      0   334918      0      0      0 BRU
lo    16436   0    58804      0      0      0    58804      0      0      0 LRU
```

Colisões indicam uma rede sobrecarregada, enquanto erros indicam problemas de cabeamento, físicos da interface ou de *drivers*. Taxas aceitáveis para uma rede funcionando corretamente são: 3% de colisões em relação ao número de pacotes transmitidos, e não mais de 0,5% de erros [97].

netstat pode também exibir estatísticas sobre uma interface em intervalos periódicos, através da especificação do número de segundos. Este modo é útil para se rastrear a origem dos erros.

2.9.5.4 Contadores de Protocolos

netstat –s exibe o conteúdo de vários contadores espalhados pelo código de rede. A saída é separada em seções para IP, ICMP, TCP e UDP.

Para o serviço Web a seção mais importante é a do TCP. O contador `tcpListenDrop`, por exemplo, especifica o número de conexões recusadas devido ao esgotamento da fila de requisições (ver 3.3.1 "O three-way-handshake", página 93).

2.9.6 Ferramentas de Monitoramento de Tráfego

Ferramentas dessa classe são denominadas *sniffers*, cuja função é ouvir o tráfego de uma rede e gravar ou exibir pacotes que contêm atributos que coincidem com um certo conjunto de critérios fornecido pelo usuário. Por exemplo, gravar todos os pacotes relacionados a uma determinada porta ou a um determinado *host*.

Sniffers são úteis para se solucionar problemas conhecidos ou mesmo detectar novos. Para que funcionem é necessário que a tecnologia de hardware permita acesso a todos os pacotes. Isso é fato para tecnologias orientadas à *broadcast* como é o caso da Ethernet via *hub*, por exemplo, mas não funcionam quando o tráfego é separado em segmentos através de *switches*. Além disso, é necessário que a interface de rede suporte o modo "promíscuo" onde ela passa a aceitar todos os pacotes lançados à rede e não somente aqueles cujo endereço físico corresponda ao seu próprio endereço.

O tcpdump é um dos *sniffers* mais conhecidos e é presente na maioria das distribuições Unix.

O snoop é um clone do tcpdump desenvolvido pela Sun, disponível nas distribuições do Solaris.

Outro exemplo de *sniffer*, agora com interface gráfica, é o *ethreal* [52].

Capítulo 3

O TCP

O protocolo TCP (*Transmission Control Protocol*) atua juntamente com o UDP na camanda de transporte provendo serviços para a camada de aplicação, incluindo o protocolo HTTP. Foi definido formalmente na RFC 793, sofreu correções na RFC 1122 e possui extensões definidas na RFC 1323.

O TCP tem como principal função prover à camada de aplicação um serviço orientado a conexão entre dois processos em máquinas distintas (comunicação entre processos ponto-a-ponto) que seja *full-duplex*, fluxo-controlado e principalmente confiável, independente das redes existentes entre esses dois pontos. Para tanto, utiliza-se de vários mecanismos como *timers* e algoritmos de controle de congestionamento, tornando-o sem dúvidas em um dos protocolos mais complexos da hierarquia TCP/IP. Sendo assim, possui inúmeros pontos de estudo que afetam diretamente o desempenho do tráfego Web, e que serão aboradados neste capítulo.

3.1 Interface com a camada de Aplicação

As 2 APIs mais populares para criação de programas que usam os protocolos Internet são: *sockets* e TLI (*Transport Layer Interface*). A primeira é comumente chamada de *Berkeley Sockets* e é originária das distribuições BSD enquanto a segunda é originária das distribuições AT&T (System V). A interface BSD é a mais popular sendo utilizada em inúmeros sistemas, derivados ou não do Unix, e será abordada neste texto [117].

3.1.1 Sockets

A camada de aplicação obtém serviços de comunicação de arquitetura cliente/servidor da camada de transporte através do conceito de *sockets*. Duas aplicações, presentes em máquinas distintas e que desejam trocar informações entre si, utilizam-se de um *socket* como ponto de comunicação. Um *socket* é composto pelo endereço IP da máquina mais um número interno de 16 bits chamado de *porta*. Assim, o conceito de porta é um endereçamento interno à máquina possibilitando que ela possa estabelecer simultaneamente diferentes conexões à rede controlada por diferentes aplicações ou instâncias de aplicações.

O TCP reserva as identificações de portas no intervalo de 0-255 para as "portas bem conhecidas", que são utilizadas para atender serviços que são padrões da Internet, de forma que os clientes possam abrir conexões sem a necessidade de "adivinhar" as portas onde devem se conectar. O restante do espaço de portas pode ser livremente alocado a outros processos. Uma lista das "portas bem conhecidas" associadas aos seus respectivos serviços encontra-se na RFC 1700.

Essa noção pode ser estendida adicionando uma terceira sub-divisão para o espaço de portas TCP: as portas reservadas, que são em geral, utilizadas para serviços específicos do sistema operacional. Na maioria das implementações esse espaço de porta varia entre 0 a 1023 (havendo assim intersecção com as portas "bem conhecidas"). Tal intervalo também é protegido, permitindo-se que somente usuários privilegiados recebam conexões TCP por meio destas portas.

No Unix, o arquivo `/etc/services` lista as portas com os respectivos nomes dos serviços existentes na máquina. A identificação padrão para o serviço HTTP é a porta 80.

3.1.2 Primitivas

O conjunto de primitivas é a interface pela qual as aplicações fazem uso dos serviços oferecidos pelo TCP, por onde é possível se realizar a comunicação entre processos. São funções, através das quais os *sockets* são manipulados, desde sua criação, controle, transferência de informações até seu encerramento.

Além de aceitar comandos, o TCP deve também retornar informações de controle para o processo que está servindo, como respostas que indicam o sucesso ou os vários tipos de falhas que podem ocorrer durante a tentativa de execução da função.

A especificação (RFC 793) provê uma descrição de alto nível bastante flexível de forma que cada Sistema Operacional, que pode possuir diferentes funcionalidades, acrescentem características próprias. Todavia, todas as implementações TCP devem prover um conjunto mínimo de serviços para garantir que todas possam suportar a mesma hierarquia de protocolos.

Abaixo são apresentadas as primitivas utilizadas na interface *sockets* de Berkeley. As primitivas são implementadas no Sistema Operacional através de chamadas ao sistema (*system calls, ver 2.3.2 "Modos de execução e as System Calls", página 28).*

Socket

Primitiva utilizada para a criação de um novo *socket*. Como parâmetros, recebe o formato do endereço a ser utilizado (em geral o Internet) e o tipo de serviço (datagrama ou fluxo - a mesma primitiva é usada também para a criação de *sockets* UDP). Retorna um *descritor de socket* (similar ao descritor de arquivos) que é utilizado para posteriores ações sobre o *socket* (ver *2.6.1.5 "Descritor de Arquivo", página 55.*)

Bind

Associa um endereço (número de porta) a um descritor de *socket*. Seus argumentos são o descritor do *socket* e uma referência para uma estrutura que contém o endereço que deve ser associado ao descritor.

Pode parecer mais razoável se ter somente uma única primitiva para ambos: a criação do *socket* e atribuição da identificação. Todavia, a vantagem de se possuir duas primitivas é o fato de que existem aplicações que necessitam de um endereço fixo por ser padrão (portas bem conhecidas), ao passo que outras aplicações não exigem este endereçamento fixo, como por exemplo os clientes, que podem se utilizar da primeira porta livre. Para essas últimas o bind sequer é invocado.

Listen

Usado para sinalizar que o processo está pronto para aceitar conexões em um *socket*. Além disso, serve para se alocar espaço para a fila de conexões, de modo que vários clientes possam requisitar conexões simultaneamente. Esta é uma situação bastante comum, como veremos, para servidores Web de grande demanda, onde múltiplos clientes podem requisitar conexões

simultaneamente à porta 80, gerenciada pela aplicação servidora HTTP. Neste contexto, um parâmetro bastante importante no controle de desempenho, é o tamanho da fila de conexões a ser alocada, pois ultrapassando este limite, novos pedidos de conexão são rejeitados. Maiores detalhes sobre este assunto são discutidos em *3.3.1 "O three-way-handshake", página 93*.

Accept

Bloqueia a aplicação (servidora) aguardando por conexões. Quando uma requisição de nova conexão chega, é criado um novo *socket* com as mesmas características do original e seu descritor é retornado. Neste instante a aplicação deve se utilizar do conceito de multiprocessos (*fork*) ou *multithreading* para atender a nova conexão (e as possíveis posteriores), enquanto o *socket* original continua sendo utilizado para se aguardar as novas requisições. Somente neste momento é que a conexão é retirada da fila especificada em *listen*, liberando espaço para outro pedido de conexão. As aplicações servidoras Web multiprocessos ou *multithreading* são discutidas em 5.1 "Arquitetura de Servidores Web", página 129.

Connect

Usada na aplicação (cliente) para se requisitar o estabelecimento de uma conexão com uma outra máquina (servidor). Passa-se como parâmetro o descritor de *socket* previamente criado e o endereço do servidor (IP + endereço do *socket*). Não é obrigatório invocar o *bind* anteriormente, uma vez que a primitiva já associa o descritor do *socket* a um endereço livre aleatório. É uma primitiva bloqueante. Quando uma resposta apropriada do servidor é recebida, o processo é desbloqueado e a conexão estabelecida.

Send e Receive

São primitivas usadas tanto pelo cliente quanto pelo servidor para a transferência dos dados sobre a conexão *full-duplex* criada. *Send* é similar ao *write* (para arquivos). Ela envia os dados passados como parâmetro via o *socket* e retorna o número real de caracteres enviados. Além disso, através desta primitiva envia-se também as *flags* “URG” e “PUSH”, abordadas em 3.2 "Segmentos TCP", página 90. Já o *receive* é análogo ao *read*: recebe os bytes em um *buffer* passado como parâmetro no espaço de memória do usuário e retorna o número de bytes lidos.

Close

Libera estruturas de dados internas como *buffers* e encerra o fluxo unidirecional.

O fechamento da conexão exige ação conjunta. A conexão é desfeita quando ambos os lados executam a primitiva close.

Primitiva	Função
SOCKET	Criar um novo ponto de conexão.
BIND	Associar um endereço local (número de porta) ao socket.
LISTEN	Tornar o socket disponível para receber conexões e definir o tamanho da fila de requisições.
ACCEPT	Bloquear o processo até que uma requisição chegue. Retornar o descritor de um novo socket para o tratamento da nova requisição.
CONNECT	Tentar o estabelecimento de uma conexão ativamente.
SEND	Envia dados da conexão.
RECEIVE	Receber dados da conexão.
CLOSE	Fechar a conexão.

Tabela 3.1: Primitivas TCP

3.1.3 Ilustrando conexão cliente servidor via *sockets*

A figura 3.1 ilustra uma conexão cliente/servidor utilizando as primitivas descritas [45].

`s = socket(AF_INET,SOCKET_STREAM,0)`
`(..)`
`(..)`
 	
`connect(s, ServerAddress)`
`(..)`
`send(s, “data”, length)  -----→`
 	
Requisitando uma conexão (cliente)
	`s = socket(AF_INET, SOCK_STREAM, 0)`
	`(..)`
	`bind(s, ServerAddress)`
	`listen(s,5)`
	`(..)`
	`sNew = accept(s, from)`
	`n = receive (sNew, buffer, nbytes)`
	
	Aguardando e aceitando uma conexão (servidor)

Figura 3.1: Uso das primitivas.

- O servidor inicialmente cria o *socket* TCP usando a primitiva *socket*. Em seguida, o *bind* é invocado, associando o descritor do *socket* ao endereço do *socket* (uma porta "bem conhecida"). A primitiva *listen* é então chamada tornando o *socket* disponível para aceitar pedidos de conexões. O segundo argumento do *listen* especifica o *backlog*, número máximo de requisições que podem estar na fila aguardando estabelecimento de conexão, no exemplo igual a 5.
- O servidor utiliza a primitiva *accept* para aceitar a conexão requisitada por um cliente, obtendo um novo *socket* para a comunicação com este cliente. O *socket* original pode ainda ser utilizado para atender pedidos de novas conexões.
- O processo cliente utiliza a primitiva *socket* para a criação de um novo *socket*, invocando em seguida a primitiva *connect* que requisita uma conexão através do endereço do *socket* do processo servidor. Devido ao fato de *connect* invocar o *bind* automaticamente, a associação prévia de nome não é necessária.
- Após a conexão, cliente e servidor se comunicam através de *send* e *receive*.

3.2 Segmentos TCP

Embora sob o ponto de vista da aplicação o TCP forneça uma conexão através de um fluxo de bytes, o TCP implementa esta conexão através de segmentos como unidade de comunicação.

A cada segmento enviado pelo remetente, este aguarda seu reconhecimento (*ACKnowledgment*) pelo destinatário, para que o segmento seja considerado como enviado corretamente. Para se indicar a perda de segmentos, torna-se necessária a implantação de *timers* e identificadores de segmentos. Na realidade, esta identificação é feita para cada *byte* transmitido, através de um número seqüencial de 32 bits. Além do controle de reconhecimento (ACKs) este número seqüencial é utilizado também para o controle de fluxo de segmentos .

Um segmento TCP consiste de um cabeçalho que é formado por uma parte de tamanho fixo (20 bytes) e uma parte variável que abriga possíveis opções de comunicação não providas pela parte fixa. O cabeçalho é seguido pelos bytes de dados. Desta forma, o tamanho de um segmento TCP pode, teoricamente, variar de 20 bytes a 64KB dependendo do ambiente. A figura 3.2 ilustra um segmento TCP.

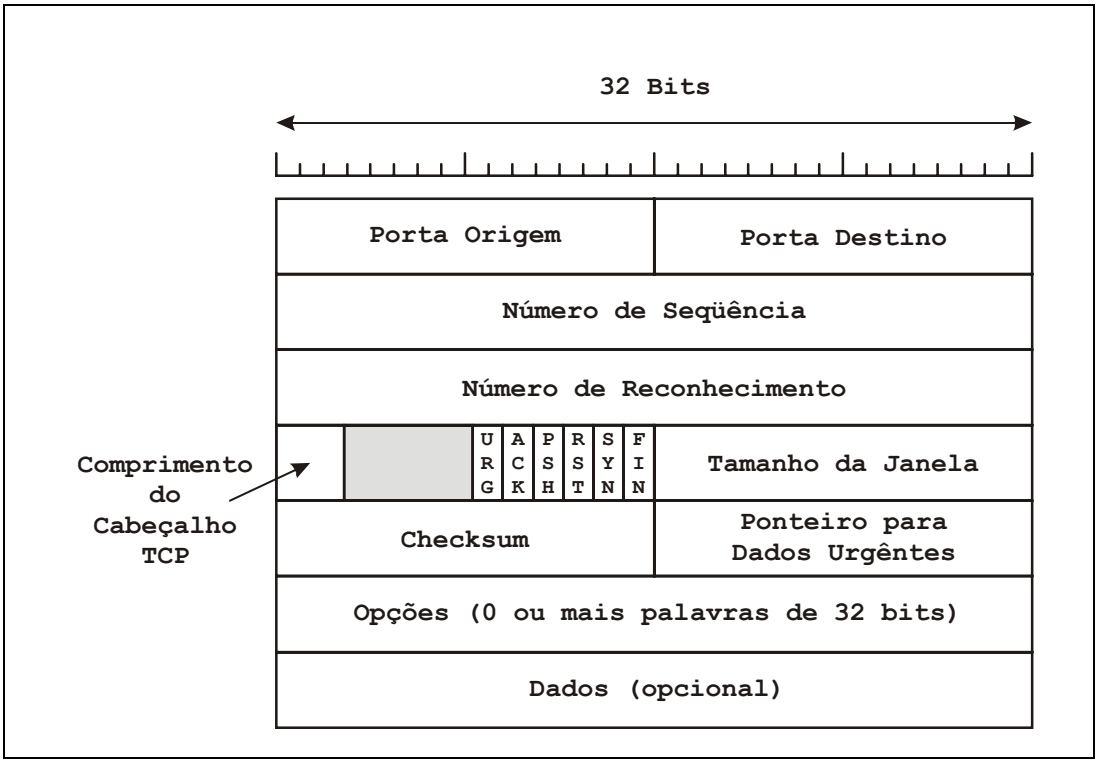

Figura 3.2: Cabeçalho de um segmento TCP.

Os campos iniciais, *Porta Origem e Porta Destino,* indicam os endereços dos *sockets* envolvidos na comunicação em cada máquina.

O *Número de Seqüência* é a identificação do 1º byte de dados presente no segmento. Já o *Número de Reconhecimento* indica qual o próximo byte esperado. Através deste campo é implementado o mecanismo de *piggybacking*: aproveita-se um segmento indo em direção oposta para o reconhecimento de segmentos recebidos anteriormente, evitando-se o envio de um único segmento exclusivamente para isto. Para se aumentar a eficiência deste mecanismo, associa-se a técnica de "*delayed ACK*", onde o reconhecimento é atrasado por um momento aguardando o envio natural de um segmento para se realizar o *piggybacking*.

O *Tamanho do Cabeçalho* indica quantos *bytes* foram usados para o cabeçalho. Em outras palavras, indica onde começa a parte de dados do segmento. Este torna-se necessário devido a presença do campo *Opções* que possui tamanho variável.

A *flag URG* indica que o campo *Urgent Pointer* deve ser considerado. *Urgent Pointer* sinaliza o deslocamento a partir do número seqüencial inicial do segmento onde a informação urgente está localizada .

A *flag ACK* indica se o segmento está sendo utilizado também para o reconhecimento de um outro segmento que teve caminho inverso (*piggybacking*). Caso seu valor seja 1, o campo *Número de Reconhecimento* deve ser considerado.

PUSH indica que o dado de ser enviado diretamente à aplicação, não devendo ser *bufferizado*. *RST* é utilizado para se "derrubar" uma conexão que tornou-se instável devido a problemas em um dos pontos de comunicação (um *crash* da máquina por exemplo). Além disso, serve também para recusar a abertura de uma nova conexão.

SYN é utilizado para sinalizar o início, o estabelecimento de uma conexão, que é implementada através do método *three-way-handshake* (SYN=sincronizar números seqüenciais). FIN tem efeito contrário: indica o pedido de fim de conexão.

Tamanho da Janela é o campo utilizado para o controle de fluxo de segmentos. Indica quantos bytes o remetente estaria apto a receber (quantos ainda caberiam em seus *buffers*) a partir do byte reconhecido.

Checksum valida o cabeçalho TCP, os dados e também um "pseudo-cabeçalho" que inclui inclusive campos do IP, como o endereço IP origem e destino, com objetivo de identificar pacotes entregues erroneamente.

Finalmente, *Opções,* foi criado com intuito de prover funcionalidades adicionais não cobertas pelos outros campos do cabeçalho. Um dos parâmetros mais comumente utilizados é o tamanho máximo de segmento (MSS) que cada um dos *hosts* envolvidos pode manipular. Isto é dependente da tecnologia de rede utilizada, e é um parâmetro combinado no momento da conexão. Caso não especificado o valor padrão é 536. Outras opções para controle de fluxo são associadas a este campo e serão detalhadas na Seção 3.5.

3.3 Estabelecimento e Fechamento de Conexão

3.3.1 *O three-way-handshake*

Como visto, o TCP é um protocolo de transporte orientado a conexão. Desta forma, antes que os dados sejam trocados entre os processos é necessário que a conexão seja estabelecida. Isto é feito através do algoritmo conhecido como *three-way-handshake*.

Este algoritmo recebe este nome pois demanda a troca de três segmentos iniciais para o estabelecimento da conexão, como descrito a seguir:

1. O processo da máquina requisitante (normalmente o cliente) envia o segmento inicial com a *flag* SYN ativada e especificando o número da porta onde o processo servidor deve estar aguardando requisições além do número de seqüência inicial (ISN) X. Neste segmento também é especificado o MSS, no campo "opções".

2. O servidor responde com um segmento contendo a *flag* SYN também ativada para a porta especificada pelo cliente e também um número inicial seqüencial Y. O servidor também reconhece o segmento inicial do cliente retornando o ISN do cliente (Y) + 1, no campo ACK.

3. O cliente reconhece o segmento enviado pelo servidor (Y + 1). A partir deste instante a fase de estabelecimento de conexão é terminada.

A figura 3.3 ilustra o *three-way-handshake*.

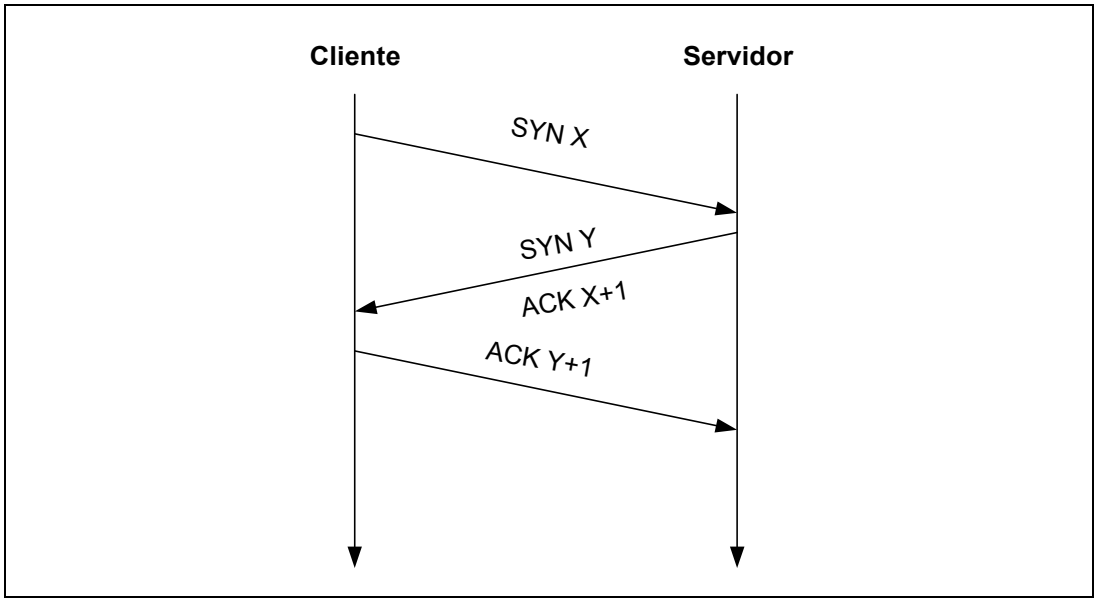

Figura 3.3: Segmentos trocados no *three-way handshake*

3.3.1.1 Implementação

Derivados do BSD padrão implementam o *three-way-handshake* através de duas filas no servidor. A primeira abriga conexões pendentes, ou seja, apenas o primeiro segmento foi recebido, o reconhecimento foi enviado pelo servidor, porém este ainda aguarda o recebimento do terceiro segmento. Nesta situação a conexão encontra-se no estado `SYN_RECEIVED` (conexões *half-open,* fila denominada `so_q0`). No momento em que o servidor recebe este último segmento, a conexão é tida como estabelecida e movida para uma segunda fila (`so_q`). Nesta, são mantidas as conexões estabelecidas mas que aguardam ainda o tratamento por um processo ou *thread*, momento em que são retiradas desta fila pelo `accept()`. Assim, `so_q` serve como uma interface entre a pilha TCP/IP e a aplicação. Essas duas filas correspondem a um único *socket*.

A figura 3.4 ilustra o funcionamento das duas filas.

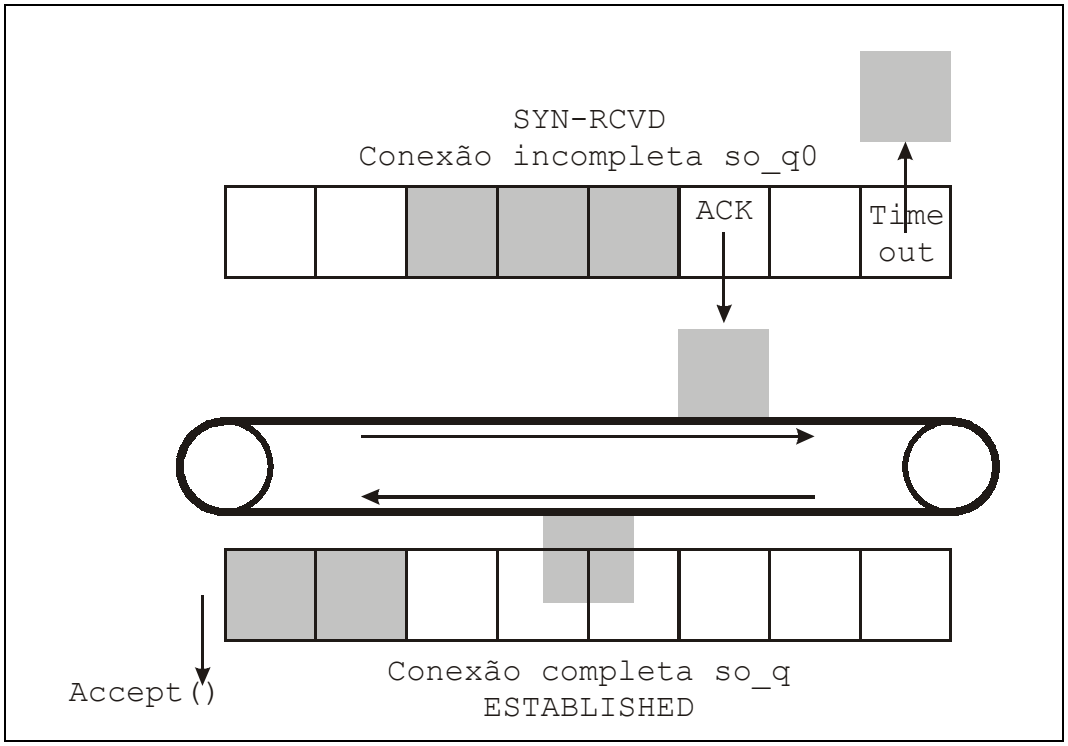

Figura 3.4: Filas (`so_q0` e `so_q`) mantidas para o atendimento de conexões.

3.3.1.2 Tamanho das filas de requisições

Um limite fundamental de servidores Web é a taxa pela qual o servidor consegue atender a pedidos de conexão. Isso depende da taxa com que pedidos são feitos e do tamanho das filas. Quando as filas enchem, o TCP passa a descartar silenciosamente posteriores pedidos de conexão[1]. Para o usuário, um servidor que não pode aceitar novas conexões é tão ruim quanto um servidor *down* [74]. Torna-se então necessário, dimensionar o tamanho das filas de acordo com a carga.

Em sistemas Unix é possível a configuração do tamanho dessas filas. Segundo Vöckler em [129], `so_q0` necessita de mais entradas em servidores sobrecarregados do que `so_q`. Isso baseia-se em experimentos que indicam que um servidor com relativa carga tem um `so_q` vazio durante 99% do tempo e um `so_q0` necessitando 15 ou menos entradas em 98% do tempo.

1. O RST não é enviado intencionalmente para que ocorra o *timeout* do cliente e este reenvie o pedido de conexão, possivelmente num momento em que a fila não esteja cheia.

O Linux implementa as duas filas, sendo que o tamanho máximo da primeira (a de conexões incompletas) pode ser alterado em `/proc/sys/net/ipv4/tcp_max_syn_backlog` via interface *sysctl*. O tamanho padrão do *kernel* 2.2 é 128, e no 2.4 é 1024. Já o tamanho da segunda fila é determinado pela aplicação, na chamada `listen()`. Para evitar ataques de negação de serviço do tipo SYN *flooding,* o Linux implementa também o conceito de "syn-cookies" explicitado na próxima seção.

Versões do Solaris anteriores à 2.6 possuiam uma única fila, não diferenciando os dois estados anteriormente descritos. A partir da versão 2.6, a implementação passou a utilizar-se de duas filas como especificado, podendo ter o tamanho configurado através dos parâmetros `tcp_conn_req_max_q0` e `tcp_conn_req_max_q`. Vöckler recomenda um valor entre 1024 e 10240 para a primeira fila e um entre 128 e `tcp_conn_req_q` para a segunda.

Tais alterações surtirão efeito somente quando a aplicação for alterada na chamada `listen()`.

É importante frisar que esses parâmetros não influenciam no número máximo de conexões estabelecidas, após serem atendidas via `accept()`. Esse limite está relacionado com o número máximo de descritores de arquivos especificados em 2.6.1.5 "Descritor de Arquivo", página 55.

Uma forma de identificar se o sistema está atendendo menos conexões que poderia, pelo tamanho limitado de uma ou das duas filas, é através do comando netstat –s observando-se o parâmetro `tcpListenDrop`, que especifica o número de conexões que foram recusadas por esgotamento da fila. Números altos para este parâmetro indicam a necessidade do aumento das filas (ver 2.9.5 "netstat", página 81).

O número de pedidos que podem ser atendidos em função do tamanho da fila so_q0 pode ser calculado aproximadamente através da seguinte fórmula [60]:

```
Tempo_na_fila: T(Round_Trip) + T(proc_ack_cliente)+T(accept).

    Taxa_Max_Aceitação: Tamanho da fila / Tempo_na_fila.
```

O parâmetro tempo na fila, seria calculado baseando-se no tempo médio de *Round_Trip* das conexões com os clientes, e evidentemente não é constante. Será mais alto quando da existência de clientes acessíveis por caminhos lentos, diminuindo o *throughput*. Ele depende também do

tempo que o cliente leva para processar o ACK T(proc_ack_cliente), e do tempo que `accept()` leva para retirar a conexão da fila (T(accept)).

3.3.2 Atendimento da conexão pela aplicação

Como visto, para o estabelecimento de uma conexão TCP, este passa pelas duas filas so_q0 e so_q. `accept()` retira esta conexão da segunda fila. Neste momento é necessário que o servidor invoque um novo processo ou *thread* para tratar desta conexão.

Internamente o que ocorre é a criação de um outro *socket* que servirá de suporte para a comunicação entre o processo filho recém criado e a aplicação cliente, enquanto o processo pai e o *socket* original continuam cuidando da chegada de novas conexões.

O novo *socket* será identificado com o mesmo número da porta do processo pai que se manteve no estado LISTEN. A demultiplexação dos segmentos que chegam é feita através dos quatro componentes que identificam uma conexão: IP origem, porta origem, IP destino e porta destino.

3.3.3 Ataques de negação de serviços e os "syn_cookies" do Linux

Ataques de negação de serviço (*Denial of Service*) exploram basicamente a natureza de implementações da pilha de protocolos sobre um montante de recursos limitado, induzindo o servidor a consumir todos esses recursos, fazendo-o rejeitar conexões legítimas. A implementação das filas de conexão de tamanho finito, abordadas nesta seção, é um caso típico de recurso limitado que é explorado por um dos mais conhecidos e eficientes ataques de negação de serviço, o "syn flooding".

