TIAGO SEMPREBOM

DIFERENCIAÇÃO DE SERVIÇOS EM SERVIDORES WEB BASEADA EM ESCALONAMENTO ADAPTATIVO E CONTROLE DE ADMISSÃO

FLORIANÓPOLIS
2007

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO
EM ENGENHARIA ELÉTRICA

DIFERENCIAÇÃO DE SERVIÇOS EM SERVIDORES WEB BASEADA EM ESCALONAMENTO ADAPTATIVO E CONTROLE DE ADMISSÃO

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

TIAGO SEMPREBOM

Florianópolis, Março de 2007.

DIFERENCIAÇÃO DE SERVIÇOS EM SERVIDORES WEB BASEADA EM ESCALONAMENTO ADAPTATIVO E CONTROLE DE ADMISSÃO

TIAGO SEMPREBOM

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Sistemas*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'

Carlos Barros Montez, Dr.
Orientador

Nelson Sadowski, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Carlos Barros Montez, Dr.
Presidente

Rômulo de Oliveira, Dr.
Co-orientador

César Albenes Zeferino, Dr.

Frank Augusto Siqueira, PhD.

Ubirajara Franco Moreno, Dr.

Aos meus amados pais, Dersi e Elizabeth
e à minha namorada Alline ...

AGRADECIMENTOS

No decorrer desse trabalho tive a ajuda e compreensão de muitas pessoas. Agradeço primeiramente a Deus, por me dar à oportunidade de chegar até aqui e por estar comigo em todos os momentos da minha vida, e agradeço também a essas pessoas:

Meus pais, pela educação que me proporcionaram, me concedendo muito além do que tiveram. Agradeço também a força por eles prestada durante todas as fases de concepção deste trabalho.

A minha namorada, pelo apoio mesmo estando longe, pela compreensão e paciência, me ajudando sempre que necessário e aos seus pais: Dulce e Luís.

Meus irmãos Cristiano e Cléverson, que me ajudaram sempre que foi preciso.

A meu orientador Prof. Carlos Montez, pela ajuda, amizade e constante disponibilidade durante a realização desta obra.

A meu co-orientador, Prof. Rômulo Silva de Oliveira, pela ajuda prestada.

Agradeço aos membros dessa banca examinadora que contribuíram na revisão deste trabalho, colaborando com sugestões.

Agradeço, também ao Programa de Pós-Graduação em Engenharia Elétrica, pela oportunidade de desenvolver minha pesquisa junto à UFSC.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelos meses de concessão de bolsa de mestrado, sem a qual não conseguiria realizar este trabalho.

A todos os professores do Departamento de Automação e Sistemas (DAS).

Agradeço aos participantes da lista de discussão de desenvolvedores oficial do Apache, pelos esclarecimentos prestados.

Um agradecimento especial aos amigos, que fiz durante o mestrado, principalmente os do Laboratório de Controle e Microinformática (LCMI): André Piazza, Augusto Carlson, Christiano Casanova, Douglas Bertol, Ebrahim El´yousself, Henry Lopes Salamanca, Jim Lau, Luciano Costa, Marcus Americano, Robson Costa, Robson Mitmam (Bob), Rodrigo Sumar, Ricardo Shima, Susan Möller, Underlea Corrêa e Warody Lombardi.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

DIFERENCIAÇÃO DE SERVIÇOS EM SERVIDORES WEB BASEADA EM ESCALONAMENTO ADAPTATIVO E CONTROLE DE ADMISSÃO

TIAGO SEMPREBOM

Março/2007

Orientador: Carlos Barros Montez, Dr
Área de Concentração: Controle, Automação e Informática Industrial
Palavras-chave: QoS, Serviços Diferenciados, Servidor Web, Apache
Número de Páginas: xv + 88

Com o paradigma de melhor esforço *(best-effort)* oferecido pela Internet atual, clientes de novos tipos de aplicações (VoIP e IPTV) têm que conviver com aplicações tradicionais da Internet como: e-mail e simples *web sites*. Contudo, verifica-se que nem todos os tipos de tráfego têm os mesmos requisitos de confiabilidade, segurança e urgência. Considerando-se a infra-estrutura de rede, existem algumas propostas nesse sentido, elaboradas sob a coordenação da IETF *(Internet Engineering Task Force)*. Entretanto, em nível de aplicação os esforços ainda são limitados, não sendo encontrados na grande maioria dos servidores *web*.

Esta dissertação traz um levantamento das principais tecnologias de provimento de QoS disponíveis. Um modelo de provimento de qualidade de serviço em servidores *web* também é proposto. O modelo incorpora uma abordagem de escalonamento adaptativo com política de atribuição dinâmica de prioridades e de escolha de versões precisas/imprecisas de páginas *web*. O modelo proposto também contempla aspectos de controle de admissão, procurando evitar a sobrecarga do sistema caso a demanda dos usuários cresça inesperadamente. Os resultados obtidos foram avaliados na presença e ausência do módulo de controle de admissão.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

DIFERENTIATION OF SERVICES IN WEB SERVERS BASED ON ADAPTIVE SCHEDULING AND ADMISSION CONTROL

TIAGO SEMPREBOM

March/2007

Advisor: Carlos Barros Montez, Dr.
Area of Concentration: Control, Automation and Industrial Informatics
Key words: QoS, Differentiated Services, Web Servers, Apache
Number of Pages: xv + 88

With the paradigm of best-effort offered by the current Internet, customers of new types of applications (VoIP and IPTV) have to coexist with traditional applications of the Internet, such as: e-mail and simple web sites. However, it is verified that not all types of traffic have the same requirements of dependability, security and urgency. In concern of the network infrastructure, there are some proposals in this direction, that are elaborated under the coordination of the IETF (Internet Engineering Task Force). However, in application level the efforts still are limited, not being found in the great majority of the web servers.

This work brings a survey of the main provisions technologies of QoS available. A model for providing quality of services in web severs is also considered. The model contemplates an adaptative scheduling approach with dynamic priority attribution and a mechanism for choosing precise/imprecise versions of web pages. The suggested model also contemplates aspects of admission control, seeking to avoid the overload of the system when the request of the users grows unexpectedly. The obtained results were evaluated in the presence and absence of the admission control module.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

APXS	*Apache eXtenSion toll*
API	*Application Programming Interface*
AF	*Assured Forwarding*
BA	*Behavior Aggregate*
DSCP	*Differentiated Services Codepoint*
DSO	*Dynamic Shared Object*
EF	*Expedited Forwarding*
FCF	*Fastest Connection First*
HTML	*HyperText Markup Language*
IETF	*Internet Engineering Task Force*
IP	*Internet Protocol*
MF	*Multi-Field*
MPM	*Multi-Processing-Modules*
PHB	*Per-Hop Behavior*
QoS	*Quality of Service*
RFC	*Request for Comments*
RSVP	*Resource Reservation Protocol*
SLA	*Service Level Agreement*
SRPT	*Shortest Remaining Processing Time*
TCA	*Traffic Conditioning Agreement*
TCP	*Transmission Control Protocol*
URL	*Universal Resource Locator*
VC	*Valor Cumulativo*
VQ	*Valor de Qualidade*
VoIP	*Voice over IP*
WFR	*Weight Fair Queue*

Capítulo 1

Introdução

A convergência das tecnologias de comunicação e computação tem propiciado o surgimento de novos tipos de aplicações e serviços oferecidos pela Internet, contribuindo substancialmente para o seu crescimento. Atualmente várias transações comerciais, antes realizadas apenas com a presença do cliente ou por telefone, agora podem ser realizadas com a facilidade e flexibilidade propiciada pelo uso da Internet. Exemplos desses novos serviços são: o comércio eletrônico, comércio móvel *(m-commerce)*, transações bancárias, telefonia sobre IP (VoIP), educação à distância, entretenimento, multimídia, entre outros. Todos esses serviços adotam a Internet como via de informação global e apresentam novas demandas em questões de desempenho, segurança e confiabilidade. Clientes dessas novas aplicações têm de conviver com clientes de aplicações convencionais da Internet, cujas necessidades de serviços são menos exigentes e podem ser atendidos por abordagens de melhor esforço *(best-effort)*.

Uma característica observada, relacionada a essas novas aplicações, é que suas demandas não podem aguardar o surgimento de uma nova tecnologia de Internet, tendo em vista que muitas dessas aplicações já estão sendo utilizadas. Os desenvolvedores dessas aplicações desejam utilizar código legado e soluções já existentes de banco de dados e servidores *web*, entre outros. Entretanto, se deparam com o fato de que essas aplicações foram concebidas tendo como meta o serviço de melhor esforço oferecido pela Internet.

Dentro deste cenário, motivado pelo crescimento do número de serviços oferecidos pela Internet, dentro do paradigma de "mesmo serviço para todos" da Internet atual, a comunidade científica desenvolve pesquisas na área de QoS (Qualidade de Serviço).

Com o reconhecimento da inadequação do modelo de melhor esforço oferecido pela Internet atual, é premente a necessidade de se adotar soluções que forneçam qualidade de

serviço (QoS) diferenciadas para os clientes dessas aplicações. Assim, organizações padronizadoras tais como a IETF *(Internet Engineering Task Force)* vêm buscando padrões abertos para fornecimento de QoS às aplicações, como as arquiteturas IntServ [19] e DiffServ [18], que oferecem qualidade de serviço nos elementos do núcleo da rede (roteadores). Entretanto, pouco adianta o emprego dessas soluções na infraestrutura da rede, caso as aplicações nos pontos finais também não se adeqüem a esta nova realidade. Um exemplo é o fato notório que as aplicações servidoras existentes (ex. servidores *web*) adotam a política de atendimento de requisições segundo sua ordem de chegada (FIFO).[1]

Segundo [60] o servidor *web*, o elemento principal na maioria das aplicações de comércio eletrônico, não apresenta mecanismos de diferenciação de serviços, tratando as requisições a essas páginas de maneira uniforme, sem levar em consideração a identidade dos clientes, as condições de carga do servidor ou o tipo de requisição.

1.1 Motivações

Com o objetivo de ressaltar as motivações deste trabalho, experimentos foram realizados utilizando o gerador de carga sintética *httperf* em um servidor *web* Apache. A Figura 1.1 apresenta o comportamento do servidor *web* submetido a uma sobrecarga (quando a carga excede a capacidade de processamento). É interessante observar que os tempos de resposta fim a fim obtidos (tempos de resposta medidos no cliente) crescem demasiadamente com a chegada de novas requisições, resultando assim em tempos impraticáveis para usuários humanos. Clientes que experimentam longos períodos para obterem resposta a suas requisições provavelmente deixarão o *site* em questão.

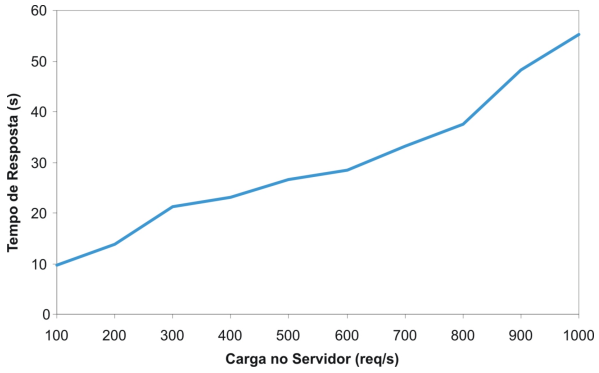


Figura 1.1: *Tempos de resposta de um sítio web de comércio eletrônico.*

[1] Alguns servidores *web* implementam LIFO – *Last In, First Out* – priorizando requisições recém-chegadas.

Segundo [27, 60], um servidor *web* deveria favorecer os clientes que pagam para terem suas requisições atendidas, concedendo-lhes as maiores prioridades do sistema em relação a outros clientes que não pagam pelo serviço. Similarmente os clientes com as maiores prioridades receberão quantidade de recursos proporcional à quantia paga pelo serviço.

1.1.1 Experimentos com Prioridades Estáticas

Uma abordagem imediata e direta para diferenciar serviços seria a de atribuir diferentes prioridades às requisições *web*.

Neste trabalho houve o interesse em avaliar o comportamento do Apache frente a requisições com diferentes prioridades (principalmente, considerando que o sistema operacional possui pontos de execução não preemptivos propensos ao problema de inversão de prioridades). Em alguns servidores *web*, como o Apache, é possível criar filas de prioridades para atendimento de requisições. Foram usados três computadores clientes A, B e C cujas requisições – enviadas para um outro computador com servidor *web* – foram classificadas (pelos seus endereços IP) com ordem decrescente de prioridades.

A Figura 1.2 apresenta resultados de experimentos, com o eixo x representando a passagem do tempo. Enquanto o servidor se encontrava com carga baixa, entre os tempos 1 a 6, este conseguia executar requisições dos três clientes. Após, esse período de tempo, o servidor em sobrecarga se dedicou a executar quase que exclusivamente as requisições mais prioritárias (Cliente A). Após o instante 25, aproximadamente, a demanda do Cliente A cessou de existir, e o servidor passou a executar requisições do Cliente B. Somente quando a demanda deste último cliente terminou que o servidor passou a atender requisições do Cliente C, cujas requisições são classificadas como menos prioritárias.

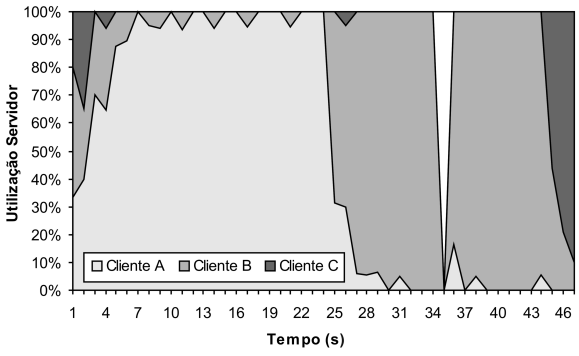


Figura 1.2: *Experimentos com prioridades estáticas.*

A Tabela 1.1 apresenta os tempos de resposta médios obtidos nos clientes. Os tempos de resposta do Cliente C foram bem maiores, devido aos tempos de espera na fila, aguardando o atendimento de requisições mais prioritárias.

Tabela 1.1: *Tempos de resposta fim a fim com prioridades estáticas.*

	Cliente A	**Cliente B**	**Cliente C**
Prioridade	0 (maior)	1	2
Tempos de resposta fim a fim	2378 ms	21487 ms	43610 ms

Os resultados obtidos indicam que a adoção de prioridades estáticas (prioridades fixas) não é indicada, pois pode levar as requisições com prioridades menores à inanição *(starvation)*.

1.1.2 Múltiplas Versões de Páginas *Web*

A adoção de prioridades dinâmicas, no entanto, apenas redistribui a capacidade de processamento entre as requisições. Muitas vezes em situações de sobrecarga é necessário adotar alguma abordagem complementar para reduzir a carga. Neste trabalho há interesse em utilizar técnicas de redução de carga no servidor, tal como controle de admissão e implementação de múltiplas versões de páginas *web*.

A técnica de múltiplas versões – técnica da computação imprecisa [39] que será vista na Seção 2.7.1 – consiste em modelar cada página *web* com mais de uma versão. Neste trabalho estamos modelando as páginas com duas versões: reduzida (ou precisa) e completa (ou precisa). A escolha por parte do servidor de uma versão reduzida para atender requisições de clientes é uma técnica efetiva para reduzir a sua carga.

Buscando avaliar a viabilidade da abordagem dessa técnica em servidores *web*, experimentos foram realizados utilizando o gerador carga sintética *httpperf* em um servidor *web* Apache e podem ser vistos na Figura 1.3. Os resultados apresentam os tempos de resposta medidos no próprio cliente. Para efetuar essas medições, uma página *web* brasileira de comércio eletrônico foi totalmente remodelada com uma versão reduzida, removendo-se todas as figuras, substituindo-as com texto. (As versões dessas página *web* são apresentadas no Apêndice A.)

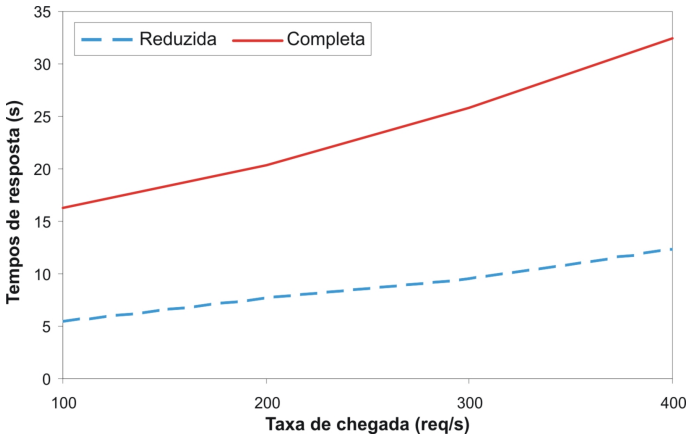


Figura 1.3: *Tempo de resposta versão completa vs versão reduzida.*

Apesar de os valores absolutos mostrados aqui não serem relevantes, pois dependem da implementação, configuração do servidor *web*, velocidade do processador, rede, etc, é possível observar que os valores dos tempos de resposta para versões reduzidas podem alcançar menos de 30% dos tempos de resposta de versões completas.

1.2 Objetivos do Trabalho

Este trabalho tem como objetivo principal a proposta de um modelo de diferenciação de serviços proporcional com controle de admissão. O modelo proposto incorpora técnicas de escalonamento dinâmico de requisições *web* baseadas em múltiplas versões (computação imprecisa) e prioridades dinâmicas. Com objetivo de avaliar o modelo proposto um protótipo foi desenvolvido e implementado em um servidor *web* Apache. Os experimentos realizados foram obtidos através de medições e analisados posteriormente. Como objetivos subjacentes são também avaliadas e comparadas com outras técnicas de QoS *(Quality of Service)*.

Este trabalho também apresenta detalhadamente maneiras de se adaptar modelos de qualidade de serviço em servidores *web* com características modulares, como é o caso do servidor *web* utilizado neste trabalho, o Apache.

1.3 Organização do Texto

Este trabalho é dividido em seis partes. O Capítulo 2 faz um levantamento dos principais conceitos sobre qualidade de serviço. Apresenta as arquiteturas de diferenciação de serviço propostas pela IETF, IntServ e DiffServ e aponta as principais técnicas utilizadas no desenvolvimento do trabalho, como o da Computação Imprecisa.

O Capítulo 3 descreve os principais trabalhos relacionados encontrados na literatura e as ferramentas utilizadas na implementação e validação de mecanismos de qualidade de serviço.

O Capítulo 4 propõe um modelo de QoS, apresentando o modelo de adaptação e os módulos que compõe a arquitetura. O algoritmo sobre o modelo é detalhadamente descrito.

No Capítulo 5 são apresentado detalhes da implementação do protótipo do modelo proposto juntamente com os experimentos realizados. Considerações interessantes sobre a implementação são mostradas, bem como maneiras para instrumentalização de servidores *web*. Os experimentos foram realizados através de medições e os dados validados através da utilização de métodos estatísticos como intervalos de confiança sobre as amostras coletadas.

Por fim, o Capítulo 6 acrescenta os últimos comentários acerca da dissertação, as conclusões e perspectivas futuras.

Capítulo 2

QoS e Diferenciação de Serviços

2.1 Introdução

A evolução tecnológica, a miniaturização de componentes e o desenvolvimento de novos suportes algoritmicos têm propiciado o surgimento de novos tipos de aplicações e, conseqüentemente, exigido o desenvolvimento de suportes e novas técnicas que dêem sustentação a essas aplicações. Exemplos dessas novas aplicações são: comércio eletrônico, voz sobre IP, multimídia, etc. Todas adotam a Internet como uma via de informação global, e apresentam necessidades cada vez mais sofisticadas em questões de desempenho, segurança e confiabilidade. Clientes dessas novas aplicações têm de conviver com clientes de aplicações convencionais da Internet, como serviço de e-mail, por exemplo, que possuem requisitos bem menos estritos.

As demandas dessas novas aplicações não podem aguardar o desenvolvimento de uma "nova" Internet, uma vez que muitas dessas aplicações já estão sendo usadas em nosso dia a dia [46]. Muitos desenvolvedores desejam utilizar código legado e soluções já existentes, como servidores *web*, por exemplo. Entretanto, esbarram no fato que essas soluções são construídas tendo como objetivo o fornecimento do mesmo serviço para todos, desconsiderando as necessidades individuais dos clientes. Muitas vezes, embora desejável, nem a rede nem os elementos finais priorizam o tráfego. Técnicas de especificação de níveis de qualidade de serviço, de implementação de diferenciação de serviços e de escalonamento adaptativo podem e devem ser empregadas nesse sentido. Este capítulo apresenta os principais conceitos relacionados a essas técnicas, as quais podem ser aplicadas a servidores *web*.

2.2 Qualidade de Serviço

Qualidade de Serviço (QoS) é um conceito que tem sua base fundamentada na idéia de que nem todas as aplicações necessitam do mesmo desempenho em suas execuções [29]. Dessa forma, costuma-se caracterizar as diversas necessidades de recursos das aplicações distribuídas em termos de requisitos de QoS [17].

Pesquisas na área de QoS tiveram origem, principalmente, devido ao surgimento e a proliferação de aplicações na Internet com demandas de valores mínimos de confiabilidade, garantia de entrega e correção temporal *(timeliness)* [28].

Exemplos de tipo de aplicação que apresentam requisitos de correção temporal com restrições não rígidas *(soft real-time)* [43] são as aplicações multimídia, tais como rádio na Internet, vídeo sob demanda e teleconferência. Recentemente, a necessidade de suporte a QoS tem se exacerbado com o emprego, cada vez maior, de aplicações de telefonia e voz sobre IP (VoIP), jogos interativos massivos *(massively multiplayer games)*, telemedicina, televisão sobre IP (IPTV) e computação distribuída em larga escala usando *grids* [28].

Um dos objetivos das pesquisas na área de QoS é o de permitir que programadores dessas aplicações distribuídas possam solicitar à infraestrutura subjacente (sistema operacional, rede) níveis de QoS apropriados para suas aplicações; e que esses níveis de QoS sejam mantidos desde a fonte da informação até o destino, passando por toda a infraestrutura da rede (ou seja, fim a fim) [10].

Três dos principais princípios de QoS que precisam ser observados na construção de um serviço de QoS são [10]:

- **Integração:** Estados de QoS devem poder ser configurados, preditos e mantidos fim a fim, de forma integrada e em todas as camadas de uma arquitetura de QoS.
- **Separação de responsabilidades:** As atividades de transferência de dados, controle e gerência são atividades distintas dentro de uma arquitetura de QoS.
- **Transparência:** A complexidade existente nas subcamadas de especificação e gerência de QoS deve ser ocultada das aplicações.

Para lidar com esses princípios em aplicações distribuídas na Internet diferentes abordagens têm sido propostas. Alguns oponentes de abordagens mais complexas sugerem que a

simples adição de maior largura de banda, oriunda do desenvolvimento e barateamento das tecnologias de enlace, seria suficiente para atendimento das demandas das aplicações citadas. Contudo, essa abordagem de super-provisionamento *(overprovisioning)* lida apenas com os requisitos de largura de banda (taxa de bits) mínima de aplicações, não atendendo aos seus outros requisitos temporais (atrasos máximos e *jitter* de atraso, por exemplo) [28].

Portanto, considerando a inadequação do paradigma de melhor esforço da Internet atual, e compreendendo que o simples aumento de capacidade dos enlaces não é suficiente, organizações padronizadoras tais como a IETF *(Internet Engineering Task Force)* [20, 21] vêm propondo novos protocolos e melhorias no protocolo IP, de forma a tornar possível suportar a proteção e a priorização de aplicações distribuídas na Internet. Essas arquiteturas são conhecidas como Serviço Integrado (IntServ) [19] e Serviços Diferenciados (DiffServ) [18] e são descritas a seguir.

2.3 Arquitetura de Serviços Integrados

Na arquitetura IntServ [19] a aplicação sinaliza explicitamente para a rede a necessidade de garantia de serviço (largura de banda e atraso, por exemplo). Nessa arquitetura existe a necessidade da manutenção dos estados pelos elementos da rede através de um protocolo específico de reserva de recursos, o RSVP *(Resource Reservation Protocol)*, que lida com estados individuais de fluxos[1] de datagramas em roteadores.

Os três principais componentes da arquitetura IntServ são: o **controle de admissão**, o qual verifica se a rede pode conceder o serviço solicitado; o **mecanismo de encaminhamento de pacotes**, que atua por operação de pacote ou em classificação de fluxos, modelando, escalonando e gerenciando as filas nos roteadores; e o **protocolo de reserva de recursos**, RSVP, o qual ativa os estados dos fluxos (reserva de largura de banda, contas e filtros) nos roteadores em que o fluxo passa.

A abordagem IntServ é baseada em reserva de recursos antecipada mantendo o estado do fluxo em todos os roteadores por onde os pacotes vão passar. Portanto, essa arquitetura é limitada, principalmente, com respeito à **escalabilidade**. Os problemas de escalabilidade surgem porque a arquitetura IntServ necessita que os roteadores mantenham o estado de

[1] Apesar de não haver explicitamente a definição de "fluxo de datagramas" na Camada de Rede da Internet, boa parte da Literatura usa esse conceito quando trata sobre QoS em roteadores. Segundo [52], esse tipo de fluxo – caracterizado por uma seqüência de pacotes com o mesmo par origem/destino e que flui pelo mesmo caminho de roteadores – deveria ser denominado de "fluxo sem conexão".

controle e encaminhamento de todos os fluxos que passam por ele. Como o tamanho da Internet sempre cresce, manter os estados de todos os fluxos nos roteadores acaba se tornando inviável em redes IP.

Por outro lado, não há problemas de escalabilidade na abordagem DiffServ descrita a seguir, onde os fluxos de pacotes de clientes com requisitos de QoS são classificados e agrupados em classes de serviço. Por trás de cada uma dessas duas abordagens – IntServ e DiffServ – está o conceito de **granularidade de QoS** [26]. Enquanto a arquitetura IntServ lida com estados individuais de fluxos em roteadores, a arquitetura DiffServ trabalha com um número reduzido de classes, com a finalidade de resolver o problema da escalabilidade da arquitetura IntServ.

2.4 Arquitetura de Serviços Diferenciados

Ao contrário do IntServ, que faz reserva de recursos e mantém o estado para cada fluxo de dados que passa no roteador, na arquitetura DiffServ [18] os aplicativos não necessitam fazer reserva de recursos para seus fluxos de dados [44].

A arquitetura do DiffServ é composta por funções implementadas nos nós da rede. Estas incluem funções de classificação de pacotes, características de encaminhamento por nó e funções de condicionamento de tráfego tais como medição, marcação, formatação e vigilância.

A escalabilidade dessa arquitetura vem do fato de implementar essas funções apenas nos nós de borda da rede, e aplicar as características, ou o comportamento desejado, por nó. Isso é chamado de *Per-Hop Behavior* (PHB), onde para cada conjunto de tráfegos *(aggregate traffic)* que foi previamente marcado utilizando o campo DS, um tratamento diferenciado será dado. Já a flexibilidade vem do fato de que o DiffServ provê componentes funcionais, que são partes da arquitetura de rede, as quais os serviços podem ser implementados [38]. Isso faz com que a arquitetura suporte novas classes de serviços, ou, até mesmo, permita que uma determinada classe de serviços se torne obsoleta.

2.4.1 Domínio de Serviços Diferenciados

Na arquitetura DiffServ, todo pacote é classificado, através de uma marca que é carregada no cabeçalho do datagrama IP chamado de *DS Codepoint* (DSCP). No IPv4 essa informação vai dentro do campo Tipo de Serviço *(Type Of Service, TOS)* e no IPv6 no campo Tipo de Tráfego (Figura 2.1). Sempre que um fluxo entra em um roteador DiffServ, esse campo é analisado, resultando em diferentes classes de tráfego recebendo desempenhos diferentes (diferentes PHBs).

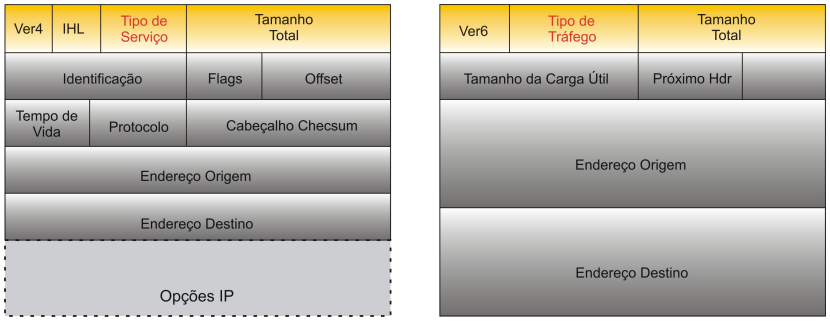


Figura 2.1: *Cabeçalhos dos datagramas IP versão 4 e 6.*

Um domínio de Serviços Diferenciados pode ser definido como um conjunto contíguo de nós que operam com uma política de provimento de serviços comum e um conjunto de grupos PHB que são implementados em cada nó [18]. Esse domínio consiste de nós de borda e nós internos, e ambos devem ser capazes de aplicar o PHB apropriado aos pacotes com base no DSCP (Figura 2.2).

Os nós de borda do domínio são os responsáveis por classificar e condicionar o tráfego de entrada para, dessa forma, garantir que os pacotes que transitem dentro do domínio estejam sempre marcados. Só assim o tráfego pode ser tratado de forma diferenciada. Os nós de borda podem atuar também como nós de entrada e nós de saída do domínio e são responsáveis por interconectar domínios com outros domínios, mesmo que estes não suportem Serviços Diferenciados.

Já os nós internos irão encaminhar os pacotes baseando-se no comportamento, que é previamente definido pelo DSCP [49]. Devem ser capazes também de fazer um tipo de condicionamento do tráfego limitado, como por exemplo, a remarcação de DSCP.

É importante que todos os nós internos do domínio suportem diferenciação de serviços, pois a inclusão de nós não condicionados dentro do domínio pode causar um mau funciona-

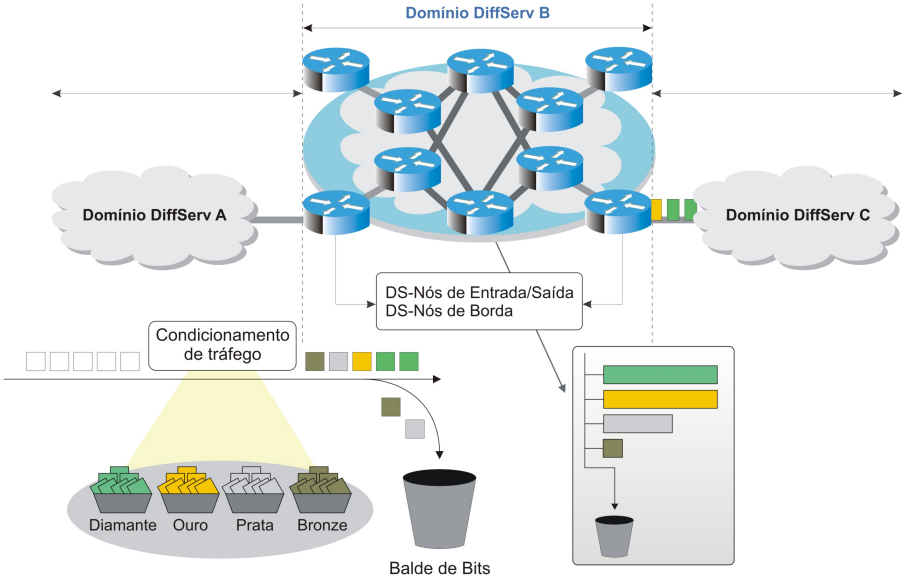


Figura 2.2: *Domínio DiffServ (baseado em [23]).*

mento da rede, e também a perda da habilidade dos nós se comunicarem para estabelecer rotas e prioridades diferenciadas.

Um outro conceito importante é o de região de Serviços Diferenciados, que é um conjunto de domínios contíguos. Os domínios dentro em uma região são capazes de suportar diferentes grupos PHB internamente e diferentes mapeamentos de DSCPs. Porém, para permitir serviços que atravessem diferentes domínios, os domínios devem estabelecer um Acordo de Nível de Serviço (SLA – *Service Level Agreement*) [18], que irá definir um Acordo de Condicionamento de Tráfego (TCA – *Traffic Conditioning Agreeement*), o qual especifica como o tráfego em trânsito de um domínio para outro é condicionado.

2.4.2 Classificação, Perfil e Condicionamento do Tráfego

Serviços diferenciados são estendidos através de um domínio de Serviços Diferenciados através do estabelecimento de um SLA entre a rede comum (sem diferenciação de serviços) e o domínio de Serviço Diferenciado [18]. Esse acordo irá especificar as políticas de classificação e remarcação de pacotes e perfis de tráfego, para que dessa forma, o subconjunto de um tráfego do domínio possa receber um serviço diferenciado através do condicionamento e do mapeamento para um ou mais comportamentos agregados.

O classificador de pacotes, localizado na borda do domínio, seleciona os pacotes em

um fluxo de dados baseado no conteúdo do cabeçalho do pacote [18]. Existem dois tipos de classificadores:

Behavior Aggregate (BA): Classifica os pacotes baseando-se apenas no DSCP.

Multi-Field (MF): Classifica os pacotes baseando-se no valor de um ou mais campos do cabeçalho, como por exemplo, porta de origem e destino, campo DSCP, etc. O classificador seleciona um fluxo de tráfego e guia os pacotes desse fluxo para uma instância lógica de um condicionador de tráfego.

O perfil de tráfego especifica as propriedades temporais de um fluxo selecionado pelo classificador [18], e provê regras para determinar se um pacote específico está dentro do perfil *(in-profile)* ou fora do perfil *(out-of-profile)*. Dependendo da situação em que se encontra o pacote *(in ou out)*, diferentes ações de condicionamento devem ser tomadas. Por exemplo, pacotes dentro do perfil podem ter permissão para entrar no domínio sem a necessidade de um condicionamento mais adiante. Já os pacotes fora do perfil podem ser enfileirados até que estejam ou dentro do perfil, ou descartados ou marcados com um novo *codepoint*. Essas ações tomadas com pacotes fora do perfil são todas feitas pelo condicionador.

O condicionamento é feito através da medição, marcação, formatação e vigilância, de forma que garanta que o tráfego que entre no domínio respeite os acordos previamente estabelecidos no TCA (Figura 2.3). Abaixo uma breve explanação das funções de cada item do condicionamento:

- **Medidores:** Avaliam as propriedades temporais de um fluxo de pacotes (previamente selecionados pelo classificador) em relação ao perfil de tráfego especificado pelo TCA. Um medidor passa informações de estado para outras funções de condicionamento para executar uma ação em particular para cada pacote que esteja dentro ou fora do perfil.

- **Marcadores:** Configuram o campo DS de um pacote para um *codepoint* em particular e depois adicionam o pacote para um DS de comportamento agregado. O marcador pode marcar todos os pacotes com um único *codepoint*, ou marcar os pacotes com um dos vários *codepoints* que são usados para selecionar um PHB em um grupo PHB, de acordo com o estado do medidor.

- **Formatadores:** Têm a função de atrasar os pacotes em um fluxo de tráfego para tornar o fluxo compatível com o perfil do tráfego.

- **Descartadores:** Da mesma forma que o formatador, têm a função de tornar o fluxo de tráfego compatível com o perfil do tráfego. Porém, ao invés de atrasar, os pacotes são descartados. Esse processo funciona como se fosse uma "vigilância" do tráfego.

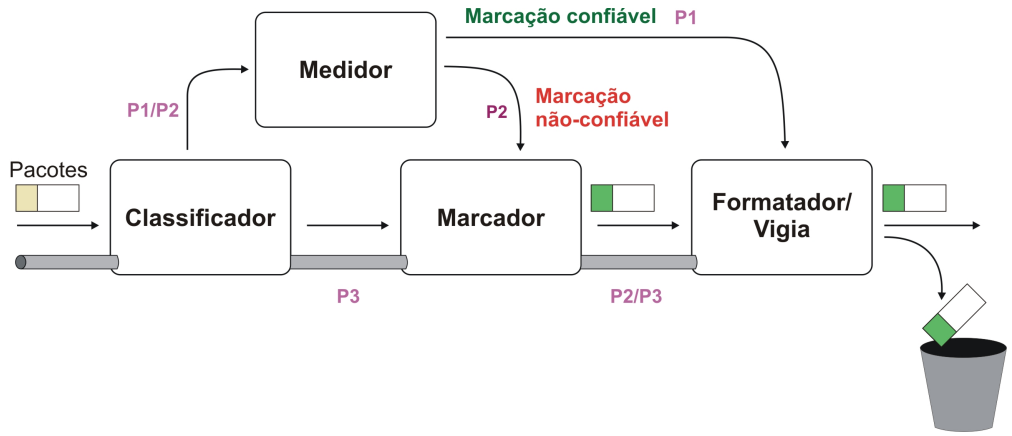


Figura 2.3: *Condicionamento de tráfego DiffServ (baseado em [23]).*

A localização dos classificadores e condicionadores de tráfego, isto é, o momento em que os pacotes são marcados com a devida forma de tratamento, pode variar de acordo com normas administrativas. Normalmente estão localizados nos nós de entrada e saída do domínio, porém também podem ser localizados em nós dentro do domínio ou até mesmo em domínios não DS.

2.4.3 Definição do Campo Serviço Diferenciado

Conforme visto, o cabeçalho IP possui um campo de Diferenciação de Serviços onde o datagrama é marcado (Figura 2.4). Este campo é composto por oito bits, sendo que dois bits são reservados e atualmente não usados. A outra porção, composta por seis bits, é denominada DSCP, e é utilizada para definir o PHB, ou seja, o comportamento do pacote nos nós DiffServ. O campo DSCP é definido de forma bem estruturada, para facilitar a definição de um futuro PHB.

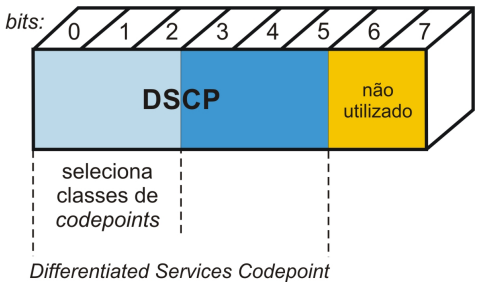


Figura 2.4: *DS codepoint*

Algumas particularidades, como o mapeamento de valores para PHBs podem ser configuradas. Um nodo DS deve suportar a lógica equivalente à mapeada com valores para os PHBs. Implementações devem suportar os valores PHB mapeados em suas configurações padrão, operações devem escolher usar diferentes valores para PHB, ou adicionar no local do padrão recomendado. Devem então ser permitidas as operações de escolha e remarcação dos campos DS, necessárias em limites administrativos ou até mesmo em ambos os lados do limite DS.