Neste, o atacante tenta inundar a so_q0 de forma que o servidor descarte pedidos de conexão posteriores. Em geral, o atacante envia múltiplos pacotes com *flag* SYN ativada, requisitando a abertura de uma conexão e fornecendo um endereço IP de origem inválido ("IP spoofing"). Assim o servidor responde com o 2º segmento do *three-way-handshake*, mas nunca recebe o retorno (3º segmento), uma vez que o endereço fornecido pelo cliente é inválido. Isso faz com que o pedido de abertura de conexão fique na fila so_q0 até que ocorra o *timeout*, ou que se receba uma mensagem ICMP, indicando que não existe rota para o *host* indicado. Uma forma de

se identificar um ataque de *syn-flooding* é se verificar via netstat, o número de conexões no estado SYN_RECVD.

Soluções paliativas para este ataque são o aumento da fila so_q0 e a diminuição do tempo de *timeout*. Todavia, caso a taxa de envio de conexões inválidas seja mais rápida do que o servidor pode expirar, o ataque ainda terá sucesso.

Uma solução interessante e exclusiva do Linux é a chamada "TCP-Syn-Cookies", que se baseia na escolha particular de um número de seqüência por parte do servidor, calculado à partir do *socket* do cliente, *socket* do servidor e uma semente secreta. Esse segmento é enviado ao cliente como segundo estágio do *three-way-handshake*. Após isso, o servidor libera a conexão, ou seja, não guarda nenhum estado dela na fila so_q0. Quando eventualmente o servidor recebe um segmento correspondente ao 3º estágio *three-way-handshake*, ele valida o valor de reconhecimento e caso esteja correto a conexão entra diretamente para a fila so_q, no estado TCP_ESTABLISHED. Desta forma, o servidor evita manter controle sobre conexões no estado *half-open*, que poderiam representar pedidos inválidos gerados por um ataque de *syn-flooding* [26].

É importante notar que o Linux passa a adotar esta técnica somente após o esgotamento da fila so_q0. Para se habilitar esta funcionalidade no Linux é necessário que o *kernel* esteja compilado com a opção "TCP SynCookie Support" (ver 2.4.1.1 "Configurando o novo kernel", página 31). Além disso, é preciso habilitar este parâmetro via sysctl na variável `/proc/sys/net/ipv4/tcp_syncookies` =1. O default é 0.

3.3.4 Tamanho e Fragmentação de Segmentos

Como visto, um segmento TCP pode ter tamanho variável, cabendo ao TCP definir este tamanho e quando um segmento deve ser enviado, de acordo com diferentes variáveis como por exemplo, a *bufferização* ou ainda *flags* URG e PSH. O tamanho do segmento tem um papel importante no desempenho e é definido no momento da conexão. Quanto menor, maior é o *overhead* causado pelo cabeçalho, além de controles adicionais como *timers* e *ACKs*. Então, o ideal seria enviar o segmento com o maior tamanho possível. Existem 2 limites para o tamanho de um segmento TCP, para que este seja enviado sem que se ocorra a fragmentação, sendo ambos decorrentes dos limites das unidades de transferência das camadas inferiores. O primeiro é o máximo número de bytes que um pacote IP pode enviar, que é 64KB. O segundo, e

determinante por ser o menor, é o MTU (*Maximum Transfer Unit*), o tamanho máximo do pacote, que é específico para cada tipo de rede física (nível de enlace) por onde trafegará o segmento. Assim, o menor MTU do caminho determina o MSS (*Maximum Segment Size*), tamanho máximo do segmento TCP para a conexão, de modo a não ocorrer fragmentação de segmentos. Em redes *Ethernet*, o valor do MTU é 1500 bytes. A figura 3.5 mostra a relação entre o MTU e o MSS.

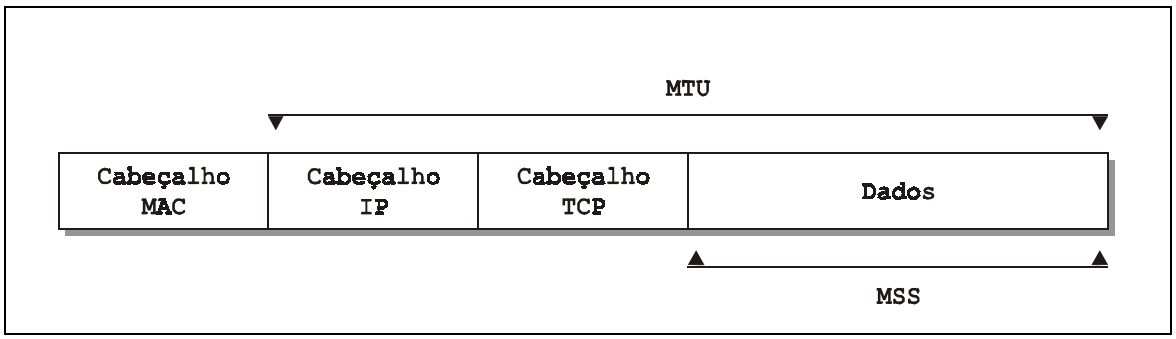

Figura 3.5: Relação entre MTU e MSS.

A fragmentação de segmentos insere, além do *overhead* nos roteadores no nível IP, um problema mais sério de controle de fragmentos. Por exemplo, pode ocorrer de um segmento ser quebrado em vários pedaços, e nem todos atinjam seu destino. Assim deve-se haver um controle para que segmento (como um todo) não seja reconhecido. Outro problema que pode surgir é o fato de um segmento ser enviado, fragmentado, mas nenhum reconhecimento deste segmento chegar ao remetente. Assim, estoura-se o *timer*, e este segmento é reenviado e possivelmente fragmentado de modo diferente do seu 1º envio. Ocasionalmente pedaços de ambos os envios podem chegar esporadicamente ao destino, requerendo um tratamento refinado de tais situações de forma a prover um serviço de comunicação confiável à camada de aplicação

Como já dito, o TCP tem a funcionalidade de negociar o MSS no momento da conexão, quando os *hosts* envolvidos na comunicação enviam, por meio do campo "Opções" (veja figura 3.2), um valor de MSS, baseando-se no MTU suportado pelas interfaces locais. O menor dos dois números seria utilizado para o restante da conexão. Isto permitiria que duas implementações que podem manipular grandes segmentos o façam. Todavia, isto não resolve completamente a situação. O problema mais sério é o fato de nenhum dos *hosts* saber ao certo qual o MTU dos caminhos entre eles, em se tratando de Internet, considerando-se ainda que cada segmento pode

seguir caminhos distintos (soluções para isso envolvem o *Path* MTU *Discovery*, descrito na próxima seção).

Devido ao fato de que a complexidade do processo de remontagem dos segmentos possa implicar em um grande número de implementações com *bugs*, muitos implementadores adotam uma filosofia conservadora de se enviar sempre datagramas com o tamanho máximo (MTU) de 576 bytes. Este é considerado um tamanho seguro, de forma a evitar a fragmentação. O Solaris permite a configuração deste tamanho padrão em `tcp_mss_def`.

3.3.5 Path Maximum Transmission Unit Discovery

A RFC 1191 [91] descreve a *Path* MTU *discovery* que é uma técnica para o descobrimento dinâmico do MTU de um caminho arbitrário da Internet. Essa técnica consiste em uma pequena mudança na forma como os roteadores geram um tipo de mensagem ICMP indicando o MTU de uma dada subrede.

Como visto anteriormente, quando uma conexão é estabelecida, os dois *hosts* envolvidos trocam os valores dos MSS. O menor dos dois valores será o usado na conexão. O MSS para o sistema é usualmente o MTU do *link* menos 40 bytes, sendo 20 bytes do cabeçalho IP e 20 do cabeçalho TCP.

Quando segmentos TCP são enviados, a *flag* DF ("Don't Fragment") é ativada no cabeçalho IP. Qualquer subrede no caminho pode ter um MTU que seja diferente daquele combinado na conexão. Caso seja menor que o combinado, ele teria de ser fragmentado. Como a *flag* DF está ativada, o roteador, não poderá fragmentá-lo e deverá retornar uma mensagem ICMP (*Destination Unreachable*) de volta ao remetente indicando que o datagrama não poderá ser enviado sem a fragmentação. Os roteadores podem enviar juntamente com a mensagem ICMP o MTU por ele suportado, colocando este valor em um campo "não utilizado" da mensagem ICMP. Quando esta mensagem chega, o TCP ajusta o seu MSS para ser o MTU retornado menos o tamanho dos cabeçalhos TCP e IP (em geral 40). Assim os próximos pacotes serão enviados com o tamanho menor que o limitado pelo roteador em questão, evitando a fragmentação até aquele ponto. O MTU mínimo permitido especificado pelas RFCs é 68 bytes. Algumas implementações mais antigas do software de bombeamento de pacotes do roteador podem retornar somente a mensagem ICMP sem incluir o MTU por ele aceito. Nesse caso, o MSS correto deve ser determinado por tentativa e erro.

Pelo fato da Internet ser uma rede de passagem de pacotes sem uma conexão fixa determinada, a rota pela qual os pacotes IP de uma mesma conexão trafegam pela rede pode variar com o tempo. Assim, a RFC 1191 recomenda o descobrimento do MTU para a conexão ativa a cada 10 minutos. Dependendo da implementação utilizada, este valor pode ser configurado de acordo com as características da aplicação e do ambiente em questão. Em uma intranet, por exemplo, onde o pacote poderia circular por poucas subredes, o freqüente redescobrimento de rotas poderia ocasionar um tráfego desnecessário.

Em geral, a maioria das implementações TCP permite a configuração de parâmetros relativos ao *Path* MTU *Discovery* que podem interferir de forma significativa no desempenho.

No Linux o único parâmetro existente é o `/proc/sys/inet/ipv4/ip_no_pmtu_discovery` que desativa o mecanismo de *Path* MTU *Discovery*. Já o Solaris possibilita a configuração do intervalo de redescobrimento do MTU para a conexão via parâmetro `ip_ire_pathmtu_interval`.

3.4 Fechamento de conexão

Se a abertura de conexão demanda a troca de 3 segmentos, o fechamento exige 4: 2 para cada sentido da conexão, uma vez que o fechamento pode ser unilateral.

A figura 3.6 ilustra o processo de fechamento de conexão.

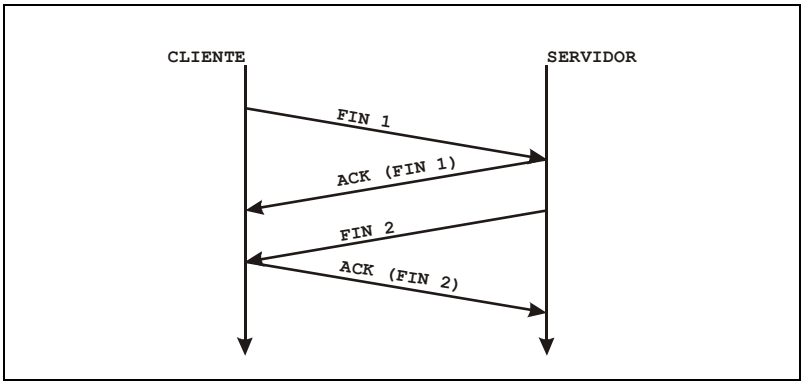

Figura 3.6: Troca de segmentos durante o fechamento de conexão.

Ao contrário do que ocorre na maioria das conexões cliente-servidor [116], no protocolo Web quem faz o *close* ativo é o servidor Web, governado pelos parâmetros de configuração do servidor, que em geral utiliza-se de conexões persistentes, atendendo quantas requisições forem necessárias dentro do limite estipulado no arquivo de configuração.

3.5 Controle do Fluxo de Dados

Após o estabelecimento da conexão passa-se para a fase de transferência dos dados em si. Nesta, o TCP emprega diversas técnicas para controle de congestionamento e estouro de *buffers* dos destinatários como veremos a seguir.

3.5.1 Fluxo de bytes e as *flags* Push e Urgent

Na visão da aplicação, a transferência de dados via conexão TCP é um fluxo de bytes não havendo delimitações. O TCP pode acumular um conjunto de bytes em *buffer* e enviá-los de uma só vez, mesmo que esses bytes sejam oriundos de múltiplas chamadas à primitiva *send*. Assim, sob o ponto de vista da aplicação, não existe o conceito de mensagem ou unidade de envio. Cada byte é simplesmente um byte, cabendo à camada de aplicação determinar o significado de um conjunto deles.

Em algumas situações, a aplicação necessita que os dados sejam enviados imediatamente ao invés de serem *bufferizados*. Por exemplo, no caso de um terminal remoto, quando o usuário termina a digitação de um comando de linha e aguarda pela resposta. Neste momento o TCP não deve aguardar por mais dados para se preencher o *buffer*, mas enviá-lo imediatamente ao servidor para que este possa executar a tarefa. Neste caso a aplicação faz uso da *flag PUSH*, através da primitiva SEND, que indica ao TCP a necessidade de transmissão imediata.

Outra característica TCP em relação ao fluxo de bytes, é a presença dos "dados urgentes". Suponha que se deseje cancelar interativamente um processamento remoto já iniciado (CTRL-C). A aplicação cliente deve enviar informações de controle sinalizados com a *flag URGENT*. Isso instrui ao TCP que deixe de acumular dados, enviando-os imediatamente. Quando o agente TCP do destinatário recebe esta informação, ele interrompe a aplicação indicando a presença de dados urgentes no fluxo TCP. Assim, a aplicação pode interromper seu processamento e identificar o pedido de cancelamento pelo usuário, abortando sua execução.

3.5.2 *Buffers* e Janelas de envio e recepção

Buffers de Recepção

Ao invés de aguardar o reconhecimento de cada segmento, o TCP permite ao remetente enviar novos segmentos, mesmo que ainda não tenha recebido os reconhecimentos referentes aos segmentos anteriores. Isso permite um melhor desempenho na conexão onde o tempo de latência da rede seja muito grande, além de permitir que múltiplos segmentos sejam reconhecidos através de um único ACK.

A quantidade de dados que o receptor está preparado a receber num dado instante é definido pelo tamanho da janela de recepção especificado através do campo "tamanho da janela" na figura 3.2. Este tamanho é definido em *bytes* a cada segmento enviado e previne o estouro do *buffer* do destinatário. Quando um tamanho de janela é divulgado como "zero", o remetente cessa o envio de dados até que receba um novo segmento indicando um tamanho de janela maior que 0 (*window update*). Embora o *buffer* esteja cheio, pode ocorrer a situação de todos os segmentos nele contidos já terem sido reconhecidos. Nesta, a camada TCP recebeu os dados com sucesso, porém a aplicação ainda não teve chance de recuperar os dados do *buffer* e transferí-los para seu espaço de memória.

Janela de Envio

Todo dado da aplicação é copiado para o *buffer* de envio do *socket*. Caso tenha um tamanho insuficiente, a aplicação será bloqueada. A partir do *buffer* o TCP irá criar os segmentos cujo tamanho não exceda o MSS.

Somente quando o dado é reconhecido pelo receptor ele pode ser removido do *buffer* de envio. Para conexões ou receptores lentos, isso implica que os dados permanecerão em *buffer* ocupando espaço em memória por um tempo maior.

Esse fato reforça a importância da presença de memória no servidor de alta demanda que em geral trata múltiplas conexões, cada uma demandando *buffers* próprios.

O Linux 2.4 controla o tamanho máximo dos *buffers* de recepção e envio além da quantidade máxima de memória a ser usada pelo TCP, baseado no montante de memória disponível no sistema e através dos parâmetros contidos em `/proc/sys/inet/ipv4`. O arquivo `tcp_wmem` especifica em bytes os tamanhos mínimo, padrão e máximo do *buffer* de envio, `tcp_kmem` os

tamanho do *buffer* de recepção e `tcp_mem` define limiares para a quantidade de memória utilizada pelo TCP [76].

As aplicações podem ainda, controlar o tamanho dos *buffers* até os limites contidos nos arquivos listados anteriormente, através da chamada de sistema `setsockopt()` com os parâmetros `SO_SNDBUF` ou `SO_RCVBUF`.

3.5.3 Tamanho da Janela e o *bandwidth-delay product*

O tamanho ideal da janela de recepção depende da medida chamada *bandwidth-delay product* que é a velocidade da conexão em bytes por segundo multiplicado pelo tempo de *round-trip* (RTT[1]). Esta medida representa a capacidade do "duto" de comunicação.

```
Capacidade(bytes) = largura de banda(bytes/seg) x tempo de round-trip(seg)
```

É possível obter-se alguma estimativa do tempo de *round-trip* através de *ping* –s.

Citando-se um exemplo, caso cliente e servidor estejam numa rede Ethernet a 10 Mbits/s com uma latência de aproximadamente 1 ms, o produto será 1 Kbyte. Assim, a partir deste cálculo, o tamanho padrão de *buffer* de 8 Kbytes é suficiente para LANs onde o RTT permaneça abaixo de 8 ms [36].

Um tamanho menor de *buffer* ocasionaria uma sub-utilização da rede, que ficaria mais tempo ociosa, já que o tamanho limitado da janela interromperia a transmissão de dados até que anteriores tenham sido reconhecidos.

1. RTT (*Round Trip Time*) é o tempo que se leva para enviar um segmento e receber seu reconhecimento.

As figuras 3.7 e 3.8 [116] representam graficamente a forma como a largura de banda e o RTT influenciam na capacidade do duto. Em 3.7, duplicando o RTT (comprimento do duto) duplica-se a capacidade do duto que pode abrigar 8 segmentos ao invés de 4.

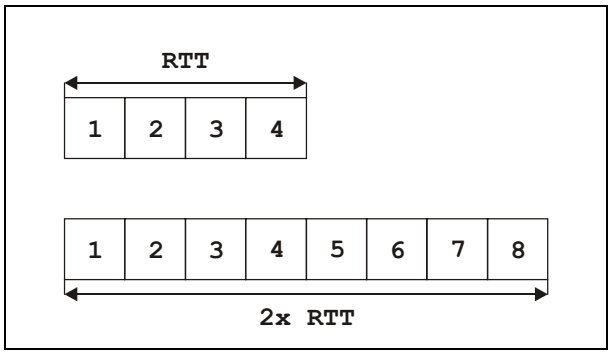

Figura 3.7: Duplicando-se o RTT, duplica-se a capacidade do duto.

Igualmente, duplicando-se a largura de banda (diâmetro do duto), duplica-se a sua capacidade. Assim pode-se enviar 4 segmentos na metade do tempo.

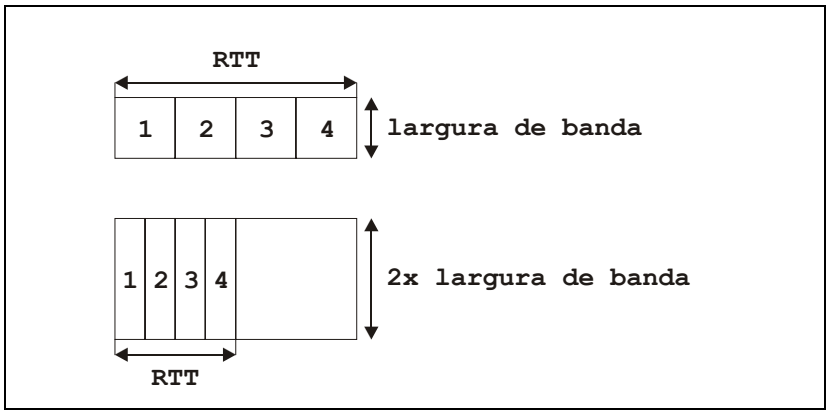

Figura 3.8: Duplicando-se a largura de banda, duplica-se a capacidade do duto.

Desta forma, o objetivo é manter o duto de comunicação cheio a maior parte do tempo, dimensionando a janela de recepção adequadamente.

Deve-se lembrar ainda, que o TCP possui o limite de 64KB (2^{16}) para o tamanho de janela. Para se manipular janelas maiores deve-se fazer uso da opção Window-scale (3.7.1 "Opção Window-scale", página 113).

3.5.4 *Slow – Start*

Embora o servidor conheça o espaço disponível no *buffer* de recepção do cliente, no início da conexão, não é claro qual a largura de banda disponível ou qual é o tempo de *round trip* (RTT) na conexão entre o cliente e o servidor. Caso a rede já esteja congestionada, o envio de dados na máxima capacidade somente ocasionará mais congestionamento. Desta forma, o mecanismo de *slow-start* é implementado para se determinar qual a taxa ideal de envio de segmentos sem o recebimento de reconhecimento baseando-se na carga atual da rede.

Essa quantidade de segmentos enviados sem reconhecimento é limitada pela "janela de congestionamento". O padrão TCP especifica que o tamanho inicial desta janela deve ser um segmento e ter seu tamanho duplicado a cada reconhecimento recebido até o limite da janela de recepção do cliente.

Pelo fato de que, em geral, objetos transferidos pelo HTTP possuem um tamanho pequeno, o *slow-start* irá penalizar este tipo de transferência, pois quando a taxa de transferência estiver próxima ou atingir um ponto ótimo, baseado na largura de banda disponível, faltarão poucos ou mesmo nenhum segmento a ser transmitido. Em outras palavras, grande parte da transferência é feita subutilizando-se a largura de banda disponível.

Cockcroft [36] sugere a configuração do sistema do servidor para iniciar a janela de congestionamento com tamanho 2, devido a um *bug* existente em implementações clientes BSD e Windows, que consideram o estabelecimento de conexão como o primeiro passo do *slow-start*. Sendo assim, após a conexão, clientes destas plataformas só enviam o reconhecimento após dois segmentos de dados. Caso o servidor envie um único segmento ao início da conexão, tais clientes não irão reconhecer imediatamente, causando a demora adicional no início da transferência [129] [116].

3.5.5 Algoritmo de prevenção de congestionamento

Como visto, o *slow-start* é uma forma de se iniciar o fluxo de dados da conexão entre o cliente e o servidor. O tamanho da janela de congestionamento cresce exponencialmente até o tamanho da janela de recepção do cliente ou até o limite de capacidade da rede. No segundo caso, o TCP deve manter o fluxo de envio compatível com a capacidade da rede.

O primeiro passo é detectar o congestionamento. Isso é feito através da observação de perda de pacotes. Assume-se que a freqüência de perda por ruído é muito baixa (menos de 1 %). Assim

uma perda de pacote sinaliza congestionamento em um dos trechos da rede entre o cliente e o servidor. Tais perdas são identificadas por *timeouts*.

Pode-se dizer que o *slow-start* e o algoritmo de prevenção de congestionamento são independentes, com objetivos distintos. Porém, sempre quando ocorre o congestionamento, o *slow-start* é reinvocado e na prática são implementados juntos [118].

Para a implementação do controle de congestionamento, utiliza-se da janela de congestionamento (*cwind*), da janela do receptor e um parâmetro adicional: o limiar.

A combinação dos dois algoritmos funciona da seguinte forma:

1. A variável limiar é inicializada com um valor de 64KB e a janela de congestionamento (*cwind*) com o valor 1.

2. O TCP nunca irá enviar mais que a janela divulgada pelo receptor nem mais que *cwind*.

3. *Slow-start* é invocado e opera seu crescimento exponencial até que ocorra um congestionamento, sinalizado por um *timeout* ou a recepção de um ACK duplicado.

4. Na detecção de congestionamento a variável limiar é setada para a metade da menor das janelas (*cwind* ou de recepção). *Cwind* é setada para 1 (início do *slow-start*).

5. A partir de então, a cada novo reconhecimento, de dados, aumenta-se *cwind*, porém de duas formas: caso *cwind* seja menor ou igual ao limiar, utiliza-se o *slow-start* (crescimento exponencial). Caso contrário, utiliza-se o crescimento linear (prevenção de congestionamento).

A figura 3.9 ilustra o processo [124].

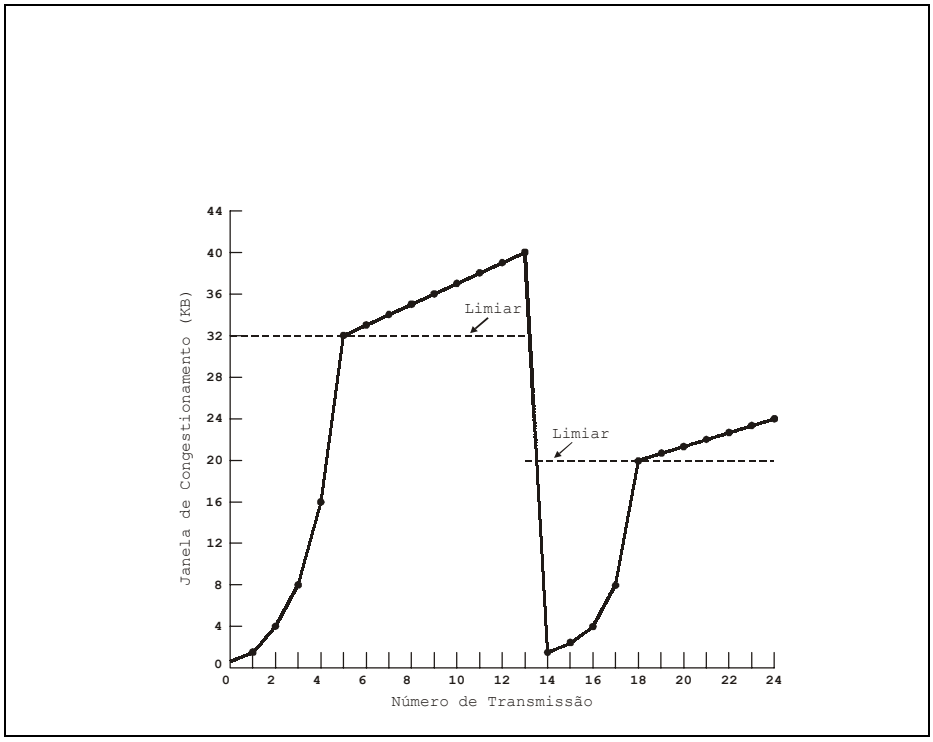

Figura 3.9: Visualização do slow-start e prevenção a congestionamento.

No exemplo, inicialmente assume-se a variável limiar com tamanho de 32 KB decorrente de um prévio *timeout*. Nota-se que *cwind* cresce exponencialmente até o limiar 32 KB (*slow-start*). Após isso, ocorre o crescimento linear (prevenção à congestionamento).

Na transmissão 13 ocorre um *timeout*. *Cwind* é setado para 1 (*slow-start* recomeça) e limiar é setado para 20KB (metade de *cwind*, assumindo-se que a janela de recepção seja maior).

O processo repete-se caso não ocorra mais *timeouts, cwind* irá crescer até o limite da janela de recepção do cliente, permanecendo com este tamanho até que ocorra a atualização da janela de recepção.

Fast retransmit e *fast recovery*

Fast retransmit e *fast recovery* foram modificações propostas para a melhoria de desempenho no algoritmo de prevenção a congestionamento anteriormente descrito.

Na chegada de um segmento cujo número de seqüência esteja fora ordem, o receptor deve enviar imediatamente um ACK duplicado (que já havia sido enviado referente ao segmento

anterior ao esperado), cuja função é informar esta situação do remetente, além de indicar qual o número de seqüência esperado.

Um segmento fora de ordem pode significar duas coisas: uma simples questão de atraso de um dos segmentos (talvez por ter seguindo por um caminho distinto dos outros no roteamento) ou a perda de um segmento. Para se distinguir uma situação da outra, observa-se o número de ACKs duplicados que são enviados. Em geral, para o primeiro caso, apenas um ou dois ACK's duplicados são tolerados. A partir do terceiro, assume-se como um segmento perdido.

O *fast retransmit* consiste no reenvio do pacote indicado pelo ACK duplicado na chegada do 3º ACK antes mesmo da ocorrência do *timeout*. Mudanças também são aplicadas à *cwind* e limiar. Limiar é setada para a metade de uma das janelas como no algoritmo anterior. Porém para *cwind* é atribuído o valor de limiar mais 3 vezes o tamanho do segmento (e não 1, como no algoritmo original). Ou seja, é aplicado prevenção à congestionamento e não *slow-start*. Isso é conhecido como *fast recovery*.

No Solaris a variável `tcp_dupack_fast_retransmit` indica a partir de quantos ACKs duplicados deve-se aplicar o fast retransmit.

SACK (*Selective Acknowledgment*)

Devido ao limitado conjunto de informações provido pelo campo ACK, o remetente só pode saber sobre um único pacote perdido por RTT. Um remetente agressivo pode optar por retransmitir todos os segmentos desde o indicado como perdido, dos quais muitos já podem ter chegado ao destino com sucesso.

O mecanismo de reconhecimento seletivo combinado com uma política de transmissão seletiva visa eliminar este problema. O receptor pode enviar pacotes SACK informando ao remetente sobre quais segmentos foram recebidos. O remetente pode assim transmitir somente os segmentos faltantes. Informações sobre a implementação do SACK podem ser obtidas pelas RFCs 1072 [70] e 2018 [85].

No Linux esse mecanismo é habilitado a partir de `/proc/sys/net/ipv4/tcp_sack`. Clientes como o Windows 98 possuem essa opção habilitada por default [58], sendo importante desta forma, o uso pelo servidor.

FACK (*Forward Acknowledgment*)

FACK é um aprimoramento do SACK com uma mecanismo de recuperação mais rápido. FACK utiliza-se de opções do SACK para colher informações adicionais sobre o estado do congestionamento, adicionando um controle mais preciso para a injeção de dados na rede durante a recuperação de segmentos perdidos. FACK separa os algoritmos de controle de congestionamento dos algoritmos de recuperação para prover uma forma mais simples e direta de utilizar o SACK, melhorando o controle de congestionamento [86] [95].

No Linux, o FACK é habilitado pela *flag* `/proc/sys/net/ipv4/tcp_fack.`

3.6 Controle de *Timers*

O TCP utiliza-se de múltiplos *timers* para cumprir suas funções. Para cada conexão são mantidos 4 *timers,* descritos nesta seção.

3.6.1 *Timer* de Retransmissão

O *timer* de restransmissão é o mais importante e tem a função de detectar a perda de um segmento. O *timer* é iniciado no momento do envio de um segmento. Caso seu reconhecimento chegue antes da expiração do *timer*, este simplesmente é ignorado. Caso contrário, o segmento deve ser reenviado.

A maior dificuldade na implementação de *timers* de retransmissão é como se determinar qual deve ser o período de expiração. Na camada de transporte da Internet, o tempo de *round-trip* (RTT) é bastante imprevisível se comparado à camada de enlace, pelo maior percurso e conseqüentemente maior possibilidade de congestionamento.

Caso se atribua um tempo muito curto de *timeout,* retransmissões desnecessárias podem ocorrer, inundando a Internet com pacotes redundantes. Por outro lado, um tempo muito longo pode ocasionar a degradação de desempenho quando da necessidade de retransmissão (como visto, o *fast retransmit,* tenta amenizar este problema).

A situação torna-se mais crítica quando se considera que um servidor trata múltiplas conexões com clientes de RTT distintos.

Para resolver este problema, o TCP adota um algoritmo dinâmico que constantemente ajusta o tempo de *timeout* baseando-se em testes de desempenho da rede. Estimativas, médias e desvios de RTT são calculados. [116] fornece maiores detalhes da implementação.

Juntamente com o cálculo dinâmico do RTT, aplica-se o chamado "*exponential backoff*" que exponencialmente aumenta o tempo de *timeout* a cada expiração consecutiva.

O tempo total de *timeout* para se abortar uma conexão, em geral, não é alterável pelo administrador. Solaris é uma exceção através da variável `tcp_ip_abort_interval.`

3.6.2 *Timer* de Persistência

O *timer* de persistência (*persistence timer*) tem a função de evitar a seguinte situação de *deadlock*: o receptor envia um reconhecimento juntamente com uma divulgação de tamanho de janela igual a zero. Ou seja, diz ao remetente que seus *buffers* estão cheios, e portanto é necessário interromper o envio de dados. Eventualmente, quando a aplicação lê dos *buffers* liberando espaço para a chegada de mais informação, o receptor envia um segmento do tipo *window update* informando o novo valor de janela, liberando-se novamente o fluxo de dados. Caso esse segmento seja perdido, ambos, remetente e receptor ficam bloqueados, um aguardando uma ação do outro.

O *timer* de persistência é então implementado no remetente. Ele é iniciado quando uma divulgação de janela de tamanho zero chega. Quando se esgota, o remetente envia um segmento especial de *window probe* indagando sobre o atual tamanho de janela. Caso seja ainda zero, o processo se repete. Caso seja maior que zero, dados podem ser enviados.

3.6.3 *Keepalive timer*

Este *timer* tem a função de identificar conexões perdidas. Quando uma conexão está ociosa por muito tempo o *keepalive timer* esgota-se, fazendo com que um dos lados da conexão verifique se o outro lado ainda responde. Caso não obtenha resposta a conexão é finalizada. Isto é útil para que conexões perdidas não permaneçam no servidor consumindo recursos. Para servidores Web este parâmetro torna-se significante quando um cliente aborta a conexão (sem o conhecimento do servidor) após ter enviado o pedido do objeto HTTP [36].

No Solaris este parâmetro é setado em `tcp_keepalive_interval`. No Linux `tcp_keepalive_interval` determina a freqüência de envios de segmentos de verificação.

`tcp_keepalive_probes` especifica quantas tentativas são feitas antes que a conexão seja abortada. Para o primeiro parâmetro, o valor padrão é 75s e para o segundo 9 tentativas. Neste caso a conexão só será abortada depois de aproximadamente 11 minutos [76].

Esta opção no TCP não está relacionada com o *timer keepalive* do HTTP (conexões persistentes).

Caso o comando netstat mostre um grande número de conexões no estado FIN-WAIT 2, significa que clientes não estão fechando a conexão corretamente. Pode-se amenizar este problema reduzindo-se o intervalo de *keepalive* [74].

3.6.4 *Close-Wait Interval Timer*

Quando uma conexão é fechada pelo servidor, ela é mantida em memória por alguns minutos para que se possa identificar possíveis pacotes atrasados correspondentes à conexão. O padrão TCP define este intervalo como duas vezes o máximo tempo de vida de um segmento TCP (2MSL), porém nem todas as implementações usam o mesmo valor. É comum 120 ou 240 segundos no Solaris. Quanto menor esse *timer*, mais rápido recursos estarão novamente disponíveis.

Em situações de grande carga, a tendência é se acumular conexões TCP, tornando a localização na estrutura de dados do *kernel* ineficiente, degradando o desempenho. Uma forma de se amenizar este problema é diminuir o tempo de *close-wait*. Cockcroft [36] aconselha-se setar este tempo para 60 segundos. Outra providência é aumentar-se o tamanho da tabela *hash* de conexões de forma que consultas sejam mais eficientes. Aconselha-se para esta variável o valor 8192.

No Solaris o tempo de *close-wait* é setado pelo parâmetro `tcp_close_wait_interval` e o tamanho da tabela *hash* é configurável pela variável `tcp_conn_hash_size`. No Linux `tcp_fin_timeou`t configura este *timer*.

3.7 Extensões do TCP para Redes rápidas

O TCP foi inicialmente concebido para operar sobre redes não muito rápidas e com um *delay* curto, tendo a Ethernet como padrão por muitos anos. À medida que redes mais rápidas surgiram como a Fast Ethernet, FDDI, dentre outras, operando sob um *delay* alto, certos limites do

protocolo TCP foram alcançados. Para se dar suporte a estes ambientes, extensões ao TCP foram propostas na RFC 1323 e são descritas nesta seção.

3.7.1 Opção *Window-scale*

Em redes que possuem um grande *bandwidth-delay product,* como conexões via satélite, o tamanho de janela máximo do TCP padrão (2^{16} = 64KB) pode não ser suficiente e limitar a taxa de transferência da conexão. Para contornar este problema surge opção *window scale* que aumenta a definição da janela TCP de 16 para 32 bits. O cabeçalho padrão não é alterado, de forma a manter a compatibilidade com implementações anteriores, e a opção para 32 bits é setada através do campo "Opções" no cabeçalho TCP. Esta opção é especificada no momento do estabelecimento da conexão.

No Solaris os parâmetros `tcp_always` e `tcp_cwnd_max` governam esta funcionalidade. Caso o primeiro seja diferente de zero, esta funcionalidade sempre será negociada durante a inicialização da conexão. Caso contrário, *window scale* será utilizado somente quando a segunda variável for maior que 64KB [36].

No Linux `/proc/sys/net/ipv4/tcp_window_scale` habilita o *window scale*.

3.7.2 Opção *Timestamp*

Em redes rápidas surge também a necessidade de uma melhor estimativa do RTT. No TCP convencional, em geral é feita uma estimativa de medida para cada janela, o que é razoável para janelas pequenas. Janelas maiores necessitam de uma melhor medição do RTT. Isto é conseguido através da opção *timestamp*, que permite que o receptor calcule o RTT para cada reconhecimento.

Da mesma forma que o *window scale,* esta opção deve ser especificada no início da conexão e se utiliza do campo "Opções". *Timestamp* não requer nenhuma forma de sincronização de relógio entre os *hosts* envolvidos.

Esta opção é habilitada pela variável `tcp_tstamp_if_wscale` no Solaris como adição ao *window scale*. O Linux configura esta opção através de `tcp_timestamps`.

3.7.3 PAWS (*Protection Against Wrapped Sequence Numbers*)

Da mesma forma que o tamanho máximo da janela TCP é limitado para conexões rápidas, limitado também é o espaço de números seqüenciais que identificam os dados que trafegam em um sentido da conexão. Assim, após um tempo determinado pela largura de banda, um número seqüencial deverá ser reutilizado, o que poderá causar conflitos no sistema de reconhecimento, caso isso ocorra antes de se expirar o tempo de vida do segmento (MSL).

Em uma rede Ethernet (10 Mbits/seg) o tempo para que ocorra o reuso de um número seqüencial é de 60 minutos, ou seja, ainda maior que o MSL. Todavia, a medida em que a largura de banda aumenta esse tempo diminui. Numa rede FDDI ou *Fast Ethernet* (100 Mbits/seg) o tempo é de 5 minutos e em uma rede gigabit (1000 Mbits/Seg) em 34 segundos. Nota-se que este problema está relacionado apenas com a largura de banda e não com o *bandwidth-delay product*.

O algoritmo PAWS evita este problema utilizando-se a opção *timestamp* como uma extensão do número seqüencial. PAWS também não requer sincronização entre os relógios dos *hosts*. A única condição imposta é que os valores de *timestamp* sejam incrementados por pelo menos uma unidade por janela [116].

3.8 T/TCP - *Transaction TCP*

O TCP baseia-se em 3 fases distintas para o ciclo de vida de uma conexão: estabelecimento, transferência de dados e término. Para aplicações longas como *login* remoto ou transferência de arquivos, esta é uma arquitetura eficiente. Todavia, existem aplicações que são projetadas para usar um serviço baseado em transações. Transações são requisições do cliente seguidas de respostas do servidor, onde o *overhead* causado pelo processo de estabelecimento e finalização da conexão deve ser evitado.

Neste contexto, o UDP poderia surgir como uma solução para estas aplicações. Entretanto, ele não provê qualidades desejáveis como cálculo de *timeout* dinâmico, retransmissão, controle de congestionamento, dentre outras.

Desta forma, o T/TCP foi definido na RCP 1379 [23]. As principais mudanças no T/TCP são a supressão do *three-way-handshake* e dos quatro segmentos necessários para o fechamento da conexão (evita-se assim o 2MSL).

Embora o T/TCP seja proposto como uma extensão do TCP, permitindo compatibilidade com implementações já existentes, este protocolo não teve grande aceitação, sendo pouco utilizado. Maiores detalhes sobre o T/TCP podem ser obtidos em [118].

Uma alternativa ao T/TCP é o VMTP (*Versatile Message Transaction Protocol*). Ao contrário do T/TCP, é um protocolo de transporte completo que usa o IP. A RFC 1045 [32] descreve o VMTP.

Capítulo 4

O HTTP

4.1 Histórico

A Web é sem dúvidas a aplicação mais difundida e a principal responsável pela popularização da Internet fora do mundo acadêmico e de pesquisa. Foi criada no Instituto de Pesquisa Nucleares CERN na Suíça, por Tim Berners-Lee [16] em 1989, com o objetivo inicial de ser um mecanismo pelo qual os cientistas do CERN, espalhados pelo mundo, pudessem compartilhar artigos, relatórios, fotos e outros documentos. O protocolo tornou-se popular e em 1993 surgiu a público o Mosaic, a primeira interface gráfica que se tornou tão popular que o autor Marc Andressen deixou o NCSA (*National Center of SuperComputer Applications*), onde o Mosaic foi desenvolvido, para fundar a Netscape.

Em 1994, o CERN e o MIT assinaram um acordo fundando o World Wide Web Consortium (W3C), tendo Tim Berners-Lee como diretor, com a função de dar continuidade ao desenvolvimento da Web e de protocolos de padronização. Desde então, centenas de universidades e companhias se juntaram ao consorcio. http://www.w3.org é o *site* onde podem ser encontradas as informações mais recentes sobre o desenvolvimento da Web.

4.2 O protocolo HTTP

O protocolo central da Web é o HTTP (*HyperText Transfer Protocol*) que atua na camada de aplicação. Na porção cliente é implementado nos navegadores, sendo os principais representantes o Netscape e o Internet Explorer. No lado do servidor é implementado através de

servidores Web como o Apache, IIS da Microsoft e o Netscape (iPlanet). O HTTP define a estrutura das mensagens trocadas entre cliente e servidor e como essa troca é feita [111].

O HTTP foi projetado para ser simples, extensível e servir páginas estáticas, o que facilitou sua implementação e popularização. Porém essas foram as origens de sua ineficiência para o processamento de transações. Graças à implementação de tipos MIME no cabeçalho do documento retornado pelo servidor, o cliente sabe como tratá-lo, possibilitando ao HTTP transportar qualquer tipo de documento, sendo o HTML o principal deles.

O HTML (*HyperText Markup Language*) especifica uma Web Page, que é constituída de objetos. Cada objeto é simplesmente um arquivo, tal como o próprio arquivo HTML, figuras JPG ou um *applet* Java, dentre outros, que são referenciados por um único endereço. Isso diferencia o conceito de *page views* e *HTTP hits*. O primeiro retrata a operação sob o ponto de vista do usuário, onde a página é uma entidade única e o *download* uma ação unificada. O segundo especifica a operação sob o ponto de vista do servidor, que ignora o conceito de conjunto entre os objetos. Nada passa de uma série de requisições por arquivos distintos [74]. Cada *HTTP hit* corresponde a uma operação HTTP, e serve como uma unidade de medida de desempenho.

A versão 1.0 do HTTP está documentada na RFC 1945 [17] e foi padrão nos *browsers* e servidores até 1997. A nova versão 1.1 incorpora mudanças significativas melhorando o desempenho, e está especificada na RFC 2068 [18].

HTTP 1.1 é compatível com 1.0, ou seja, clientes e servidores podem trocar informações mesmo implementando versões distintas do HTTP.

4.3 Características

4.3.1 Sem Estados

O HTTP não possui estados, ou seja, o protocolo não reserva nenhuma porção de memória sobre o que o cliente ou servidor fizeram no passado. Isso facilita a implementação e faz com que seja um protocolo escalável, uma vez que recursos não são gastos armazenando-se estados. Todavia, prejudica aplicações que necessitam de transações, tais como as de comércio eletrônico. Nestas,

o conceito de transação deve ser mantido pela aplicação por meio de *cookies*[1] [75] ou outros mecanismos.

4.3.2 Assimétrico

A transferência de dados HTTP é bastante assimétrica: o fluxo de dados no sentido do servidor para o cliente é bem maior do que o fluxo no sentido inverso. Assim, em relação à quantidade de dados, o gargalo tende-se a localizar mais na saída do que na entrada do servidor.

4.3.3 Baseado em Texto

Assim como o SMTP, o HTTP é um protocolo baseado em trocas de mensagens em formato texto. Desta forma é fácil interagir com um servidor Web via telnet, conectando-se à porta 80 :

```
$telnet w3.convest.unicamp.br 80 | more
 Connected to w3.convest.unicamp.br.
 Escape character is '^]'.
 GET / HTTP/1.0
HTTP/1.1 200 OK
Date: Thu, 11 Oct 2001 16:12:26 GMT
Server: Apache/1.3.19 (Unix)  (Red-Hat/Linux)
Last-Modified: Fri, 13 Jul 2001 20:02:31 GMT
Accept-Ranges: bytes
Content-Length: 270
Connection: close
Content-Type: text/html
<TITLE> Comiss&atilde;o Permanente para os Vestibulares </TITLE>
<FRAMESET ROWS="*, 30">
(...)
```

No exemplo, digitando-se o comando "GET / HTTP/1.0", obtém-se a resposta do servidor. Isso é útil para se averiguar versões de servidores, além de se diagnosticar problemas.

4.4 Formato das mensagens HTTP

As especificações do HTTP 1.0 [111] e do HTTP 1.1 [17] definem o formato das mensagens HTTP, que são basicamente de dois tipos: de requisição e de resposta.

1. O conceito de *cookies* insere um grande debate sobre a questão de privacidade. *Cookies Central* (http://www.cookiecentral.com) dá maiores detalhes.

4.4.1 Mensagens de Requisição HTTP

Abaixo um típico exemplo de mensagem de requisição HTTP:

```
GET /provascomentadas/quest2001.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
Accept-Language: pt-br
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 95; DigExt)
Connection: Keep-Alive
```

Como já referido, a mensagem é descrita em formato ASCII e foi capturada via ferramenta *snoop* a partir da comunicação entre o *browser* e o servidor Web. Cada uma das linhas é seguida por um CR (*Carriage Return*) e um LF (*Line Feed*). Dependendo da requisição que pode envolver navegadores, opções ou versões de protocolos distintos, o número de linhas pode ser diferente. A figura 4.1 ilustra o formato de uma mensagem de requisição.

A primeira linha é chamada de *linha de requisição* e as subseqüentes de linhas *do cabeçalho*.

4.4.1.1 Linha de Requisição

A linha de requisição é composta por 3 campos:

- **Método**: Especifica o tipo de operação. Pode assumir diferentes valores, incluindo-se GET, POST e HEAD. Grande maioria das requisições HTTP utiliza o GET, que indica o pedido de um objeto do servidor.
- **URL**: Especifica qual o objeto requisitado pelo GET. Uma vez que a conexão TCP já está estabelecida com o servidor, não é necessário explicitar o nome do *host* neste campo.
- **Versão**: Especifica qual versão do protocolo HTTP está sendo utilizada.

4.4.1.2 Cabeçalho

O cabeçalho é composto por uma série de opções, separadas em linhas, que se encontram no formato *Opção:Valor*. No exemplo são especificados 5 opções que caracterizam esta requisição. Existe um grande conjunto de opções especificadas na documentação do HTTP 1.0 e 1.1 [111] [17].

A primeira opção, *Accept* indica ao servidor quais são os tipos de objetos que o cliente está apto a visualizar. A opção *Accept-language* indica que o cliente prefere receber uma versão em língua portuguesa se existir no servidor. Caso contrário, o servidor deve enviar a versão *default*.

A terceira opção, *Accept-Encoding*, informa ao servidor os formatos de compressão que o cliente está apto a manipular. O quarto campo, *User-Agent,* especifica qual o navegador utilizado. Esta informação é útil na situação em que o servidor mantém diferentes versões do objeto, destinados a diferentes navegadores ou versões.

A última opção, *Connection*, com valor *Keep-Alive* especifica o uso de conexão persistente.

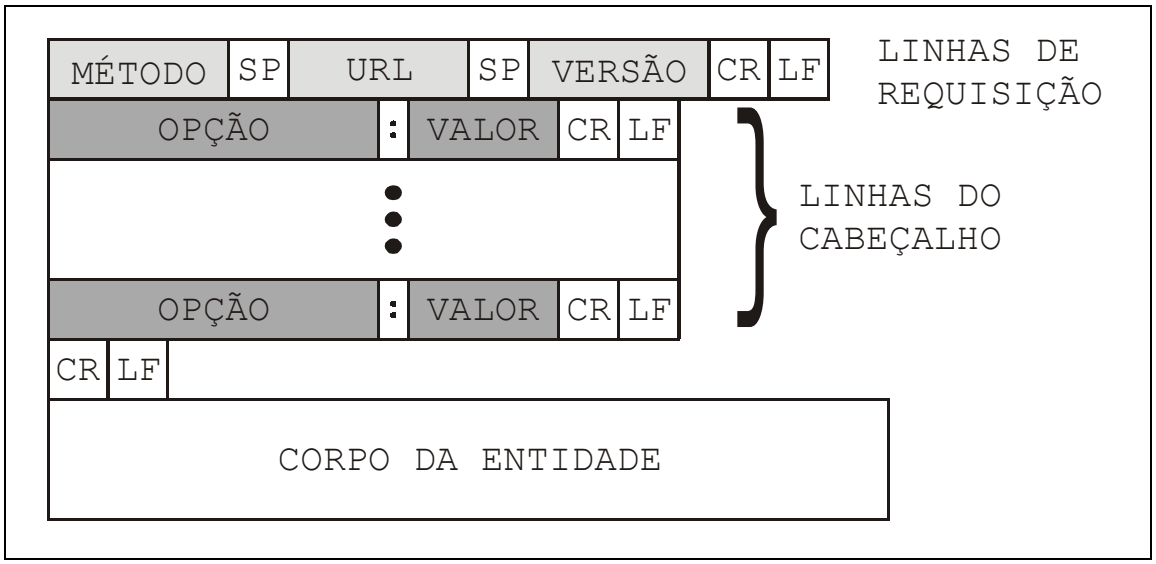

Figura 4.1: Formato Geral de uma mensagem de requisição HTTP.

Na figura 4.1 , o último campo, corpo da entidade, não é utilizado em requisições do tipo GET, mas nas requisições do tipo POST. O método POST é usado em situações onde o cliente especifica parâmetros para a obtenção de uma página, em geral dinâmica, via formulários.

4.4.2 Mensagens de Resposta HTTP

A mensagem de resposta à requisição anterior é a seguinte:

```
HTTP/1.1 200 OK
Date: Thu, 18 Oct 2001 15:39:26 GMT
Server: Apache/1.3.19 (Unix)  (Red-Hat/Linux)
Last-Modified: Thu, 19 Jul 2001 13:14:59 GMT
Accept-Ranges: bytes
Content-Length: 2542
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
(...)
```

Esta mensagem possui basicamente 3 seções: linha de *status*, cabeçalho e corpo da entidade, como visualizado na figura 4.2.

4.4.2.1 Linha de *Status*

A linha de *status* é constituída de 3 campos: a versão do HTTP, um código numérico de *status* e uma mensagem associada a esse código. No exemplo, indica que o servidor utilizar-se-á a versão 1.1 do HTTP e que tudo está OK (o objeto foi encontrado e a transferência está sendo iniciada).

Alguns dos códigos de *status* mais comuns associados às suas respectivas mensagens são:

- **200 OK**: A requisição foi aceita e a informação está sendo enviada
- **301 Moved Permanently**: O objeto requisitado foi movido permanentemente para o endereço especificado na opção *Location* da mesma mensagem. O navegador deverá redirecionar-se automaticamente.
- **304 Not Modified**: Indica que a versão do objeto que se encontra no cache do cliente é a mais recente, não necessitando-se a retransferência (ver 4.6.1 "O GET condicional", página 127).
- **400 Bad Request**: Um código genérico de erro indicando que o servidor não entendeu a requisição do cliente.
- **404 Not Found**: O objeto requerido não se encontra no servidor.

4.4.2.2 Cabeçalho

Da mesma forma que na requisição, o cabeçalho é formado pelas opções e seus respectivos valores. No exemplo, destacam-se, a opção *Date* que indica o dia e hora em que a resposta HTTP foi gerada pelo servidor. Note que este campo não indica a data de criação ou última alteração do objeto. Isto é função do *Last-Modified,* campo crítico para controle de caches, que afeta consideravelmente o desempenho e é visto com mais detalhes em 4.6.1 "O GET condicional", página 127.

O campo *Server* é análogo ao *User-Agent* da requisição e especifica qual o servidor Web gerou a mensagem. *Accept-Ranges* especifica a unidade para transferência de apenas parte de um objeto (ver 4.6.2 "Range Requests", página 128). *Content-Length* indica o tamanho (em *bytes*) do objeto sendo transferido.

Keep-Alive indica o *timeout* da conexão, ou seja, por quanto tempo a conexão persistente permanecerá aberta enquanto inativa. O campo *Connection* indica o tipo de conexão, no caso persistente, e por último, *Content-type* indica o tipo do objeto sendo enviado. Oficialmente o tipo é indicado por este campo e não pela extensão do nome do arquivo.

A figura 4.2 Ilustra o formato geral de uma mensagem de resposta HTTP [111].

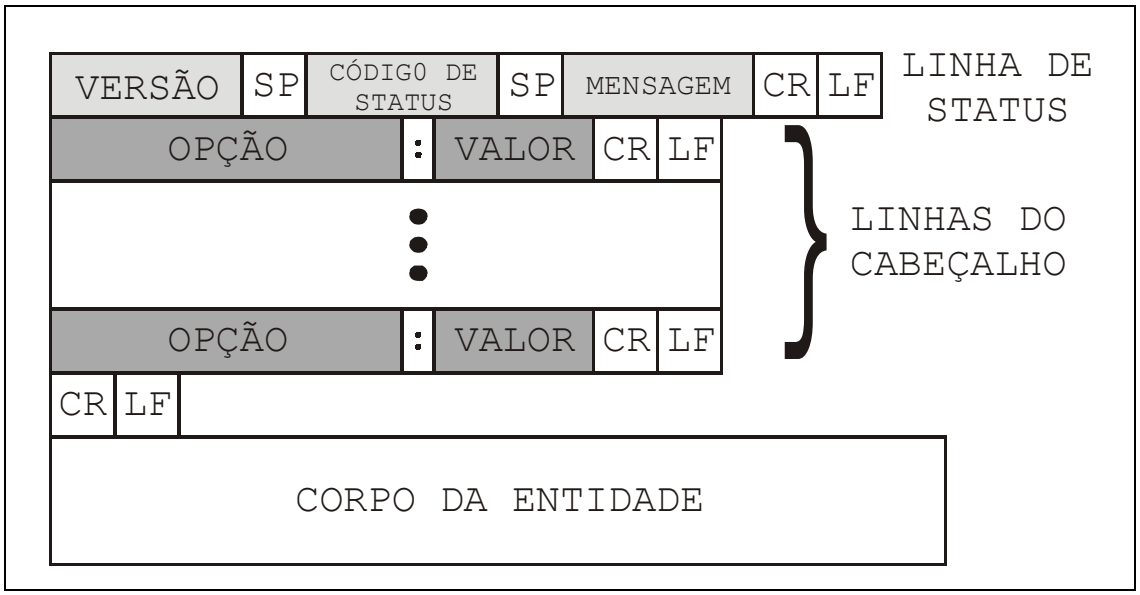

Figura 4.2: Formato geral de uma mensagem de resposta HTTP.

Nesta seção foi coberta apenas uma pequena fração das possíveis opções presentes nos cabeçalhos. A diversidade e quantidade dessas opções dependem da implementação do cliente

e do servidor e de suas respectivas configurações. [17] e [18] apresentam a descrição completa dessas opções.

4.5 Interações com o TCP

O HTTP utiliza o TCP, e não o UDP como se poderia imaginar pelo seu caráter "sem estados". Versões iniciais do HTTP (0.9 e 1.0) interagiam de modo sofrível com o protocolo de transporte TCP, por se utilizar de conexões curtas. A principal vantagem da versão 1.1[1] é a implementação de conexões persistentes, que podem transferir vários objetos sob uma mesma conexão, diminuindo o *overhead* ocasionado pela abertura e manutenção de uma conexão TCP. Detalhes desta interação entre o HTTP e o TCP são exibidas nesta seção.

4.5.1 Conexões não Persistentes

Neste tipo de conexão, para cada objeto a ser transferido de uma página, uma nova conexão TCP é aberta. Supondo que uma página Web seja composta por 10 imagens JPGs e que todas residam em um mesmo servidor, 11 objetos deverão ser transferidos através de 11 conexões TCP distintas.

Para cada conexão deve ser computada o atraso com o ***three-way-handshake*** (ver 3.3.1 "O three-way-handshake", página 93), que insere um limite inferior de 2 RTTs para qualquer transmissão por menor que seja.

Para ilustrar a situação, analisemos o que ocorre em termos de conexão TCP após o usuário clicar sobre um *link* (ver figura 4.3 [79]). Inicialmente o *browser* abre uma conexão TCP com o servidor, o que acarreta a aplicação do *three-way-handshake*. O cliente envia um pequeno segmento com a *flag* SYN ativada, o servidor reconhece este segmento com outra mensagem e o cliente reconhece esta última. Nas primeiras duas interações do *three-way-handshake* um RTT já se passou. No terceiro segmento, o cliente já envia a requisição HTTP, além do último reconhecimento (3ª fase do *three-way-handshake*) através da técnica de *piggy-backing*. Uma vez que este segmento chega ao sevidor, a conexão é estabelecida, um novo *socket* é criado e será, em geral, tratado por uma nova instância do servidor Web que iniciará a transferência dos dados

1. Embora a documentação oficial do HTTP 1.0 não faça referência a conexões persistentes, algumas implementações da versão 1.0 a permitiam através do uso da opção Connection: *keep-alive*.

requisitados. Esta interação, de requisição e resposta HTTP, consome mais um *round-trip*. Só a partir deste instante é que o objeto HTML começará a chegar ao cliente. Após a chegada a conexão é fechada através dos 3 segmentos FIN, ACK + FIN, ACK. Na figura supõem-se que o objeto HTML foi pequeno o bastante para caber em um único segmento, o que em geral não é realidade.

Paralelamente a isso o cliente interpreta o HTML e requisita a abertura de mais uma conexão para a transferência de uma figura JPG. Esta nova conexão requer igualmente o *three-way-handshake*. Na prática, a primeira imagem somente poderá ser exibida após 4 RTTs. O processo se repete para cada figura da página.

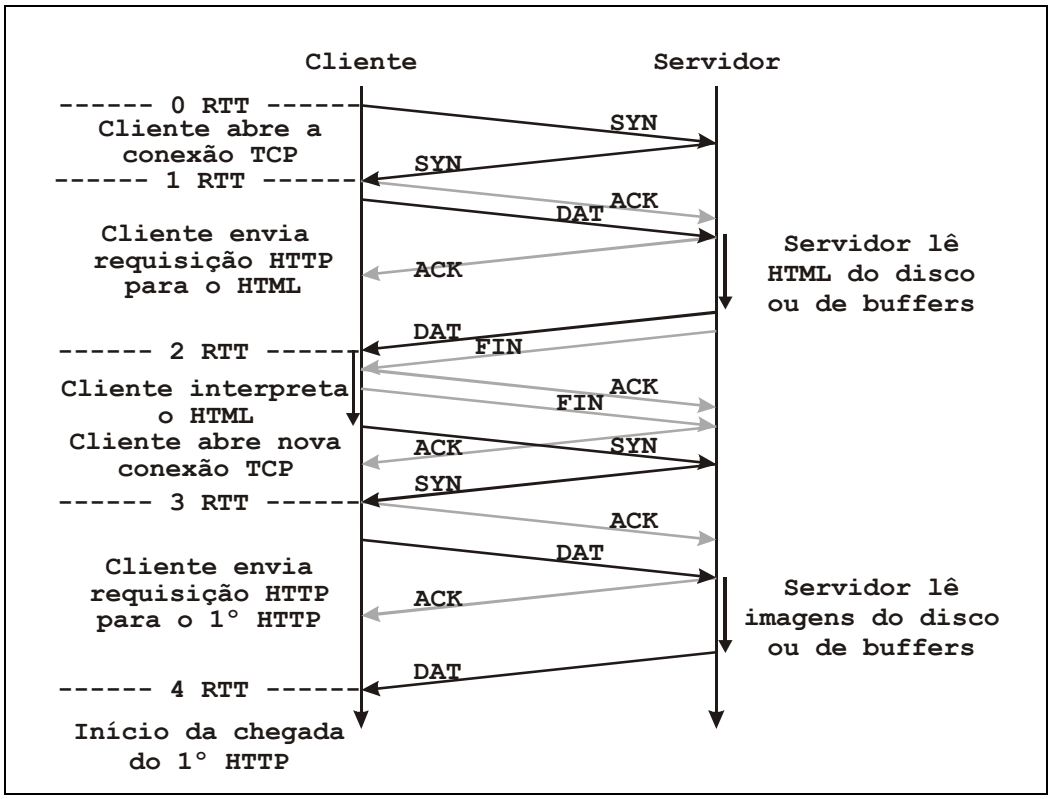

Figura 4.3: Troca de pacotes e RTTs para uma conexão TCP. Linhas claras mostram pacotes requeridos pelo TCP que não afetam a latência da comunicação

Além de ser penalizada pelo processo de *three-way-handshake*, cada conexão será penalizada também pelo processo de ***slow-start*** que estabelece uma taxa limitada de envio para o início de cada transferência (ver 3.5.4 "Slow – Start", página 106).

Em geral, a capacidade de rede será subutilizada, uma vez que o "duto" não se manterá cheio (ver 3.5.3 "Tamanho da Janela e o bandwidth-delay product", página 104).

Navegadores podem obter os 10 JPGs restantes através de conexões TCP abertas paralelamente. Isso diminui o tempo de resposta, amenizando (não haverá acúmulo) mas não eliminando os problemas do *three-way-handshake* e do *slow-start*. Em geral os navegadores possibilitam a configuração desse grau de paralelismo.

A abertura excessiva de conexões (uma para cada objeto) empregada por este método, acarreta também em um consumo excessivo de recursos por parte do servidor. Supondo um servidor que atenda a centenas de clientes, cada um com várias conexões abertas (considere o *close-wait*, ver 3.6.4 "Close-Wait Interval Timer", página 112) ou em fase de abertura (*three-way-handshake*), estruturas como a fila de requisições (ver 3.3.1.1 "Implementação", página 94) e *buffers* podem se esgotar rapidamente. Além disso, o *three-way-handshake*, o *slow-start* e o ***timer de close-wait***, farão com que estas conexões sejam mantidas (consumindo recursos) por mais tempo no servidor.