2.4.4 Per-Hop Behavior

O comportamento de um pacote em um nó DiffServ é definido através do DSCP que é mapeado e comparado com os PHBs definidos na arquitetura DiffServ [49]. Um PHB é definido como sendo comportamentos individuais aplicados em cada roteador (com base no tipo de tráfego agregado), por isso isoladamente não garantem QoS fim a fim.

PHBs devem ser especificados em termos de prioridade de recursos (por exemplo, *buffers*, largura de banda) relativo a outros PHBs, ou baseado em observações de características do tráfego, como por exemplo, perda de pacotes e atrasos. PHBs devem ser usados como um grupo (grupo PHB) consistente, os quais compartilharão normalmente as mesmas características aplicadas para cada PHB dentro do grupo, tal como agendamento de pacotes ou políticas de gerência de *buffers* [18].

Os grupos PHB devem ser definidos de forma que a alocação de recursos entre grupos possa ser inferida, e mecanismos integrados podem ser implementados e então suportar dois ou mais grupos. Alguns PHBs padrões foram definidos, entre eles o PHB *default*, PHB AF *(Assured Forwarding)* e PHB EF *(Expedited Forwarding)*.

- **PHB *default:*** O padrão PHB *default* deve estar disponível em um nodo compatível com DS. Este é um comportamento comum no serviço de melhor esforço disponível nos roteadores existentes [11]. Quando não há nenhum outro acordo no lugar, assume-se que estes pacotes pertencem a esse padrão. O valor recomendado para o padrão é a sequência de bits prenchidos com zeros. Este *codepoint* escolhido como padrão é compatível com o valor existente recomendado na RFC [53].

- **PHB EF *(Expedited Forwarding:)*** O padrão de grupo PHP EF *(Expedited Forwarding)* tem como objetivo fornecer baixa perda, baixo atraso, baixa variação do atraso (*jitter* de atraso), garantia de largura de banda, e garantia que os pacotes encontrem as filas vazias ou com pouco enfileiramento. Deve garantir que o tráfego receba largura de banda suficiente para que a taxa de transmissão seja a que foi definida, permitindo que se quantifique o atraso máximo e variação do atraso para este tráfego [18].

  A taxa em que o tráfego EF é servido em uma dada interface de saída deve ser ao menos a taxa R configurada, sobre um intervalo de tempo conveniente, independente do tráfego não-EF sobre esta mesma interface. O *codepoint* 101110 é o recomendado para uso pelo EF PHB.

- **PHB AF *(Assured Forwarding:)*** O padrão de grupo PHB AF *(Assured Forwarding)* é um meio para o provedor de um domínio DS fornecer diferentes níveis de garantia de encaminhamento de pacotes IP [31]. Foram definidas quatro classes AF, sendo que cada uma contém três precedências de descarte de pacotes: baixa, média e alta. Cada classe AF é garantida o fornecimento de uma quantidade mínima de largura de banda e *buffering*. Os DSCPs recomendados para uso no AF são identificados na (Tabela 2.1):

Tabela 2.1: *Tabela de precedência de descarte (baseado em [23]).*

Precedência	Classe AF1	Classe AF2	Classe AF3	Classe AF4
Baixa	001010	010010	011010	100010
Média	001100	010100	011100	100100
Alta	001110	010110	011110	100110

Os três primeiros bits do DSCP identificam as classes de transmissão e os três últimos a precedência de descarte.

2.5 Comparação entre Serviços Diferenciados e Serviços Integrados

As principais diferenças entre as arquiteturas DiffServ e IntServ são mostradas na Tabela abaixo (Tabela 2.2).

Tabela 2.2: *Comparação entre arquitetura DiffSer vs IntServ (baseado em [26]).*

Ítens Comparados	**IntServ**	**DiffServ**
Granularidade da diferenciação de serviços	Fluxos individuais	Fluxos agregados
Manutenção do estado	Por fluxo	Por agregado
Tipo de diferenciação de serviços	Garantia determinística ou estatística	Garantia absoluta ou proporcional
Coordenação	Fim a fim	Local (por salto)
Escalabilidade	Limitada pelo número de fluxos	Limitada pelo número de classes de serviços
Sinalização	Características de fluxos ou requisitos de QoS	Convenção das classes

Compilando as informações obtidas a partir das discussões sobre as arquiteturas IntServ e DiffServ e com base na Tabela 2.2 é possível identificar as diferenças entre as arquiteturas. A principal diferença entre as arquiteturas diz respeito à **escalabilidade**; enquanto a arquitetura IntServ provê reserva de recursos individual (fim a fim), a arquitetura DiffServ promove manutenção dos estados por agregado (local), através da convenção de classes de serviço. A limitação no caso do IntServ está no número de fluxos que um roteador deve manter; por outro lado, na arquitetura DiffServ o limite é ditado pela quantidade de classes de serviço.

2.6 Diferenciação de Serviços Absoluta e Proporcional

Conforme visto, a diferenciação de serviços pode ser implementada através da arquitetura de Serviços Integrados (IntServ) ou da arquitetura de Serviços Diferenciados (DiffServ). Devido ao problema da escalabilidade, a arquitetura DiffServ tem se mostrado mais promissora para ser implementada na Internet, e diversos aprimoramentos vêm sendo propostos nesse sentido. Um desses aprimoramentos se refere à abordagem dinâmica de “espaçamento” entre classes de QoS visando implementar a diferenciação de serviços proporcional [26].

Segundo Dovrolis e Ramanathan [26] a diferenciação de serviços pode ser classificada como **absoluta** ou **proporcional**. O modelo de diferenciação de serviços absoluto é caracterizado pela existência de níveis de serviços fixos, estáticos, onde o suporte se esforça para atender valores mínimos e máximos de qualidade estipulados pelas aplicações. Esse modelo pode ser implementado, por exemplo, por uma abordagem de atribuição estática de prioridades às classes. O problema desse modelo é que a carga submetida a cada classe é ignorada. Ou seja, caso uma classe de serviço "superior" receba uma demanda contínua, as classes "inferiores" poderão sofrer um problema de inanição *(starvation)*, pois nunca serão atendidas. Além disso, não há formas de ajustar dinamicamente o "espaçamento" de qualidade entre as classes.

No modelo de diferenciação de serviços proporcional (ou relativo proporcional) se busca manter valores proporcionais de QoS entre as classes de serviços existentes, independentemente da carga. A maior justificativa desse modelo é que em ambientes onde a carga pode crescer de forma inesperada, como é o caso da Internet, torna-se inadequado qualquer esquema de gerenciamento estático de recursos, como por exemplo, atribuição fixa de prioridades entre as classes. O modelo proporcional possui classes de serviços, onde as aplicações são previamente classificadas e recebem a qualidade de serviço em conformidade com a classe à qual pertence. Este modelo leva em consideração dois aspectos [26]:

1. **Controlabilidade:** Este importante atributo se refere à capacidade de os operadores de rede poderem ajustar de forma apropriada a qualidade de serviços entre as classes, ou seja, o espaçamento de qualidade entre as classes baseados em seus preços ou uma política definida.

2. **Previsibilidade:** Capacidade de o sistema manter a diferenciação de qualidade de serviço consistente (classes altas são melhores) entre as classes independente das condições de sobrecarga do sistema. Em ambientes como a Internet, que em determinados momentos atua sobre condições severas de sobrecarga, o modelo de diferenciação de serviços relativo proporcional é indicado.

Conforme visto na seção 2.2, esses aspectos compõem o **princípio da integração**, que precisa ser respeitado em arquiteturas de QoS, conforme proposto em [10].

Apesar do conceito de diferenciação proporcional de serviços ter sido formalizado em [26], alguns trabalhos apresentam abordagens semelhantes no escalonamento na fila de roteadores no núcleo da rede. Por exemplo, em [25] é proposto o algoritmo conhecido como

Fair Queue, que implementa uma distribuição proporcional de carga entre as filas de entrada nos roteadores. Em [36, 61] uma variação desse trabalho foi proposta, onde foi introduzido o conceito de "peso" nas filas utilizando uma variação do algoritmo de escalonamento circular *(round-robin)*, no qual os datagramas são classificados na entrada dos roteadores em classes de serviço e a qualidade do serviço é dada conforme o "peso" de cada classe. Este algoritmo é conhecido como *Weight Fair Queue* (WFR).

2.7 Escalonamento Adaptativo

As políticas de escalonamento têm por finalidade determinar como serão ocupados os recursos do sistema (processador, disco, largura de banda, etc) necessários para a realização das tarefas mediante algumas restrições ou objetivos (ex. restrições temporais). Apesar de abordagens de QoS e escalonamento adaptativo constituirem em temas usualmente desenvolvidos por grupos de pesquisa independentes, técnicas de escalonamento podem ser usadas como um mecanismo para implementar abordagens de QoS. Por esse motivo, em [54] é proposta uma taxonomia para especificação de QoS (Figura 2.5) onde alguns dos parâmetros considerados referem-se a métricas de desempenho de correção temporal *(timeliness)* da aplicação.

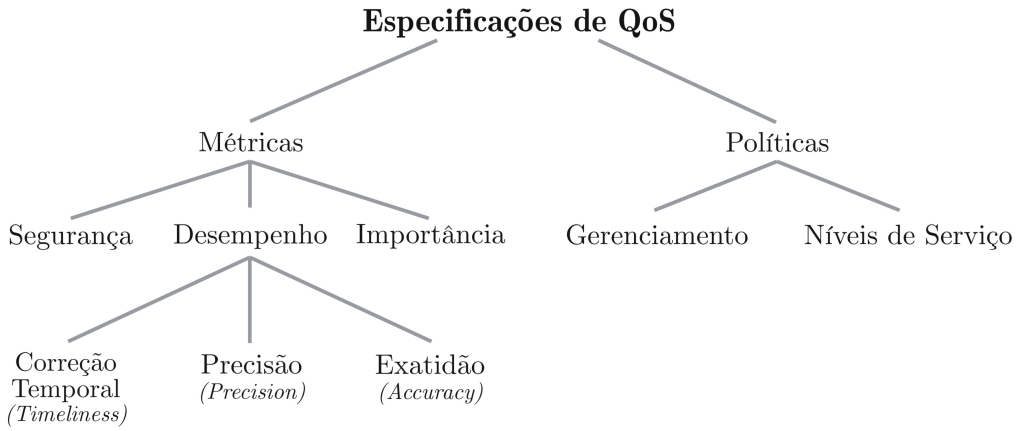


Figura 2.5: *Uma taxonomia para especificação de QoS (baseado em [54]).*

Técnicas de escalonamento adaptativo monitoram as condições do sistema (ex. sobrecarga, utilização de recursos) para realizar as suas tomadas de decisão [46]. Vários trabalhos têm estudado a utilização dessas técnicas em ambientes como a Internet, onde a carga é muito dinâmica e pode crescer arbitrariamente [1, 41, 42, 46, 58]. Servidores *web*, por exemplo, lidam com um grande número de requisições de vários clientes. Essas requisições possuem

características temporais aperiódicas, ou seja, é difícil precisar quando um determinado cliente vai enviar novas requisições ao servidor, tornando praticamente impossível manter os estados de todos os clientes que acessam o servidor.

O uso do escalonamento adaptativo, nesse caso, vem prover um esquema de adaptação dinâmica ao servidor *web*, para situações onde a taxa de chegada de requisições seja maior do que a capacidade do servidor processar respostas, caracterizando assim uma sobrecarga. Em condições de sobrecarga, um servidor *web* pode operar de maneira indesejada (ex. longos tempos de resposta, descarte de requisições). A característica de carga dinâmica deste tipo de servidor torna inadequado qualquer esquema de atribuição estática de provimento de recursos.

Em servidores *web*, o modelo de serviços executa o atendimento as requisições de forma binária: atende ou não atende (descarta) a requisição. Técnicas de escalonamento adaptativo, tais como a da computação imprecisa, ajudam a lidar com essas questões, oferecendo uma solução de compromisso.

2.7.1 Computação Imprecisa

A técnica de computação imprecisa tem mostrado ser útil em várias áreas nas quais o tempo de processamento das tarefas ou serviços é enfatizado. A computação imprecisa [39, 40] propõe uma maneira de se lidar com sobrecargas transientes e aprimorar a tolerância a faltas em sistemas de tempo real. Um sistema baseado nessa técnica é chamado de sistema impreciso. Em um sistema impreciso, cada tarefa ou cada conjunto de tarefas contém uma parte opcional. Sob condições normais, a parte opcional é completada e o resultado da tarefa tem a qualidade desejada, sendo assim **precisa**. Durante condições de sobrecarga, a parte opcional ou parte dela pode ser omitida com o objetivo de conservar os recursos do sistema. Quando esta parte opcional é omitida, a tarefa produz um resultado **impreciso**.

O uso de múltiplas versões de tarefas é uma opção para fornecer flexibilidade no escalonamento, e pode ser realizada com pelo menos duas versões de tarefas: primária e alternativa. A versão primária de cada tarefa produz um resultado desejado e preciso, mas tem um longo tempo de processamento. Uma versão alternativa oferece um menor tempo de processamento, mas produz resultados imprecisos. Durante situações de sobrecarga transientes, onde não é possível atender as tarefas com as versões primárias, o escalonador pode optar por versões alternativas. A qualidade do dado aumenta à medida que a imprecisão do objeto diminui; similarmente, a qualidade do dado diminui à medida que a imprecisão aumenta [5].

Com objetivo de aplicar técnicas de computação imprecisa em servidores *web*, em [1] é apresentado um modelo de adaptação de conteúdo. Neste modelo, as requisições são adaptadas em conformidade com as condições de carga do servidor. Dessa forma, permite-se que mais clientes acessem os servidores em condições de sobrecarga, e também se reduz a quantidade de recursos desperdiçados (ex. sistema operacional, rede, processador, memória, etc) quando a capacidade de atendimento de requisições é excedida e requisições são abortadas por *timeouts*, por exemplo. A técnica de adaptação de conteúdo é utilizada também para executar as requisições de acordo com as limitações de recursos de cada cliente. Por exemplo, a adaptação de conteúdo pode fornecer versões mais apropriadas de conteúdo para clientes com restrições de memória, disco, processador, rede, capacidade de visualização, etc.

Páginas *web* costumam ter uma grande quantidade de objetos, com grande apelo visual, contendo várias imagens *gif, jpg* e animações *flash*, facilitando a implementação de computação imprecisa pela abordagem de múltiplas versões [18, 19, 26]. Neste caso, para uma mesma página, mais de uma versão é implementada. Abaixo são apresentadas três técnicas para implementação de versões de páginas *web* imprecisas apresentadas por [1]:

1. **Degradação na qualidade das imagens através de métodos de compressão:** Uma pesquisa realizada por [1] aponta que em 80 *web sites* de compras (comércio eletrônico), cerca de 65% do total dos bytes são de imagens *gif* ou *jpg*. Na grande maioria dos casos essas páginas podem ser reduzidas de tamanho sem que os clientes percebam. Este fator melhora em muito a taxa de transmissão dessas imagens na rede devido à compressão.

2. **Diminuição no número de objetos nas páginas:** Na visão do servidor, o número de objetos contidos em uma página *web* é mais importante do que o tamanho dos objetos, portanto a redução no número de objetos nas páginas diminui a taxa de acessos *(hits)* ao servidor.

3. **Redução de *links* locais:** Esta redução irá surtir efeito no comportamento do *browser* do cliente, que tende a diminuir a carga no servidor devido à diminuição de conteúdo de acesso.

Em casos extremos, onde versões de páginas são formadas apenas por textos HTML, experimentos mostram que de 30% a 90% das páginas apresentaram melhoria de 400% [1].

2.7.2 Função Benefício

A abstração de função benefício tem sido proposta em alguns trabalhos como forma de modelar os requisitos e preferências de uma aplicação [54]. Uma função benefício é um gráfico multidimensional que especifica o benefício obtido pela aplicação (ou pelo usuário desta) quando o sistema oferece um determinado nível de QoS.

Funções benefício podem ser empregadas também em sistemas de tempo real [43]. Nessas abordagens, uma função benefício pode modelar o valor obtido por um sistema de tempo real em função do seu tempo de resposta. A Figura 2.6 apresenta um exemplo de uma função benefício que modela a existência em uma tarefa de um *deadline soft* d1 e um *deadline* firme d2. Nessa tarefa, o benefício (ou valor) obtido decai após o *deadline soft* d1 até alcançar um valor nulo após o *deadline* d2.

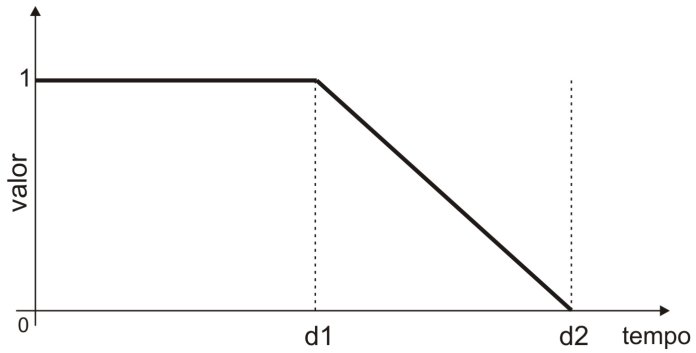


Figura 2.6: *Função Benefício de uma tarefa com deadline soft e firme.*

Segundo [39], um *deadline* pode ser classificado como: *soft e firm* (de acordo com a utilidade do resultado para a aplicação:

- ***Soft:*** Restrição temporal em que o resultado que a ela está associado mantém alguma utilidade para a aplicação mesmo depois de um tempo limite, embora haja uma degradação da qualidade de serviço.
- ***Firm:*** Restrição temporal em que o resultado qua a ela está associado perde toda utilidade para a aplicação depois de um tempo limite.

A adoção de uma função benefício para representar o valor obtido por um sistema também pode ser empregado em servidores *web*. Apesar de modelos tradicionais considerarem estes como sistemas convencionais (não tempo real) que possuem uma função benefício cujo

valor nunca reduz com o tempo [43], essa é uma modelagem pouco realista, pois mesmo os sistemas de propósito geral têm seus benefícios reduzidos com o passar do tempo.

Além disso, a demanda de serviços de tempo real em aplicações *web* como comércio eletrônico, por exemplo, tem se tornado premente. Em muitas aplicações de comércio eletrônico é desejável processar as requisições dos clientes dentro de *deadlines* especificados [37]. No entanto, essa é uma tarefa difícil devido à variação de carga na rede e nos servidores *web*, que muitas vezes agem sob condições de sobrecarga transientes (momentos onde a demanda de serviços excede a capacidade do servidor) [55].

2.8 Conclusões do Capítulo

Este capítulo apresentou os principais conceitos relacionados à implementação de QoS e diferenciação de serviços. Foram discutidas abordagens de QoS na infraestrutura da rede que vem sendo propostas, como IntServ e DiffServ. Técnicas de escalonamento adaptivo, como a computação imprecisa e uso de função benefício foram introduzidas. No próximo capítulo são listados alguns trabalhos encontrados na literatura, que propõem o uso dessas técnicas em servidores *web*.

Capítulo 3

Ferramentas e Trabalhos Relacionados

3.1 Introdução

Estima-se que atualmente existam mais de 100 mil *web sites* na Internet[1]. Pequenos *sites web*, comerciais ou simplesmente *blogs* de usuários, têm contribuído significativamente para esse crescimento. Várias empresas oferecem domínios gratuitos ou com baixo preço, favorecendo assim o aparecimento desses novos *sites*.

Os componentes responsáveis por processar e armazenar as informações contidas nesses *web sites* são os servidores *web*. Estudos apontam que em 1995 o servidor *web* NCSA dominava o mercado de servidores *web* com 57%, em segundo lugar estava o CERN com 19%. Nesse mesmo ano surgiu o servidor *web* Apache com apenas 3,5% do uso. Atualmente o Apache lidera o mercado de servidores *web* com 60,3%, a Microsoft com 31% e a Sun com 1,7% (Tabela 3.1). A Figura 3.1 traz um levantamento do uso dos servidores *web* desde agosto de 1995 até novembro de 2006.