Um outro ponto negativo, quanto a presença de múltiplas conexões curtas, relaciona-se ao fato de que o cálculo de *timers* de retransmissão torna-se mais preciso à medida que mais amostras são colhidas, ou seja, em conexões mais longas (ver 3.6.1 "Timer de Retransmissão", página 110).

4.5.2 Conexões Persistentes

Através de conexões persistentes, o servidor mantém a conexão aberta após enviar as respostas. Requisições subsequentes entre o mesmo cliente e servidor podem ser feitas através da mesma conexão TCP. Em particular todos os objetos de uma mesma página Web (como no exemplo) podem ser transferidas através de uma mesma conexão. Mais que isso, múltiplas páginas Web podem usar a mesma conexão. Isto diminui drasticamente o *overhead* causado pela abertura de múltiplas requisições.

O *three-way-handshake* será executado apenas uma vez, assim como o *slow-start*. Desta forma, uma vez alcançada a taxa ótima de envio, esta será mantida para os demais objetos. Além disso, o consumo de recursos do cliente e principalmente do servidor será bem menor.

A conexão é fechada pelo servidor após um tempo de inatividade, que em geral é configurável (ver Seção 5.3.4.2).

Este parâmetro também possui significativa importância. Se por um lado, a abertura, sucessiva de novas conexões TCP degradam o desempenho, manter-se uma conexão aberta e ociosa também consome recursos (como *buffers* de *socket* pré-alocados).

O controle deste *timer* em geral é implementado como um parâmetro estático em servidores como o Apache. [37] apresenta políticas que exploram informações embutidas nas mensagens de requisições HTTP, tentando identificar padrões de requisições a objetos, para uma definição dinâmica de tempo de fechamento de conexões persistentes.

4.5.3 Pipelining

Existem duas versões de conexões persistentes: as que não usam *pipelining* e as que usam. Para o primeiro caso o cliente requisita um novo objeto somente quando o anterior foi completamente recebido. Isto insere um atraso de 1 RTT para cada objeto devido à interação de requisição e resposta HTTP. Embora seja um avanço em relação as conexões não persistentes, um melhor aproveitamento pode ser feito através do uso de *pipelining*.

Em conexões persistentes com *pipelining,* o cliente requisita um objeto assim que encontra referência a ele sem precisar esperar o término da requisição anterior. Desta forma, a requisição de um novo objeto pode ser enviado com um segmento que seria enviado de qualquer forma transportando o reconhecimento (mais uma vez, via *piggybacking*) de parte de um objeto anterior. Atrasa-se portanto apenas um RTT para a requisição de todos os objetos.

Embora o *pipelining* seja padrão no HTTP 1.1 nem todos os navegadores o utilizam, como mostra [111]. Além disso, o navegador pode abrir múltiplas conexões persistentes para um mesmo servidor.

4.6 Controle de Cache e Otimização de Banda

4.6.1 O GET condicional

Pelo armazenamento de objetos previamente transferidos, o cliente pode reduzir o tráfego de dados sobre a Internet. Estes objetos podem ser armazenados no cliente ou em uma máquina intermediária para toda uma rede (um servidor *proxy*).

No advento de uma requisição, o cliente analisa se o arquivo já se encontra em cache. Em caso afirmativo, ele envia uma requisição GET com a opção *If-Modified-Since* no cabeçalho,

indicando ao servidor que somente deverá transferir o documento caso exista uma versão mais recente do objeto do que aquela que se encontra em cache (compara-se com a data de transferência, campo *Date*). Caso não exista, o servidor envia uma mensagem de retorno com o *status 304 Not Modified*, sem anexar o conteúdo do objeto. Em caso contrário, envia todo o objeto normalmente, atualizando o cache do cliente.

Além do campo *If-Modified-Since* especificado pelo cliente, uma outra forma de controle de cache parte do servidor, que também pode enviar o campo *Expires* que força o objeto a expirar no cache do cliente depois de um determinado tempo, obrigando-o a recuperar uma nova versão do servidor.

Nota-se que, embora o teor das mensagens seja pequeno no caso de um objeto válido no cache, uma nova conexão deverá ser aberta (caso não existam conexões persistentes referentes a objetos anteriores). Assim uma simples conferência da validade do cache é penalizada pelo processo de conexão TCP.

4.6.2 *Range Requests*

A partir da versão 1.1, tornou-se possível a transferência de apenas parte de um objeto (campo *Accept-Ranges*). Esta opção é útil em duas situações [93]:

- quando se deseja obter apenas a parte inicial da figura que contem sua dimensão, de forma a determinar o *lay-out* da página antes que a figura seja transferida por completo.
- Para completar uma transferência anterior que foi interrompida de alguma forma. Em outras palavras, para converter uma entrada parcial do cache em uma resposta completa, evitando-se assim, retransmissão de dados.

4.6.3 Compressão de Dados

Uma das formas mais conhecidas de se economizar banda passante é o uso de compressão. Embora a maioria dos formatos de imagem como GIF e JPEG já sejam pré-comprimidos, outros tipos de arquivos que circulam via HTTP não são. Estudos feitos em [93] mostram uma economia de até 40 % de *bytes* enviados utilizando-se um método agressivo de compressão.

A versão 1.0 do HTTP inclui um suporte rudimentar para compressão. Na versão 1.1 uma forma mais avançada de negociação é implementada através dos campos *Content-Encoding* e *Transfer-Encoding*, onde se especifica o formato de compressão.

Capítulo 5

Software Servidor Web

O software servidor Web atua na camada de aplicação, sendo responsável pela implementação do protocolo HTTP (descrito no capítulo anterior) na porção servidora.

A um nível mais básico, tem a função de aceitar requisições HTTP e retornar respostas, que podem ser páginas estáticas, conteúdo dinâmico ou mensagens de erro.

Neste capítulo serão abordados aspectos de arquitetura, interação com o Sistema Operacional, parâmetros configuráveis, exemplos de servidores Web, além de detalhes do servidor Apache, atualmente o mais utilizado. Extensões ao servidor Web que possibilitam o suporte às aplicações mais avançadas baseadas em conteúdo dinâmico também são apresentadas.

5.1 Arquitetura de Servidores Web

Servidores Web tradicionais são implementados no espaço do usuário e são atualmente os mais conhecidos e utilizados. Recentemente, tem surgido uma nova tendência de implementação de servidores Web objetivando melhor desempenho: a implementação direta no *kernel*. Nesta seção, são caracterizadas as duas arquiteturas.

5.1.1 Questões de Desempenho na Implementação de Servidores Web

Técnicas para melhoria de desempenho na implementação de servidores Web se focam nos seguintes objetivos: eliminação de cópia e leitura de dados, redução de escalonamento e *overhead* causado pela notificação de eventos e a redução da comunicação entre as várias camadas da pilha de protocolos de rede [72].

5.1.1.1 Cópia e Leitura de Dados

A necessidade de se copiar dados do cache do sistema de arquivos para o espaço de memória do usuário e daí para a interface de rede, técnica adotada por servidores convencionais, é bastante custosa. Uma forma de se evitar isso é a adoção de um mecanismo que copie dados diretamente do cache do sistema de arquivos para a interface de rede. Isto pode ser alcançado a partir dos servidores Web implementados diretamente no *kernel* ou por primitivas especiais do Sistema Operacional, como veremos mais adiante.

5.1.1.2 Notificação de Eventos

É definido como notificação de eventos o enfileiramento de requisições de clientes que aguardam por uma resposta do servidor. No caso de uma requisição HTTP, como visto, isto envolve o tratamento do *three-way-handshake*, a montagem da mensagem de requisição HTTP e sua transferência para o espaço de memória do usuário. Para se manipular múltiplos clientes, o servidor deve suportar concorrência através da distribuição de requisições entre múltiplos processos filhos ou utilizando-se de chamadas de sistemas assíncronas para se manipular várias requisições com um conjunto pequeno de processos. A opção por qual método de concorrência utilizar direciona a forma de implementação e posterior classificação dos servidores implementados no espaço do usuário, como veremos em 5.1.2.

5.1.1.3 Caminho de comunicação dos dados através das camadas de rede

A interface padrão de chamadas de sistemas para transmissão de dados não é otimizada para serviços Web. Assim, versões recentes de Sistemas Operacionais tem criado novas interfaces especificamente para serviços Web, reduzindo o número de chamadas de sistema. Por exemplo,

o Linux implementa chamada de sistema `sendfile()` que combina a leitura de dados do sistema de arquivos e a escrita de dados para o *socket*.

5.1.2 Implementação do Servidor Web no espaço do usuário

Esta arquitetura é a tradicional para a implementação de aplicações em geral. O servidor Web é somente mais uma aplicação que se utiliza de recursos do sistema, com privilégios limitados, através da interface de *system calls*. Desta forma, o servidor Web é completamente dependente da interface fornecida pelo Sistema Operacional, e quão otimizada é implementada essa interface, baseando-se nas questões de desempenho vistas na seção anterior.

Devido à demanda por serviços mais rápidos, tem-se notado a descoberta de formas de melhoria do desempenho e escalabilidade do servidor Web através da maior integração entre servidores Web e Sistemas Operacionais. Esta tendência tem provocado uma busca pela otimização das interfaces e implementações existentes nos Sistemas Operacionais especialmente para atender o serviço Web [81]. Por exemplo, implementações de primitivas como `select()` e `poll()` tem sido otimizadas em Sistemas Unix para se reduzir o *overhead* de notificação de eventos em arquiteturas que se utilizam de chamadas não bloqueantes.

Os servidores implementados no espaço do usuário dividem-se basicamente em quatro grupos, por evolução na implementação: os invocados pelo *inetd*, servidores Web multiprocessos (MP), servidores Web *multithreads* (MT) e servidores Web SPED.

5.1.2.1 Invocados pelo *inetd*

A primeira geração de servidores Web surgiu como mais um serviço invocado sob demanda pelo *inetd*, que na sua inicialização lê o arquivo `/etc/inetd.conf`, aguardando requisições das portas especificadas e invocando um servidor adequado. Esta prática objetiva a conservação de recursos, uma vez que os *daemons* (processos persistentes) servidores somente serão executados quando necessário.

Todavia, este mecanismo só é válido para serviços que são requisitados esporadicamente (o que incluía o servidor Web no seu início). Para serviços de alta demanda, este modelo tem o efeito contrário, pois requer a constante execução de chamadas de sistema: `fork()` para clonar o *inetd* e `exec()` para sobrescrever este processo com o servidor correspondente. Lembrando

que a visualização de uma única página Web pode requerer múltiplas requisições HTTP, isto torna-se desastroso para um servidor Web de grande carga.

Servidores atuais como o Apache, permitem ser configurados para este modo de inicialização. Mais detalhes são encontrados na Seção 5.3.4.2.

5.1.2.2 Servidores Web Multi-Processos (MP)

Um avanço sobre serviços Web que se utilizam do *inetd,* é o uso de uma instância do httpd (*daemon* http) para simplesmente executar um `fork()` (evitando o `exec()`) para o atedimento de uma conexão.

O `fork()` funciona razoavelmente bem para aplicações cliente/servidor, mas pelo fato de conexões Web serem tão curtas e freqüentes, na maioria dos casos, o tempo gasto com a criação de um novo processo e o escalonamento entre múltiplos processos pode ser maior do que o gasto com o atendimento da conexão em si. Além disso o consumo de recursos, principalmente de memória, é bastante significativo (ver 2.5.1.3 "Criação de Processos", página 43). Por esse motivo, servidores que se utilizam de *threads*, ao invés de processos, têm ganhado espaço.

Uma forma de se evitar o atraso com a criação de processos é a adoção de um *pool* de processos criados a priori que ficam aguardando por novas requisições. A desvantagem desta técnica é o consumo permanente de recursos, independente da necessidade.

Na arquitetura de múltiplos processos, um processo é designado para executar todos os passos do atendimento de uma conexão para um cliente seqüencialmente. Uma vez que são utilizados múltiplos processos, múltiplas requisições podem ser atendidas concorrentemente. A intercalação de uso de CPU, disco ou rede ocorre naturalmente, devido ao próprio controle de processos do Sistema Operacional, diferentemente do que ocorre na arquitetura SPED, abordada mais adiante.

Como cada processo possui seu próprio espaço de endereçamento, não é necessária uma sincronização entre as múltiplas requisições.

A arquitetura de MP é ilustrada na figura 5.1.

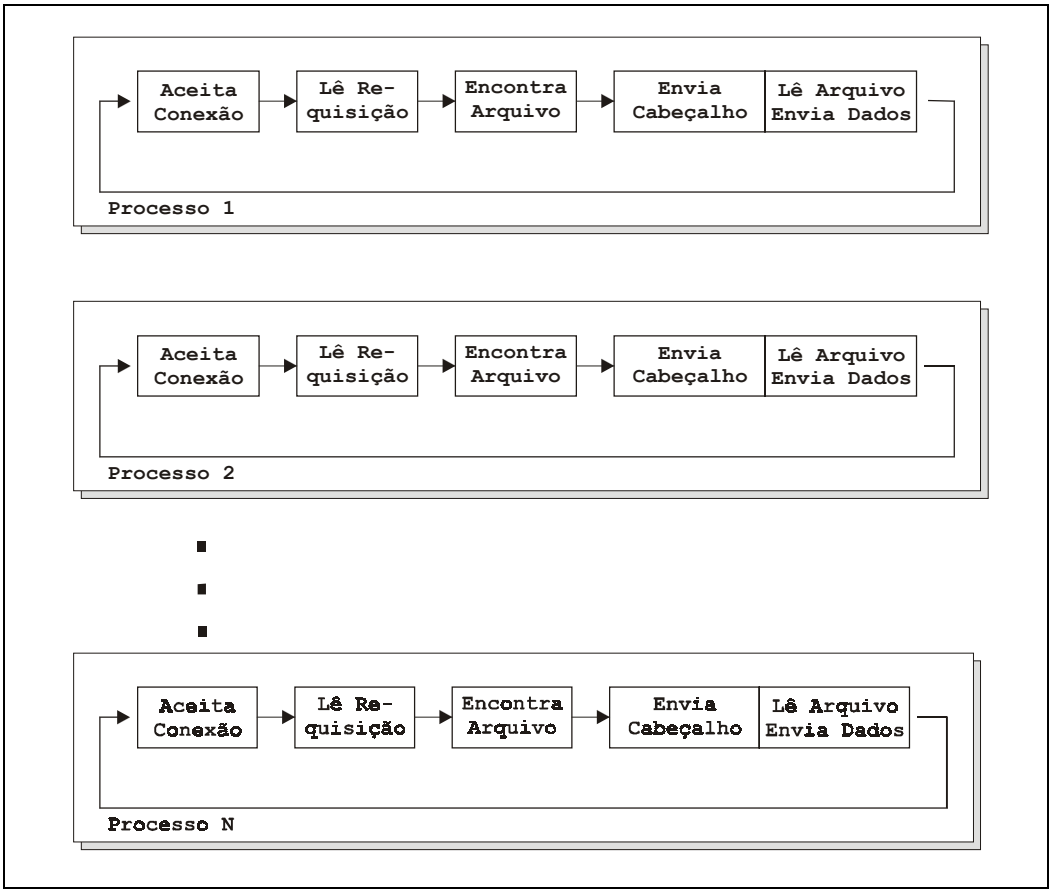

Figura 5.1: No modelo MP cada servidor manipula uma requisição por vez. Processos executam estágios seqüencialmente.

O servidor Apache na versão 1.3 para Unix é um típico exemplo de servidor MP. Servidores MP também são conhecidos como servidores Web *forked*. A grande vantagem da arquitetura MP é a maior portabilidade.

5.1.2.3 Servidores Web *Multi-Threads* (MT)

Para atender a natureza efêmera das conexões HTTP, os servidores Web estão sendo implementados via *threads*. Como visto, a criação e escalonamento de *threads* são ordens de grandeza mais rápidos que os equivalentes de processos. Experimentos realizados em [72] demonstram ganhos de duas ordens de grandeza no tempo de resposta de servidores MT em relação aos servidores MP, principalmente pela melhoria no processo de notificação de eventos.

Servidores MT empregam múltiplas *threads* independentes dentro de um único processo, num único espaço de memória. Cada *thread* executa seqüencialmente todos os passos para o atendimento da requisição de modo semelhante ao MP. A grande diferença é o espaço de memória comum que evita o tempo gasto com alocação e facilita a cooperação/ compartilhamento de dados entre as *threads*. Todavia isso insere também a necessidade de sincronismo para o controle de acesso aos dados compartilhados.

A figura 5.2 ilustra a arquitetura MT.

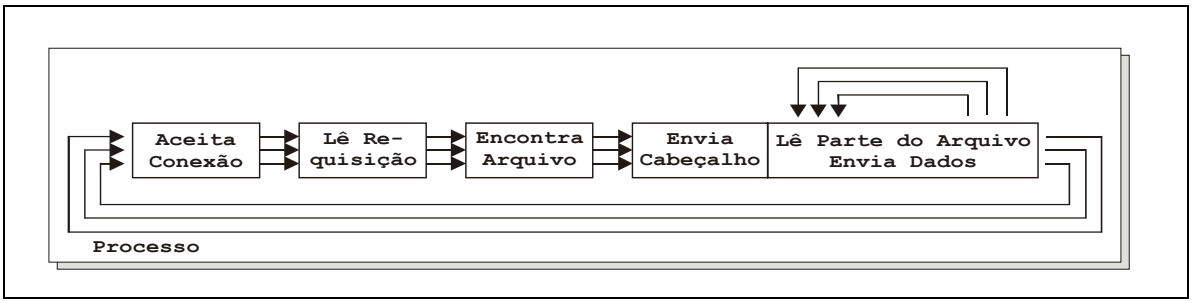

Figura 5.2: O modelo MT utiliza-se de um único espaço de endereçamento e múltiplas *threads* concorrentes de execução, cada uma manipulando uma requisição distinta.

O servidor Netscape utiliza o conceito de *threads*. No mundo Apache para Unix, a versão 2.0 (beta) implementa um modelo híbrido.

Para a plataforma Windows, o conceito de *threads* já é adotado desde as primeiras versões do Apache e do IIS. Este relativo atraso das versões *threaded* para Unix deve-se, em grande parte, à falta de padronização das inúmeras versões de Sistemas Unix [9], como visto em 2.1 "O Unix", página 22.

5.1.2.4 *Single Process Event Driven* (SPED)

A arquitetura SPED utiliza-se de um único processo servidor dirigido por eventos para atender múltiplas requisições HTTP concorrentemente. O servidor faz uso de chamadas de sistemas não bloqueantes para executar operações de I/O assíncronas. Uma operação como o `select` (BSD) ou `poll` (System V) é usada para checar se operações de I/O foram completadas.

Através de chamadas de sistemas não bloqueantes o servidor pode executar cada requisição HTTP em etapas, podendo intercalar etapas de múltiplas requisições, funcionando mais ou menos como um escalonador. Em cada interação ele checa através do `select()` eventos de I/

O completados, como a chegada de novas conexões, finalização de operações de manipulação de arquivos ou requisições de clientes via *buffers* de *sockets* [100].

Um servidor SPED pode intercalar operações de CPU, disco ou redes associadas ao atendimento de múltiplas requisições HTTP concorrentes, utilizando-se de um único processo e um único fluxo de execução, o que elimina o *overhead* com mudança de contexto e sincronização de *threads* existente nas arquiteturas MP e MT.

Um problema que surge é o fato de muitos Sistemas Operacionais não proverem suporte adequado às operações assíncronas de disco. Somente operações assíncronas de rede são suportadas.

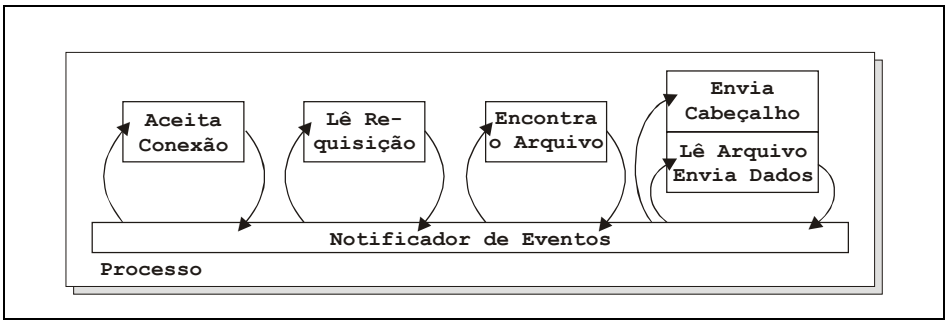

Figura 5.3: O modelo SPED se utiliza de um único processo para executar todo o processamento e atividade de disco baseado em eventos. Múltiplas requisições são manipuladas por um único processo.

A figura 5.3 ilustra a arquitetura de um servidor Web SPED, explicitando os passos executados para o atendimento de uma requisição. O processo de leitura e envio de dados é repetitiva até o envio de todo o arquivo.

Para conteúdos que podem ser cacheados, ou seja, onde toda a hierarquia de páginas pode ser mantida em memória, a arquitetura SPED apresenta um desempenho bastante superior em relação aos modelos MP e MT [100]. Além disso, a arquitetura SPED consome menos memória, uma vez que existe um único processo e uma única pilha. As arquiteturas MT e MP consomem memória adicional e recursos do *kernel* proporcionalmente ao número de requisições simultâneas.

Pelo fato de o atendimento das conexões ser centralizado, a manutenção do cache é facilitado por não se exigir sincronização no acesso ou atualização dos dados, como acontece nas arquiteturas MT e MP.

Exemplos expressivos de servidores que usam esta arquitetura são o Zeus e o IIS.

5.1.2.5 Modelos Híbridos

Processos e *Threads*

Embora o conceito de *threads* apresente nítidas vantagens de desempenho sobre múltiplos processos, a idéia de se manter um único processo gigante contendo centenas de *threads* também não é atraente, principalmente se o servidor estiver carente de memória demandando execução de *swapping* [74].

Assim, a forma mais comum de implementação, utilizada pelo Netscape e o Apache 2.0, é o modelo híbrido, onde se mantém vários processos, cada um contendo múltiplas *threads*.

No Apache existe um servidor principal que é responsável por criar processos filhos de acordo com a carga.

Cada processo filho por sua vez, é responsável pela criação de um número fixo de *threads* cuja função é atender requisições à medida que chegam.

Desta forma, o número máximo de clientes que podem ser atendidos simultaneamente é determinado pela multiplicação do número máximo de processos filhos pelo número de *threads* criadas por cada processo. Os valores máximos dessas duas variáveis são configuráveis, como veremos na Seção 5.3.4.2.

Múltiplos Processos/*Threads* e o SPED (AMPED)

Combinando-se o alto desempenho de servidores SPED para cargas que se baseiam em conteúdo mantido em memória com servidores MP ou MT, que possuem um desempenho mais alto em cargas cujo acesso a disco é mais constante, surge a arquitetura AMPED; acrônimo para *Asymetric Multi-Process Event-Driven*.

Nela o servidor comporta-se como um SPED quando o conteúdo a ser requisitado está em memória. Quando uma operação de disco é necessária (o arquivo requisitado não se encontra em cache) o processo servidor principal invoca um processo ou *thread* auxiliar que executará uma operação de I/O bloqueante. Desta forma, a arquitetura AMPED preserva a eficiência da

arquitetura SPED em operações que não envolvem leitura de disco, ao mesmo tempo que evita problemas de desempenho sofridos pelo SPED devido ao suporte inapropriado às operações assíncronas de disco em muitos Sistemas Operacionais.

A figura 5.4 ilustra o AMPED.

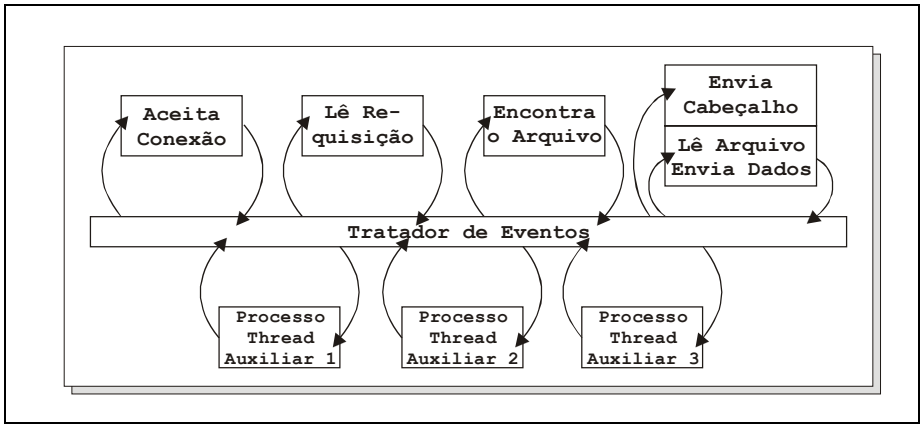

Figura 5.4: O modelo AMPED utiliza-se de um único processo para o processamento orientado à evento, mas vários auxiliares para manipular operações de disco.

O servidor Web Flash apresentado em [100] é uma implementação desta arquitetura. Além do Flash, o TUX também possui arquitetura semelhante, porém diretamente no *kernel*, via *kernel threads* e não *threads* de usuário.

5.1.3 Implementação do Servidor Web no espaço do *kernel*

Na busca por servidores Web mais velozes, tem surgido recentemente uma nova abordagem na implementação de servidores Web: a implementação direta no *kernel*. Nela, integra-se o processamento HTTP à pilha de protocolos TCP/IP de forma a prover a "copia-zero" entre as estruturas internas do *kernel*, evitar cópia de dados entre processos no modo usuário e o *kernel* (o que evita a mudança de contexto entre esses dois modos), além de possibilitar uma notificação de eventos mais eficiente.

Citando-se um exemplo, na implementação do TUX (ver Seção 5.2.7), a interface `accept()` é ignorada. A requisição é movida diretamente da fila do socket para a fila de entrada de requisições da *thread*, tornando o servidor diretamente orientado por eventos da camada de rede.

Respostas completas são armazenadas diretamente no cache de rede do *kernel* para se reduzir o *overhead* associado com operações de cópia de dados e cálculos de *cheksum*.

Scripts CGIs para a geração de páginas dinâmicas podem ser chamadas diretamente pelo *kernel* e a saída desses direcionada diretamente para a camada de rede também através da "cópia-zero" para ser reduzir, além da cópia da dados, as mudanças de contexto entre modo usuário e modo *kernel* (ver 2.3.2 "Modos de execução e as System Calls", página 28) [81].

A idéia de se incorporar funções da camada de aplicação para o *kernel* não é nova, como pode-se observar em servidores de arquivos. [72] mostra um melhoria de desempenho de até 2 vezes em relação aos mais rápidos servidores implementados em modo usuário.

O TUX e o kHTTPd, concebidos para arquitetura Linux, são os principais exemplos de servidores Web implementados em *kernel* e serão abordados na próxima seção.

5.2 Exemplos de Servidores Web

5.2.1 Apache

O Apache é atualmente o servidor mais utilizado segundo a Netcraft [98]. Derivado do NCSA, tem como uma das principais qualidades o custo: é disponível gratuitamente juntamente com o código fonte. O Apache será detalhado na Seção 5.3.

5.2.2 IIS

O IIS (*Internet Information Server*) da Microsoft é o segundo servidor mais utilizado. Grande parte do seu sucesso deve-se à sua integração com outros produtos da Microsoft, facilidade de programação de páginas dinâmicas via ASP (*Active Server Pages*) além da facilidade de instalação e configuração.

O IIS é implementado segundo a arquitetura SPED, assim como o Zeus.

Um problema sério do IIS tem sido suas inúmeras falhas de segurança. É o servidor com maior incidência de *page defacements* (pichação de páginas) segundo [2] e alvo dos recentes ataques de *worms* como o Cod Red e o Nimda [90].

Mais informações sobre o *tuning* do IIS podem ser obtidas em [115].

5.2.3 NCSA

O httpd do *National Center of Supercomputing Application*s foi o primeiro servidor Web após o CERN. É o ancestral do Apache e do Netscape, e através do *Spyglass,* do IIS. Muitas características de outros servidores, como controle de acesso e CGI, foram originários do NCSA. Grande parte dos esforços de desenvolvimento foi transferido para o projeto Apache.

5.2.4 iPlanet (Netscape)

Em julho de 2000 a Netscape foi comprada pela AOL. Posteriormente, à essa junção criou-se uma aliança com a Sun, batizada de iPlanet, para o desenvolvimento servidores Web voltados ao comércio eletrônico.

Servidores iPlanet são comerciais e ocupam a 3ª posição entre os mais utilizados.

É uma das poucas opções *multithreaded* com certo grau de amadurecimento para plataforma Solaris. Possui uma interface gráfica via HTML bastante intuitiva para sua configuração, além de uma ferramenta de análise de estatísticas de desempenho, chamada *perfdump*. As informações providas relacionam-se a *status* de *sockets*, informações de cache, dentre outras.

Entre suas principais versões encontram-se a *Fast Track*, direcionada a *sites* pouco complexos, sendo de fácil manutenção, e a *Enterprise*, um *upgrade* da *Fast Track*, que inclui funcionalidades como gerenciamento de conteúdo e administração centralizada. Mais informações sobre o iPlanet, incluindo seu *tuning*, podem ser obtidas em [69].

5.2.5 Zeus

Zeus é um produto comercial considerado um dos servidores Web mais rápidos. Como o IIS, utiliza-se da arquitetura (SPED) trabalhando sob um único processo e utilizando-se somente de primitivas de entrada e saída de rede não bloqueantes.

5.2.6 kHTTPd

kHTTPd é um servidor Web implementado no *kernel* do Linux, disponível a partir das versões 2.4.x. kHTTPd manipula somente páginas Web estáticas e repassa todas as requisições de páginas dinâmicas para um servidor convencional como o Apache ou Zeus.

Para a sua utilização, é necessário que seja compilado diretamente no *kernel* ou como um módulo. A opção pelo kHTTPd na configuração do *kernel* do Linux encontra-se em *Network Options – kernel httpd acceleration*.

A sua configuração é feita via diretório virtual (ver 2.4.2.1 "Estrutura do /proc", página 37) `/proc/sys/net/kttpd`, onde localizam-se arquivos que abrigam parâmetros do servidor Web, tais como porta servidora (`server_port`), diretório raiz dos documentos estáticos (`document_root`), dentre outros. [125] oferece mais detalhes de configuração.

O suporte ao protocolo HTTP não é completo e muitas funcionalidades descritas no capítulo 4 como *range requests* não são implementadas. O TUX, descrito a seguir, é uma alternativa mais completa ao kHTTPd.

5.2.7 TUX

Criado por Ingo Molnar da Red Hat, é outro exemplo de servidor Web integrado ao *kernel* do Linux. TUX é acrônimo para *Threaded linUX HTTP layer*.

De acordo com Molnar, TUX é o próximo passo na evolução tomada pelo TCP, quando integrado aos *kernels* do Unix como uma funcionalidade padrão para redes [81].

Possui muito mais recursos que o kHTTPd. Pode por exemplo, executar todas as tarefas de um servidor Web convencional, não necessitando assim repassar tais tarefas para um servidor no espaço do usuário, embora o possa fazer no caso de aplicações mais complexas ou cabeçalhos que possivelmente não possa reconhecer. Neste caso, a requisição é repassada para um servidor convencional através de um mecanismo rápido de redirecionamento de *socket*. Não são necessárias alterações no servidor Web de suporte.

TUX suporta as operações básicas das versões 1.0 e 1.1 do HTTP como “Connection”, “Cookies” ou “Range Requests”.

A configuração do TUX se inicia pelo processo de compilação do *kernel*. Dependendo da distribuição do Linux é necessária a aplicação de um *patch*, que habilitará no menu de configurações “Networking Options” a opção “*Threaded linux HTTP layer* (TUX)”.

Assim como kHTTPd, configurações dinâmicas são possíveis através do diretório `/proc` via `sysctl`. Exemplos de parâmetros passíveis de modificação são: raiz do diretório de documentos, arquivo de *log*, porta a ouvir, porta de redireção, número máximo de conexões,

tamanho máximo da fila de requisição (ver 3.3.1.2 "Tamanho das filas de requisições", página 95), keepalive/timeout (ver 3.6.3 "Keepalive timer", página 111), dentre outros.

Logs são gravados num formato binário a fim de minimizar tempo com I/O e uso de espaço em disco. *Threads* responsáveis por atender clientes compartilham a mesma cadeia de *buffers* de *log*. Uma *kernel thread* (ver 2.5.2.4 "Implementação do kernel via threads", página 47) é exclusivamente dedicada ao *flushing* deste *buffer* a cada segundo ou quando o *buffer* está 95% cheio. Estes valores são constantes, não são alteráveis dinamicamente.

A cópia de dados e o cálculo de *ckecksums* para pacotes a serem enviados, são possivelmente as operações mais custosas no processo de se servir conteúdo estático que já resida na memória. Um método adotado pelo TUX, para se acelerar este processo, é manter os dados em cache nos próprios *buffers* de *socket* no *kernel*, ao invés do cache do sistema de arquivos ou em algum outro cache no nível da aplicação.

TUX utiliza um esquema similar ao Flash [100] (AMPED), para manipular rapidamente requisições em memória através do mecanismo orientado a eventos diretos da rede, enquanto gerencia operações de I/O assincronamente através de *threads* separadas, que o fazem escalar bem em ambientes SMP.

Em suma, o servidor TUX possui ganhos consideráveis em desempenho e escalabilidade, através da passagem de funcionalidade do servidor Web para mais próximo da pilha de protocolos de rede do *kernel*. Orientando o serviço Web diretamente por eventos oriundos da rede e mantendo-se dados em cache na camada de rede, além de evitar a mudança de contexto entre o modo usuário e *kernel*, possibilita um atendimento muito mais rápido às requisições, se comparado a servidores Web tradicionais.

Mais informações sobre o TUX, incluindo seu código fonte, podem ser obtidos em [106].

5.3 Apache

5.3.1 Histórico

O Apache surgiu em 1995 a partir de uma série de “patches” sobre o NCSA; e daí a origem de seu nome: “A patchy server”.

Graças à contribuição de muitos voluntários (o Grupo Apache) ele tem sido desenvolvido constantemente e mantido o *status* de servidor Web mais utilizado, segundo a Netcraft, estando

instalado em cerca de 10 milhões de servidores, o que corresponde a 60% dos servidores Web. Sua vantagem sobre outros servidores é bastante grande. O segundo servidor Web mais utilizado é o IIS da Microsoft com cerca de 20%[1].

Grande parte desta popularidade deve-se ao seu caráter livre. Seu código fonte pode ser obtido gratuitamente de forma a ser estudado/adaptado para atender necessidades próprias. Além disso, é conhecido por sua confiabilidade, configurabilidade, portabilidade e boa documentação.

5.3.2 Os Módulos do Apache

A arquitetura modular, que distingue o Apache de outros servidores, permite aos *webmasters* a habilidade de incluir qualquer característica desejada em um servidor Web, ao mesmo tempo que evita que tais características estejam presentes em todos os servidores Apache. Juntando-se isso a um núcleo simples e à não dependência entre os módulos, o Apache pode acomodar necessidades específicas de cada *site*, atingindo grande flexibilidade.

O núcleo do Apache (`http_core`) tem as funções básicas de manipular recursos (como descritores de arquivos, por exemplo), manter *pool* de processos *pré-forked*, ouvir *sockets* TCP/IP (inclusive para possíveis servidores virtuais configurados), dentre outros.

Uma ótima documentação dos módulos do Apache, pode ser obtida em [51].

5.3.3 Arquitetura para Atendimento de Conexões Simultâneas

Para o atendimento a conexões concorrentes no ambiente Unix, o Apache utiliza-se do modelo MP para a versão 1.3 e um modelo híbrido para a versão 2.0. No ambiente Windows o modelo adotado é o MT.

5.3.3.1 Apache 1.3 para Unix

Na versão 1.3 para Unix, o Apache baseia-se no modelo multiprocessos. Um único processo pai é simples e tem a função de inicializar e controlar o número de processos filhos para o atendimento de múltiplas requisições. Os processos filhos são mais complexos e têm a função de estabelecer a conexão com o cliente e processar a requisição. Essa transferência de

1. *Dados de Setembro 2001.*

complexidade para os filhos garante uma maior confiabilidade do servidor, uma vez que um mal funcionamento, que poderia causar um término abrupto do processo, é mais provável de ocorrer em um processo filho (que será rapidamente substituído), afetando apenas uma única conexão, não comprometendo todo o serviço [19].

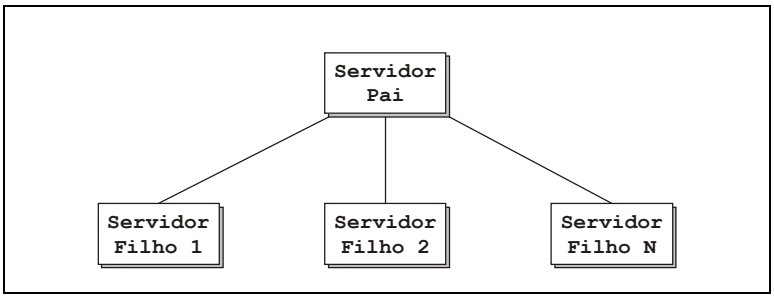

Figura 5.5: Modelo de processos do Apache 1.3.

O servidor pai controla o número de servidores filhos baseado na carga atual do sistema. Um número fixo de servidores filhos ociosos são mantidos, para que o atendimento a uma conexão seja mais ágil, pois evita-se o atraso com a criação de um novo processo. Os números mínimo e máximo de processos ociosos são configuráveis respectivamente pelos parâmetros `MinSpareServers` e `MaxSpareServers` (ver Seção 5.3.4.2).

Na inicialização do processo pai é criado um número fixo de processos filhos especificado pelo parâmetro `StartServers`. Após a criação inicial desses processos filhos, novos processos são criados a uma taxa exponencial até que se atinja o `MinSpareServers`. Inicialmente cria-se 1 processo, aguarda-se 1 segundo; depois 2 processos e aguarda-se 1 segundo; depois 4 processos e assim por adiante até se atingir a taxa de 32 processos filhos por segundo [59].

Assim, novos processos são criados para se manter a taxa de `MinSpareServers`. Analogamente, processos filhos se auto-destroem para se manter o número de processos ociosos menor do que `MaxSpareServers`. Um número muito alto de processos ociosos aumenta o tempo gasto com escalonamento, além de ocupar memória desnecessariamente.

O número máximo de processos a serem criados é limitado por `MaxClients`.

Este modelo, embora não possua boa escalabilidade, tem se mostrado bastante robusto e facilmente portável.

5.3.3.2 Apache 2.0

A versão 2.0 do Apache (atualmente beta) tem como principal inovação sobre a 1.3 a implementação de *threads* para se atender requisições concorrentes. Mais do que isso, através da sua arquitetura modular, implementa o MPM (*Multiple-process module*) [4]. Por este módulo é possível se determinar, em tempo de compilação, como as requisições são mapeadas para *threads* ou processos, de acordo com a plataforma utilizada. Por exemplo, em plataformas que possuem bom suporte para processos e *threads*, como o Solaris, é interessante se adotar o modelo híbrido: vários processos com múltiplas *threads*. Já em plataformas que não manipulam bem ou não dão suporte a múltiplos processos (como o Windows) um único processo com múltiplas *threads* é a melhor estrutura. Ainda, plataformas que não possuem suporte a *threads* devem ser consideradas. Neste último caso o método de *forking* convencional deve ser adotado.

O principal motivo da adoção de MPMs é a possibilidade de se suportar uma grande variedade de Sistemas Operacionais mais facilmente e eficientemente, aproveitando características próprias de cada um deles. Por exemplo, a versão 2.0 para Windows é muito mais eficiente, uma vez que a opção de MPM mpm_winnt pode usar características de rede nativas ao invés do padrão POSIX utilizado no Apache 1.3 . Este benefício é igualmente expansível para outros Sistemas Operacionais através da implementação de MPM especializados.

Atualmente, existem 5 opções disponíveis para MPM, sendo que todas adotam a estrutura de um servidor pai que monitora servidores filhos (sejam *threads* ou processos).

A escolha de qual opção MPM utilizar é definida através do argumento "`-with-mpm` *`modulo`*" no *script* de compilação.

O padrão para plataformas Unix é o modelo híbrido (`mpm_threaded_module`), onde existem múltiplos processos, cada um controlando múltiplas *threads*. Da mesma forma que na versão 1.3, o Apache 2.0 mantém um *pool* de *threads* servidores. Ele avalia o número de *threads* ociosas em todos os processos, criando-os ou destruindo-os, baseado nos parâmetros de configuração `MinSpareThreads` e `MaxSpareThreads`. O número máximo de requisições simultâneas a serem atendidas é determinado pela multiplicação de `MaxClients` (número máximo de processos) por `ThreadsPerClild`.

5.3.4 Otimizando o Apache

5.3.4.1 Compilação Otimizada

Assim como o Linux (ver 2.4.1 "Compilação otimizada", página 31) o Apache pode ser compilado de acordo com situações e plataformas específicas. É aconselhável que se escolha somente os módulos que serão realmente utilizados e que esses sejam compilados estaticamente ao núcleo, permitindo uma execução mais rápida [130].

5.3.4.2 Configurando parâmetros de execução

A instalação de um servidor Web é um processo relativamente simples e automatizado por *scripts* na maioria das implementações. Por outro lado, a otimização de um servidor Web para um melhor desempenho, é um processo pouco documentado e que dificilmente pode ser automatizado, pelas características próprias de cada *site*.

Grande parte do processo de otimização do servidor Web envolve a alteração de parâmetros, chamados de diretivas, nos arquivos de configuração do Apache. Essas diretivas permitem controlar o comportamento do servidor Apache pós-compilado no nível de processos e protocolos. O principal arquivo de configuração do Apache é o `conf/httpd.conf`. Distribuições do Apache disponibilizam `conf/highperformance.conf`, que é um arquivo de configuração compacto, que serve como um ponto de partida para o *tuning* do servidor.

Controlando Processos Filhos

`TypeOfServer` - Pode assumir dois valores: “inetd” ou “standalone”. Quando atuando com a opção “inetd”, o Apache utiliza-se do aplicativo inetd (ou xinetd) para servir como intermediário para suas conexões, o que degrada bastante o desempenho em servidores de alta demanda.

`StartServers` – Especifica o número inicial de processos que serão criados na inicialização do Apache. Assim, é um parâmetro que tem efeito somente na inicialização do servidor e em geral pode ser útil em servidores que são reinicializados constantemente e possuem alta demanda. (vide Seção 5.3.3)

`MinSpareServers` - Parâmetro responsável pelo controle do número de processos que devem estar disponíveis em um dado instante. À medida que a carga de requisições simultâneas aumenta, o Apache começa a criar novos processos a fim manter o *pool* de processos disponíveis com um tamanho mínimo. Pelo fato da criação de novos processos ocorrer sob demanda, o aumento deste valor torna-se importante somente em servidores que vivenciam crescimentos repentinos na taxa de requisições.

`MaxSpareServers` – Especifica o número máximo de processos ociosos em um dado momento. Na situação onde muitos processos tenham sido ativados para se manipular um pico na demanda, e posteriormente essa demanda cai, esta diretiva evita que o número excessivo de processos continue ativo.

`MaxClients` – Este parâmetro, um dos mais importantes, controla o número máximo de processos filhos que o Apache pode ativar independente da carga. Um valor baixo implica na rejeição de novas requisições, mesmo que o hardware ainda comporte a carga. Por outro lado, um valor muito alto ocasiona um consumo excessivo de recursos do servidor pelo Apache, podendo levar inclusive à queda do sistema. Invariavelmente o valor *default* (128) deve ser alterado em servidores de grande demanda, principalmente para aqueles que se utilizam da geração de páginas dinâmicas via CGI. Este valor pode ser alterado até o *hard-limit*, que em geral é 256. Caso se deseje ultrapassar esse valor (isso é comum em servidores de grande demanda) é necessária a recompilação do Apache, alterando a definição `HARD_SERVER_LIMIT` em `src/include/httpd.h`. Um valor aconselhável é 2048. Uma forma de se identificar se este parâmetro está sendo a causa de latência do servidor é se verificar, via *ps*, se o número de processos httpd ativos está próximo ou igual a este limite (vide 2.9 "Ferramentas de monitoramento", página 78).

`MaxRequestsPerChild` – Este parâmetro determina o número máximo de conexões que um processo servidor filho poderá atender antes de terminar voluntariamente. Este parâmetro é útil para se evitar “vazamentos de memória”, ocasionado por *bugs* na própria implementação do processo filho ou no sistema operacional, o que ocasionaria um consumo crescente de memória pelo Apache. Como na maioria das configurações do Apache, o valor 0 (zero) especifica que não

existe limite de requisições a serem atendidas, e é a melhor escolha quando se tem certeza de que não existem vazamentos de memória consideráveis. Ferramentas como ps e top podem ser úteis para se identificar vazamentos de memória. Um valor baixo para esta diretiva pode degradar o desempenho, uma vez que o Apache criará e destruirá processos mais freqüentemente. Um valor razoável, para plataformas com problemas de vazamento de memória, situa-se na faixa de 1000 a 10000 requisições [130].

Versões *MultiThreads*: Apache 1.3 para Windows e Apache 2.0

Em plataformas Windows, o Apache se utiliza da arquitetura *multithreads* pura, em vez de multiprocessos ou híbrida como nas versões Unix. Desta forma, as diretivas `StartServers`, `MinSpareServers` e `MaxSpareServers` não têm efeito. Teoricamente, a implementação *multithreads* no ambiente Windows é mais eficiente. Todavia, a imaturidade de implementações *threaded* em plataformas Windows, implica em uma certa instabilidade do servidor [130].

O número máximo de conexões simultâneas, manipulado pelo Apache para Windows, é configurado pela diretiva `ThreadsPerChild`, que é análoga à diretiva `StartServers`, nas plataformas Unix. Uma vez, que existe apenas um processo filho, este também é o número máximo de requisições ao servidor como um todo. Isto difere das implementações híbridas para Unix, onde o valor máximo de conexões é determinado pela multiplicação de `MaxClients` (número de processos) por `ThreadsPerChild` (número de *threads* por processo). Os parâmetros análogos ao `MinSpareServers` e `MaxSpareServers` são `MinSpareThreads e MaxSpareThreads.`

O arquivo `conf/httpd.conf` para o Apache 2.0 possui seções distintas de configuração para cada módulo MPM utilizado.

No Linux com o Apache 2 compilado no modo híbrido, um ps exibe todas as *threads* como se fossem processos. O número padrão de *threads* por processo (`ThreadsPerChild)` é de 25, e o número máximo de processos filhos (`MaxClients`), 4. O *hard-limit* para o número máximo de processos é 8. Para valores acima desse limite é necessária a recompilação do servidor, alterando a constante `HARD_SERVER_LIMIT` no arquivo fonte `/mpm/threaded/mpm_default.h`. A definição desse parâmetro acima do limite provoca a seguinte mensagem de erro:

```
[root@andromeda bin]# ./apachectl restart
 WARNING: MaxClients of 10 exceeds compile time limit of 8 servers,
 lowering MaxClients to 8.  To increase, please see the
 HARD_SERVER_LIMIT define in server/mpm/threaded/mpm_default.h.
```

Parâmetros de Desempenho relacionados aos Protocolos de Rede

`KeepAlive` -- Ativa ou não o processo de conexões TCP persistentes descrito em 4.5.2 "Conexões Persistentes", página 126. O valor *default* é "on", e deve ser mantido para que o servidor não sofra com o *overhead* inserido por conexões não persistentes. Esta diretiva permite um diálogo muito mais rápido entre o cliente e o servidor, ao custo de impedir que o processo servidor atenda qualquer outra requisição até que o cliente desconecte. Para lidar com esta questão, o Apache faz uso de duas diretivas adicionais, que controlam o tempo de vida de uma conexão persistente: `KeepAliveTimeout e MaxKeepAliveRequests`, discutidas a seguir.

`KeepAliveTimeOut` – Esse parâmetro é especificado em segundos e controla quanto tempo a conexão TCP é mantida, aguardando por novas requisições HTTP de um mesmo cliente. Um valor ideal é aquele onde as conexões não são mantidas abertas, consumindo recursos do servidor, por muito tempo após o usuário já ter desconectado, mas longo o suficiente para que conexões não necessitem ser recriadas constantemente. Isso é dependente da velocidade de conexão utilizada pela maioria dos clientes. Por exemplo, se todos os clientes acessam o servidor através de uma conexão discada a 1200 bps, o valor padrão de 15 segundos pode ser muito baixo. Por outro lado, caso trate-se de um servidor em uma intranet, esse valor pode ser muito alto.

`MaxKeepAliveRequests` – Este parâmetro controla quantas requisições uma única conexão persistente irá atender antes de ser encerrada. Aconselha-se um valor alto para se evitar o *overhead* com o estabelecimento de novas conexões. O valor 0 especifica um número ilimitado de requisições na situação onde `KeepAliveTimeout` não é excedido e o cliente não desconecte. Isto pode tornar o servidor vulnerável a ataques de negação de serviço.

`SendBufferSize` – Esta diretiva determina o tamanho do buffer de saída utilizada em conexões TCP, útil para se enfileirar informações para conexões onde a latência é alta. Cada *buffer* TCP criado pelo Apache, um por conexão de cliente, será dimensionado de acordo com este parâmetro. Desta forma, um valor alto pode ter grande efeito no consumo de memória em servidores com grande número de conexões simultâneas.

`Timeout` – É uma diretiva bastante abrangente, que determina por quanto tempo o Apache irá considerar uma conexão quando aparentemente ela esteja inativa, utilizando-se um dos seguintes critérios:

- O tempo decorrido entre o estabelecimento da conexão e o recebimento da requisição. Isto não afeta as conexões persistentes, para as quais é usada a diretiva `KeepAliveTimeout`.
- O tempo desde o último ACK

O *default*, 300 segundos, é considerado muito alto, aconselhando-se diminuir para 150 ou mesmo 75.

`ListenBacklog` – Determina o tamanho da fila de requisições por novas conexões TCP. O valor padrão é 512. Para que esta mudança surta efeito, é necessário que este limite seja também alterado no sistema, como visto em 3.3.1 "O three-way-handshake", página 93, que provê uma discussão mais profunda sobre este tópico.

Diretivas Relacionadas ao HTTP

`LimitRequestBody` – Limita o tamanho do corpo de uma requisição HTTP, no caso de um método PUT ou POST. O valor *default* é 0, o que significa que não há limite especificado. O tamanho máximo de uma requisição é 2GB. Para se prevenir ataques de negação de serviço explorando esta característica, é importante que se limite este parâmetro. 10 KB é um valor razoável, que suporta com folga o tamanho máximo de uma resposta de usuário via formulários [130].

`LimitRequestFields` – Limita o número de opções adicionais presente em um cabeçalho de requisição HTTP. O valor *default* é 100. Em situações reais é bastante difícil que o número de opções ultrapasse 30, mesmo em casos onde é feita uma extensiva negociação de conteúdo. Um grande número de opções em uma requisição pode ser indício de um cliente tentando executar requisições hostis. Um valor razoável para se amenizar este tipo de ataque, sem prejudicar a negociação com o cliente, é 50.

Outras otimizações

`Options Multiviews` – Este parâmetro localiza-se na seção *Directory* em httpd.conf. Caso não se esteja servindo versões de documentos em múltiplas línguas, esta opção deve ser mantida "off". Existe um processo de negociação entre o *browser* e o servidor para se determinar o que o *browser* pode manipular e o que o servidor pode oferecer. Em geral esse processo é desnecessário e pode afetar o desempenho do servidor.

`FancyIndexing` – Habilita a listagem de conteúdo de diretório. Aconselha-se evitar o uso deste método. O *design* do *site* deve ser responsável pela não necessidade de se listar conteúdo de diretórios. Este parâmetro permite ainda várias opções que podem prejudicar o desempenho de um servidor de grande carga, como a exibição de diferentes ícones para diferentes extensões.

`HostNameLookups` - Permite que Apache registre a informação baseada no nome do cliente ao invés do endereço IP. É um processo bastante custoso, mesmo se considerando que o Apache armazena resultados de resolução de nomes em cache. Isso porque para cada novo cliente deve-se fazer uma consulta ao servidor DNS, o que pode sobrecarregar o servidor e gerar tráfego desnecessário. Desta forma, é recomendável se desativar esta opção. Caso se necessite de estatísticas por domínio, por exemplo, pode-se utilizar analisadores de *logs* que fazem sua própria resolução de nomes em um momento oportuno.

`LogLevel` - O processo de gravação de *logs* é um dos grandes consumidores de ciclos de CPU e de tempo de disco. Desta forma, é importante não se gravar informações que raramente serão utilizadas. Ao extremo, pode-se evitar completamente o processo de *logging*, embora esta prática não seja aconselhável, visto que *logs* podem armazenar informações valiosas sobre o

comportamento do servidor. É possível, através da diretiva LogLevel, configurar qual o nível de *logging* desejado.

`Options FollowSymLinks` - Permite que o Apache sirva documentos sem a necessidade de se checar se o arquivo em si, ou qualquer diretório acima do arquivo é um *link* simbólico, processo que pode consumir tempo extra.

`MMapFile` – Esta opção não faz parte do Apache padrão executável, estando disponível através do módulo `mod_mmap_static.` Quando ativado, permite especificar arquivos para serem mapeados em memória, caso o Sistema Operacional suporte (ver 2.7.4 "Mapeando arquivos em memória", página 72). Arquivos mapeados são mantidos em memória permanentemente, permitindo ao Apache disponibilizá-los ao cliente rapidamente, sem a necessidade de acessá-lo a partir do disco. Essa diretiva não é flexível na sua sintaxe, não permitindo coringas, por exemplo. Também não é possível agrupar arquivos em um diretório e mapeá-los em memória de uma só vez. É importante lembrar que, uma vez o arquivo estando mapeado em memória, este não será acessado novamente do disco mesmo que ocorra alguma atualização. Para isto, é necessário se reinicializar o Apache.

Esta opção só é aplicável a arquivos estáticos. Para conteúdos dinâmicos pode-se utilizar do módulo `mod_perl ou` FastCGI vistos na próxima seção.

`Mod_bandwidth` – é um módulo que dá ao Apache a capacidade de limitar a quantidade de dados a serem enviados por segundo, baseado em um domínio ou endereço IP do cliente, ou ainda no tamanho do arquivo a ser transferido. Limites de banda podem também ser usados para se dividir a largura de banda disponível de acordo com o número de clientes, permitindo que o serviço seja mantido para todos os clientes, mesmo que teoricamente não exista banda disponível para todos. Mais informações sobre o `mod_bandwidth` podem ser obtidas em http://www.cohprog.com

5.4 Geração de Páginas Dinâmicas

5.4.1 CGI (*Common Gateway Interface*)

O CGI foi o primeiro método padrão para disponibilização de conteúdo dinâmico, inicialmente introduzido como parte do NCSA.

CGI provê uma interface padrão entre servidores Web e programas que podem gerar páginas HTML ou outro conteúdo, baseado em parâmetros fornecidos pelo usuário. A figura 5.6 mostra o funcionamento básico da geração de páginas dinâmicas via CGI [20].

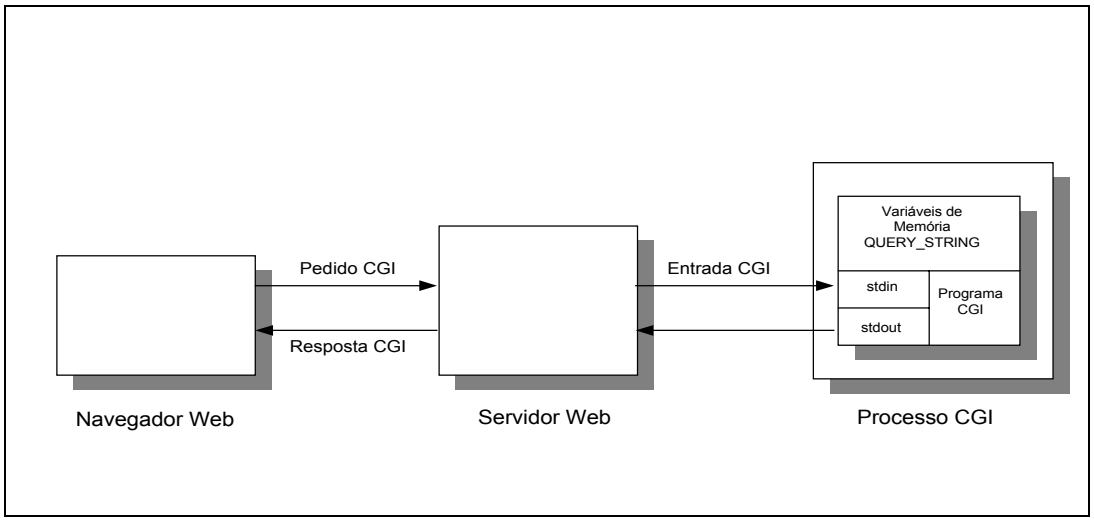

Figura 5.6: Servidor Web e o processo CGI

O processo CGI herda um conjunto de variáveis do servidor que descreve o estado do cliente, servidor e dados de entrada (QUERY_STRING).

5.4.1.1 Problemas de Desempenho do CGI

Embora o mecanismo de geração de conteúdo via CGI seja bastante simples e versátil, a sua estrutura básica limita consideravelmente seu desempenho, principalmente pela necessidade de criação de uma nova instância do programa a cada requisição do cliente. Logo após retornar o conteúdo ao servidor, este processo é então destruído.

Como visto, o processo de criação de um novo processo é bastante custoso. Além disso, outras operações são afetadas. Por exemplo, caso o CGI utilize acesso a um banco de dados,

uma nova conexão com esse banco deve ser estabelecida a cada instância do CGI. Em outras palavras, não existe o conceito de transações.

Esta carga gerada sobre o Sistema Operacional, limita o número de requisições simultâneas que podem ser atendidas, uma vez que exige mais processamento da CPU e outros recursos, se comparado à disponibilização de conteúdo estático.

Quando é recebida uma requisição por um programa CGI, o servidor deve interpretar a URL enviada pelo *browser* e detectar que o cliente deseja a execução de um CGI. O programa CGI é localizado, e são executadas as chamadas de sistemas `fork()` e `exec()`. Essas duas chamadas, como visto, pesam bastante para o mal desempenho. Variáveis de ambiente são então repassadas ao processo filho pela entrada padrão. A partir de então é de responsabilidade do CGI identificar os parâmetros de entrada e gerar o conteúdo de volta ao servidor. Esta etapa é o corpo do CGI e varia em complexidade, dependendo da aplicação. Após retornar os dados para o servidor, o CGI é finalizado. O servidor então acrescenta cabeçalhos HTTP adicionais e retorna o conteúdo ao cliente.

Um programa CGI bem escrito é fundamental para um bom desempenho. Dicas de programação otimizada podem ser obtidas em [74].

Quando se trata de um CGI implementado através de linguagens interpretadas, como Perl por exemplo, deve-se acrescentar o *overhead* associado ao carregamento do interpretador a cada requisição.

Embora o CGI possua um desempenho ruim, ele ainda é bastante popular, e continuará sendo por alguns anos, pelos seguintes motivos:

- O conceito do CGI é simples
- CGI é um padrão suportado pela maioria dos servidores Web, independentemente do hardware ou do Sistema Operacional. Em outras palavras, CGI é portável
- Programas CGI são fáceis de serem escritos, e o podem ser na maioria das linguagens de programação.
- Pelo fato de serem executados em um processo separado do servidor Web, não se corre o risco de um CGI mal escrito afetar todo o serviço.

5.4.2 FastCGI

FastCGI é uma variação da CGI, lançado pela OpenMarket, Inc. Ela oferece um método alternativo para se reduzir a sobrecarga durante o carregamento e término de processos CGI.

FastCGI substitui o processo CGI com um processo *daemon*, que uma vez iniciado, nunca é terminado. A lógica do programa CGI é colocada dentro do *daemon*. Quando o cliente inicia um pedido CGI, o servidor Web se conecta ao *daemon* CGI, por meio de um canal *full-duplex*, se o *daemon* estiver na mesma máquina, ou por um *socket*, se o *daemon* estiver em uma máquina diferente.

Essa conexão é usada tanto para entrada como para saída, a partir do *daemon* CGI. O servidor envia dados para o *daemon*, usando um protocolo simples de pacotes. Em seguida, o *daemon* demultiplexa cada pacote, processa o pedido, envia de volta os dados de saída pelo protocolo de pacotes e fecha a conexão. Nesse ponto, o *daemon* não desaparece, mas apenas aguarda o próximo pedido de conexão do servidor.

Ao contrário da maneira pela qual o programa CGI recebe os dados de entrada (pelas variáveis de ambiente, dos dados de entrada padrão ou da linha de comando), o FastCGI usa um protocolo simples de mensagens para empacotar essa mesma informação, antes de enviá-la pelo canal de conexão até o *daemon* CGI.

Removendo a sobrecarga da criação do processo, o método FastCGI é significativamente mais rápido do que o CGI, com tempos de resposta que se aproximam dos de transmissão de recursos estáticos [20]. Além da melhora no desempenho, os outros benefícios da FastCGI são:

- Capacidade de distribuir aplicativos CGI pela rede, oferecendo escalabilidade.
- Capacidade de ocultar aplicativos CGI por trás de um *firewall*.
- Independência de qualquer arquitetura ou fornecedor de servidores.
- Independência de linguagens de programação.

Mais informações sobre o FastCGI podem ser obtidas em [56].

5.4.3 APIs

Com intuito de se estender os serviços providos pelo servidor Web, servindo como uma alternativa mais veloz ao CGI e oferencendo arquiteturas de aplicativos mais avançadas, surgiram

as extensões aos servidores Web, através do conceito de API (*Application Programming Interface*). Utilizando-se deste método, a lógica, que de outra maneira seria codificada em um programa CGI, é implementada por meio de um conjunto de funções de aplicativos (geralmente em C), e ligada ao próprio servidor Web. Citam-se como exemplos a *Internet Server API* (ISAPI) da Microsoft, e a *Netscape Server API* (NSAPI) da Netscape.