Além de ser o servidor *web* mais utilizado no mundo, o Apache possui código aberto, com comunidade amplamente difundida e vasta documentação disponível. Por esses motivos foi o servidor escolhido na implementação do modelo proposto nesse trabalho. Este capítulo apresenta detalhes da implementação interna do Apache. Algumas das principais ferramentas e trabalhos relacionados à implementação de QoS em servidores *web* também são descritos.

[1] www.netcraft.com (nov/2006)

Tabela 3.1: *Distribuição do uso de servidores web no mundo atualmente.*

Desenvolvedor	Nov/2006
Apache	60,3%
Microsoft	31,0%
Sun	1,7%
Zeus	0,5%

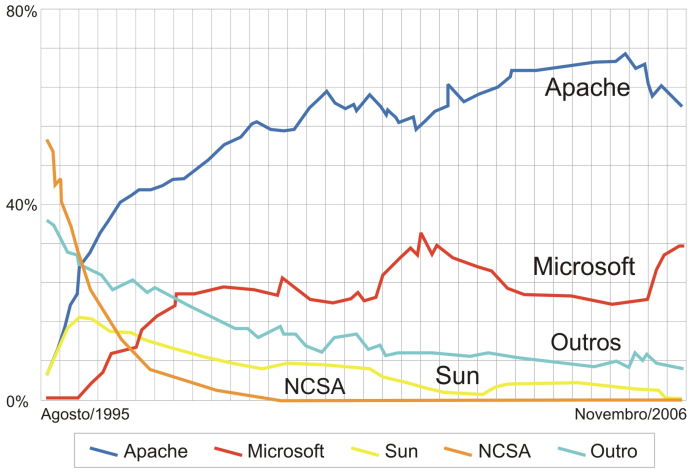


Figura 3.1: *Evolução no uso dos servidores no mundo (baseado em [30]).*

3.2 Aspectos Internos do Apache Versão 2

A definicão mais simples possível de um servidor *web* é a de um programa que escuta por requisições HTTP e retorna as respostas quando as recebe. No Apache esta é a tarefa do gerador de conteúdo, o "coração" do servidor *web* (Figura 3.2). Um gerador de conteúdo deve ser executado para tratar cada uma das requisições HTTP que chegam ao servidor. Qualquer módulo pode registrar geradores de conteúdo, geralmente isto é realizado através das diretivas *SetHandler* ou *AddHandler*, inseridas no arquivo de configuração *httpconf*.

Se uma requisição não possui um gerador de conteúdo associado, esta é manipulada pelo gerador de conteúdo padrão do Apache, que simplesmente retorna o arquivo indicado pela requisição para o sistema de arquivos. Módulos que implementam um ou mais geradores de conteúdo são chamados de módulos manipuladores *(handlers)*.

A princípio, um gerador de conteúdo pode implementar todas as funções de um servidor *web*, porém, como em outros servidores, o Apache divide as requisições em diferentes fases (Figura 3.3). Existem várias fases a serem executadas antes do gerador de conteúdo. Estas

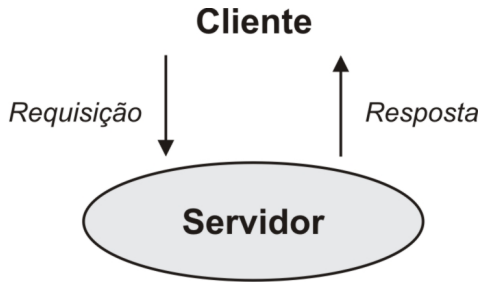


Figura 3.2: *Gerador de conteúdo do Apache.*

podem examinar e, talvez, manipular os cabeçalhos das requisições, e determinar o que fazer com as requisições. Por exemplo:

1. Verificar se a URL *(Universal Resource Locator)* da requisição será compatível com a configuração para determinar qual gerador de conteúdo será utilizado;
2. Mapear a URL para o sistema de arquivos (ex. para um arquivo estático, CGI, etc.);
3. Encontrar a versão que mais se enquadra com o navegador *web*, como por exemplo, o idioma solicitado pela requisição;
4. Controlar o acesso ao servidor, com adição de algumas regras de autenticação e de acesso;
5. Alterar a URL contida nas requisições segundo alguma estratégia.

Existe ainda a fase de *log* que é concebida antes do gerador de conteúdo enviar uma resposta para o cliente. Módulos localizados nas fases anteriores ao gerador de conteúdo são conhecidos como módulos de metadados. Os que tratam as requisições depois do gerador de conteúdo são chamados de módulos de *log* (Figura 3.3).

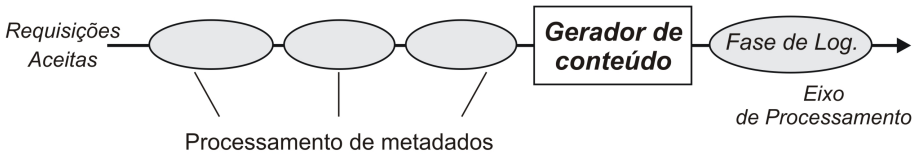


Figura 3.3: *Fases de processamento das requisições no Apache.*

A maior inovação da versão 2 do Apache, que o transforma de um simples servidor *web* (como era a versão 1.3 do Apache) em uma poderosa plataforma de aplicação é a cadeia

de filtros (Figura 3.4). A cadeia de filtros é representada na figura como um eixo de dados ortogonal ao eixo de processamento das requisições. Os dados das requisições podem ser processados por filtros de entrada antes de alcançarem o gerador de conteúdo, e a resposta pode ser processada pelos filtros de saída antes de ser enviada para o cliente.

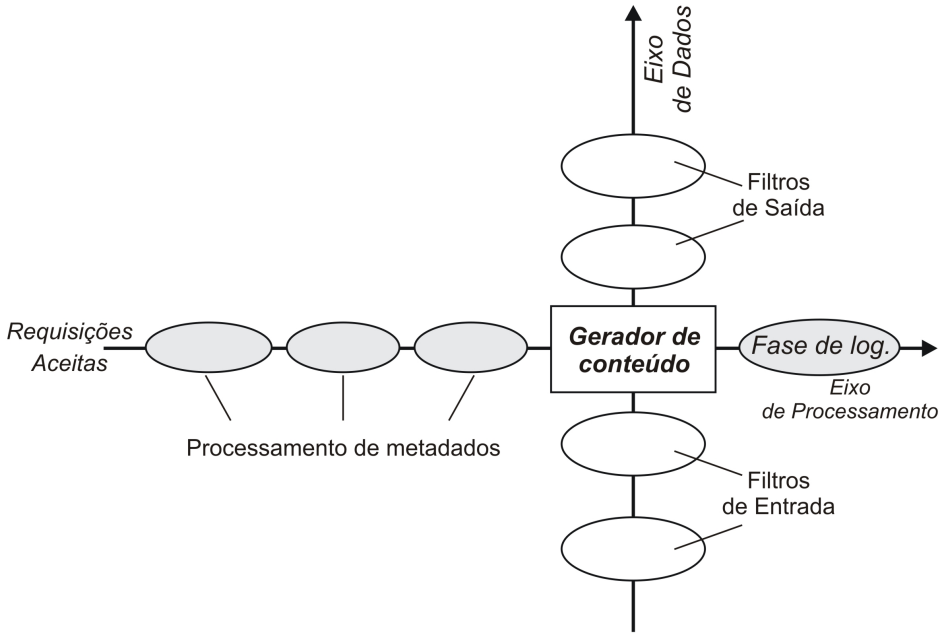


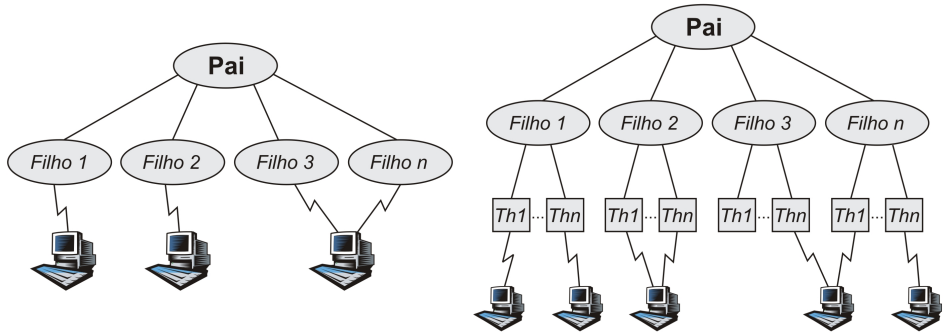
Figura 3.4: *Eixo de dados e Filtros do Apache.*

O eixo de processamento das requisições é estritamente ordenado, ou seja, as fases de processamento acontecem sempre na mesma ordem. Porém, o eixo de dados ocorre de forma paralela, portanto o gerador de conteúdo e os filtros não são executados em uma ordem determinística. Por exemplo, não se pode realizar alguma função em um filtro de entrada e esperar que isso seja aplicado no gerador de conteúdo ou nos filtros de saída.

A ordem de processamento está de fato centrada no gerador de conteúdo, que é responsável por pegar os dados do filtro de entrada e publicá-los no filtro de saída. Quando o gerador ou filtro precisa alterar algo em uma requisição, este deve fazê-lo antes do dado passar pelo gerador ou filtro.

3.2.1 Módulos Multiprocessamento

O servidor *web* Apache é projetado para trabalhar com uma ampla variedade de plataformas e ambientes. Isto é possível devido à sua implementação modular. A versão 2



(a) Prefork (b) Worker

Figura 3.5: *Tipos de Módulos Multiprocessamento do Apache.*

do Apache introduziu o modelo de processos MPM (Módulos Multiprocessamento - *Multi-Processing-Modules*), responsáveis por gerenciar as portas de comunicação, aceitar conexões e alocar processos ou *threads* para atendimento das requisições.

A escolha do MPM a ser utilizado depende de vários fatores, por exemplo, se o sistema operacional oferece suporte a *threads*, disponibilidade de memória do sistema, escalabilidade versus estabilidade. Os módulos MPM disponíveis na versão 2 do Apache são o MPM *Prefork*, *Worker* e *Event*.

1. O MPM *Prefork* é parecido com a configuração existente no Apache 1.3. Utiliza múltiplos processos filhos, sendo que cada filho manipula uma conexão por vez, o que garante máxima estabilidade, pois cada servidor é executado em seu próprio processo. Assim, se um processo falhar, este não afeta os outros processos servidores (Figura 3.5(a)). O MPM *Perfork* é adequado para sistemas que necessitam ser tolerantes a faltas. Porém, utilizam demasiadamente a memória.

2. O MPM *Worker* utiliza múltiplos processos filhos, onde cada filho pode disparar *threads* para manipular uma única conexão. É mais escalável e possui melhor desempenho, porém perde em confiabilidade, pois caso um processo falhe, os demais processos e threads ligados a esse processo também irão falhar (Figura 3.5(b)).

3. O MPM *Event* é um módulo com características semelhantes ao MPM *Worker*, porém atualmente está em fase de testes.

3.2.2 Interceptadores

O atendimento das requisições no Apache pode ser subdividido em quatro fases (Figura 3.6):

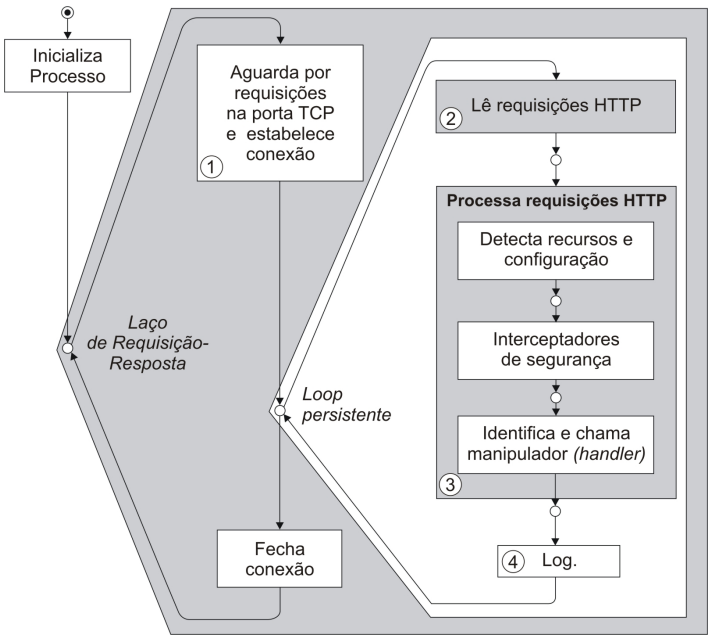


Figura 3.6: *Ciclo de atendimento das requisições no Apache.*

1. **Estabelecimento da conexão:** Inicialização da conexão, criação da estrutura *connection_rec* (possui todas as informações referentes a uma conexão).

2. **Leitura da requisição HTTP:** Criação da estrutura *request_rec* (estrutura mais importante para o Apache, possui informações sobre a requisição).

3. **Processamento da requisição HTTP:** Fases de autenticação de usuário, segurança, tradução da URL da requisição para o sistema de arquivos e, principalmente, a chamada do gerador de conteúdo adequado para a requisição.

4. **Fase de log:** Executada depois de concluído o tratamento da requisição.

A estrutura dos módulos no Apache declara várias (algumas opcionais) funções como membros. A Figura 3.7 apresenta a declaração de uma estrutura padrão para o desenvolvimento de um módulo Apache. O módulo aqui chamado de **meumodulo** possui funções

relacionadas à configuração de diretórios, configurações do servidor, alocação dinâmica de memória do Apache, etc. Porém a função mais relevante para criar o manipulador de requisições é o último membro, aqui chamado de **register_hooks**.

```
module AP_MODULE_DECLARE_DATA meumodulo ={
    STANDARD20_MODULE_STUFF,
    my_dir_conf,        /*conf. de estrutura por diretório */
    my_dir_merge,       /*unifica conf. de est. por diretetório*/
    my_server_conf,     /*conf. de estrutura por servidor*/
    my_server_merge,    /*unifica conf. de est. por servidor*/
    my_cmds,            /*comando apr_table_t*/
    register_hooks      /*registra manipuladores*/
};
```

Figura 3.7: *Estrutura de declaração de módulos Apache.*

A Figura 3.8 mostra a fase de criação do módulo, registrando as funções que serão efetivamente chamadas para atender as requisições que chegam até o servidor. Nesse momento deve-se especificar também em qual fase o módulo deve tratar as requisições. Por exemplo, na Figura 3.8 a função **meu_manipulador** implementa um gerador de conteúdo ou manipulador *(handler)* e, seu endereço para chamada *(upcall)* é registrado no Apache através da função *ap_hook_handler*. A partir deste momento, a função **meu_manipulador** será chamada a cada vez que uma nova requisição alcançar a fase do gerador de conteúdo. Esta função irá tratar as requisições de acordo com as políticas estabelecidas pelo código contido na função (Figura 3.9).

```
static void register_hooks(apr_pool_t * pool)
{
    ap_hook_handler(meu_manipulador, NULL, NULL, APR_HOOK_MIDDLE);
}
```

Figura 3.8: *Registro de módulos.*

Alguns manipuladores podem ser declarados de forma similar ao apresentado aqui, porém em diferentes fases de processamento das requisições. Algumas funções comumente utilizadas são:

1. *ap_hook_post_read_request*: primeira chance para verificar uma requisição logo após ela ser aceita;
2. *ap_hook_fixups*: última chance de verificar a requisição antes do gerador de conteúdo;
3. *ap_hook_log_transaction*: interceptador da fase de *log*.

Existem ainda outras fases designadas para propósitos específicos como, por exemplo, módulos de acesso e autenticação possuem interceptadores específicos para verificar as permissões.

```
static int meu_manipulador (request_rec* r)
{
    /*manipula a requisição*/
}
```

Figura 3.9: *Função de tratamento das requisições no Apache.*

A estrutura *request_rec* [7] é a principal estrutura de dados do Apache, ela representa todos os aspectos de uma requisição HTTP. Esta estrutura é criada quando o Apache aceita a requisição e é possível através das API´s *(Application Programming Interface)* oferecidas pelo Apache acessar essa estrutura e alterar seus valores. Os resultados possíveis de se obter na função **my_handler** são:

- **OK:** A função manipulou a requisição corretamente. A fase de manipulação é concluída;
- **DECLINED:** A função não está interessada nessa requisição. Possivelmente outro manipulador irá tratar a requisição;
- **Qualquer código de resposta HTTP:** Uma condição de erro ocorreu aconteceu durante o processamento da requisição. O processo servidor é abortado e um documento contendo uma mensagem de erro é retornado para o cliente.

3.3 Caracterização de Tráfego *Web* e Geração de Carga Sintética

Uma importante conseqüência da expansão da Internet está relacionada com o crescimento na complexidade do tráfego experimentado nestas redes [24]. O desenvolvimento de modelos que permitam representar o desempenho de servidores *web* é uma importante área de estudo, pois bons modelos permitem que se possa avaliar a capacidade de um servidor. Vários trabalhos vêm sendo propostos na literatura [12, 14, 57, 59], introduzindo modelos que tentam refletir as características comportamentais dos clientes e servidores.

3.3.1 SURGE

Um dos trabalhos mais citados na literatura é o modelo chamado SURGE *(Scalable URL Reference Generator)* proposto por [14]. O SURGE é baseado em um autômato *ON-OFF* que captura as características comportamentais do sistema e distribuições de probabilidade [34] para caracterizar o tamanho dos arquivos armazenados no servidor. Quando o sistema está no estado *ON*, a sessão está ativada enviando os objetos requisitados na sessão. O intervalo de tempo entre os arquivos enviados durante a sessão é denominado de tempo *active-off*. O tamanho dos arquivos e o número de referências em uma sessão de usuário também são utilizados. As principais variáveis deste modelo são apresentadas abaixo:

- **Tempo OFF:** tempo que o usuário permanece pensando, modelado por uma distribuição Pareto (cauda-pesada).
- **Tamanho dos arquivos:** tamanho dos objetos transmitidos. Normalmente modelado por uma distribuição Pareto.
- **Número de referências:** número de arquivos transmitidos em uma sessão de usuário. Também modelado normalmente por uma distribuição Pareto.
- **Tempo active-off:** é o intervalo de tempo entre os arquivos transmitidos em uma sessão de usuário. Modelada pela distribuição Weibull.
- **Popularidade:** número relativo de acessos realizados e um arquivo individual.
- **Localidade temporal:** a localidade temporal indica que, uma vez tendo sido requisitado um arquivo, a probabilidade de ele ser novamente requisitado no futuro aumenta.

A principal desvantagem no uso do SURGE é o fato da carga sintética ser gerada *pré-runtime*. Ou seja, a carga sintética que essa ferramenta cria é um arquivo, gerado antes da execução do servidor *web*, contendo, na forma de um *log*, todas as requisições que o servidor irá receber. Essa abordagem dificulta o uso do SURGE em abordagens dinâmicas, como é o caso do trabalho desenvolvido nesta dissertação.