Essas extensões também são conhecidas no servidor como *in-process* porque a biblioteca é compartilhada e associada dentro do espaço de endereço do servidor. Isso permite às funções do aplicativo o acesso direto às estruturas de dados do servidor.

As extensões *in-process* oferecem um desempenho melhor que a CGI, pois grande parte da sobrecarga da criação de processos e da comunicação entre eles desaparece. Normalmente, a conversão de um método CGI para um *in-process* aumentará o desempenho em até cinco vezes [20].

A chamada a um programa é substituída pelo carregamento de um objeto compartilhado, que normalmente é feito somente uma vez. Além disso, a sobrecarga do carregamento de um objeto compartilhado é significativamente menor do que a criação de um novo processo. Uma vez carregado, o objeto compartilhado permanece dentro do espaço de endereço do servidor, e a sobrecarga se restringe às chamadas de função.

As extensões *in-process* também evitam as restrições do compartilhamento de dados da CGI, pois conseguem armazenar o estado do aplicativo mesmo entre as funções, utilizando estruturas de dados na memória, armazenadas no espaço de endereço do servidor. Um objeto compartilhado possui acesso total às estruturas de dados internas do servidor. Essa acessibilidade é bastante poderosa, todavia perigosa. Embora o servidor forneça APIs e protocolos para formalizar o acesso ao seu interior, também o torna alvo fácil de um objeto compartilhado mal-intencionado, ou mal escrito que pode inclusive ocasionar a queda do servidor.

Sob o ponto de vista do desenvolvedor, um dos problemas graves do uso de APIs é que elas são específicas para cada servidor. Não há compatibilidade de código. Além disso fornecedores de servidores Web podem adotar modelos diferentes de computação distribuída. Como exemplos: a Netscape adota a arquitetura CORBA (*Common Object Request Broker Architecture*) enquanto a Microsoft a DCOM (*Distributed Common Object Model*).

5.4.4 *Scripts* embutidos

5.4.4.1 SSI

SSl (*Server-Side Include*) é outra alternativa para a geração de páginas dinâmicas e é apresentado como trechos de código, também denominados *tags*, inseridos em um documento Web e que serão executados pelo servidor Web no momento em que um cliente solicitar o documento.

Por meio de código SSI é possível obter dados dinâmicos em um documento Web de forma bastante simples.

Apesar da simplicidade, existem duas desvantagens de se usar SSIs para o desenvolvimento de aplicações Web. Em primeiro lugar, o servidor Web deve verificar em cada documento solicitado a existência de códigos SSIs. Essa interpretação do fonte acaba afetando o desempenho do servidor. Em segundo lugar, é um mecanismo que pode ser perigoso sob o ponto de vista da segurança.

Arquivos SSI têm extensão "shtml". Mais informações sobre o SSI são obtidas em [96]. Atualmente não é uma técnica muito utilizada, sendo substituída por PHP ou ASP.

5.4.4.2 PHP

PHP (acrônimo recursivo para *PHP Hypertext Processor*) é uma linguagem e interpretador de *scripts,* também usado para criação de páginas dinâmicas, que está disponível gratuitamente e é utilizada principalmente em servidores Web rodando em máquinas com Linux e Apache.

Em um documento HTML é embutido, através de *tags* especiais, um *script* PHP que possui sintaxe similar a Perl ou C.

Pelo fato de ser embutido em *tags*, o autor das páginas pode alternar entre código HTML e PHP rapidamente ao invés de depender de um programa compilado ou interpretado (um CGI) com várias linhas de código, cuja saída é um HTML. Por ser executado no servidor, o cliente não visualiza o código PHP.

Um fato interessante sobre o PHP é que é uma linguagem desenvolvida puramente para servir páginas Web, ao invés de ser baseado em uma linguagem já existente como Perl, Python, Visual Basic ou Java. Isso torna as aplicações escritas em PHP bastante sucintas se comparadas às equivalentes escritas em outras linguagens.

[103] é uma boa referência para o PHP.

5.4.4.3 ASP

ASP (*Active Server Pages*) é uma especificação para criação de páginas dinâmicas sobre o servidor IIS da Microsoft, que utiliza ActiveX, em geral código VBScript ou JScript. É o similar ao PHP para o mundo Microsoft, pois, da mesma forma, o *script* é embutido no código HTML. O arquivo possuirá extensão .asp. No momento da requisição pelo cliente, o arquivo é processado pelo servidor (o que demanda um interpretador VBScript ou JScript embutido no IIS), e o resultado enviado ao cliente como código HTML puro, interpretado pela maioria dos *browsers*.

5.4.5 Mod_Perl

O projeto de integração Apache/Perl [6] reúne o poder da linguagem de programação Perl e o servidor HTTP Apache.

Mod_Perl é uma solução para aceleração de código Perl na geração de conteúdo dinâmico sob o Apache. Nela o interpretador Perl é embutido no Apache como um módulo. Isso afeta bastante o desempenho, pois o interpretador persistente embutido no servidor evita o *overhead* de iniciar o interpretador externo. Além disso, os próprios *scripts* são cacheados no servidor para se melhorar o desempenho.

Mod_Perl é tão intrinsicamente acoplado ao Apache que é possível a um desenvolvedor Perl literalmente mudar a forma como o Apache trabalha. Inclusive o *tuning* do Apache pode ser feito através de programação Perl [7]. Antes isso somente era possível através de programação C via APIs do Apache. Módulos Apache podem ser completamente escritos em Perl.

5.4.6 Java

5.4.6.1 Porção Cliente - *Applets*

Como visto, uma das principais limitações do HTTP é seu pouco suporte para implementação de transações. Java surge como uma solução a este problema, ao transportar para o cliente certa capacidade de processamento, através das *applets*, evitando carga no servidor, e em alguns casos na rede. Uma vez que objetos Java são carregados no cliente, é necessário um método padrão de comunicação com o servidor, sendo CORBA e RMI (*Remote Method Invocation*) os padrões mais utilizados para comunicação entre objetos distribuídos. Uma outra vantagem de Java na

porção cliente é que se diminui a preocupação com vazamentos de memória no servidor, o que poderia levar a constantes inicializações, que são comuns em clientes, mas não em servidores.

5.4.6.2 Porção servidora – *Servlets*

Servlets é uma tecnologia desenvolvida também pela Sun para se estender a funcionalidade do servidor Web. Para tanto, o servidor Web deve estar habilitado através de uma API para executar código Java via Máquina Virtual Java acoplada ao servidor Web. Essa API especifica como o servidor deve carregar e executar arquivos de classes, ou *servlets*.

Desta forma, *servlets* é o análogo a *applets* na porção servidora. São carregadas e invocadas pelo servidor Web de modo semelhante ao que os *applets* são carregadas e invocados pelos navegadores. É independente de protocolo ou plataforma. *Applets* introduzem a independência de plataforma no nível do cliente, enquanto *servlets* estendem esse conceito no nível do servidor. Essa é uma grande vantagem de *servlets* sobre APIs convencionais que, como visto, são totalmente dependentes do servidor Web.

Por Java ser uma linguagem orientada a objetos, pode se beneficiar das qualidades inerentes a esse paradigma, como reuso e modularidade. É possível, por exemplo, criar-se uma *framework* de *servlets* comuns e reutilizá-la em aplicações futuras, estendendo funcionalidades. Outra qualidade de Java, empregada por *servlets,* são as *threads*, utilizadas para isolar o tratamento de clientes entre si.

Além disso, *servlets* têm total acesso às funcionalidades de rede presentes na linguagem Java. *Servlets* podem se comunicar com outras máquinas utilizando-se do conceito de *sockets* ou RMI (*Remote Method Invocation*). Da mesma forma, um *servlet* pode facilmente se conectar a um banco de dados relacional utilizando-se de JDBC (*Java Database Connection*) [48].

A utilização de *daemons* também pode ser observada no conceito de *servlets* em Java. É possível se inicializar *servlets* e fazer freqüentes conexões a eles, evitando-se o *overhead* da inicialização. Na primeira vez em que um *servlet* é requisitado ele é carregado no espaço de memória do servidor Web. Requisições subseqüentes ao *servlet* resultam em chamadas à instância da *servlet* em memória. Esta é uma vantagem sobre ASP, PHP ou SSI, onde as páginas com *scripts* embutidos são interpretadas a cada pedido do cliente.

O fato de Java ser uma linguagem interpretada tem pouca importância se comparado ao tempo perdido com constantes reinicializações de CGIs. Porém, é mais lento se comparado a APIs programadas em C, pois *bytecodes* de Java são mais lentos do que código nativo.

Uma grande vantagem de Java (*applets* juntamente com *servlets*) sobre CGIs é a habilidade de se distribuir a funcionalidade entre o cliente e servidor quando necessário. Ao invés de colocar toda a funcionalidade no programa CGI ou sofrer com o tempo de *download* de um *applet* completo, é possível baixar somente alguns dos *applets*, mantendo outros no servidor. Dinamicamente, baseado na carga da rede e do servidor, decide-se se a *applet* deve ser executada no cliente ou no servidor.

Para que se utilize o conceito de *servlets*, o servidor deverá estar habilitado para isso. O Tomcat do projeto Jakarta [5], desenvolvido pela *Apache Foundation* e pela Sun, é um dos exemplos mais expressivos. Outros servidores que podem ser habilitados para *servlets* são o iPlanet, IIS, Zeus, dentre outros listados em [122]. Servidores desta categoria passam do estágio de Servidores Web para "Servidores de Aplicação", ou seja, servem como uma interface para aplicações mais complexas, não estando restritos à comunicação via HTML (no nível da aplicação), podendo desenvolver comunicação via protocolo customizado [67].

JSP (JavaServer Pages)

A tecnologia JSP, também desenvolvida pela Sun, é uma extensão à tecnologia de *servlets* concebida para suportar a criação de páginas HTML e XML. Ela facilita a combinação de um esqueleto estático com conteúdo dinâmico possibilitando a separação da interface do usuário da geração de conteúdo. Assim, permite aos *webdesigners* mudarem o *layout* completo da página sem se alterar os mecanismos de obtenção do conteúdo dinâmico. Pode-se considerá-lo, um similar à SSI, ASP e PHP. Mais informações sobre JSP são obtidas em [122].

Capítulo 6

Distribuição de carga

Por mais otimizado que seja um servidor Web, baseado nos fatores descritos nos capítulos anteriores, em *sites* de alta carga, um único *host* pode não conseguir preencher os requisitos de tolerância a falhas, escalabilidade e principalmente desempenho, demandados por estes sites. Como visto, o consumo de recursos para o atendimento de múltiplas conexões simultâneas, como memória e poder de processamento, pode facilmente atingir o limite do hardware. *Upgrade* de hardware, em geral, é uma solução paliativa e que não tem uma boa relação custo-benefício. Além disso, a causa de latência pode não ser apenas o servidor em si, mas a largura de banda, o que demanda a localização diferenciada dos servidores [27]. A arquitetura baseada em múltiplos servidores é uma alternativa efetiva e relativamente barata que tem sido adotada pela maioria dos grandes sites.

Um mecanismo de distribuição de carga ideal deve prover a melhor associação entre clientes e servidores, objetivando o melhor tempo de resposta, de modo mais transparente possível ao usuário.

Pode-se classificar técnicas de distribuição de carga em dois grupos: Local e Global. O primeiro se concentra na distribuição de carga sobre servidores distribuídos localmente, fisicamente próximos, sendo útil quando se atinge o limite de hardware de um servidor. Já o segundo se foca na utilização de servidores dispersos geograficamente e é adotado quando o problema de desempenho se concentra no limite da rede. Atualmente a segunda técnica ainda é pouco explorada, com poucas soluções apresentadas se comparada com a primeira, sendo assim abordada em menor profundidade neste trabalho. Estes mecanismos serão categorizados, salientando-se vantagens e desvantagens, além de exemplos comerciais.

6.1 Servidores Web Distribuídos Localmente

A distribuição local de servidores, em um único ponto da Internet é útil para a situação onde um único *host* não está sendo capaz de atender a demanda de requisições, embora a rede suporte a carga.

Dado um conjunto de servidores que hospedam um *site* em uma única localização, pode-se identificar duas arquitetura principais:

- Arquitetura baseada em *cluster*: onde os nós servidores ocultam seus endereços IP na visão dos clientes. O único endereço IP visível é o do servidor virtual correspondente a um dispositivo (hardware ou software) localizado entre os clientes e o conjunto de servidores, responsável pela distribuição de carga, denominado *distribuidor*.
- Arquitetura Web Distribuída: quando os endereços IP reais dos nós servidores são visíveis às aplicações.

A arquitetura distribuída é mais antiga, onde o roteamento é, em geral, decidido pelo sistema DNS. A solução baseada em *cluster* é mais recente e o roteamento de requisições é inteiramente conduzido pelos componentes internos do *cluster*, o que proporciona um maior controle na atribuição de tarefas aos servidores, além de mecanismos de segurança e alta disponibilidade. Em geral, a arquitetura distribuída é mais aconselhável para servidores dispersos geograficamente.

6.1.1 Arquitetura baseada em *cluster*

Um Web *cluster* corresponde a um conjunto de servidores que são posicionados em uma localização única, inter-conectados por uma rede de alta velocidade e que apresentam uma única imagem (endereço IP). Cada servidor do *cluster* em geral apresenta seu próprio disco e um sistema operacional completo.

Embora o *cluster* possa ser composto por dezenas de nós, somente um nome e um endereço IP é divulgado. Assim, o servidor DNS autoritativo executa o mapeamento de um para um, traduzindo o nome no endereço IP de um entidade dedicada a rotear requisições entre os servidores, fazendo com que a distribuição de carga seja totalmente transparente aos clientes. Desta forma, esta entidade atua como um distribuidor centralizado que possui um controle bastante fino sobre o escalonamento a ser feito.

O distribuidor pode ser implementado tanto como um dispositivo de hardware dedicado plugado na rede, ou como módulos de software de um sistema operacional de propósito geral ou não. Em geral, a primeira alternativa é a de melhor desempenho, porém menos flexível e mais cara.

6.1.1.1 Mecanismos de roteamento

O distribuidor pode identificar univocamente cada servidor real através de um endereço que pode estar em diferentes níveis do protocolo, dependendo da arquitetura utilizada. Mais especificamente, um servidor do *cluster* pode ser identificado através do próprio endereço IP ou ainda do endereço MAC, da camada de enlace.

As informações que o distribuidor utiliza para decisões de roteamento também se baseiam nos níveis de rede, e servem para classificar o tipo de distribuidor. A escolha do mecanismo de roteamento a se utilizar possui um grande impacto nos algoritmos de escalonamento, uma vez que as informações disponíveis para decisão são bastante distintas. Atualmente existem dois tipos de distribuidor: o de nível 4 e o de nível 7[1].

Distribuidor de nível 4

Executa o chamado roteamento "cego" em relação ao conteúdo da camada de aplicação, uma vez que a escolha do servidor do *cluster* é feita no momento do estabelecimento da conexão TCP, na chegada do primeiro SYN (ver Seção 3.3.1). Visto que os pacotes do cliente não “sobem” até a camada de aplicação, a distribuição é mais rápida. Todavia, as políticas de escalonamento não podem se basear no conteúdo das requisições do cliente.

Pelo fato de pacotes pertencentes à mesma conexão TCP deverem ser associados ao mesmo servidor real, o distribuidor mantém uma tabela de associação, relacionando cada sessão TCP com o servidor adequado. Em geral a tabela é mantida em memória e acessada através de uma função de *hash*, para a melhoria de desempenho.

Isso permite que requisições HTTP, sob a mesma conexão TCP, sejam atendidas por um único servidor, de forma a evitar o *overhead* associado com estabelecimento e fechamento de conexões TCP (ver Seção 4.5.1). Todavia, isso não garante o conceito de transação no nível de

1. Taxonomia correspondente à classificação padrão OSI.

aplicação, uma vez que diferentes requisições HTTP sob uma mesma transação no nível de aplicação podem ser direcionados para servidores distintos.

Distribuidor de nível 7

Pode executar o roteamento baseado no conteúdo. Primeiramente o distribuidor estabelece uma conexão TCP completa com o cliente (incluindo o *three-way-handshake*), examina a requisição HTTP no nível de aplicação (nível 7) e repassa a conexão para um servidor escolhido. É menos eficiente em relação ao distribuidor de nível 4 porque possui um mecanismo de roteamento com informações mais detalhadas, além da necessidade de cópia de dados e mudanças de contexto entre modo *kernel* e usuário. Por outro lado, pode suportar políticas de roteamento mais sofisticadas.

Por trabalhar no nível 7, pode implementar em si mesmo o conceito de transação no nível de aplicação, usando informações de estado como *cookies* para direcionar requisições ao servidor adequado. Isso é útil em aplicações como o comércio eletrônico ou para melhoria de desempenho em aplicações que exigem a transmissão de dados via SSL (onde a fase de estabelecimento de transação é bastante custosa), dentre outras aplicações.

Outra vantagem desta arquitetura é a possibilidade de se combinar o cacheamento das páginas estáticas no distribuidor, porque é possível verificar qual arquivo está sendo requisitado no momento do roteamento. Esta solução é conhecida como *proxy* reverso (Seção 6.1.5).

6.1.1.2 Classificação quanto ao fluxo de saída

Os distribuidores, de ambos os níveis, podem ainda ser classificados de acordo com o fluxo de saída, ou seja, as respostas das requisições para o cliente. Obrigatoriamente o fluxo de entrada deve passar pelo distribuidor. Todavia, o fluxo de saída pode ou não passar por ele. Na classificação *two-way*, ambos os fluxos, o de entrada e o de saída, passam obrigatoriamente pelo distribuidor. Por outro lado, na classificação *one-way* apenas o fluxo de entrada. O fluxo de saída segue direto do servidor escolhido para o cliente.

Distribuidor de nível 4 - Arquitetura two-way

Nesta arquitetura cada servidor do *cluster* possui um endereço IP distinto. Ambos os pacotes de entrada e saída são reescritos no nível TCP pelo distribuidor. A reescrita de cabeçalhos de

pacotes é baseada na técnica de NAT (*Network Address Translation*) que será mais detalhada na Seção 6.1.4.

Pacotes que entram têm o endereço IP de destino alterado para o endereço IP do servidor escolhido do *cluster*. A resposta desse servidor retorna ao distribuidor que altera o endereço origem para seu próprio endereço e o repassa ao cliente.

Essa arquitetura não exige a alteração da implementação dos clientes, tampouco dos servidores reais. Todo o trabalho é feito pelo distribuidor. Pelo fato da reescrita de cabeçalhos não ser uma função trivial, demandando recálculos de *checksum* tanto de pacotes IP quanto de segmentos TCP na comunicação nos dois sentidos, o distribuidor pode tornar-se o ponto de lentidão, comprometendo sua escalabilidade.

Distribuidor de nível 4 - Arquitetura one-way

Nesta arquitetura, o fluxo de saída (em geral maior que o de entrada) não passa pelo distribuidor, diminuindo assim o trabalho de reescrita de pacotes demandado pelo modelo *two-way*. Contudo, este trabalho não é eliminado, mas sim distribuído entre os servidores do *cluster*, exigindo a alteração da implementação da pilha de protocolos dos mesmos. Assim, não é uma solução transparente quanto à arquitetura dos servidores.

Além disso, este mecanismo exige uma conexão separada com a rede externa para a transmissão dos dados a partir dos servidores diretamente para os clientes.

A comunicação entre o distribuidor e os servidores do *cluster* pode ser feita tanto através do nível de rede (endereços IP) quanto através do nível de enlace (endereços MAC). O roteamento pode ser feito por diversos mecanismos, como NAT, tunelamento de pacotes ou repasse de pacotes, descritos a seguir:

- NAT: A diferença deste mecanismo ao aplicado no *one-way* é o fato de que agora são os servidores do *cluster* que têm a responsabilidade da reescrita dos pacotes de saída, e não mais o distribuidor. O endereço de origem desses pacotes deve ser o endereço do *cluster,* para que os pacotes de entrada continuem obrigatoriamente passando pelo distribuidor. Isso acaba adicionando complexidade à implementação, pela necessidade da alteração na implementação da pilha de protocolos de cada servidor.

- Tunelamento de pacotes: É uma técnica que encapsula datagramas IP em novos datagramas IP. Isso é feito pelo distribuidor, endereçando esses novos pacotes para os servidores reais que devem desencapsular o pacote IP que contêm o endereço do cliente, e retornar a requisição diretamente para este, fornecendo como o endereço origem o endereço do distribuidor. A vantagem desta técnica é a não necessidade de recálculo de *checksums* demandado pela técnica de reescrita de cabeçalhos de pacotes. Por outro lado, exige que os servidores do *cluster* tenham suporte ao tunelamento IP.

- Repasse de pacotes: Esta técnica trabalha com endereços MAC, e exige que o distribuidor e os servidores do *cluster* estejam em uma mesma rede local, ou seja, que suas interfaces de rede estejam fisicamente conectadas a um mesmo *link* ininterrupto. Um mesmo endereço IP virtual é compartilhado pelo distribuidor e todos os servidores do *cluster* através do uso de endereços IP primários e secundários. Cada servidor é configurado com o endereço do distribuidor como secundário. Isso pode ser feito através do *aliasing* da interface de *loopback* via `ifconfig` em servidores Unix. Mesmo estando todos com o mesmo endereço IP, apenas o distribuidor irá receber o pacote de recepção já que os servidores deverão desabilitar respostas ao protocolo ARP, caso contrário ocorreriam colisões. Assim o distribuidor pode repassar pacotes para o servidor escolhido usando seu endereço físico (MAC), sem a necessidade de se modificar cabeçalhos TCP/IP. Essa técnica exige que o distribuidor reescreva o endereço de destino no nível de enlace. Por esse motivo esta técnica também é definida como "tradução de endereços MAC". O projeto ONE-IP descrito em [47] foi o precursor desta técnica.

Distribuidor de nível 7 - Arquitetura *two-way*

TCP *Gateway*

Nesta arquitetura utiliza-se de um *proxy* no nível de aplicação no distribuidor. Este *proxy* recebe conexões do cliente e mantém conexões persistentes com todos os servidores. Quando uma requisição do cliente é recebida, esta é repassada para um dos servidores através da conexão TCP persistente. Quando a resposta do servidor é recebida, esta é repassada ao cliente através da primeira conexão.

TCP *Splicing*

Este mecanismo corresponde a uma melhoria ao mecanismo anterior. O repasse de pacotes agora funciona em nível de rede e é feito diretamente pelo sistema operacional. Isso tem o

objetivo de se reduzir a cópia de dados e mudança de contexto entre modo usuário e modo *kernel* (uma das causas que incentivou a implementação de servidores Web diretamente no *kernel*, como visto na Seção 5.1.3). Uma vez que a conexão cliente-distribuidor foi estabelecida, e a conexão distribuidor-servidor foi escolhida em nível de aplicação, os pacotes serão agora comutados sem a necessidade de "subirem" até o nível de aplicação, melhorando o desempenho. Para isto é utilizada a técnica de reescrita dos cabeçalhos dos pacotes IP e segmentos TCP.

6.1.2 Arquitetura Web Distribuída

Como visto, nesta arquitetura cada servidor Web não oculta seu endereço IP real em relação aos clientes e não existe a figura do distribuidor fisicamente entre os clientes e os servidores. Os mecanismos de roteamento aplicados nesta arquitetura podem ser classificados de acordo com o local onde é tomada a decisão sobre qual servidor atenderá a requisição do cliente. Este local pode ser no próprio cliente, no serviço de resolução de nomes ou nos servidores.

6.1.2.1 Distribuição selecionada pelo cliente

A forma mais básica de se efetuar a distribuição de carga é se fornecer uma lista de servidores com suas respectivas localizações geográficas, deixando a cargo do usuário a seleção de qual servidor utilizar. Embora seja bastante simples de se implementar, já que não se adiciona nenhuma complexidade ao servidor, é bastante deselegante, por não ser transparente ao usuário, e não se basear em nenhum critério de seleção, senão a posição geográfica. Nem sempre o servidor mais próximo é o que provê melhor tempo de resposta. Além disso, para o caso de múltiplos servidores localizados em um mesmo *cluster* (não dispersos geograficamente), não haverá nenhum critério de escolha para o usuário, a não ser o "psicológico", e em geral os primeiros da lista sofrerão maior carga. Não é possível se ter nenhum controle da distribuição de carga pelos administradores.

6.1.2.2 Mecanismo de roteamento por DNS puro

DNS foi a primeira solução proposta para se distribuir a carga entre múltiplos servidores Web e continua sendo uma das mais simples. Originalmente foi concebida para múltiplos servidores distribuídos localmente. Todavia, atualmente é mais utilizado para servidores Web geograficamente distribuídos.

O DNS demanda a presença de múltiplas cópias do *site* em servidores idênticos.

Este mecanismo funciona da seguinte forma: para um nome de domínio são associados múltiplos endereços IP correspondentes a servidores distintos. Quando é feita uma consulta a um domínio, uma lista de servidores é retornada, variando-se a ordenação desta lista a cada consulta. Em geral o *resolver* (na aplicação Web, traduzida como *browser*) utiliza como endereço IP o primeiro da lista. Como para cada requisição DNS o primeiro da lista varia, clientes distintos acessam endereços IPs distintos correspondentes a diferentes servidores. Isto provê uma forma básica de distribuição de carga, baseada no algoritmo de *round-robin*.

Embora bastante fácil de se aplicar, este mecanismo possui várias desvantagens:

- A distribuição de carga é completamente "cega", ou seja, não se utiliza de nenhum critério, como carga atual, congestionamento ou proximidade geográfica. Isso obriga que todos os servidores tenham a mesma capacidade de processamento, e estejam localizados geograficamente próximos. Assim, pode-se considerar que o *Round Robin* DNS puro não é um mecanismo de balanceamento de carga completo.

- Dificulta a implementação de aplicações que dependem da manutenção do estado, ou transação como as de comércio eletrônico. Resultados DNS são armazenados em cache pelo cliente durante um tempo chamado TTL (*Time To Live*). Caso a transação seja longa, de maior duração que o TTL, é possível que o cliente continue a transação a partir de um outro servidor. Para o tratamento coerente desse tipo de requisição é necessária a utilização de um *timeout* de transação ou então que o servidores compartilhem dados sobre o estado do usuário. Ambas as alternativas adicionam complexidade. *Round Robin* DNS é mais indicado quando o conteúdo é somente para leitura.

- A natureza distribuída do DNS dificulta o controle dos endereços IP a serem utilizados, visto que a propagação dos dados não é instantânea, e consultas podem ficar armazenadas em cache de servidores DNS intermediários. Em suma, os caches intermediários acabam ignorando a distribuição de carga provida pelo DNS. Assim, no caso da retirada de um dos servidores do *cluster* este ainda pode continuar sendo requisitado por resoluções DNS armazenadas em cache de clientes ou servidores DNS intermediários, dentro do período TTL. Uma solução para isso seria a divulgação do registro DNS com TTL igual a zero, o que obriga uma nova consulta ao servidor "authoritative" a cada requisição, o que traz como efeito colateral, o aumento do tráfego DNS. Mesmo assim, não há detecção automática de falha em algum dos servidores, ou seja, não provê tolerância a falhas.

Embora o DNS não seja um mecanismo de distribuição de carga ideal, pelas deficiências apresentadas, ele continua sendo o mais utilizado pela sua facilidade de implementação e baixo custo. Além disso, não demanda o posicionamento dos servidores em uma única sub-rede, como as soluções baseadas em *cluster* discutidas anteriormente. Round-Robin DNS é especificado formalmente na RFC 1794 [25]. [38] apresenta propostas de mudanças com algoritmos de roteamento mais sofisticados. O *lbnamed* descrito em [113] é uma outra modificação ao DNS para melhor distribuição de carga.

6.1.2.3 Mecanismo de roteamento no servidor Web

Redirecionamento HTTP

O protocolo HTTP permite que o servidor responda a uma requisição do cliente (em nível de aplicação) com um código de status 301, que instrui o cliente a submeter uma nova requisição a um outro servidor.

Uma vantagem desta técnica é que a replicação dos dados pode ser gerenciada a partir de uma granularidade média de diretórios até a páginas Web individuais. Além disso, permite um roteamento baseado no conteúdo, uma vez que o primeiro servidor pode reconhecer o tipo de conteúdo da requisição e rotear o pedido para um outro nó baseado nessa informação.

A principal desvantagem é o fato de se acrescentar, a cada requisição, um tempo de *round-trip* extra pelo estabelecimento de uma nova conexão TCP com o segundo servidor.

Distribuição manual via *links*

Para situações onde se sabe de antemão qual será o período de pico, e que este não será longo, a ponto de ser possível o acompanhamento *on-line* pelo administrador, a distribuição de carga por redirecionamento via *links* pode ser viável. Nesta, o administrador desvia o tráfego em tempo real entre os servidores, baseado em parâmetros de carga e tempo de resposta de cada servidor, via ferramentas de monitoramento. Pode-se por exemplo, direcionar todo o conteúdo dinâmico mais acessado para uma máquina otimizada para isto. Assim, como no mecanismo anterior, o redirecionamento pode ser feito em uma granularidade menor, chegando até a uma única página. Além disso, pode-se conseguir um roteamento baseado no conteúdo.

A página principal continua acessível apenas a partir de um único servidor, cujo endereço IP é aquele resolvido pelo DNS. A partir da página principal, os acessos a outras seções do *site* são direcionados para IPs de servidores auxiliares.

O fato da página principal poder ser cacheada pelo navegador, fazendo com que a distribuição de carga seja ignorada no caso da mudança pelo administrador, pode ser evitado através da diretiva HTTP <meta http-equiv="refresh" content="300"> que especifica de quantos em quantos segundos a página deve expirar no cache (5 minutos no exemplo). Assim, o problema de cacheamento neste modelo possui um impacto bem menor do que ocorre na distribuição via *Round-Robin DNS*, explicado na Seção 6.1.2.2.

Reescrita de URL

Esta técnica é semelhante à anterior, porém baseada na geração de páginas dinâmicas (ver Seção 5.4), onde o primeiro servidor contactado altera dinamicamente os *links* para objetos embutidos na página ou referências a outras páginas, direcionando o tráfego para outros nós. Akamai [1] é exemplo de mecanismo de distribuição que se utiliza desta técnica.

Um ponto negativo é o fato de introduzir uma carga adicional ao servidor com a geração de páginas dinâmicas. Além disso, pode ocasionar aumento no tráfego DNS caso a reescrita de URLs baseie-se em nomes e não endereços IP.

6.1.3 Algoritmos de Escalonamento

Após a análise dos mecanismos de roteamento de requisições, esta seção descreve as políticas que podem ser adotadas para definir qual, dentre os servidores disponíveis, irá atender a requisição. A política a ser adotada tem um grande impacto tanto no desempenho quanto na escalabilidade do sistema.

Algoritmos de escalonamento de tarefas são aplicados em diferentes áreas da computação, possuindo uma infinidade de classificações distintas. Na prática, para a distribuição de carga Web, a principal classificação reside entre algoritmos estáticos ou dinâmicos.

Algoritmos estáticos são os mais rápidos e previnem o congestionamento na entidade que gerencia o escalonamento, em geral o distribuidor, em sistemas baseados em *cluster*. São mais simples por não levarem em consideração o estado atual de carga do sistema no momento da

decisão, podendo todavia, levar a decisões pobres de escalonamento. Exemplos desses algoritmos são o *Round-Robin* e o aleatório.

Algoritmos dinâmicos têm a capacidade de melhores decisões de escalonamento por se utilizar de alguma informação de estado. Todavia, isso insere um *overhead* devido ao gasto com a coleta e análise das informações.

Colajanni em [43] descreve alguns requisitos básicos para algoritmos de escalonamento de requisições Web:

- Baixa complexidade computacional, uma vez que as decisões devem ser feitas em tempo real.
- Total compatibilidade com padrões e protocolos existentes no escopo da aplicação Web.
- Todas as informações necessárias ao escalonamento deverão estar acessíveis pelo escalonador. Em particular, somente o escalonador e os servidores são as entidades responsáveis pela coleta e troca dessas informações.

Os algoritmos dinâmicos podem ser classificados em outras três categorias:

- Baseado em informações do cliente: Utiliza-se de informações do cliente como endereço IP ou porta TCP, no nível 4; ou a requisição HTTP completa do cliente, no nível 7.
- Baseado em informações do servidor: Utiliza-se de informações como carga atual ou passada, número de conexões, tempo de resposta e disponibilidade, para a associação de uma requisição a um servidor. Caso trabalhe no nível 7, pode se utilizar de informações de conteúdo, levando em conta informações sobre os caches dos servidores.
- Baseado em informações do cliente e do servidor: Utiliza-se da combinação dos estados do cliente e do servidor. Na realidade, a maioria dos algoritmos pertencente a primeira categoria, estão também contidas nesta, visto que em geral se utilizam também de informações dos servidores, mesmo que seja apenas a disponibilidade deles. A afinidade entre clientes e servidores também pode ser explorada, caso a arquitetura se baseie no nível 7. Nesta, pode se utilizar de um identificador de transação como um *cookie* ou identificador SSL.

6.1.4 NAT (*Network Address Translation*)

NAT é uma técnica baseada na reescrita de cabeçalhos de pacotes, que permite que uma rede local utilize um conjunto de endereços IP para tráfego interno e um segundo conjunto para o

tráfego externo. O software NAT fica localizado no roteador, entre a rede local e a Internet, constituindo o roteador NAT, fazendo todas as traduções de endereço necessárias.

A tradução de endereços baseia-se em tabelas internas, e o mapeamento entre os endereços internos e externos pode ser da forma M para N, e a variação desse mapeamento caracteriza um tipo de NAT. Citando-se um exemplo, o *masquerading*, uma forma de NAT descrita na próxima seção, possui mapeamentos da forma M para 1. Ou seja, M endereços IP externos são mapeados em um único IP interno. O redirecionamento dos pacotes é feito através da reescrita dos cabeçalhos.

O principal objetivo de NAT é diminuir a utilização do número de endereços IP (já bastante limitados na versão 4 do IP) dentro de uma organização, partindo do pressuposto de que apenas uma pequena porção do número de *hosts* da organização se conecta com o mundo externo, em um dado momento. *Hosts* em comunicação recebem dinamicamente um endereço Internet oficial, dado pelo roteador NAT.

Embora o objetivo principal seja "salvar" espaço de endereçamento valioso, como visto, NAT pode ser utilizado para outros propósitos, sendo os principais: segurança de rede e distribuição de carga.

6.1.4.1 Segurança (*Masquerading*)

No primeiro caso, parte-se do pressuposto de que é muito mais fácil proteger um único ponto de entrada do que múltiplos. A variação de NAT neste caso é a chamada "*masquerading*", onde múltiplos endereços IPs são ocultos por trás de um único endereço do roteador/*firewall*.

O cabeçalho IP de pacotes que saem é reescrito de forma a parecer que são originários de um único ponto. Respostas a esses pacotes são traduzidas pelo roteador NAT e repassadas para máquinas internas apropriadas. Assim, clientes internos podem se conectar ao mundo externo. Todavia, máquinas externas não podem sequer encontrar as internas, uma vez que só conhecem um único endereço: o do roteador ou *firewall* NAT. Desta forma, máquinas internas não podem ser atacadas diretamente.

Por esta técnica utilizar de informação de porta TCP para multiplexar múltiplas conexões, é também conhecida como NAPT (*Network Address Port Translation*) [128].

6.1.4.2 Distribuição de Carga (Servidores Virtuais)

Como visto na Seção 6.1.1.1, NAT pode ser utilizado para distribuição de carga através do conceito de "servidor virtual", que é representado por um único endereço IP, diferentemente do que ocorre com o *Round Robin* DNS. Este endereço corresponde a um dispositivo (hardware ou software) que emprega NAT, reescrevendo o endereço de destino de uma requisição de conexão, repassando-a a um servidor real. A escolha de qual será este servidor real a atender a conexão depende do algoritmo de escalonamento utilizado. Assim, possui-se um controle de distribuição real, baseado em parâmetros, quando comparado ao *Round-Robin DNS*.

Nos pacotes de retorno (arquitetura *two-way*), o mesmo roteador se encarrega da mudança de endereço da origem, não se necessitando assim de alterações na pilha de protocolos dos servidores reais.

O produto comercial, *Cisco Local Director*, descrito na Seção 6.4.1.1, é um exemplo da utilização de NAT que possui várias políticas de escalonamento configuráveis. A maioria dos produtos comerciais, para distribuição de carga em um *cluster* de máquinas localizadas em uma mesma rede utilizam-se do conceito de NAT para a distribuição de carga.

Um problema inerente a todas as utilizações de NAT é o fato de que um roteador NAT não tem somente a função de bombear pacotes entre interfaces, assim como um roteador comum, mas também deve manter informações de estado de comunicação. Isto requer atenção para que o roteador NAT não seja em si o ponto de congestionamento.

6.1.5 *Proxy* Reverso

Mecanismos de *proxy* convencionais em geral situam-se na porção cliente, servindo como um cache de páginas mais acessadas, evitando múltiplos acessos a partir de clientes de um mesmo domínio. Utilizando-se da mesma idéia de cacheamento, o *proxy* reverso serve como um cache das páginas mais acessadas, porém residente do lado servidor.

O *proxy* reverso, assim como o NAT, situa-se entre o cliente e o servidor, ou *cluster* de servidores. Ao receber uma conexão do cliente, é verificado se o conteúdo a ser servido já não se encontra em seu cache, atendendo imediatamente o cliente em caso positivo, ou atualizando seu cache a partir dos servidores originais caso contrário.

Pelo fato do conteúdo dinâmico em geral não poder ser armazenado em cache, mas gerado sob demanda, o *proxy* reverso pode apenas servir conteúdo estático. Com isso, os servidores

principais podem focalizar seu poder de processamento para o atendimento de requisições a conteúdo dinâmico. *Proxy* reverso pode ainda ser usado em conjunção com a técnica de NAT discutida anteriormente, onde requisições por conteúdos estáticos e dinâmicos podem ser divididas entre múltiplos servidores reais e o *proxy* reverso utilizado para conteúdo estático somente [73].

6.2 Servidores Web Distribuídos Geograficamente

Este tipo de solução é adotado quando o ponto de latência é a rede, o que demanda a localização diferenciada dos servidores ou *clusters* de servidores na Internet, provendo uma solução mais escalável para o atendimento de milhões de acessos diários.

Este modelo de arquitetura possui muitos pontos em comum com o explicitado na Seção 6.1.2, podendo ser tratado como um caso particular deste, onde os servidores estão obrigatoriamente localizados em regiões diferentes, não sendo apenas uma questão de se deixar visível ou não o seu endereço IP.

6.2.1 Arquiteturas

6.2.1.1 Servidores Espelhados com seleção pelo cliente

Esta arquitetura é análoga à descrita na Seção 6.1.1.1, onde fica a cargo do cliente a escolha do servidor a se requisitar o pedido, baseada na localização geográfica deste.

6.2.1.2 Servidores Web Distribuídos Geograficamente

Nesta arquitetura, servidores únicos são espalhados em pontos estratégicos da Internet. Pode ser classificado quanto aos níveis de escalonamento possíveis.

Único nível de escalonamento

Aqui o DNS autoritativo, ou outra entidade, seleciona diretamente um servidor, baseando-se em algoritmos próprios. O servidor selecionado fica responsável pelo atendimento da requisição.

Dois níveis de escalonamento

O primeiro nível de escalonamento é o descrito no ítem anterior. Após a escolha do servidor pelo DNS ou pela entidade responsável, este servidor pode atender a requisição ou então redirecionar o pedido a um outro servidor, via diretiva HTTP *Redirect*. Cria-se assim uma política de escalonamento distribuída, onde todos os nós servidores participam da associação de requisições a servidores. Os critérios a serem adotados para o reescalonamento podem ser a carga atual do servidor, o conteúdo da requisição ou outro critério de seleção.

Este nível adicional de escalonamento é útil, pois algoritmos de seleção geográfica ainda podem ocasionar um balanceamento não eficiente por confiar em conceitos de proximidade, que ainda é uma questão aberta no contexto da Internet [42].

Esta redireção é transparente ao usuário, porém não ao cliente (*browser*), o que pode causar certo atraso, visto que requer a abertura de duas conexões HTTP.

6.2.1.3 *Cluster* de Servidores Distribuídos Geograficamente

Nesta arquitetura, ao invés de se distribuir servidores únicos em pontos específicos da Internet, distribui-se *clusters* inteiros. Os *clusters* podem ser interconectados por uma rede de alta velocidade, o que facilita a troca de informações entre os centros.

O escalonamento primário, feito em tempo de resolução de nome via DNS autoritativo, redireciona a requisição a um dos *clusters* distribuídos geograficamente. Neste *cluster* a arquitetura é a mesma da exposta na Seção 6.1, onde a associação será feita localmente.

Assim, pode possuir até três níveis de escalonamento.

Primeiro nível de escalonamento

Executado pelo DNS ou outra entidade no momento da resolução do nome do domínio. A associação a um *cluster* é feita baseando-se, em geral, no critério de proximidade com o cliente.

Segundo nível de escalonamento

Executado pelo "distribuidor" do cluster que seleciona um dos servidores reais, segundo algoritmos e arquiteturas descritas na Seção 6.1

Terceiro nível de escalonamento

Cada servidor real pode redirecionar a requisição recebida para outro servidor através de mecanismos de redirecionamento HTTP.

6.2.2 Questões sobre Escalonamento Global

O escalonamento global possui certas peculiaridades que podem produzir um balanceamento não muito eficiente.

A primeira delas é a presença de fusos horários. Caso se opte por uma distribuição que leva em conta apenas a distância geográfica, certos servidores ou *clusters* serão requisitados em apenas certos períodos do dia, enquanto outros poderão estar ociosos nesses mesmos períodos. Haverá, assim, picos de acesso que não serão distribuídos entre todos os servidores.

Outro ponto é o fato de que o conceito de proximidade na Internet é ainda uma questão aberta. A proximidade geográfica nem sempre implica um melhor tempo de resposta. Algumas informações que podem ser utilizadas na determinação do melhor servidor no contexto geográfico são:

- Informações estáticas:
  - Endereço IP do cliente para se determinar a posição geográfica
  - Número de saltos (*hops*): Número de roteadores entre o cliente e os servidores
- Informações dinâmicas:
  - *Round-Trip Time* (via ping)
  - Largura de banda disponível

[46] discute mais aprofundadamente questões relacionadas à "distância" na Internet.

6.3 Outras Considerações

6.3.1 HTTP é escalável

Embora o HTTP seja ineficiente nas interações com o TCP, devido à natureza sem estados e com conexões curtas como visto no Capítulo 4, essas são exatamente as características que o tornam escalável. Por serem curtas, torna-se menos provável a concorrência entre múltiplas conexões, o que possibilita um melhor acesso aos recursos do sistema, quando se observa a partir da óptica

de cada conexão. Pelo fato de não se possuir estados, é inteiramente transparente ao usuário se uma requisição foi atendida por um ou por outro servidor. Isso facilita a adição de mais servidores "*on the fly*" para se atender aumento de carga com relação ao conteúdo estático [27]. O conteúdo dinâmico, como visto na Seção 5.4.2, pode ser escalável através da adoção do FastCGI, ou o uso de servidores dedicados a este fim.

6.3.2 Replicação e Sincronização

O processo de servir documentos Web através de múltiplos servidores, via múltiplas técnicas aqui citadas, insere a seguinte questão: deve-se replicar todo o conteúdo estático, repartí-lo entre os múltiplos servidores ou mantê-lo centralizado ? Caso se escolha a primeira alternativa, replicar o conteúdo entre servidores idênticos, é necessário se garantir que os dados estejam sincronizados entre si. Para isso pode-se utilizar ferramentas como o rsync ou rdist fornecidos em plataformas Unix, ou ainda o protocolo DRP (*Distribution and Replication Protocol*) [84]. Encontram-se também, ferramentas comerciais como o *GLOBAL-SITE Controller* da F5 Networks [55].

A segunda técnica, distribuir o *site* por máquinas distintas, tem a desvantagem de se dificultar a manutenção do *site*, além de poder ser pouco efetiva em situações onde apenas parte do conteúdo é mais extensivamente acessado (a página principal por exemplo).

A terceira técnica, manter todo o conteúdo centralizado em discos compartilhados, é a mais simples em termos de manutenção e pode ser atingida através do uso do NFS, caso os múltiplos servidores estejam localizados fisicamente próximos (arquitetura distribuída localmente), de forma que o atraso na comunicação entre o servidor Web e o sistema de arquivos comum não degrade o desempenho. Mesmo assim, o servidor de arquivos, no caso, se consistirá em um único ponto de falha.

6.4 Implementações Comerciais e protótipos para balanceamento de carga

Existem diversos produtos comerciais, em geral implementados em hardware, cuja função é a distribuição de carga local ou geográfica. Estão seção apresenta os dois produtos atualmente

mais utilizados, além de uma classificação, baseada nos critérios apresentados, dos produtos e protótipos mais expressivos.

6.4.1 CISCO

6.4.1.1 Cisco LocalDirector

Na solução adotada pelo LocalDirector da Cisco, o nome do domínio é associado a um único endereço IP, que é o endereço do balanceador de carga, que atua como um servidor Web virtual, através de NAT. Esta solução pode ser classificada como Local baseada em *cluster*, com fluxo *two-way*. Quando requisições são recebidas pelo balanceador, ele se encarrega por decidir qual dos servidores do *cluster* deve atendê-la, baseando-se em diferentes algoritmos, que podem ser escolhidos de acordo com o *site* em particular:

- Servidor com menor número de conexões: LocalDirector rastreia quando uma conexão é aberta e fechada para cada servidor. Baseado nesse algoritmo ele redireciona a conexão para o servidor que possui o menor número de conexões abertas.
- *Round-Robin*: o balanceador irá rotacionar requisições pelos servidores, um de cada vez, não levando em conta o número de conexões ou a carga do servidor. Todos os servidores serão tratados igualmente. Esta técnica é similar ao round-robin DNS exceto pelo fato de que um servidor com problemas, será automaticamente detectado e retirado do *pool*.
- Porcentagem de carga: Esta opção permite que o administrador associe um peso de desempenho para cada servidor. Para cada será atribuída uma porcentagem de carga, baseada na configuração e capacidade de cada servidor.
- Número máximo de conexões: O administrador pode atribuir um número máximo de conexões que cada servidor pode suportar. Esta opção tem a vantagem de prevenir que o servidor seja totalmente sobrecarregado, garantindo um tempo de resposta mínimo para as requisições que já estão sendo atendidas.

Nota-se que o algoritmo de decisão não leva em conta o uso de recursos de cada servidor, como memória ou tempo de processamento, não obrigando desta forma a presença de um processo responsável pela coleta dessas informações em cada servidor. Assim, não se possui um conjunto de informações mais exato, que reflita com mais fidedignidade a real carga dos servidores.

6.4.1.2 Distributed Director

É o produto da Cisco Systems [34] para a distribuição de carga entre servidores que se localizam geograficamente dispersos. Determina qual servidor está mais próximo do cliente baseado no número de *hops* (roteadores) entre o cliente e o servidor. Assim, ele não analisa o desempenho dos servidores. Em outras palavras, o cliente pode estar acessando o servidor mais próximo, mas não necessariamente o de melhor desempenho.

A conexão é transparentemente redirecionada para o servidor escolhido. Distributed Director pode ser adotado juntamente com o LocalDirector como uma solução completa. Através do Distributed Director localiza-se qual o *cluster* de servidores mais próximo do cliente, sendo que cada *cluster* é gerenciado pelo LocalDirector, que escolhe qual servidor do *cluster* deverá atender à requisição.

6.4.2 F5 Networks

6.4.2.1 F5 – Big IP Controller

Big IP Controller [53] é um produto comercial da F5 Networks análogo ao LocalDirector da Cisco. Assim, tem a função de distribuir carga entre um *cluster* de servidores locais, não contemplando a distribuição geográfica.

É uma solução em *hardware*, que pode trabalhar em duplicata de forma a prover tolerância a falhas. Utiliza-se de NAT para a distribuição, via reescrita de cabeçalhos.

Big IP não exige a presença de sensores de cargas nos servidores. O mecanismo de escalonamento pode ser configurado das seguintes formas:

- Servidor mais rápido: servidor cuja requisição possui o menor tempo de *round-trip.*
- *Round Robin*
- Menor número de conexões
- Porcentagem de carga

Assim como o Cisco Local Director, não mantém um *daemon* para coleta de informações de carga no servidor, tornando a implementação mais transparente às possíveis arquiteturas adotadas.

6.4.2.2 F5 – 3DNS Controller

3DNS é uma solução de distribuição de carga para servidores dispersos geograficamente, baseada na melhoria do DNS padrão para garantir ao cliente acesso ao servidor ativo e com melhor tempo de resposta.

Clientes são direcionados para servidores, ou grupo de servidores, baseando-se em métricas de desempenho coletadas a partir de balanceadores de carga locais (como o BIG IP). As decisões podem ser baseadas em uma métrica única, como servidor com menor número de conexões perdidas ou tempo de *round-trip*, ou ainda através da combinação de métricas, atribuindo-se pesos às mais importantes. Esses indicadores podem também ser variáveis de acordo com períodos pré-definidos.

Os algoritmos de balanceamento de carga consistem dos seguintes métodos:

- *Round-Robin*
- Disponibilidade Global: primeiro servidor disponível.
- Geográfica: Atribuição por país de origem do cliente, permitindo, além de teoricamente um acesso mais rápido ao cliente, a possibilidade de se criar *sites* em línguas específicas para cada país, com encaminhamento automático.
- Menor número de conexões
- Pacotes por segundo
- Kilobytes por segundo
- Tempo de Round Trip
- Número de *hops* (roteadores) entre o cliente os servidores.
- Aleatória

Mais informações podem ser obtidas em [54]

6.4.3 Classificação de Produtos e protótipos

Nesta seção são categorizados alguns dos produtos e protótipos mais expressivos baseando-se nos critérios adotados neste capítulo, para a arquitetura local

6.4.3.1 Distribuídores de nível 4

Two-Way	One-Way
Cisco's Local Director	IBM Network Dispatcher
Linux Virtual Server	Linux Virtual Server
F5 Network Big-IP	ONE-IP
Foundry Networks ServerIron	Foundry Networks ServerIron
Coyote Pont's Equalizer	Nortel Network's Alteon

Exemplos de distribuídores Locais de nível 4

6.4.3.2 Distribuídores de nível 7

Two-Way	One-Way
IBM Network Dispatcher	Reasonate's Central Dispatcher
Foundry Networks ServerIron	
Cisco's CCS	
HydraWEB's Hydra2500	
Zeus Load Balancer	

Exemplos de distribuídores Locais de nível 7

Capítulo 7

Conclusão

A crescente popularização da Internet, em especial do serviço Web, tem demandado cada vez mais *sites* com tempos de resposta satisfatórios.

Todavia, nota-se na bibliografia existente uma falta de estudos que abordem de maneira completa este assunto, em especial questões relacionadas ao processo de otimização de desempenho de servidores Web. Assim, o maior objetivo e maior contribuição deste trabalho foi o levantamento de forma clara dos aspectos envolvidos, reunindo em um único texto toda a informação esparsa existente, estabelecendo relações entre os tópicos, além de agregar embasamento teórico, soluções e sugestões.

Como resultado importante, observou-se que o *upgrade* de hardware, em geral a primeira alternativa adotada, não é a única e nem sempre é a que possui melhor relação custo-benefício. A correta otimização do software pode levar a um melhor aproveitamento do hardware, de forma a poder se atender *sites* com carga considerável, a um custo bastante baixo.

Começando pelo Sistema Operacional no capítulo 2, foram exploradas todas as facetas deste componente no que tange ao desempenho Web, iniciando-se pelo histórico de Sistemas Unix. Estudou-se sua longa história e sua relação com a própria Internet, fatos que contribuem para estabilidade e para o grande número de serviços Internet que dele dependem. Todavia, nota-se também a existência de uma grande variedade de versões com características próprias, o que dificulta a implementação de aplicativos que sejam compatíveis e ao mesmo tempo otimizados para funcionarem de forma eficiente em todos os sistemas distintos.

O Linux, representante maior da categoria de software livre, foi utilizado como base para se exemplificar o processo de otimização do *kernel* para um ambiente em específico. Isso só foi

possível graças à disponibilidade de seu código fonte, o que reforça mais uma vantagem dos softwares nesta categoria: a possibilidade de um controle total sobre o seu funcionamento através de otimizações específicas.

A implementação de processos e *threads* tem um papel importantíssimo no desempenho Web, visto que são através deles que são representadas as tarefas que serão executadas pelo sistema, incluindo o software servidor Web. A implementação, além do processo de criação e manutenção dessas entidades pelo sistema operacional, têm impacto direto sobre o consumo de memória e ciclos da CPU e inclusive orientam a arquitetura de implementação de servidores Web, como visto no capítulo 5.

Não menos importante é a forma como é implementado o sistema de arquivos já que o disco é a entidade mais lenta na hierarquia de memória. Sendo assim, o acesso customizado a ele tende a trazer importantes melhorias de desempenho no sistema como um todo. Sugestões foram propostas para se atingir esse acesso customizado, citando-se como exemplos o particionamento em múltiplos discos e a desativação do registro de último acesso de leitura.

Gerenciada conjuntamente com o acesso ao disco, está a memória RAM, cuja quantidade e forma de gerenciamento está também diretamente relacionada com o desempenho do sistema. Quando em grande quantidade, melhora o desempenho, visto que são evitados acessos a disco pelo uso de *buffers*. Quando em falta, degrada-o, já que se passa a utilizar o disco como extensão de memória. Estudou-se também como o processo de mapeamento de arquivos em memória agiliza o acesso, uma vez que evita as freqüentes mudanças de contexto que ocorrem por chamadas de sistemas, no método convencional de acesso a arquivos. Este mecanismo é bastante utilizado pelos servidores Web para servir páginas estáticas.

Ferramentas de monitoramento foram apresentadas de modo a identificar possíveis gargalos, seja no limite de processos criados pelo servidor, na falta de memória, na deficiência no poder de processamento, no limite de número de conexões simultâneas ou em outros aspectos.

O TCP, abordado no capítulo 3, foi escolhido por ser o protocolo de transporte que dá suporte ao HTTP. É bastante complexo na sua implementação e por isso possui vários pontos passíveis de otimização. Por ser implementado no *kernel* do Sistema Operacional, a sua interface com as aplicações, incluindo o software servidor Web, dá-se através das chamadas de sistemas.

Demonstrou-se que o processo de estabelecimento de conexão é bastante custoso, já que utiliza-se do mecanismo de *three-way-handshake* onde são necessários no mínimo 2 *round-trips*

para que o cliente comece a receber as informações. Neste processo de conexão, encontram-se as filas so_q e so_q0, cuja função é manter os estados das conexões "semi-abertas". Por possuírem um tamanho fixo, são bastante suscetíveis aos ataques de negação de serviço. O Linux apresenta uma alternativa para este problema, pelo conceito de *syn-cookies*, que deve ser habilitado no momento da compilação do *kernel* e posteriormente nos arquivos de configuração. Soma-se ao processo de conexão, o controle de fluxo de dados que se dá através do algoritmo de *slow-start,* cuja função é prevenir o congestionamento da rede ao mesmo tempo que previne também o estouro de *buffers* de *sockets* de um destinatário mais lento. Isto é implementado através do aumento gradativo da taxa de envio de segmentos.

Os inúmeros *timers* do TCP também foram abordados, explicitando-se suas funções de modo que o administrador possa configurá-los de forma adequada, impedindo que recursos do sistema sejam consumidos desnecessariamente.

No capítulo 4 foi abordado o HTTP, o protocolo central da Web. Suas características principais foram demonstradas, dentre as quais o fato de não possuir estados, fator que simplifica o processo de implementação no servidor e no cliente. Demonstrou-se o formato das mensagens principais, que podem ser facilmente colhidas via ferramentas de monitoramento de rede por se tratar de um protocolo que transfere mensagens no formato texto.

O ponto mais importante quanto ao desempenho HTTP são suas interações com o TCP. Pelo fato do HTTP ser um protocolo sem estados, existe um grande número de requisições curtas. Supondo que se utilize de conexões não persistentes, isso penaliza consideravelmente o desempenho pela necessidade de se abrir uma nova conexão TCP para cada objeto HTTP transferido. Como visto no capítulo 3, o processo de estabelecimento de conexão é muito custoso, pela aplicação do *three-way-handshake* e do *slow-start.* A utilização de conexões não persistentes na versão 1.1 do HTTP visa evitar esse problema ao agrupar em uma única conexão TCP múltiplas requisições HTTP.

O capítulo 5 apresentou o software servidor Web, responsável pela implementação do protocolo HTTP na porção servidora, cuja função é servir as requisições dos clientes, interagindo com o sistema operacional de forma eficiente.

Apresentou-se uma classificação dos servidores baseada na arquitetura adotada. Uma primeira classificação é quanto à implementação no modo usuário ou modo *kernel*. A primeira categoria engloba os servidores considerados tradicionais, que se utilizam dos recursos do

hardware tendo como interface o sistema operacional, via chamadas de sistema. Dentro desta categoria encontram-se os servidores que utilizam múltiplos processos, múltiplas *threads* ou ainda a técnica SPED para fazer o atendimento de múltiplas requisições simultâneas. Foram apresentadas a forma de implementação, as vantagens e desvantagens de cada uma das arquiteturas, baseando-se no nível de interação com o sistema operacional.

Servidores Web implementados diretamente no *kernel* são recentes e têm como principal motivação a melhoria de desempenho quando trazem para dentro do *kernel* a implementação HTTP. Evita-se, por exemplo, as constantes mudanças de contexto entre modo usuário e modo *kernel* e a necessidade de cópia de dados entre esses dois espaços. As requisições podem ser atendidas diretamente por *kernel threads* e dados podem passar diretamente do cache do sistema de arquivos para os *buffers* de *sockets* sem a necessidade de subirem até o espaço de memória do usuário.

Exemplos de servidores Web dessas duas categorias foram apresentados, apontando suas características principais. O Apache foi visto com mais detalhes por se tratar do servidor Web mais utilizado, além de possuir, como o Linux, o código fonte livre, o que permite customizações. Foram apresentadas as principais diretivas de funcionamento e como elas podem ser configuradas para prover um melhor desempenho, dependendo do ambiente em questão. Um exemplo importante é o *MaxClients,* diretiva que controla o número máximo de processos/ *threads* filhos que o Apache irá criar para se atender às múltiplas conexões simultâneas, sendo um gargalo bastante comum na maioria das implementações de sistemas de alta demanda. Outros parâmetros relacionam-se com a taxa de criação de processos filhos, com parâmetros TCP e HTTP, e com o funcionamento do próprio servidor, como a atividade de geração de *logs* que pode envolver ou não a resolução de nomes via DNS.

A criação de páginas dinâmicas também é um outro fator de suma importância no desempenho. Páginas dinâmicas têm tido uma crescente taxa de utilização à medida que cresce também o número de aplicações Web com um nível de sofisticação mais alto. CGI, um dos métodos pioneiros, sofre em desempenho pela necessidade da criação de um novo processo para cada requisição do cliente. O FastCGI, surge como uma alternativa, quando se utiliza do conceito de "processos persistentes" evitando o *overhead* causado com as constantes criações e destruições de processos, como visto no capítulo 2. O uso de APIs de servidores é outra técnica, que possui o inconveniente de ser muito específico para cada servidor. *Scripts* embutidos no

código HTML são úteis para a geração de páginas dinâmicas mais simples. Inserem, contudo, uma carga adicional ao servidor que deve interpretar cada página a cada consulta do cliente. Uma técnica mais avançada é o uso de Java, tanto na porção cliente, via *applets*, como na porção servidora, via *servlets*, onde se beneficia com a criação de uma aplicação extremamente portável via *byte codes* Java, além de uma linguagem orientada a objetos, que fornece características como modularidade e reutilização de código.

Finalizando, no capítulo 6 foram apresentadas técnicas de distribuição de carga, necessárias quando um único servidor ou a rede de suporte, não está conseguindo atender a demanda de requisições. Torna-se assim necessária a utilização de arquiteturas e algoritmos de escalonamento para se distribuir a carga de forma homogênea entre os servidores.

A arquitetura de distribuição de carga pode ser inicialmente classificada como Local ou Global. O primeiro grupo se concentra na distribuição de carga em um *cluster* cujos nós estão localizados geograficamente próximos. O segundo abriga o conceito de servidores localizados em pontos estratégicos, geograficamente distantes, de forma a se tornar uma solução mais escalável.

A entidade mais importante na arquitetura é o distribuidor, cuja função é repassar pedidos de conexão para os servidores reais. Essa entidade pode ser representada por vários componentes. O servidor DNS autoritativo talvez seja o mais comum em uma arquitetura onde não se ocultam os endereços IPs dos servidores, por ser bastante fácil de implementar e possuir um baixo custo. Todavia, não oferece um método de controle de carga refinado.

Entre as classificações mais específicas, pode-se citar a distribuição de carga que leva em conta informações apenas da camada de transporte (nível 4), ou aquela que leva em conta informações da camada de aplicação (nível 7). O primeiro grupo tem um processamento mais eficiente, visto que os pacotes não necessitam subir até o nível de aplicação para serem analisados. Todavia, somente o segundo grupo pode ter uma avaliação baseada no conteúdo do nível de aplicação.

As duas técnicas de distribuição de carga, local e global, podem ainda ser acopladas de forma a criar a arquitetura de *clusters* distribuídos geograficamente. Tem-se assim dois níveis de escalonamento: o primeiro para se determinar qual *cluster* irá atender o pedido, baseado em geral na localização geográfica, e o segundo para se determinar qual servidor do *cluster* irá atender o pedido, baseado na carga ou no conteúdo.