3.3.2 *httperf*

Com a finalidade de mensurar o desempenho de servidores *web* foi proposta em [47] a ferramenta de geração de carga sintética chamada *httperf*. O *httperf* permite a geração de

carga utilizando o protocolo HTTP/1.1. Na Figura 3.10 é mostrado um exemplo de linha de comando de entrada utilizado pelo *httperf*.

```
httperf --server hostname --port 80 --uri /teste.html
--rate 150 --num-conn 27000 --num-call 1 --timeout 15
```

Figura 3.10: *Exemplo de comando httperf.*

Este comando busca o servidor *web* localizado no endereço IP *hostname*, rodando na porta 80. A página retornada é "/teste.html", no caso deste simples exemplo a página é requisitada repetidamente. A taxa em que as requisições são efetuadas é de 150 por segundo. O teste em questão executa 27000 conexões e em cada conexão é dirigida uma chamada HTTP.[2] O *timeout* indica o tempo em que um cliente irá esperar para receber a resposta de sua requisição. Caso o *timeout* expire a ferramenta considera que a chamada falhou. É importante ressaltar que, com um total de 27000 conexões a uma taxa de 150 requisições/segundo, o tempo de duração total do teste é de aproximadamente 180 segundos, independentemente de qual carga o servidor pode atender naquele instante. Existe também a possibilidade de representar o tempo de pensar do usuário *(think-time)*, porém, visando sobrecarregar o servidor, esse recurso não foi utilizado.

Uma vez finalizado o teste, vários dados estatísticos são impressos (Figura 3.11).

```
Total: connections 27000 requests 26701 replies 26701 test-duration 179.996 s

Connection rate: 150.0 conn/s (6.7 ms/conn, <=47 concurrent connections)
Connection time [ms]: min 1.1 avg 5.0 max 315.0 median 2.5 stddev 13.0
Connection time [ms]: connect 0.3

Request rate: 148.3 req/s (6.7 ms/req)
Request size [B]: 72.0

Reply rate [replies/s]: min 139.8 avg 148.3 max 150.3 stddev 2.7 (36 samples)
Reply time [ms]: response 4.6 transfer 0.0
Reply size [B]: header 222.0 content 1024.0 footer 0.0 (total 1246.0)
Reply status: 1xx=0 2xx=26701 3xx=0 4xx=0 5xx=0

CPU time [s]: user 55.31 system 124.41 (user 30.7% system 69.1% total 99.8%)
Net I/O: 190.9 KB/s (1.6*10^6 bps)

Errors: total 299 client-timo 299 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

Figura 3.11: *Exemplo de resultado apresentado pelo httperf.*

[2]Uma chamada HTTP consiste em enviar uma requisição e receber uma resposta.

No relatório de resultados é possível verificar seis grupos de dados: resultados gerais, dados pertencentes a conexões TCP, dados das requisições que foram enviadas, dados das respostas recebidas, utilização da CPU e da rede e um resumo dos erros (erros de *timeout* são os mais comuns em situações de sobrecarga).

3.3.3 Outras propostas

Alguns trabalhos utilizam *traces* de *logs* de servidores *web*, como o efetuado na Copa do Mundo de 1998, que possui mais de quatro milhões de acessos registrados [9]. A adoção deste tipo de abordagem apresenta a desvantagem de uma granularidade muito grossa, pois servidores *web* costumam armazenar *logs* com intervalos de 1 segundo.

3.4 Trabalhos sobre Servidores *Web* com QoS

A seguir serão apresentados alguns trabalhos importantes encontrados na literatura que tratam de qualidade de serviço em servidores *web*.

3.4.1 WebQoSL – Qualidade de Serviço em Servidores HTTP

No modelo WebQoSL [51] é implementada QoS em servidores *web* utilizando prioridades nas requisições e alocando recursos do servidor. Segundo este trabalho, apesar de os parâmetros de QoS dependerem do tipo de serviço oferecido pelo servidor, exemplos típicos utilizados são: atraso de transmissão, taxa de transferência na rede e resolução da imagem.

Segundo o modelo proposto, a visão de QoS é dada sob dois pontos de vista:

1. **visão dos clientes:** que desejam garantias de serviços específicas para suas requisições. Por exemplo, garantias de baixa vazão (bytes/segundo) ou limites superiores para tempos de resposta de certas requisições.
2. **visão do servidor:** que preocupa-se em como será provida QoS nos seus serviços. Isto inclui selecionar prioridades entre as requisições e limites nos recursos usados no servidor por essas requisições.

Na especificação das restrições de QoS das requisições, o autor aponta também duas classificações:

1. **Restrições centradas no servidor:** dependem apenas dos atributos do servidor. Tais restrições não distinguem entre diferentes requisições para uma mesma página.

2. **Restrições centradas no cliente:** dependem também dos atributos dos clientes. Neste caso, requisições para uma mesma página podem ser diferenciadas.

No modelo *WebQoSL* as páginas *web* são projetadas como objetos e as requisições para acessar as páginas como invocações de métodos. Por exemplo, uma invocação do tipo <página abc>.ler (p1,p2...pn) indica uma requisição para ler <página abc>. Os campos p1, p2...pn denotam parâmetros das requisições. Desta forma, um servidor HTTP pode ser visto como um sistema que gerencia execuções de métodos.

O artigo trata apenas de restrições absolutas, embora *WebQoSL* suporte restrições relativas, escaláveis (com suporte à adição de novos recursos) e com limites de tempo (permite relações temporais entre as páginas). O exemplo apresentado a seguir ilustra a idéia de restrições relativas, controlando a quantidade de recursos que o servidor aloca para as requisições:

<www.commerce.com/free>.server_resource < 0.1
<www.commerce.com/paid>.server_resource > 0.5

Especifica que os documentos contidos no diretório *free* devem receber 10% dos recursos do servidor enquanto os documentos do diretório *paid* recebem pelo menos a metade dos recursos do servidor.

O modelo de QoS foi implementado como um sistema distribuído de servidores *web* (Figura 3.12). Cada servidor que compõe a arquitetura foi implementado a partir de modificações feitas no servidor *web* NCSA.

O *QoS daemon* (Figura 3.12) é responsável por manter informações globais para os servidores e escalonar as requisições HTTP. Isso mantém a qualidade do modelo de serviço para várias páginas, indicando prioridades e recursos associados a diferentes requisições. Uma fila global trata das requisições HTTP e um modelo de recursos global indica a capacidade de cada servidor da arquitetura. O servidor de comunicação de cada estação é responsável por enviar mensagens entre o servidor e o *QoS daemon* e implementa o modelo de recursos. O modelo de recursos especifica a capacidade de cada servidor em dado instante (utiliza como métrica o número de bytes/segundo que o servidor pode entregar). Esta capacidade é

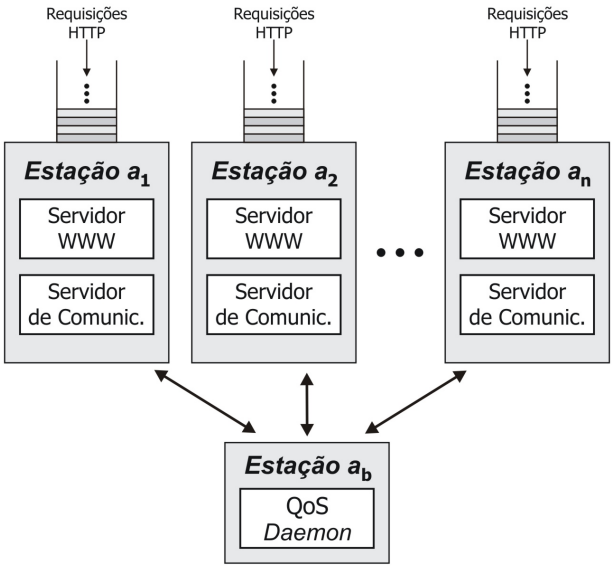


Figura 3.12: *Arquitetura do WebQoSL.*

periodicamente calculada e enviada para o QoS daemon, que então pode atualizar o modelo de recursos global.

Quando um servidor recebe uma requisição, este só pode atender a requisição se as restrições de QoS não forem violadas. O servidor então envia uma "pergunta" para o *QoS daemon* perguntando o que ele deve fazer. O *QoS daemon* pode responder de uma das três formas a seguir: processe a requisição, rejeite a requisição ou redirecione a requisição para outro servidor da arquitetura. As decisões tomadas pelo *QoS daemon* são baseadas no modelo de recursos global, que mantém a capacidade de processamento de cada estação. Um dos objetivos desse algoritmo de escalonamento e tentar satisfazer as restrições dos recursos e ao mesmo tempo utilizar o servidor de forma eficiente. Se um servidor não está sobrecarregado, a alocação de recursos para várias requisições vai ser refletida em várias requisições atendidas. Isto indica que restrições só são utilizadas em condições de sobrecarga.

O artigo apresenta uma análise de resultados com base em alguns experimentos efetuados. Um dado observado foi que os tempos de resposta obtidos com o *WebQoSL* permanecem justos e constantes, enquanto os tempos de resposta de servidores *web* padrão aumentam. Este fenômeno se deve ao fato de o modelo proposto descartar requisições depois que estas aguardam na fila por um tempo determinado. Observou-se, também, que o *throughput* do servidor *WebQoSL* é levemente inferior do que o servidor *web* padrão NCSA.

3.4.2 Diferenciação de Serviços no Sistema Operacional nos Níveis de Usuário e de *Kernel*

O modelo introduzido em [4] apresenta a implementação de um servidor com características de QoS, onde clientes podem pagar taxas para receberem melhor QoS pelos serviços requisitados. A implementação do modelo foi concebida através do escalonamento de requisições com prioridades, no sistema operacional, em ambos os níveis, de *kernel* e de usuário. Na abordagem no nível de usuário, o servidor *web* Apache [8] foi modificado com a adição de um escalonador de processos, responsável em decidir em qual ordem as requisições devem ser tratadas. O escalonador restringe o número máximo de processos destinados a servir requisições de cada prioridade. Na abordagem no nível de *kernel*, foram instrumentalizados o servidor *web* Apache e o *kernel* do sistema operacional Linux com a inserção de duas novas chamadas de sistema que permitem um mapeamento das prioridades das requisições em prioridades de processos, respeitando os níveis de prioridades também a nível se sistema operacional, indicando, por exemplo, qual processo deve utilizar a CPU naquele instante.

Existem dois importantes componentes para a política de escalonamento:

1. **política de descanso *(sleep policy)*:** é necessária quando uma nova requisição é recebida. A política deve decidir processar a requisição imediatamente ou postergá-la. A postergação é necessária quando a carga do sistema já está elevada e o processamento de outra requisição aumentará a latência (tempo de resposta) das requisições que já estão em execução, ou se a requisição é de baixa prioridade, subsequentemente chegando uma requisição de alta prioridade esta possivelmente será afetada.

2. **política de despertar *(wakeup policy)*:** é responsável por decidir qual requisição postergada deve continuar sua execução. A política de despertar é ativada quando uma requisição é completada. Em ambos os casos, as decisões são tomadas baseadas em limiares para o número máximo de requisições que podem ser manipuladas ao mesmo tempo em cada nível de prioridade. Existe um número fixo de entradas *(slots)* para cada prioridade e cada requisição de chegada deve ocupar uma das entradas, por exemplo, executar ou aguardar em uma fila até que esta seja liberada para a execução.

A política de escalonamento decide a ordem na qual as requisições devem ser atendidas. Existem dois tipos de políticas de escalonamento:

1. **política de conservação de trabalho *(work-conserving)*:** não permite que uma entrada fique vazia, isto é, se existem muitas requisições de alta prioridade, então as requisições de menor prioridade terão permissão para ocupar uma entrada pertencente a uma requisição de alta prioridade. Para a abordagem de *kernel* a política de escalonamento é preemptiva .

2. **política de não-conservação de trabalho *(non-work-conserving)*:** nunca permite uma requisição de baixa prioridade ocupar uma entrada de alta prioridade.

As implementações das políticas de escalonamento são similares para a abordagem de nível de usuário e para a abordagem de nível de *kernel*. A principal diferença é que a abordagem de nível de *kernel* pode *preemptar* um processo que já está executando, enquanto a abordagem de nível de usuário usa escalonamento não-preemptivo.

Os experimentos realizados demonstraram que restringir o número de processos que executam simultaneamente é uma estratégia simples e eficiente para obtenção de QoS. Na abordagem de nível de usuário, utilizando a política de não-conservação de trabalho, obteve-se melhoria no desempenho de 26% para requisições de alta prioridade, enquanto degradou-se as requisições de baixa prioridade em cerca de 500%, comparando-se com a versão original do servidor *web* Apache. Para a abordagem de nível de *kernel* observou-se melhora utilizando a política de não-conservação de trabalho: as requisições de alta prioridade obtiveram melhora de 24% ao custo da redução de trabalho de 208% das requisições de baixa prioridade.

Um aspecto interessante verificado é que esquemas de conservação de trabalho perdem a capacidade de diferenciar serviços se requisições menos prioritárias inundarem as entradas pertencentes a requisições mais prioritárias. Dessa forma, a única alternativa é utilizar a política de não conservação de trabalho.

3.4.3 Diferenciação de Serviços em Servidores *Web* no Nível de Aplicação

O trabalho [27] apresenta três mecanismos para prover níveis diferentes de serviço no nível de aplicação: limitando o tamanho do *pool* de processos, diminuindo a prioridade dos processos e limitando a taxa de transmissão.

Em linhas gerais, o trabalho apresenta um modelo que possui dois níveis de serviço: uma classe de requisição de primeiro plano *(foreground)* e uma classe de requisições de segundo plano *(background)*. A idéia central é que requisições de segundo plano, definidas com baixa

prioridade, apenas entram em execução no sistema se sua presença não diminuir o desempenho das requisições de primeiro plano concorrentes. Isto é, as requisições de segundo plano não recebem necessariamente o serviço de melhor-esfoço *(best-effort)*, mas sim de menor-esforço *(less-effort)*, pois estas só devem ser processadas ou transmitidas caso existam recursos ociosos disponíveis. Caso contrário, estas requisições são postergadas ou descartadas.

Os autores apontam três exemplos de utilização do modelo em servidores *web*. O primeiro é na técnica de *cache*. Com os servidores *web* atuais, tráfego conhecido (requisitado com freqüência) e desconhecido só podem ser enviados como tráfego de primeiro plano. A transferência antecipada é realizada pelo *cache* e o servidor precisa balancear a quantidade de tráfego conhecido enviado, para uma futura redução de tráfego devido a *hits* de *cache*. Se requisições conhecidas podem ser servidas em segundo plano, então a interferência com tráfego desconhecido pode ser eliminada, resultando assim em uma considerável melhora de desempenho.

O segundo exemplo relacionado com servidor *web* assinala diferentes prioridades para as requisições com base nos objetos requisitados. Por exemplo, uma página *web* é formada por partes HTML e imagens que um cliente geralmente solicita ao mesmo tempo. De qualquer forma, as respostas HTML são mais importantes para o navegador *web*, porque elas levam à *renderização* de toda a página. Desta forma, as imagens devem ser executadas em segundo plano, dando prioridade para as requisições HTML.

O terceiro exemplo apresenta a implementação de QoS com a adoção de políticas externas. O primeiro passo para a construção desse modelo é localizar o gargalo do sistema, ou seja, o recurso mais solicitado. Isto é importante, pois o recurso mais utilizado do sistema é um fator decisivo para ditar o desempenho do sistema.

Os autores apresentaram três algoritmos com o objetivo de reduzir as requisições de segundo plano, deixando assim mais recursos disponíveis para as requisições de primeiro plano. Os três algoritmos são descritos abaixo:

1. O primeiro mecanismo limita o uso dos recursos, restringindo o número de execuções simultâneas de processos de segundo plano. Se o número máximo de requisições já estiver sendo executado, então as requisições de segundo plano que chegarem serão descartadas (o valor máximo admitido no trabalho foi cinco).

2. O segundo mecanismo é idêntico ao primeiro, mas este também diminui a prioridade dos processos de segundo plano.

3. O terceiro mecanismo limita a taxa de transmissão da rede dos processos de segundo plano, coordenando e escalonando suas operações de envio. Processos de segundo plano têm intencionalmente suas transmissões atrasadas. Em adição, com este mecanismo também é limitado o número de processos de segundo plano (para cinco), sendo que cada um executa na menor prioridade.

A validação do trabalho foi concebida através de experimentos utilizando o servidor *web* Apache. Em todos os experimentos, as cargas de primeiro plano se mantiveram fixas enquanto as cargas de segundo plano cresceram durante todo o tempo. Muitas vezes, as requisições de segundo plano receberam negação de serviço, ou seja, foram descartadas ou postergadas por longos períodos de tempo.

3.4.4 Classificação e Escalonamento de Requisições em Servidores *Web*

O trabalho [16] propõe uma arquitetura escalável e transparente com a finalidade de prover diferenciação de serviços, chamada de *WebQoS*. Os dois principais objetivos da arquitetura são: (i) gerenciar efetivamente os picos de requisições HTTP dos clientes e, (ii) suportar diferentes níveis de serviço, servindo preferencialmente alguns usuários.

Na arquitetura *WebQoS*, as requisições de entrada são classificadas segundo alguma política, reordenadas e então enviadas de forma transparente para o servidor *web* para que sejam processadas. A seguir serão apresentados cinco componentes que fazem parte da arquitetura (Figura 3.13):

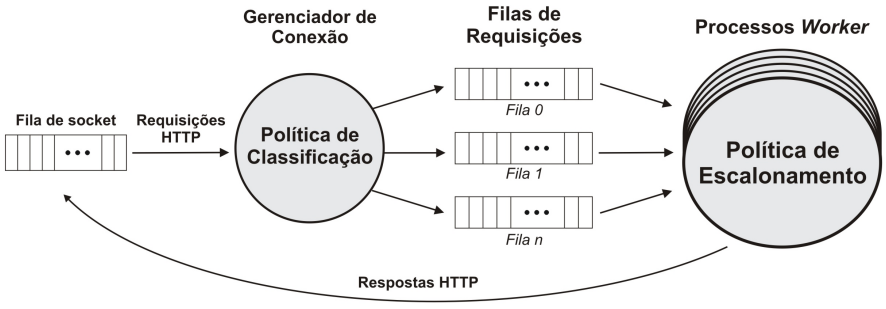


Figura 3.13: *Arquitetura de classificação de serviços [16].*

1. **Gerenciador de conexões:** é responsável por interceptar todas as requisições HTTP dos clientes e classificá-las, colocando-as em filas apropriadas de acordo com sua classe

de serviço. O gerenciador de conexões deve executar o tempo suficiente para manter as filas cheias e aceitar a maior quantidade de requisições de alta prioridade *(premium)* possível.

2. **Classificador de requisições:** a classificação pode ser realizada de duas maneiras:

   - Baseada na classificação dos usuários, caracterizando as requisições pela sua origem. No trabalho em questão foram apontadas três maneiras possíveis de uma fonte ser identificada: o endereço IP dos clientes, os *cookies* HTTP e os *plugins* que também podem conter identificadores dos clientes embutidos no corpo de uma requisição HTTP.
   - baseada na URL *(Universal Resourse Locator)* da requisição ou no nome do arquivo.

3. **Controle de admissão:** quando um servidor encontra-se em um estado de sobrecarga, as classes (privilegiadas ou não) sofrem degradação da qualidade de serviços e muitas vezes chegam a sofrer negação de serviço. Para evitar esse comportamento, requisições de baixa prioridade são rejeitadas.

4. **Escalonador de requisições:** depois que uma requisição é classificada e passa pelo controle de admissão, um processo trabalhador *(worker)* seleciona uma das classes de serviço para ser atendida de acordo com uma política de escalonamento. Este artigo apresenta cinco políticas, apresentadas abaixo de forma sucinta:

   - *Prioridade total:* requisições de classes inferiores só serão executadas se não existirem requisições de classes privilegiadas.
   - *Prioridade com peso:* existem pesos associados às classes e o atendimento às requisições é efetuado de acordo com esses pesos. Por exemplo, se uma classe possui peso três vezes maior que outra classe, essa terá três vezes mais requisições escalonadas.
   - *Capacidade compartilhada:* é determinado um conjunto de capacidades para as classes privilegiadas e alguma classe menos privilegiada pode não receber nenhuma capacidade (similar à técnica de conservação de trabalho já citada [4]).
   - *Capacidade fixa:* como no algoritmo acima, exceto pelo não compartilhamento entre as classes.
   - *EDF (Earliest Deadline First):* a requisição que tiver o seu deadline absoluto mais próximo é executada primeira.

5. **Escalonador de recursos:** garante que requisições de alta prioridade serão executadas por processos de alta prioridade no sistema operacional.

Um protótipo foi criado modificando o servidor *web* Apache. Os experimentos realizados mostraram a eficácia do modelo para os clientes de classes privilegiadas *(premium)* que obtiveram resultados amplamente melhores do que as classes menos privilegiadas ou básicas, que muitas vezes receberam negação de serviço.

3.4.5 Controle de Admissão em Servidores *Web*

Quando a taxa de chegada de requisições *web* cresce acima da capacidade, as filas e os tempos de resposta aumentam exageradamente. Um usuário que experimenta longos tempos de resposta em atendimento a uma requisição, provavelmente abandonará o *site*. Portanto, a adição de mecanismos de controle de admissão em servidores *web* é de grande importância, e tem suscitado interesse de grupos de pesquisa, resultando em vários trabalhos [1, 2, 16, 22].

Servidores *web* têm comportamento não-linear e estocástico (estatístico/aleatório), e a técnica matemática de teoria das filas pode ser uma utilizada para modelar os servidores *web*. No entanto, não existe ferramenta matemática em teoria das filas para projetar mecanismos de controle de admissão. Um controle de admissão pode ser [6]:

- **Controle de admissão estático:** admite um número fixo de requisições
- **Controle de admissão dinâmico:** este mecanismo possui um controlador que, buscando algum objetivo de controle, calcula a nova taxa de admissão em intervalos de tempo periódicos.

O artigo [6] apresenta um mecanismo de controle de admissão utilizando técnicas de controle. Segundo este trabalho, no desenvolvimento do mecanismo de controle de admissão uma importante escolha é a variável a ser controlada, ou seja, qual dos gargalos do sistema será medido (fila, processador, memória, acesso a disco, tempos de resposta, taxa de requisições, tamanho da fila, etc). A variável de controle está muitas vezes relacionada com a demanda de QoS que os cliente precisam receber. Tradicionalmente, servidores *web* costumam utilizar o comprimento das filas como variável de controle, no desenvolvimento de mecanismos de controle de admissão.

O mecanismo de controle sugerido em [6] é mostrado na Figura 3.14. O modelo é composto por três partes: um monitor, um controlador e uma entrada. O Monitor é responsável por medir a variável de controle x. Usando o valor da variável de controle o controlador decide o valor da taxa U de requisições que podem ser admitidas no sistema. O objetivo do controlador é manter o valor da variável U o mais próximo de $Xref$ possível. A entrada rejeita as requisições que não podem ser aceitas, as requisições admitidas acessam o sistema.

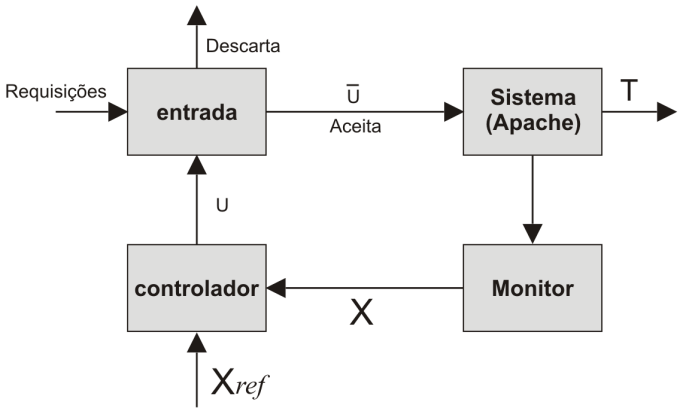


Figura 3.14: *Mecanismo de controle de admissão.*

Para a validação do modelo foi utilizado o servidor *web* Apache e o mecanismo de controle de admissão foi implementado em Java em plataforma Windows. A geração de carga HTTP foi realizada através do gerador de carga sintética S-Client. A distribuição utilizada é a Poisson [34].

Para o bom funcionamento do sistema, a escolha dos parâmetros de controle é uma tarefa fundamental. Segundo [6], todos os artigos publicados até o momento sobre análise de sistemas de filas têm utilizado apenas modelos lineares determinísticos. Porém é válido lembrar que um modelo linear em um sistema não-linear não é correto. O ponto forte desse trabalho é que os modelos refletem o comportamento real, não-linear do sistema. O maior problema encontrado foi o fato de que sistemas de teoria de filas tais como os utilizados pelos servidores *web* apresentam comportamento não linear e estocástico, isto significa que são difíceis de serem analisados e modelados com métodos de teoria de controle.

3.4.6 Servidor *Web* com Adaptação de Conteúdo

Abdelzaher [1, 3] propõe uma nova abordagem para diferenciação entre classes de serviços utilizando adaptação de conteúdo, através da técnica de computação imprecisa e estra-

tégias clássicas de controle para dirigir o servidor em condições de sobrecarga. O trabalho foi validado a partir do desenvolvimento de um módulo externo ao servidor *web*, tornando assim o modelo completamente independente do servidor *web* utilizado, evitando assim que o código precise eventualmente ser recompilado ou modificado.

O autor cita alguns fatores que motivaram como sua escolha a *web* para validar sua estratégia:

- A Internet está experimentando constante crescimento há algum tempo, mudando a maneira com que as pessoas se comunicam;
- Muitos serviços estão sendo realizados através da infra-estrutura da Internet atualmente (comércio eletrônico, videoconferência, voz sobre IP, etc).
- Os servidores *web* estão no centro das mudanças do uso da Internet, necessitando cada vez mais de qualidade de serviço, diferenciação entre as classes de serviço, confiabilidade e garantias de segurança em um ambiente construído sobre um modelo de serviço de melhor esforço.

No modelo apresentado, o conteúdo das requisições é adaptado em conformidade com as condições de carga do servidor. Este esquema não apenas permite que mais clientes acessem o servidor em situações de sobrecarga, mais também reduz a quantidade de recursos desperdiçados quando a capacidade do servidor é excedida. Um experimento realizado mostra que em condições de extrema sobrecarga (cerca de três vezes a capacidade do servidor) mais de 50% dos recursos do sistema chegam a ser desperdiçados. Estes recursos podem ser desperdiçados por requisições abortadas por excederem os tempos limites de seus clientes *(timeouts)*, por exemplo.

Além da abordagem relacionada com a sobrecarga do sistema, a técnica de adaptação de conteúdo tem outro importante benefício. Por exemplo, um determinado usuário pode ter limitações relacionadas com seus recursos (memória, rede, processador, resolução de visualização, etc.) com relação a outros usuários. A adaptação de conteúdo pode fornecer versões mais apropriadas de conteúdo para cada cliente de acordo com as restrições de cada um.

A implementação de versões imprecisas para adaptação de conteúdo é dada essencialmente de três maneiras diferentes: degradação na qualidade das imagens através de métodos de compressão, diminuição no número de objetos nas páginas e redução de *links* locais (Seção 2.7.1).

Experimentos efetuados apontam claramente melhoria no desempenho com versões imprecisas. De 30% a 90% das páginas com adaptação de conteúdo analisadas, apresentam melhoria de desempenho de pelo menos 400%. A opção de compressão para as versões imprecisas dos sites pode ser realizada de duas formas: *on-the-fly* introduzindo um *overhead* de processamento ou pré-processada *a priori* e armazenada em um diretório de versão imprecisa de uma página, por exemplo. Em situações de sobrecarga do servidor, as imagens contidas nesse diretório de versão imprecisa são carregadas, lembrando que se pode, dependendo da situação de sobrecarga, enviar apenas texto em resposta às requisições.

Como mencionado anteriormente, o modelo proposto não necessita que se altere o código do servidor *web* nem que o mesmo seja recompilado. Para implementar essa transparência existem duas possibilidade, ambas assumindo a existência de versões imprecisas armazenadas previamente em um diretório:

1. **Abordagem de processos externos:** nessa abordagem processos são executados de forma concorrente com o servidor *web*, e a adaptação de conteúdo é feita através da troca de *link* do diretório de conteúdo completo para um outro diretório compatível com as condições de carga do servidor;

2. **Abordagem de *middleware*:** a abordagem de *middleware* intercepta as requisições dos usuários e altera a URL *(Universal Resource Locator)* da requisição referente à árvore de adaptação de conteúdo correta.

Em suma, o software de adaptação de conteúdo é formado pelos seguintes componentes (Figura 3.15):

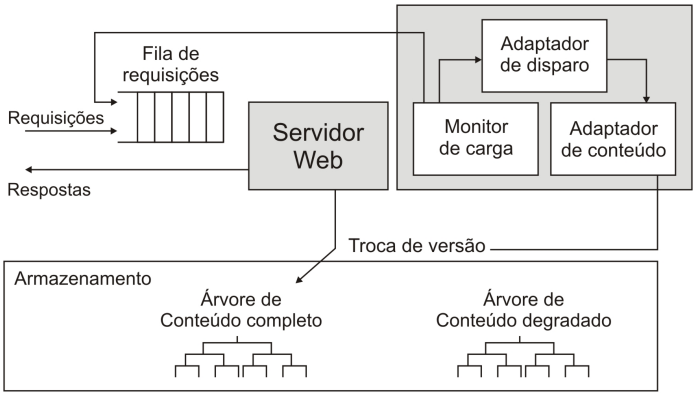


Figura 3.15: *Modelo de Adaptação de Conteúdo.*

- **Monitor de carga:** A carga do servidor *web* é monitorada com a finalidade de detectar situações de sobrecarga. Para conhecer a carga do servidor, o monitor envia requisições HTTP para o servidor que, de acordo com a política FIFO *(First-in-First-out)*, agrupa todas as requisições em uma fila (por ordem de chegada). Em seguida, o monitor analisa os tempos de resposta das requisições enviadas; requisições com tempos de resposta elevados significam fila cheia (servidor sobrecarregado), enquanto requisições com valores de tempo de resposta baixos indicam fila vazia ou com carga normal (sub-utilização do servidor).

- **Adaptador de disparo:** Verifica o valor de carga monitorado e decide invocar, desfazer ou mudar para a degradação de conteúdo apropriada.

- **Adaptador de conteúdo:** Uma vez adicionado o adaptador de disparo, indicando as condições do servidor (sobrecarga ou sub-utilização), este tem a função de restaurar a carga do servidor para as condições desejadas, se for possível.

- **Classificador de requisições:** Geralmente a decisão do adaptador de conteúdo deve ser tomada levando em consideração a idéia de punir de maneira menos severa as requisições de maior prioridade. O classificador permite dar tratamento adequado para as classes de requisições dos clientes mais importantes, como garantia de QoS.

O trabalho estudado apresenta excelentes resultados referentes à adaptação de conteúdo para prover QoS. Utiliza também técnicas de controle clássico para medir taxas de utilização do servidor. A maior contribuição do trabalho, segundo o próprio autor, foi a grande quantidade de clientes que puderam ser atendidos com versões imprecisas (conteúdo degradado).

Uma dificuldade observada diz respeito à mudança de versões. Por exemplo, um servidor *web* pode estar trabalhando com filas relativamente vazias e apresentando tempos de resposta ótimos, porém quando o adaptador de conteúdo seleciona versões precisas (com alta qualidade) para atender as requisições, pode acarretar em situações de sobrecarga inaceitáveis. Uma solução seria a escolha de uma outra métrica para avaliar a sobrecarga do servidor que não fosse o tempo de resposta. Uma possibilidade seria avaliar os gargalos do sistema medindo o uso da CPU, porém um problema com essa métrica diz respeito à implementação de processo com *busy-wait*, ocupando assim quase 100% da CPU, mesmo quando o servidor estiver ocioso. Portanto, o ideal é juntar os dois gargalos do sistema e fazer uma aproximação utilizando um método de aproximação linear.

3.4.7 Escalonamento de Conexões TCP

Em [48] é apresentada uma política de escalonamento que leva em consideração a interação entre o servidor e os *hosts* TCP. Esta política, chamada FCF *(Fastest Connection First)*, prioriza as requisições HTTP baseada no tamanho dos arquivos requisitados e no *throughput* das conexões dos usuários (menores tempos de conexão). As requisições para arquivos com tamanhos menores recebem as maiores prioridades no servidor. O objetivo desta política é melhorar os tempos de resposta das requisições. O artigo foi avaliado através de simulações utilizando o gerador de carga sintética SURGE, e comparando os resultados com outras duas políticas: FIFO *(First-Come, First-Served)* e SRPT *(Shortest Remaining Processing Time)*. Observou-se que a política FCF obteve o melhor tempo de resposta enquanto SRPT o menor atraso no servidor. Conclui-se que a política FCF proposta neste trabalho apresenta melhores resultados quando as características da rede são bem conhecidas, no caso de bandas de alta velocidade, e a política SRPT apresenta melhores resultados quando a banda é de baixa capacidade.

3.5 Comparação entre os Trabalhos

As Tabelas 3.2 e 3.3 sintetizam as principais características das ferramentas e abordagens estudadas neste capítulo.

Tabela 3.2: *Ferramentas e abordagens para geração de carga web.*

	Uso dinâmico	**Granularidade entre requisições**
SURGE	não	adequada (ms ou menos)
httperf	sim	adequada (ms ou menos)
Log da Copa do mundo	sim	inadequada (1s.)

O gerador de carga sintética utilizado neste trabalho foi o *httperf*. Esta ferramenta permite executar *hits* de página pré-estabelecidas no servidor, oferece granularidade entre as requisições da ordem de ms (ou menos) e apresenta um relatório completo ao término de sua execução contendo informações sobre tempo de resposta (fim a fim), quantidade de descartes, número de pedidos de conexão, etc.

O SURGE, por outro lado, não permite que os acessos de usuários sejam dirigidos dinamicamente a páginas pré-estabelecidas; e o log da Copa do Mundo de 1998 – assim como outros *log web* – oferece granularidade entre as requisições muito grossa (1 segundo).

Tabela 3.3: *Características dos principais trabalhos estudados.*

	[6]	**[51]**	**[4]**	**[27]**	**[16]**	**[1, 3]**	**[48]**
Controle de admissão	sim	não	não	não	sim	não	não
QoS no Sistema Operacional	não	não	sim	sim	não	não	não
Usa grupo de servidores	não	sim	não	não	não	não	não
Posterga execução de requisições	não	não	sim	não	não	não	sim
Diferentes níveis de serviço	não	não	não	sim	sim	sim	sim
Adaptação de conteúdo	não	não	não	não	não	sim	não

Com relação ao controle de admissão, apenas os trabalhos [6] e [16] tratam dessa questão. Os trabalho [6] implementa um controle de admissão sofisticado, baseado em teoria de controle realimentado. O trabalho [16], por outro lado, implementa descarte com diferenciação de serviços absoluta, descartando sempre as requisições menos prioritárias.

Quase todos os trabalhos implementam QoS fora do sistema operacional, com exceção de [4] e [27]. O problema dessas abordagens é que sua utilização implica em forte dependência no sistema operacional empregado.

O único trabalho que utiliza grupo de servidores para prover QoS é [51], onde os servidores são agrupados como um sistema de distribuído de servidores *web* e as requisições de chegada do sistema podem ser escalonadas para qualquer um dos servidores do grupo, segundo uma política de escalonamento.

Os trabalhos [4] e [48] postergam as execuções de requisições menos prioritárias como estratégia para privilegiar requisições ditas "mais importantes". O principal problema oriundo dessas abordagens é o fato de que requisições menos prioritárias podem ter seu processamento postergado indefinidamente.

Com relação a diferenciação de serviços, os trabalhos [16], [1], [3] e [48] oferecem diferentes níveis de serviço. O trabalho [27] apresenta apenas dois níves de serviço (primeiro e segundo plano).

Os únicos trabalhos que oferecem a possibilidade de adaptação de conteúdo são [1] e [3], técnica essa utilizada no desenvolvimento deste trabalho.

3.6 Conclusões do Capítulo

Com relação aos trabalhos apresentados, várias restrições podem ser verificadas. Problemas referentes à diferenciação de serviços absoluta (penalizando demasiadamente algumas classes de serviço), não garantem espaçamento de qualidade entre as classes de serviço, outros não utilizam mecanismos de controle de admissão, causando assim em situações de sobrecarga, tempos de resposta inaceitáveis.

É importante observar que existem muitos outros trabalhos relacionados ao tema, sendo que alguns começaram a surgir recentemente. Vários trabalhos utilizam o servidor *web* Apache para validar seus modelos, mas não descrevem com detalhes como se deu a implementação. Além de descrever aspectos internos do Apache, este capítulo buscou mostrar os trabalhos que são mais referenciados por outros na literatura, e também alguns trabalhos que apresentavam semelhança com o estudado nesta dissertação.

Capítulo 4

Modelo de Provimento de QoS para Servidores *Web*

4.1 Introdução

O presente capítulo propõe um modelo de provimento de QoS para clientes de servidores *web* baseado em diferentes classes de serviços. O modelo proposto, e apresentado inicialmente em [56], é uma extensão do modelo introduzido anteriormente em [45] com a adição de um novo módulo de controle de admissão.

Para lidar com situações de sobrecarga, o modelo prevê a utilização de técnicas de escalonamento adaptativo com prioridades dinâmicas (módulo de atribuição de prioridades) e computação imprecisa (módulo de seleção de versões). A estrutura proposta no modelo permite que requisições recém-chegadas ao servidor sejam devidamente classificadas e encaminhadas a filas de prioridades diferentes, conforme o nível de serviço.

Além do modelo, neste capítulo é descrita uma nova abordagem de escalonamento de requisições *web* que, diferentemente da proposta originalmente apresentada em [45], modela os tempos de resposta máximos das requisições dos clientes na forma de *deadlines soft* (através de funções benefício). Finalmente, neste capítulo é detalhada a implementação de um protótipo deste modelo no servidor *web* Apache e diversos experimentos são efetuados, comparando o desempenho das diversas abordagens possíveis.

4.2 Modelo de provimento de QoS

O modelo adotado considera a existência de classes de serviço (Figura 4.1) [45, 56]. As classes são representadas por tipos de metais, e o nível de serviço oferecido pelo servidor é proporcional ao valor do metal. Na Figura 4.1 são apresentadas as classes de serviço ouro, prata e bronze – caracterizando um serviço olímpico [26]. Apesar da Figura 4.1 representar apenas três classes de serviço, novas classes de serviço podem ser acrescentadas ao modelo.

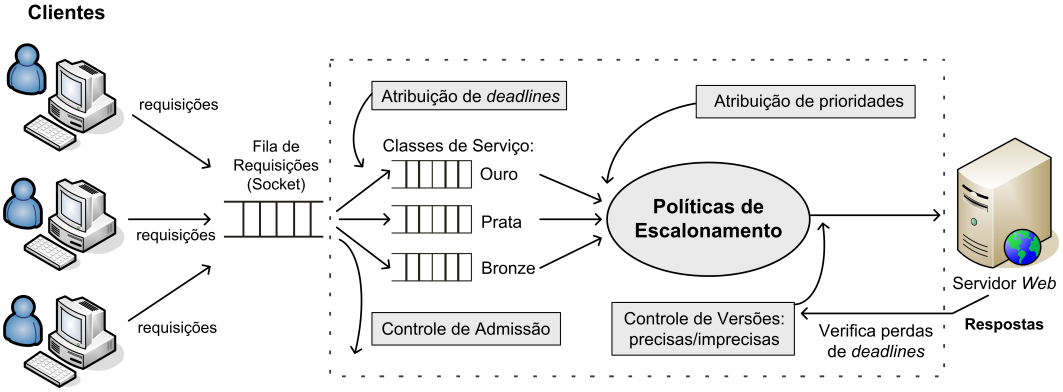


Figura 4.1: *Modelo de provimento de QoS para servidores web.*

Cada classe de serviço possui uma fila FIFO associada, responsável por armazenar suas requisições. Em cada fila, todas as requisições enfileiradas possuem a mesma prioridade da classe, a qual é atribuída dinamicamente. Essas prioridades são utilizadas para ordenar as requisições ao servidor *web*.

Existem diversas abordagens para detectar situações de sobrecarga em servidores *web* baseadas no monitoramento [1]: da ocupação da fila de *socket* do sistema operacional, da ocupação das filas de requisições do servidor, da utilização da CPU, ou tempos de resposta das requisições. Neste trabalho utiliza-se uma variação dessa última abordagem. Considerando que os clientes não irão esperar as respostas de suas requisições indefinidamente, é possível assumir uma condição de falha no servidor quando uma requisição ultrapassar um determinado valor de tempo. Esse valor de tempo é considerado como um *deadline* da requisição.

Interessante observar que existem alguns trabalhos que tentam apontar tempos de resposta "aceitáveis" para clientes *web*. Segundo [32] o valor de tempo de resposta aceitável para um requisição é de cerca de 12 segundos, enquanto para [50] este tempo é de 15 segundos. Para ilustrar os tempos de resposta praticados atualmente, a Tabela 4.1 apresenta os tempos de resposta dos dez mais importantes *web sites* de comércio eletrônico do mundo.

Tabela 4.1: *Tempos de Resposta de importantes web sites de comércio eletrônico ([32]).*

Posição	*web Sites*	Tempos de Resposta
1	Eddie Bauer	9,3 s
2	Office Depot	10,1 s
3	Amazon	11,5 s
4	Wall-Mart	14,6 s
5	Best Buy	15,0 s
6	JCPenny	15,2 s
7	Costco	15,8 s
8	Office Max	20,6 s
9	Target	23,3 s
10	Sears	23,5 s

Para o modelo proposto, no momento da chegada de uma requisição ao servidor, um *deadline* absoluto é atribuído à requisição. Quando o servidor atende uma requisição e envia a resposta ao cliente, é verificado se houve perda de *deadline* da requisição[1], ou seja, verifica se ocorreu uma falha temporal (ainda que a requisição tenha sido atendida por completo). Na abordagem de escalonamento adaptativo utilizada nesse trabalho, essa contabilização de falhas tem duas finalidades complementares:

1. É utilizada para detectar situações de sobrecarga e acionar o mecanismo adaptativo que tenta reduzir a carga no servidor, através da política de seleção de versões imprecisas.

2. É usada no cálculo do valor cumulativo que irá dirigir a política de atribuição dinâmica de prioridades. Caso o tempo levado para atender uma requisição ultrapasse o valor estipulado, é considerada uma condição de falha temporal (a requisição recebeu qualidade de serviço inadequada), e o valor cumulativo associado à execução é reduzido.

A política de atribuição de prioridades do modelo de adaptação (Figura 4.1) realiza uma atribuição dinâmica de prioridades às classes. Antes de ser executada pelo servidor, a política de seleção de versão determina se a execução da requisição será de forma precisa ou imprecisa. As duas políticas trabalham de forma integrada e são fundamentadas em um histórico de execução que armazena as k últimas execuções de cada classe.

[1]Para ser mais preciso, o *deadline* é atribuído no momento do estabelecimento da conexão TCP referente à requisição, e o teste de perda de deadline é efetuado no fechamento desta conexão. Importante observar, por outro lado, que os valores de tempo representados na Tabela 4.1 são coletados nos clientes e contabilizam também os atrasos na rede.

Testes anteriores realizados em [45] apontaram que, a partir de um determinado nível de sobrecarga, a abordagem de escalonamento proposta para o modelo (batizada de PCV – *Proportional Cumulative Value Attribution*) perdia a capacidade de realizar diferenciação proporcional de serviços. Portanto o modelo proposto neste trabalho incorpora um novo módulo de controle de admissão (Figura 4.1). Este módulo considera a existência de um mecanismo de controle de admissão que atua na fila de requisições, descartando, em situações de sobrecarga, as requisições recém-chegadas, evitando assim que o servidor fique ainda mais sobrecarregado e acabe desperdiçando os recursos do sistema (disco, largura de banda, CPU, etc) devido a *timeouts* e a cancelamentos dos clientes.

4.3 Valor Cumulativo das Classes

Antes de introduzir abordagens de escalonamento que podem ser adotadas neste modelo é importante discutir qual métrica é adequada para avaliar o desempenho das abordagens. Levando em consideração que o modelo assume a existência de mecanismos de controle de admissão e de controle de versões, torna-se inadequada a adoção de tempos de resposta como métrica. Isso ocorre por dois motivos: (i) considerando a possibilidade de execuções imprecisas, mensurar apenas os tempos de resposta das requisições não consegue quantificar a "perda de qualidade de serviço" advinda da escolha de uma versão imprecisa na execução de uma requisição; e (ii) da mesma forma, mensurar tempos de resposta não consegue quantificar a "a perda de qualidade" devido a descartes de requisições.

Como um exemplo extremo, considere uma abordagem de atendimento a requisições de um servidor *web* que simplesmente descarte todas as requisições que cheguem ao servidor (ou as aceite e execute todas na forma imprecisa). Essa abordagem iria ter tempos de resposta excelentes dando a falsa impressão (caso se adotasse os tempos de resposta como métrica) de ser uma boa abordagem.

Por conseguinte, a métrica adotada neste trabalho é o valor cumulativo, originalmente introduzido em [15], a qual busca mensurar o tempo gasto pelo servidor na execução das requisições. Considerando que as requisições possuem deadlines e que podem ser atendidas por diferentes versões, a definição do valor cumulativo VC obtido por uma requisição pode ser definido como [45]:

Quando uma requisição é atendida na sua forma imprecisa considera-se que ela oferece ao sistema um valor σ proporcional ao tempo gasto em seu atendimento. Ou seja, caso uma

$VC = 0$, se a requisição perdeu *deadline*.
$VC = 1$, se a requisição executou de forma precisa.
$VC = \sigma$, se a requisição executou de forma imprecisa, onde $0 < \sigma < 1$

página *web* tenha sido implementada na forma imprecisa e seu tempo de acesso represente 30% do tempo gasto para acessar a versão precisa, o valor cumulativo de uma requisição a essa página é de 0.3.

Cada classe de serviço do modelo possui um valor cumulativo calculado dinamicamente. Este valor é usado para guiar as políticas que irão implementar a diferenciação de serviços entre as classes. Para contabilizar os valores cumulativos, um histórico é mantido para cada classe. O histórico é uma *k-tupla*, que leva em conta as últimas k execuções de requisições de cada classe. Para cada novo estado produzido, o valor mais antigo no histórico é descartado e o novo estado é adicionado.

A manutenção do histórico atualizado permite o cálculo do valor cumulativo absoluto *(VCabs)* de uma classe i, realizado a partir do seguinte cálculo:

$$\forall\ classe_i \quad VCabs_i^k := VCabs_i^{k-1} - Vativ\_antiga_i + Vativ\_recente_i$$

Porém o valor cumulativo absoluto não mantém relação com os valores cumulativos das outras classes, tornando assim necessário o cálculo do valor cumulativo proporcional *(VC)*. Esse cálculo é alcançado através de uma normalização, primeiramente somando todos os valores absolutos das classes *(TotalVC)* e posteriormente realizando regra de três para cada classe i.

É importante ressaltar que depois de normalizados os valores, a seguinte propriedade deve ser verificada: $\sum_{i=1}^{n} VC_i = 1$.

4.4 Políticas de Atribuição de Prioridades e Seleção de Versões

O modelo é guiado pelas políticas de atribuição de prioridades e de seleção de versões que foram propostas inicialmente em [45]. Para implementação do modelo de adaptação, considera-se que cada classe de serviço possui um valor de qualidade (VQ) associado. Esse valor é uma percentagem atribuída estaticamente pelo administrador da rede, representando

a qualidade almejada para cada classe. A soma dos valores de qualidade de todas as classes deve totalizar 100%: $\sum_{i=1}^{n} VQ_i = 1$.

As políticas de atribuição de prioridades e de seleção de versões agem de forma integrada, objetivando manter os valores cumulativos das classes o mais próximo dos seus valores de qualidade estipulados. Ou seja, as duas políticas da heurística PCV agem buscando a situação ideal: $\forall_i \; VC_i = VQ_i$.

Quando o valor cumulativo de uma classe, em um determinado momento, é maior ou igual a seu valor de qualidade *($VC_i >= VQ_i$)*, a classe de requisições em questão encontra-se em um estado estável e portanto, a qualidade média dos serviços oferecidos àquela classe está acima ou dentro do esperado. Entretanto, quando o valor cumulativo de uma classe for menor que seu valor de qualidade, esta se encontra em um estado de falha. A política de atribuição de prioridades distribui as prioridades proporcionalmente ao valor $VQ_i - VC_i$ (classes estáveis possuem valores negativos e portanto, recebem as menores prioridades do sistema).

A política de seleção de versão é acionada pela ocorrência de condições de sobrecarga no servidor. A cada indicação de uma perda de *deadline*, uma classe é selecionada para executar na sua versão imprecisa. Classes que executam versões imprecisas têm seus valores cumulativos reduzidos. Dessa forma, quando precisa escolher uma classe para executar na forma imprecisa, a política de seleção de versão escolhe somente classes que estejam em um estado estável. Ou seja, a política de seleção de versões escolhe sempre a classe com o maior valor: $VC_i - VQ_i$. A busca por esse equilíbrio dinâmico é conquistado através da integração das duas políticas na heurística de escalonamento PCV *(Proportional Cumulative Value Attribution)*.

4.5 Política de Controle de Admissão

O controle de admissão tem como objetivo principal limitar a chegada de novas requisições ao servidor em situações de sobrecarga.

Esse trabalho utiliza um **controle de admissão estático**, ou seja, após uma determinada carga o servidor passa a cessar o atendimento a novas requisições. Essa carga é delimitada pelo tamanho da fila de espera das requisições, que após um determinado tamanho de fila, simplesmente descarta as requisições récem-chegadas. A escolha dessa abordagem simples ocorreu depois de diversos experimentos que indicaram que essa abordagem estática era suficiente para manter a diferenciação de serviços, mesmo em cargas muito altas.

4.6 Heurística PCV – *Proportional Cumulative Value Attribution*

A heurística de escalonamento é descrita através do pseudo-código 1. Na inicialização da aplicação da heurística (linhas 01-07), todas as classes começam com execução precisa, partindo de um estado estável *(VC = VQ)*, e recebem as mesmas prioridades (no caso, a menor prioridade do sistema). É possível se observar três fluxos de execução:

1. O primeiro fluxo (linhas 09-16) é responsável por receber as requisições, classificá-las e detectar, através do mecanismo de controle de admissão, situações de sobrecarga e nesse caso descartar as requisições recém chegadas (linha 11). Cada requisição descartada é contabilizada como uma perda de *deadline*. Para este trabalho, o limite adotado para o tamanho da fila de requisições é 600 (esse valor foi escolhido segundo experimentos que serão apresentados na Seção 5.5.2).

2. O segundo fluxo de execução (linhas 17-21) é responsável por pegar as requisições nas cabeças das filas de chegada de cada classe e aplicar a política de escalonamento.

3. No terceiro e último fluxo (linhas 22-31) é verificado no término de execução de cada requisição se esta perdeu *deadline*, atualiza o histórico e o valor cumulativo, e caso seja necessário, seleciona uma classe para executar na forma imprecisa em sua próxima ativação.

O modelo proposto não consegue prever a ocorrência de futuras sobrecargas e perdas de *deadline*, já que a política de seleção de versões (precisas/imprecisas) é dirigida pelas ocorrências de sobrecarga. A cada indicação de uma perda de *deadline*, uma classe é selecionada para executar na sua forma imprecisa (com redução de qualidade), tomando como base a sua distância para falha. A variável ($prox\_exec_j$) armazena a informação se a próxima execução de uma classe j será precisa ou imprecisa.

Ao término de atendimento de uma requisição, quando é detectada uma sobrecarga (linha 27), a política de seleção de versões escolhe a classe com maior valor de distância para falha, para executar na forma imprecisa (linha 28). Esta classe é então selecionada para executar na forma imprecisa na próxima requisição (linha 29).

Pseudo-código 1 Heurística de escalonamento PCV.