Este foi apenas um apanhado geral dos principais tópicos levantados neste trabalho. Ao longo do estudo foi interessante notar como detalhes inicialmente irrelevantes ou ocultos surgiram como um importantes pontos de otimização quando observados em suas interações, o que implicou em uma descrição mais extensa do que se plajenara no início. Ao final do trabalho, reforça-se o fato de que questões de desempenho na porção servidora de sistemas Web só podem ser analisadas corretamente quando se observam todas as "engrenagens" em funcionamento e se possui um embasamento teórico suficiente para se identificar pontos de otimização e propor modificações.

7.1 Resultados práticos obtidos

O conhecimento adquirido no desenvolvimento deste trabalho foi diretamente aplicado no desenvolvimento do site de alta demanda da Comissão Permanente para os Vestibulares, que possui a peculiaridade de picos de carga altíssimos em momentos de divulgação das listas dos aprovados.

As primeiras divulgações em meados de 1996 foram feitas em um hardware bastante ultrapassado mesmo para a época: uma Sun Sparc 2 com apenas 32MB, com Solaris 2.5.1, que sucumbiu à carga. A aplicação baseava-se em acesso a banco de dados MySQL. Posteriormente foi substituída por aplicações CGIs bastante otimizadas, que se utilizavam de mecanismos como índices e buscas binárias, construídas especialmente para a aplicação. A utilização das CGIs tornou o impacto sobre o hardware mais leve. Mesmo assim era patente que o ponto de latência era o hardware ultrapassado.

Consultando diferentes profissionais da área, que tinham idéia da carga gerada pela divulgação dos resultados dos vestibulares, sobre qual o hardware mais indicado, as respostas sempre indicavam servidores de médio/grande porte, com preços beirando os 50 mil dólares.

Com o apoio do Centro de Computação da Unicamp, em especial de Fábio Mengue, posteriores divulgações foram feitas utilizando-se a "obelix", uma máquina de uso comunitário da Unicamp, que é uma Sun Enterprise 5000, 6 processadores, e 2GB de RAM. Um hardware inquestionável, mesmo para os padrões atuais. Porém, as primeiras divulgações nesta máquina não tiveram o resultado esperado, e houve uma latência considerável. Em uma das ocasiões, o resultado foi divulgado às 9 horas, e até às 12 horas as consultas não eram atendidas. Embora a obelix não fosse uma máquina inteiramente dedicada ao serviço Web, a carga gerada por outras

aplicações era irrisória frente ao pico de divulgação das listas de vestibulares. Além disso, a maioria desses serviços era desabilitado nestes momentos. Neste ponto, era claro que o gargalo não era o hardware, e sim o software. Isso se comprovou quando da substituição do software servidor Web, do Netscape Enterprise para Apache. Recompilando-se o Apache, configurando-se adequadamente o parâmetro `MaxClients` e outros, a obelix serviu posteriores divulgações com facilidade. O tempo de resposta durante o pico de carga era praticamente idêntico àquele que se tinha durante os dias normais. Não houve mudança quanto aos programas utilizados para geração de páginas dinâmicas. Contudo, embora se tivesse obtido um sistema funcional, o preço do hardware envolvido era um obstáculo à Comissão Permanente para os Vestibulares.

Posteriores experimentos foram feitos utilizando-se hardware mais modesto, em geral plataforma Intel, com configurações que se aproximam daquelas utilizadas por desktops. Utilizou-se em grande parte do sistema Linux. Durante o processo, através de ferramentas de *stress*, foram-se encontrando os gargalos referentes ao Sistema Operacional, como relatados no capítulo 2, além de se aplicar outras otimizações também relatadas neste trabalho. Como resultado, chegou-se à seguinte configuração (máquina batizada como "orion"): Pentium III 750 monoprocessado, 512 MB de RAM e discos SCSI, rodando Linux e Apache, ambos customizados. Esta configuração, foi colocada em produção, e embora modesta, foi capaz de atender sozinha toda a carga de divulgação, que em geral ultrapassa os 100 hits por segundo com extensiva geração de páginas dinâmicas.

Assim, pôde-se comprovar na prática a grande importância da otimização do software em servidores de alta demanda principalmente no orçamento utilizado para a implementação deste serviço.

Atualmente, por precaução configuramos duas servidoras com conteúdo idêntico: a orion e a obelix. Inicialmente a carga é toda desviada para a orion. O tráfego é monitorado em tempo real e caso o tempo de resposta caia, o tráfego seria desviado para a obelix, através de redirecionamento via links, técnica explicitada no capítulo 6. Todavia, isso nunca ocorreu e somente a orion tem conseguido atender à demanda.

As tabelas seguintes apontam as principais customizações executadas na orion.

Parâmetro	Valor Final	Referência
NR_TASKS	4096	Seção 2.5.3
NR_OPEN	8192	Seção 2.6.1.5
/proc/sys/fs/inode-max	24576	Seção 2.6.1.3
/proc/sys/fs/file-max	8192	Seção 2.6.1.3
/proc/sys/vm/bdflush	70, 1200, 64,,20,60,90	Seção 2.6.3.3
/proc/sys/vm/kswapd	512, 32, 16	Seção 2.7.3.2
/proc/sys/net/ipv4/tcp_max_syn_backlog	2048	Seção 3.3.1.2
/proc/sys/net/ipv4/tcp_syncookies	1	Seção 3.3.3
/proc/sys/net/ipv4/ip_no_pmtu_discovery	1	Seção 3.3.5
/proc/sys/net/ipv4/tcp_sack	1	Seção 3.5.5
/proc/sys/net/ipv4/tcp_fack	1	Seção 3.5.5
/proc/sys/net/ipv4/tcp_keepalive_intvl	30	Seção 3.6.3
/proc/sys/net/ipv4/tcp_keepalive_probes	5	Seção 3.6.3
/proc/sys/net/ipv4/tcp_fin_timeout	60	Seção 3.6.4
/proc/sys/net/ipv4/tcp_window_scale	1	Seção 3.7.1
/proc/sys/net/ipv4/tcp_tstamp_if_wscale	1	Seção 3.7.2

Tabela 7.1: Principais parâmetros customizados relacionados ao Sistema Operacional (Linux) e ao TCP.

Parâmetro	Valor Final
StartServers	10
MinSpareServers	15
MaxSpareServers	30
MaxClients	1500
HARD_SERVER_LIMIT	3000
MaxRequestsPerChild	15000
KeepAliveTime	on

Tabela 7.2: Principais parâmetros customizados relacionados ao Apache. Referência: Seção 5.3.4

Parâmetro	Valor Final
KeepAliveTimeOut	30
MaxKeepAliveRequests	200
ListenBacklog	1256
LimitRequestBody	10240
LimitRequestFields	50
Options Multiviews	on
Options FancyIndexing	off
HostnameLookups	off

Tabela 7.2: Principais parâmetros customizados relacionados ao Apache. Referência: Seção 5.3.4

7.2 Trabalhos Futuros

Como possíveis trabalhos futuros relacionados diretamente a este, pode-se citar o estudo de diferentes mecanismos de simulação de carga, métricas e questões de hardware.

Mecanismos de distribuição de carga se monstram como um campo de pesquisa bastante promissor, como observado neste estudo.

Outro ponto importante é o estudo do impacto de técnicas voltadas à segurança no desempenho. Segurança e desempenho têm tido importâncias similares na usabilidade da arquitetura Web nas mais variadas aplicações. Dentre os tópicos, podem-se citar: o impacto de *firewalls*, via regras de filtragem, ou ainda o emprego de aceleradores criptográficos, negociação de chaves, dentre outros.

Bibliografia

[1] AKAMAI Inc. Página Principal. Disponível na Internet: http://www.akamai.com/. Outubro 2001.

[2] ALLDAS. **Alldas.de Defaced Archives** . Disponível na Internet: http://defaced.alldas.de/. Setembro 2001.

[3] ALVES, Maria Bernardete; ARRUDA, Suzana Margareth. **Como Fazer Referências: bibliográficas, eletrônicas e demais formas de documentos.** Disponível na Internet. http://bu.ufsc.br/framerefer.html. Maio 2001.

[4] APACHE SOFTWARE FOUNDATION. **Apache Multi-Processing Modules**. Disponível na Internet: http://httpd.apache.org/docs-2.0/mpm.html. Janeiro 2002.

[5] APACHE SOFTWARE FOUNDATION. **The Jakarta Project**. Disponível na Internet. http://jakarta.apache.org/. Janeiro 2002.

[6] APACHE SOFTWARE FOUNDATION. **The Apache/Perl Integration Project**. Disponível na Internet. http://perl.apache.org. Janeiro 2002.

[7] APACHE SOFTWARE FOUNDATION. **Apache/Perl Performance Tuning**. Disponível na Internet. http://perl.apache.org/guide/performance.html. Janeiro 2002

[8] APACHE SOFTWARE FOUNDATION. **Descriptors and Apache**. Disponível na Internet: http://httpd.apache.org/docs/misc/descriptors.html. Agosto 2001.

[9] APACHEWEEK. **Inside Apache on Windows.** 1998. Disponível na Internet. http://www.apacheweek.com/issues/97-11-28. Março 2001.

[10] APPLE Inc. **The grand design for Mac OS X.** Disponível na Internet. http://www.apple.com/hotnews/articles/2000/05/wwdc_keynote/part4.html. Janeiro 2002.

[11] BARM, Moshe. Tuning Linux for maximum performance. **Byte.com**. Agosto 2000. Disponível na Internet. http://www.byte.com/documents/s=429/BYT20000829S0006/.

[12] BARTON, Paul. **Linux Threads as seen in NT Magazine**. 08 Dezembro 1998. Disponível na Internet. http://www.cs.belsinki.fi/linux/linux.kernel/year_1998/1998_49/0767.html. 26 Setembro 2000.

[13] BELL LABS. **The Creation of the UNIX Operating System**. 2000. Disponível na Internet. http://www.bell-labs.com/history/unix/. 10 Julho 2001.

[14] BERGSTEN, Hans. **Servlets Are for Real!**.Web Developers Journal. Julho 1998. Disponível na Internet. http://webdevelopersjournal.com/columns/java_servlets.html. Agosto 2001.

[15] BERGSTEN, Hans. **An Introduction to Java Servlets**. Web Developers Journal. Março 1999. Disponível na Internet. http://webdevelopersjournal.com/articles/intro_to_servlets.html. Outubro 2001.

[16] BERNERS-LEE, T. **Tim Berners-lee - Página pessoal**. Disponível na Internet: http://www.w3.org/People/Berners-Lee/. Julho 2001.

[17] BERNERS-LEE, T. FIELDING, R., FRYSTYK, H.. **Hypertext Transfer Protocol -- HTTP/1.0**. RCF 1945. Maio 1996.

[18] BERNERS-LEE, T. FIELDING, R., FRYSTYK, H.. **Hypertext Transfer Protocol -- HTTP/1.1**. RCF 2068. Janeiro 1997.

[19] BESTAVROS, Azer. **Admission Control and Scheduling for High - Performance WWW Servers**. Maio 1998. Disponível na Internet: http://www.cs.bu.edu/techreports/pdf/1997-015-web-admission-control-and-sched.pdf. Setembro 2000.

[20] BEVERIDGE, Tony. **High Performance ISA/NSAPI Web Programming**. 1.ed. Editora Berkeley. 1998. 622p.

[21] BOVET, Daniel P., CESATI, Marco. **Understanding the Linux Kernel**. O Reilly. 2001. 684p.

[22] BOWMAN, Ivan. **Concrete Architecture of the Linux Kernel**. Fevereiro 1998. Disponível na Internet: http://plg.uwaterloo.ca/~itbowman/CS746G/a2/. Abril 2001.

[23] BRADEN, R.. **Extending TCP for Transactions -- Concepts**. RFC 1379. Novembro 1992.

[24] BRAN, Ray. **SMP Scalability Comparisons of Linux Kernels 2.2.14 and. 2.3.99. in Proceedings of the USENIX Technical Conference**. ALS. Outubro 2000. Disponível na Internet. http://www.usenix.org./publications/library/proceedings/als2000/bryntscale.html. 29 Maio 2001.

[25] BRISCO, T.. **DNS Support for Load Balancing. RFC 1794.** Abril 1995.

[26] BRONZESOFT. **IP SCFW, Syn Cookies Firewalls**. 1998. Disponível na Internet. http://www.bronzesoft.org/projects/scfw/. 20 Novembro 2000.

[27] BRYHNI, Haakon. **A comparison of load balancing techiniques for scalable Web Servers. IEEE Network.** Agosto 2000.

[28] CAO, Pei. **Persistent Connection Behavior of Popular Browsers**. Dezembro 1998. Disponível da Internet: http://www.cs.wisc.edu/~cao/papers/persistent-connection.html. Agosto 1999.

[29] CARNEGIE MELLON UNIVERSITY**. The Mach Project Home Page**. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html. Agosto 2001.

[30] CARNEGIE MELLON UNIVERSITY. **Enabling High Performance Data Transfers on Hosts**. Novembro 1999. Disponível na Internet: http://www.psc.edu/networking/perf_tune.html. Julho 2000.

[31] CERT. **Advisory CA-2000-21 Denial-of-Service Vulnerabilities in TCP/IP Stacks**. Dezembro 2000. Disponível na Internet. http://www.cert.org/advisories/CA-2000-21.html. Março 2001.

[32] CHERITON, David. **VMTP: VERSATILE MESSAGE TRANSACTION PROTOCOL.** RFC 1045. Fevereiro de 1988.

[33] CISCO SYSTEMS. **Cisco Local Director.** Disponível na Internet: http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml. Julho 2001.

[34] CISCO SYSTEMS. **Cisco Distributed Director.** Disponível na Internet: http://www.cisco.com/warp/public/cc/pd/cxsr/dd/index.shtml. Julho 2001.

[35] CIULLO, S., HINOJOSA, D. **HP-UX Kernel Tuning and Performance Guide**. Disponível na Internet: http://www.hp-partners.com/edaweb_public/html/technical_support/tuning.html. Junho 1999.

[36] COCKCROFT, Adrian., PETIT, Richard. **Sun Performance Tuning. Java and the Internet**. 2.ed. Prentice Hall. 587p.

[37] COHEN, E., KAPLAN, H. **Managing TCP Connections under persistent HTTP**. AT&T. Disponível na Internet: http://www.unizh.ch/home/mazzo/reports/www8conf/pdf/pd1.pdf. . Julho 2001.

[38] COLAJANNI, Michele. **Analysis of Task Assignment Policies in Scalable Distributed Web-server Systems. in IEEE Transactions on Parallel and Distributed Systems.** Junho 1998.

[39] COLAJANNI,Michele, YU, Philip S. CARDELLNI, Valeria. **Dynamic Load Balancing in Geographically Distributed. in the Proceedings of IEEE 18th Int. Conf. on Distributed Computing Systems (ICDC'98).**

[40] COLAJANNI,Michele, YU, Philip S. CARDELLNI, Valeria. **Redirection Algorithms for Load Sharing in Distributed Web-server Systems. in the Proceedings of IEEE 19th Int. Conf. on Distributed Computing Systems (ICDC'99).**

[41] COLAJANNI,Michele, YU, Philip S. CARDELLNI, Valeria. **Dynamic Load Balancing on Web-server System. IEEE Internet Computing, vol 3**. Junho 1999.

[42] COLAJANNI,Michele, YU, Philip S. CARDELLNI, Valeria. **Geographic Load Balancing for Scalable Distributed Web Systems. In IEEE Proc. of Mascots 2000.**

[43] COLAJANNI,Michele, YU, Philip S. CARDELLNI, Valeria. **The State of the Art in Locally Distributed Web-server Systems. IBM Research Report.** Outubro 2001. Disponível na Internet: http://domino.watson.ibm.com/library/cyberdig.nsf/papers?SearchView&Query=RC22209&SearchMax=10

[44] COMER, Douglas E., STEVENS, David L. **Interligação em Rede com TCP/IP**. Volume 2. 1.ed. Editora Campus. 1999. 591p.

[45] COULOURIS, George**. Distributed Systems: Concepts and Design**. 3.ed. Addison-Wesley. Agosto 2000. 672p.

[46] CROVELLA E., CARTER L. **Server selection using dynamic path characterization in wide-area networks. In Proc. of IEEE Infocom 1997**. pages 1014–1021. 1997.

[47] DAMANI, O. **ONE-IP: Techniques for hosting a service on a cluster of machines. Journal of Computer Networks and ISDN Systems.** Setembro 1997.

[48] DARBY, Chád. **Migrating CGI Scripts to Java Servlets**. Java Developers Journal. Janeiro 1998. Disponível na Internet. http://www.j-nine.com/pubs/jdj/. Outubro 2001.

[49] DIETZ, Hank. **Linux Parallel Processing HOWTO**. Janeiro 1998. Disponível na Internet. http://www.linuxdoc.org/HOWTO/Parallel-Processing-HOWTO.html. 29 Maio 2001.

[50] EGEVANG, K.. **The IP Network Address Translator (NAT). RFC 1631.** Maio 1994.

[51] ENGELS CHALL, Raft S. **Apache Desktop Reference**. Disponível na Internet: http://www.apacheref.com. Julho 2000.

[52] ETHREAL. **Site do Ethreal.** Disponível na Internet. http://www.ethereal.com/. Outubro 2001.

[53] F5 NETWORKS Inc. **BIG-IP Controller.** Disponível na Internet: http://www.f5networks.com/f5products/bigip/index.html. Janeiro 2002.

[54] F5 NETWORKS Inc. **3-DNS Controller**. Disponível na Internet: http://www.f5networks.com/f5products/3dns/index.html. Janeiro 2002.

[55] F5 NETWORKS Inc. **GLOBAL-SITE Controller**. Disponível na Internet: http://www.f5networks.com/f5products/globalsite/index.html. Janeiro 2002.

[56] FASTCGI. **FastCGI.com - Página Principal.** Disponível na Internet. www.fastcgi.com. Outubro 2001.

[57] FILCKENGER, Rob. **Speeding up Linux using hdparm**. Disponível na Internet. http://linux.oreillynet.com/pub/a/linux/2000/06/hdparm.html. Janeiro 2000.

[58] FLOYD, Sally. **SACK TCP**. Agosto 1999. Disponível na Internet: http://www.aciri.org/floyd/sacks.html. Julho 2001.

[59] GAUDET, Dean. **Apache Performance Notes**. Disponível na Internet: http://httpd.apache.org/docs/misc/perf_tuning.html. Julho 2000.

[60] GILLIGAN, Bob. **Configuring the TCP Listen Backlog for Web Servers in Solaris 2.4**. Sunsoft 1995. Disponível na Internet. http://archive.ncsa.uiuc.edu/InformationServers/Performance/Solaris/backlog.txt. 19 Novembro 2000.

[61] GLASS, Graham. **UNIX for Programmers and Users**. **A Complete Guide**, Prentice Hall. 1993. 633p.

[62] HADDAD, Ibrahim. **Apache 2.0: The Internals of the New, Improved "A PatCHy".** Agosto2001. Disponível na Internet: http//www2.linuxjournal.com/articles/misc/0050.html.

[63] HASENSTEIN, Michael. **IP Network Address Translation**. Disponível na Internet: http://www.suse.de/~mha/linux-ip-nat/diplom/. Setembro 2001.

[64] HEIDEMANN, J., OBRACZKA, K., TOUCH, J. **Modeling the Performance of HTTP Over Several Protocols**. **in IEEE/ACM Transactions on Networking**. Diponível na Internet: http://www.isi.edu/people/johnh/papers/heidemann96a.html. Outubro 1997.

[65] HEINZ HEISE. **Development Tree of UNIX Networking Code**. Novembro 1998. Disponível na Internet. http://leb.net/hzo/ioscount/ix_unix_net_pic.html. 10 Julho 2001.

[66] HOE, Janey C. **Improving the start-up behavior of a congestion control scheme for TCP**. MIT.

[67] HUGHES, Chris. **How Does a Web Server Differ From an Application Server?. 2001.** Disponível na Internet. http://webcompare.internet.com/webbasics/. Julho 2001.

[68] IBM. **AIXL Strong Linux affinity for flexible solutions**. Disponível na Internet. http://www-1.ibm.com/servers/aix/overview/linux.html. Agosto 2001.

[69] iPLANET. **iPlanet Web Server Documentation.** Disponível na Internet. http://docs.iplanet.com/docs/manuals/enterprise.html. Outubro 2001

[70] JACOBSON, V., BRADEN R.. **TCP Extensions for Long- Delay Paths**. RFC 1072. Outubro 1988.

[71] JACOBSON, V., BRADEN R., BORMAN D.. **TCP Extensions for High Performance**. RFC 1323. Maio 1992

[72] JOUBERT, Philippe., KING Robert. **High-Performance Memory Based Web Servers: Kernel and User-Space Performance**. **in Proceedings of the USENIX 2001. Annual Technical Conference**. Junho 2001.

[73] KANAL, Dua. **Balance your Web Server's Load**. Julho 2001. Disponível na Internet. http://pcquest.ciol.com/content/technology/101071402.asp. Setembro 2001.

[74] KILLELEA, Patrick. **Web Performance Tuning**. 1.ed. O'Reilly. 1998. 351p.

[75] KRISTOL, D., MONTULLI, L. **HTTP State Management Mechanism**. RFC 2109. Fevereiro 1997.

[76] KUZNETSOV, Alexey. Fonte de instalação do Linux: /usr/src/linux/Documentation/networking/ip_sysctl.txt. Novembro 2000.

[77] LAI, Kevin. BAKER, Mary. **A Performance Comparison of Unix Operating Systems on a Pentium**. In Procedings of the USENIX 1996 Annual Technical Conference. Janeiro 1996.

[78] LAM, Terance L. **Improving File System Performance by Striping.** Março 1992. Disponível na Internet. http://www.nas.nasa.gov/Research/Reports/Techreports/1992/PDF/rnd-92-006.pdf. Julho 2001.

[79] LAY, James R. **Keeping the 400Cb. Gorilla at Bay: Optimizing Web Performance**. Maio 1996. Disponível na Internet: http://eunuch.ddg.com/LIS/CyberHornsS96/j.rubarth-lay/paper.html. Agosto 2000.

[80] LEROY, Xavier. **The Linux Threads Library**. Disponível na Internet. http://pauillac.inria.fr/~xleroy/linuxthreads/. Setembro 2001.

[81] LEVER, Chuck., ERIKSEN, Marius. **An Analysis of the TUX Web Server**. Novembro 2000.

[82] **LINUX PERFOMANCE TUNING**. Disponível na Internet. http://linuxperf.nl.linux.org. 04 Setembro 2000.

[83] MacVITTIE, Don. **Tuning Your Apache Web Server. Sun How-Tos**. Outubro 2001. Disponível na Internet: http://dcb.sun.com/practices/howtos/tuning_apache.jsp. Outubro 2001.

[84] MARIMBA Inc. **The HTTP Distribution and Replication Protocol.** http://www.w3.org/TR/NOTE-drp-19970825.html. Novembro 2001.

[85] MATHIS, M., MAHDAVI, J., FLOYD, S., and ROMANOW, A.. **TCP Selective Acknowledgement Options.** RFC 2018. Abril 1996.

[86] MATHIS, M. and MAHDAVI, J.. **Forward acknowledgement: Refining TCP Congestion control**. **In Proceedings of the ACM SIGCOMM**. Agosto 1996.

[87] MAURO, Jim. **Inside Solaris - Peeling back the process layers**. **Part 1.** Agosto 1998. Disponível na Internet. http://sunsite.uakom.sk/sunworldonline/swol-08-1998/swol-08-insidesolaris.html. 30 Maio 2001.

[88] MENASCE, DANIEL A. **CAPACITY PLANNING FOR WEB PERFORMANCE**. 1.ed. 1998. PRENTICE HALL. 450p.

[89] MENTRÉ, David. **Linux SMP HOWTO**. Outubro 2000. Disponível na Internet. http://www.linuxdoc.org/HOWTO/SMP-HOWTO.html. 29 Maio 2001.

[90] MICROSOFT. **IIS: Information on the "Nimda" Worm**. 2001. Disponível na Internet: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/topics/Nimda.asp. Novembro 2001.

[91] MOGUL, J., DEERING, S..**Path MTU Discovery**. RFC 1191. Novembro 1990.

[92] MOGUL, J., PADMANABHAN, N. **Improving HTTP Latency**. Disponível na Internet: http://archive.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html. Julho 1999.

[93] MOGUL, J., KRISTOL,M. **Key Differences between HTTP/1.0 and HTTP/1.1**. AT&T Labs. Dezembro 1998.

[94] MOURANI, Gerhard. **Securing and Optmizing Linux Red Hat Edition - A Hands on Guide**. Agosto 2000. Disponível na Internet. http://www.linuxdoc.org/guides#securing_linux. Setembro 2000.

[95] NASA Glenn Research Center. **Ongoing TCP Research Related to Satellites**. RFC 2760. Fevereiro 2000.

[96] NCSA. **Server Side Includes (SSI)**. Disponível na Internet. http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html.

[97] NEMETH, Evi. SNYDER G.. SCOTT, Seebass. HEIN, Trent. **UNIX System Administration Handbook**. 2.ed. Prentice Hall. 1995. 779p.*

[98] NETCRAFT. **Site da Netcraft.** Disponível na Internet: http://www.netcraft.com/. Agosto 2001.

[99] OPERATINGSYSTEMS.NET. **Operating System Technical Comparison**. Maio 2001. Disponível na Internet. http://www.operatingsystems.net/. 10 Julho 2001.

[100] PAI, Vivek., DRUSCHEL, Peter. **Flash: An efficient and portable Web Server**. in Proceedings of the USENIX 1999. Annual Technical Conference.

[101] PCGUIDE. **Logical Geometry.** Disponível na Internet. http://www.pcguide.com/ref/hdd/geom/geomLogical-c.html. Maio 2001.

[102] PEARSON, John**. Using "Huiac" Debian Linux CDs. Planning the Partitions on your Hard Disk**. Abril 2000. Disponível na Internet. http://george.mdt.net.au/~john/cds/htm/hd-plan.htm. Dezembro 2000.

[103] PHP Net. **PHP HyperText PreProcessor**. Disponível na Internet. http://www.php.net/. Janeiro 2002.

[104] POLISERV. **Data Replication for High Availability Web Server Clusters**. Disponível na Internet: http://www.itpapers.com/cgi/PSummaryIT.pl?paperid=6177&scid=663. Setembro 2001.

[105] PRANEVICH, Joe. Wonderful World of Linux 2.4. **Linux Today**. Janeiro 2001. Disponível na Internet. http://linuxtoday.com/news_story.php3?ltsn=2000-11-23-017-06-NW-LF-KN. 07 Maio 2001.

[106] RED HAT. Disponível na Internet: http://www.redhat.com/products/software/linux/tux/. Julho 2001.

[107] RESONATE Inc. **Resonate Central Dispatch.** Disponível na Internet: http://www.resonate.com/solutions/products/central_dispatch/cd_data_sheet.php. Outubro 2001.

[108] RIEL, Rick Van. **Discussion with Matt Dillon** (**Linux Virtual Memory**). Maio 2000. Disponível na Internet: http://mail.nl.linux.org/linux-mm/2000-05/msg00419.html.Setembro 2000.

[109] RITCHIE, Denis M.; THOMPSON, Ken. **The UNIX Time-Sharing System**. 1974. Disponível na Internet. http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html. 10 Julho 2001.

[110] ROSS, Keith. TCP **Congestion Control**. Disponível na Internet: http://www.seas.upenn.edu/~ross/book/transport-layer/congestion.html. Maio 2000.

[111] ROSS, Keith W., KUROSE, James. **The World Wide Web: HTTP**. 2000. Disponível na Internet: http://www-net.cs.umass.edu/kurose/apps.html. Maio 2000.

[112] RUBINI, Alessandro. Dynamic Kernels: Modularized Device Drivers. **Linux Journal**. 1999. Disponível na Internet. http://www2.linuxjournal.com/lj-issues/issue23/1219.html. Julho 2001.

[113] SCHEMERS, R.J. **lbmnamed: A load balancing name server in Perl. in Proceedings os 9th System Administration Conference (LISA'95).**

[114] SOCOLOFSKY, T., KALE, C.. **A TCP/IP Tutorial**. RFC 1180. Janeiro 1991.

[115] STEPHENSON, Michael. **Internet Information Server 4.0. Tuning Parameters for High-Volume Sites.** 1998. Disponível na Internet: http://www.dnscomputing.com/class_files/163/Tuning%20IIS%204.0.htm. Agosto 2001.

[116] STEVENS, Richard W., WRIGHT, Gary R. **TCP/IP Illustrated.** Volume 1. 1.ed. 576p.

[117] STEVENS, Richard W., WRIGHT, Gary R. **TCP/IP Illustrated**. Volume 2. 1.ed. Addison-Wesley. 1994. 1173p.

[118] STEVENS, Richard W., WRIGHT, Gary R. **TCP/IP Illustrated.** Volume 3. 1.ed. 412p.

[119] STEVENS, Richard W. **TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithims**. RCF 2001. Janeiro 1997.

[120] SUN Microsystems Inc. **The Solaris memory System. Sizing, Tools and Architecture.** Maio 1998. Disponível na Internet: http://www.sun.com/sun-on-net/performance/vmsizing.pdf. 07 Julho 2000.

[121] SUN Microsystems. **TCP Basics - Slow Start and Delayed ACK**. Disponível na Internet: http://www.sun.com/sun-on-net/performance/tcp.dowstart.html. Agosto 1999.

[122] SUN Microsystems. **Java Servlet Technology: The Power Behind the Server**. Disponível na Internet. http://java.sun.com/products/servlet/index.html. Junho 2001.

[123] TANENBAUM, Andrew S. . **Modern Operating Systems**. Prentice Hall 1992. 728p.

[124] TANENBAUM, Andrew S. **Computer Networks**. 3.ed. Prentice Hall. 1996. 814p.

[125] THOMPSON, Evan. **Configuring kHTTPd**. Disponível na Internet: http://www.linuxnewbie.org/nht/intel/webserving/khttpd.html. Agosto 2001.

[126] VAHALIA, Uresh. **Unix Internals. The New Frontiers**. Prentice Hall. 1996. 601p.

[127] VEPSTAS, Linas. **RAID Solutions for Linux**. Disponível na Internet. http://linas.org/linux/raid.html. Julho 2001.

[128] VEPSTAS, Linas. **Linux Network Address Translation**. Novembro 1998. Disponível na Internet: http://www.linas.org/linux/load.html. Setembro 2001.

[129] VÖCKLER, Jean-s. **Solaris 2.x. Tuning your TCP/IP Stack and more**. Dezembro 2000. Disponível na Internet. http://www.sean.de/Solaris/tune.html. Janeiro 2002.

[130] WAINWRIGHT, Peter. **Professional Apache**. 1.ed. WROX PRESS. 616p.

[131] YAGER, Tom. Six Unix OS flavors run the gamut. **InfoWorld Magazine.** Disponível na Internet: http://www.itworld.com/Comp/2149/swol-0119-flavors/. Janeiro 2001.

[132] YANG Chu-Sing. **Web Server Clusters. in Proceedings of the 2nd USENIX Symposium on Internet Technologies & Systems**. Outubro 1999.

[133] ZENÁNEK,Petr. **Performance and Tuning of the Unix Operating System**.