```
1: {Inicialização}
2: for all classe_j do
3:    VC_j ← VQ_j
4:    prio_j ← menor prioridade
5:    prox_exec ← "precisa"
6:    LIMITE ← 600
7: end for
8: loop
9:    ▷ Fluxo 1: ativado com a chegada de uma nova requisição da rede ◁
10:   if chegou requisição para uma classe_j then
11:      if ocupação fila > LIMITE then
12:         descarta requisição
13:      else
14:         enfileira requisição na fila de chegada da classe
15:      end if
16:   end if
17:   ▷ Fluxo 2: ativado quando há requisições a serem executadas ◁
18:   if ∃ requisição na cabeça da fila de requisicoes para classe_j then
19:      retira requisição de sua fila de chegada
20:   end if
21:   executa requisição com prio_j e versão prox_exec_j
22:   ▷ Fluxo 3: ativado ao término da execução de uma requisição ◁
23:   if terminou execução da requisição X para uma classe_j then
24:      prox_exec_j ← "precisa"
25:      testa se perdeu deadline e calcula VC_j
26:      atribui prioridades: prio_j ← (VQ_j − VC_j) {Quanto maior o valor, maior a prioridade.}
27:      if perdeu deadline then
28:         seleciona classe_k tal que prox_exec_k = "precisa" ∧ k possua maior(VC-VQ)
29:         prox_exec_k ← "imprecisa"
30:      end if
31:   end if
32: end loop
```

4.7 Atribuição de Valores Cumulativos em Ativações Precisas e Imprecisas

No trabalho original proposto em [45] o valor cumulativo era calculado considerando que as requisições possuiam *deadlines* firmes; ou seja, conforme já foi descrito na Seção 4.3, Q = 0 caso a requisição perca seu *deadline*.

O valor cumulativo, como visto na Seção 4.3, é uma métrica utilizada para mensurar o tempo despendido pelo servidor na execução de uma requisição. No modelo de adaptação os valores cumulativos são atribuídos com base na última ativação, e podem receber valores: 0 se a requisição perdeu *deadline*, 1 se ativação foi precisa ou C (valor entre 0 e 1) se esta execução foi imprecisa (execução com qualidade reduzida). Esses valores são atribuídos tomando como referência o valor de *deadline*. Portanto, se uma requisição é atendida dentro do *deadline*

estipulado, esta é executada de forma precisa e o valor cumulativo de sua classe recebe v=1.

Caso uma requisição seja atendida após o valor de *deadline* estipulado, uma perda de *deadline* é contabilizada. Na próxima execução, uma classe (mais longe do estado de falha) irá executar de forma imprecisa e, portanto, receberá em seu valor cumulativo um valor C.

Uma função benefício [35, 43] é utilizada no modelo de adaptação para modelar o valor cumulativo a ser atribuído para as classes de serviço em função dos tempos de execução das tarefas. A Figura 4.2 apresenta a função benefício utilizada no modelo de adaptação. São utilizados dois deadlines: d1 – *deadline soft* e d2 – *deadline firm*. As execuções das requisições podem receber três valores cumulativos:

- $V = C$ se a requisição é executada dentro do *deadline* d1, sendo C = 1 se execução precisa ou C = 0,3 se execução imprecisa.
- $V = C \cdot \left(\frac{d2 - t}{d2 - d1} \right)$ se a requisição é executada entre os *deadlines* d1 e d2. À medida que o tempo de execução de uma tarefa se aproxima de d2 esta tem seu valor cumulativo reduzido até o momento crítico onde V = 0. Depois de realizado o cálculo de V, o valor obtido é multiplicado por C, e novamente C = 1 para execuções precisas ou C = 0,3 para execuções imprecisas.
- V = 0 se a requisição é executada após o *deadline* d2, pois ultrapassou o valor estabelecido pelo *deadline firm* e, portanto o benefício da tarefa é zero.

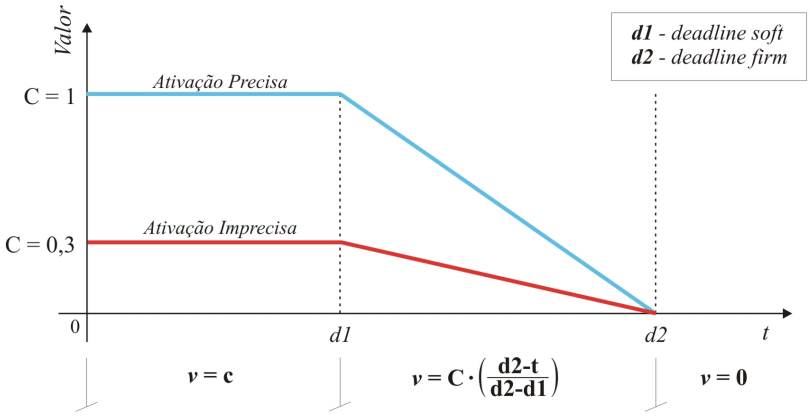


Figura 4.2: *Atribuição de valores cumulativos.*

4.8 Comparação com Trabalhos Relacionados

A Tabela 4.2 apresenta sucintamente as principais características relacionadas aos trabalhos estudados e o desenvolvido nessa dissertação (apresentada na última coluna da tabela). O modelo desenvolvido apresenta mecanismo de controle de admissão, implementa diferentes níveis de serviço (classes de serviço) e implementa a adptação de conteúdo através da técnica de computação imprecisa. Diferentemente dos outros trabalhos investigados o modelo proposto não posterga execução de requisições (utiliza diferenciação proporcional de serviços) e não utiliza grupos de servidores.

Tabela 4.2: *Análise comparativa entre os trabalhos.*

	[6]	**[51]**	**[4]**	**[27]**	**[16]**	**[1]**	**[48]**	**Modelo**
Controle de admissão	sim	não	não	não	sim	não	não	sim
QoS no Sistema Operacional	não	não	sim	sim	não	não	não	não
Usa Grupo de servidores	não	sim	não	não	não	não	não	não
Posterga execução de requisições	não	não	sim	não	não	não	sim	não
Diferentes níveis de serviço	não	não	não	sim	sim	sim	sim	sim
Adaptação de conteúdo	não	não	não	não	não	sim	não	sim

4.9 Conclusões do Capítulo

Este capítulo apresentou o modelo conceitual desenvolvido e os componentes que formam a arquitetura de provimento de QoS proposta. Foram apresentados os mecanismos de atribuição de *deadlines* as requisições, o mecanismo de classificação de requisições e atribuição de prioridades. Um controle de admissão também foi adicionado ao modelo com a finalidade de dirigir o servidor em situações onde a demanda de requisições supera a capacidade do servidor. O mecanismo de controle de admissão adotado nesse trabalho foi o controle de admissão simples, estático, diversos experimentos foram realizados, mostrando que essa abordagem é suficiente para lidar com sobrecargas severas.

A heurística de escalonamento PCV *(Proportional Cumulative Value)* busca através das políticas de atribuição de prioridades e escolha de versão buscam manter a diferenciação de serviços entre as classes, objetivando manter os valores cumulativos das classes o mais próximo possível dos valores de qualidade estipulados (VC = VQ).

Capítulo 5

Implementação e Resultados Experimentais

5.1 Introdução

Este capítulo descreve os detalhes de implementação do modelo proposto no Capítulo 4. Além disso, são apresentados experimentos realizados com a finalidade de validar o funcionamento e avaliar o desempenho da arquitetura proposta.

Esta seção apresenta os resultados obtidos através da implementação do modelo no servidor *web* Apache. Em essência, a avaliação do modelo é feita através de medições. Foram realizados experimentos sobre o modelo primeiramente sem o controle de admissão, nos quais foram observados resultados relacionados aos valores cumulativos e tempos de resposta fim a fim. Em um segundo momento, realizou-se experimentos com o controle de admissão ativado, nos quais foram observados alguns resultados: valores cumulativos, tempos de resposta, descartes, etc. Finalmente é apresentado um terceiro cenário de experimentos, objetivando avaliar a capacidade de reação do sistema perante a adição de novos pedidos de conexão ao servidor.

A preocupação principal das análises realizadas foi a de observar o comportamento do modelo em um ambiente real, utilizando geradores de carga que produzem requisições de acesso às páginas *web* estipuladas pelos clientes no servidor.

5.2 Implementação

O MPM *(Multi-Processing Module)* utilizado na implementação desse trabalho foi o MPM *Worker* (Figura 5.1). Este MPM implementa um servidor multi-processo *multi-thread*. Alterações efetuadas neste módulo apache, juntamente com a implementação de um módulo DSO *(Dynamic Shared Object)*, possibilitaram a implementação do modelo proposto. Neste trabalho, adotou-se a última versão estável do Apache, versão 2.2.2, lançada em dezembro de 2005. Essa nova versão oferece algumas funcionalidades interessantes como a possibilidade de se utilizar a API *(Application Programming Interface)* padrão oferecida pelo Apache.

A classificação dos pedidos de conexão foi implementada baseada no endereço IP dos clientes e identificada pelo servidor através da extração dessa informação contida na mensagem HTTP enviada pelo cliente. Uma *thread* ouvinte *(Listener)* aguarda novos pedidos de conexão vindos do serviço de comunicação TCP/IP (A na Figura 5.1) e os insere na fila de requisições (B na Figura 5.1), marca cada requisição com sua respectiva classe de serviço (ouro, prata ou bronze) e atribui à requisição um valor de *deadline*. O controle de admissão também é implementado nessa fase com base no tamanho da fila de requisições. Alguns pedidos de conexão são rejeitados, caso o servidor se encontre sobrecarregado.

As *threads* de atendimento de requisições (C na Figura 5.1) retiram as requisições da fila baseadas nas classes das requisições e nas prioridades de cada classe. Estas requisições então são processadas pelos módulos manipuladores *(handlers)*.

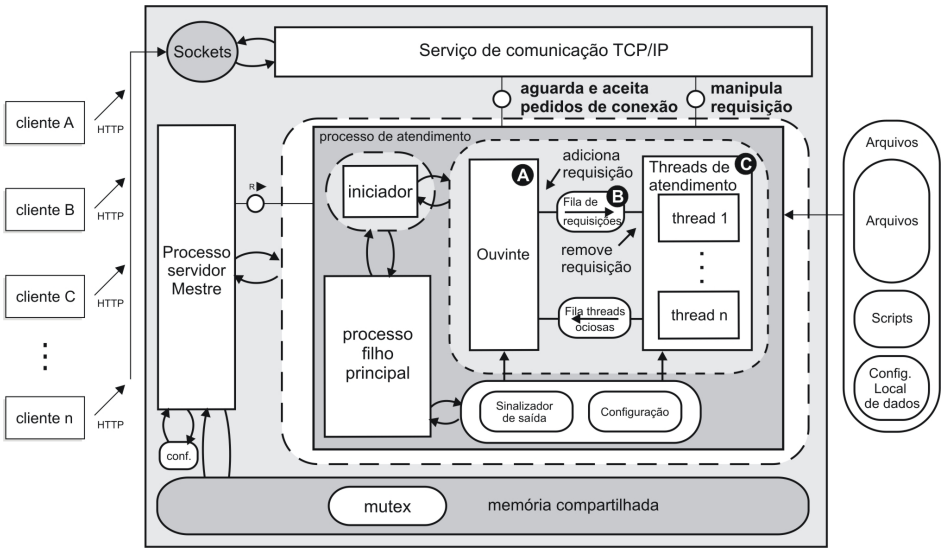


Figura 5.1: *Configuração do MPM (baseado em [30]).*

A política de seleção de versões é executada na fase de processamento das requisições por um módulo dinâmico DSO – **meumodulo** – carregado em tempo de execução através da diretiva *LoadModule* do arquivo de configuração *.httpconf* do Apache. O teste se houve perda de *deadline* no atendimento da requisição é efetuado na fase de *log* (eixo de processamento).

As trocas de informações entre os módulos MPM e o módulo *handler* são feitas utilizando áreas de memória disponibilizadas pelo Apache, conhecidas como *pools* e através do uso de memória compartilhada. Para a compilação e execução do módulo DSO desenvolvido utilizou-se a ferramenta de construção de módulos dinâmicos oferecida pelo Apache 2.2.2, APXS *(Apache eXtenSion tool)*. A Figura 5.2 mostra como executar a ferramenta APXS para gerar o módulo dinâmico *meumodulo.so* a partir de um código desenvolvido em linguagem C/C++. Na seqüência, na mesma Figura, é apresentado como se carrega o módulo utilizando a primitiva *LoadModule* no arquivo de configuração *.httpconf* do Apache.

```
apxs -c -o meumodulo.so meumodulo.c
mv .libs/meumodulo.so meumodulo.so
```

```
LoadModule meumodulo modules/meumodulo.so
```

Figura 5.2: *APXS e LoadModule no Apache.*

5.3 Monitoramento da Carga no Apache

Existem várias abordagens para detectar sobrecargas em servidores *web* [1]. Buscando avaliar algumas das abordagens, vários experimentos foram efetuados mensurando: (i) ocupação da fila de sockets do SO; (ii) ocupação da fila (B da Figura 5.1); (iii) tempo de resposta das requisições.

A abordagem (i) foi descartada neste trabalho, pois o monitoramento da fila de *sockets* não pode ser efetuada em tempo de execução alterando-se apenas o código-fonte do Apache, pois implica instrumentalizar o sistema operacional. As outras duas abordagens foram implementadas e a Figura 5.3 apresenta alguns resultados obtidos. A carga crescente submetida ao sistema foi gerada pelo gerador de carga sintética *httperf* usando três computadores clientes. O eixo x representa a passagem do tempo em segundos, o eixo y a esquerda se refere às médias dos tempos de resposta das requisições em milisegundos e do lado direito à ocupação da fila (quantidade de requisições aguardando para serem processadas).

Ao contrapor em um mesmo gráfico as duas medidas, mesmo que de grandezas diferentes, é possível observar que as curvas correspondentes crescem de maneira proporcional[1] (possuindo valores de tangente próximos). Desta forma, considerou-se a possibilidade de adotar qualquer uma dessas abordagens para mensurar o crescimento de carga submetido ao servidor. Como alguns trabalhos sugerem o uso dos tempos de resposta [1], este trabalho prosseguiu na análise deste parâmetro.

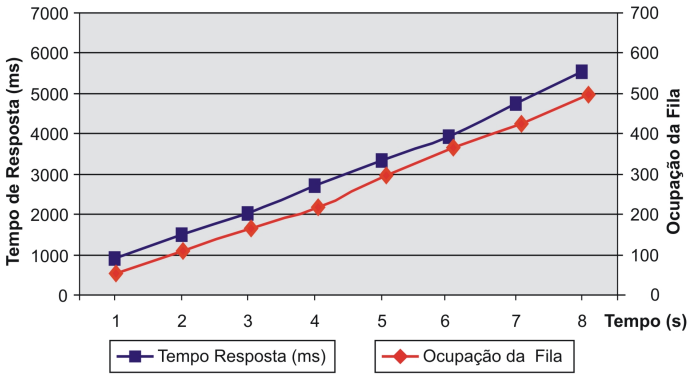


Figura 5.3: *Tempos de resposta instantâneo vs ocupação da fila em servidores web.*

A Figura 5.4 apresenta três curvas referentes a diferentes formas de avaliar os tempos de resposta: instantâneo, média aritmética em uma janela de tempo (no caso 1 segundo) e média exponencial ponderada. Como os valores instantâneos de tempo de resposta (última amostra obtida) tendem a variar bruscamente, dependendo da carga, uma média obtida em uma janela de tempo também apresenta variações, ainda que menores. A adoção de uma média móvel ponderada, como a adotada para estimação de valores de *round trip time* (tempos de ida e volta de uma mensagem) em conexões TCP proposta por Jacobson [33] tende a suavizar essa variação. No caso foi adotada a função:

$$media = \alpha \cdot media + (1 - \alpha) \cdot amostra \qquad tal\ que\ \alpha = 0,9$$

Essa função pode ser considerada como um filtro passa baixa onde o valor de α especifica a constante de tempo do filtro.

[1]Isto ocorre porque o *pool* de *threads* utilizado no servidor é de tamanho reduzido.

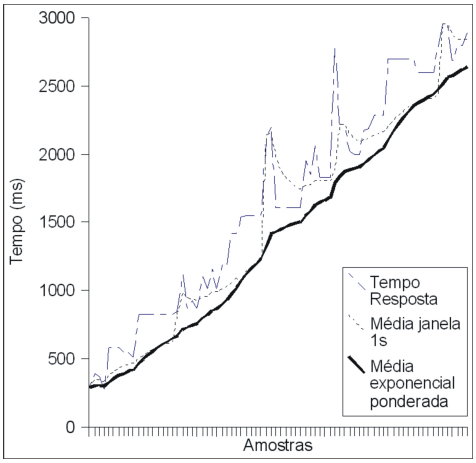


Figura 5.4: *Tempos de resposta vs média móvel ponderada.*

5.4 Cenário de Realização dos Experimentos

Experimentos foram realizados através de medições utilizando-se um protótipo do modelo descrito no Capítulo 4 construído sobre um servidor *web* Apache versão 2.2.2. A avaliação através de medições traz vantagens, pois considera diversos *overheads* (latência de rede e *buffers*, por exemplo) que acabam muitas vezes sendo ignorados em ambientes simulados.

Nos experimentos foram utilizadas quatro máquinas, sendo três clientes (A, B e C) e um servidor (Apache) interligadas por uma rede Ethernet de 10 Mbits/s. A disposição das máquinas pode ser verificada na Figura 5.5.

Na formulação do ambiente de testes foram utilizados quatro máquinas Pentium IV 3.0 GHz, 1024 MB de memória RAM e sistema operacional Ubuntu Gnu/Linux versão 6.10 *kernel* 2.6.17.

Alguns trabalhos utilizam *traces* de acessos armazenados pelo servidor, como os da copa do mundo de 1998 [9] para gerar carga de trabalho, porém essa medição não é adequada, pois os *logs* armazenados pelos servidores *web* apresentam tempos com granularidade muito grossa (intervalos de 1 segundo). Outra abordagem é o uso de distribuições matemáticas para modelar a chegada de requisições no servidor (distribuição Pareto). Contudo no sentido de tentar usar uma carga de trabalho mais próxima de um ambiente real utilizou-se na geração de carga dos clientes o gerador de carga sintética *httperf* [47] com número de conexões 1200, taxa de chegada 60 requisições por segundo e *timeout* 60 segundos. Uma característica importante do *httperf* é o fato dele não realizar *cache*.

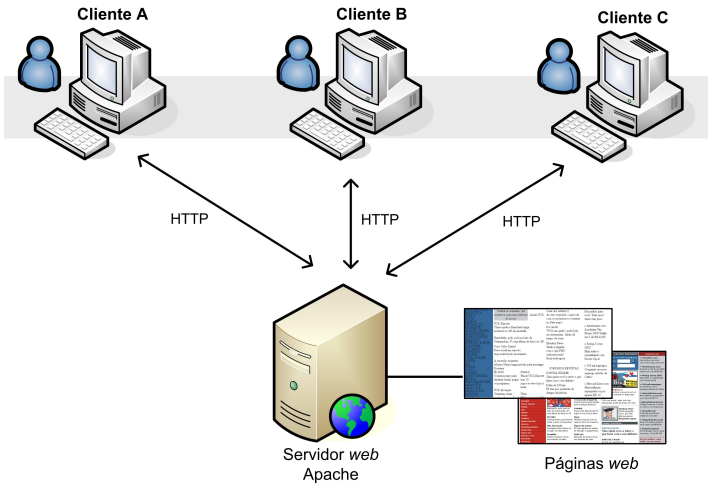


Figura 5.5: *Topologia utilizada nos experimentos.*

Objetivando facilitar a sobrecarga do servidor, algumas configurações do Apache não foram otimizadas (módulos como o *mod_cache*, foram desativados). As informações contidas no arquivo de configuração *httpd.conf* podem ser verificadas na Figura 5.6:

```
<IfModule worker.c>
    ServerLimit           1
    StartServers          1
    MaxClients            50
    MinSpareThreads       25
    MaxSpareThreads       50
    ThreadsPerChild       50
    MaxRequests PerChild  0
</IfModule>
```

Figura 5.6: *Configuração do MPM Worker.*

- *ServerLimit* : Número máximo de processos MPM *(worker)* disparados pelo servidor.
- *StartServers* : Número de processos filhos criados na inicialização do Apache.
- *MaxClientes* : Número máximo de requisições simultâneas que são suportadas pelo servidor.
- *MinSpareServers* e *MaxSpareServers* : Valores mínimos e máximos de processos filhos que se mantêm aguardando a chegada de novas requisições.
- *ThreadsPerChild* : Número máximo de *threads* que um processo pode disparar.

- *MaxRequestPerChild* : Limite do número de requisições que um processo filho irá manipular antes de morrer. O valor zero indica que o processo jamais irá expirar.

5.5 Avaliação do Modelo e da Abordagem

Nos experimentos, foram considerados para o cálculo do valor cumulativo os valores de *deadline* d1=5 e d2=20 segundos. Foram definidas três classes de serviço, Ouro, Prata e Bronze com valores de qualidade (VQ): 0,6; 0,3; 0,1, respectivamente.

A avaliação da abordagem foi dividida em três cenários diferentes: com o controle de admissão desativado (Primeiro Cenário), com o mecanismo de controle de admissão ativado (Segundo Cenário) e com adição de novos clientes nos experimentos (Terceiro Cenário).

5.5.1 Primeiro Cenário

As primeiras medições realizadas objetivaram verificar o comportamento do valor cumulativo (VC), e se estas se mantinham próximo aos valores de qualidade (VQ) especificados. Ou seja, se as propriedades de controlabilidade e previsibilidade eram respeitadas. Para esses primeiros experimentos, o controle de admissão não foi ativado. A Figura 5.7 apresenta alguns dos resultados obtidos (o eixo x representa a média de amostras consecutivas obtidas em intervalos de 1 segundo). Na Tabela 5.1 pode-se observar os valores obtidos após análise de intervalo de confiança com uso da distribuição t de *Student*[2] com nível de confiança de 95% [13].

Tabela 5.1: *Análise de confiabilidade dos dados da Figura 5.7.*

Classes	VC	Int. Confiança	Desv. Média
Ouro	0,58	0,015	2,47%
Prata	0,27	0,009	3,12%
Bronze	0,13	0,012	8,69%

[2]Para amostras pequenas, a validade do intervalo de confiança está condicionada à suposição de que os dados provenham de uma distribuição aproximadamente normal. Para $n > 30$ (onde n é o número de amostras), o teorema limite central garante a validade de intervalo de confiança.

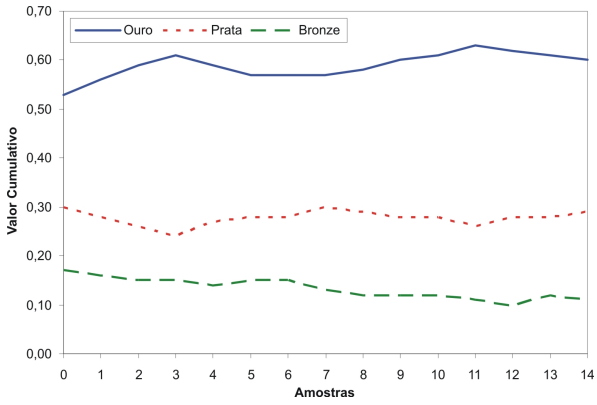


Figura 5.7: *Valores cumulativos do PCV sem controle de admissão.*

Apesar de, como já foi ressaltado, devido a descartes e execuções imprecisas, os valores de tempo de resposta não poderem ser considerados como a única métrica relevante, nesses mesmos experimentos, os valores foram obtidos e estão apresentados na Figura 5.8 (o eixo representa amostras consecutivas obtidas em intervalos de 1 segundo). A análise de confiabilidade realizada (nível de confiança de 95%) sobre os dados da Figura 5.8, pode ser observada na Tabela 5.2.

Tabela 5.2: *Análise de confiabilidade dos dados da Figura 5.8.*

Classes	Média (ms)	Int. Confiança	Desv. Média
Ouro	14391	500	3,47%
Prata	18569	424	2,29%
Bronze	30749	727	2,36%

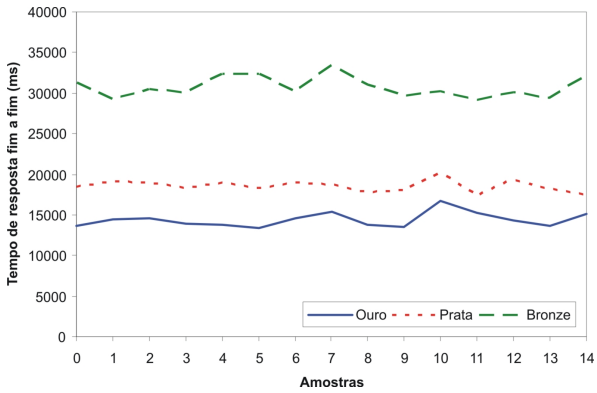


Figura 5.8: *Tempos de resposta fim a fim sem controle de admissão com mesma carga.*

Buscando avaliar a consistência do modelo proposto neste trabalho, os valores obtidos no experimento mostrado na Figura 5.8 foram adquiridos dentro de uma mesma carga no servidor, ou seja, foram executados testes sucessivos no servidor (sempre com a mesma carga) e os resultados dos tempos de resposta coletados nos clientes apresentados. Porém buscando avaliar o comportamento da heurística, em situações onde a carga gerada no servidor cresce continuamente, um novo experimento foi realizado, desta vez variando a carga no servidor e coletando novamente os valores dos tempos de resposta fim a fim. Os resultados são apresentados na Figura 5.9.

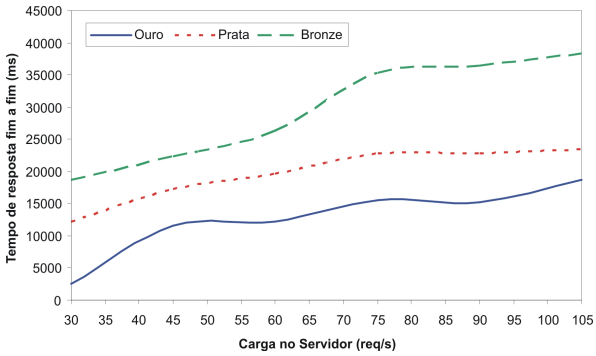


Figura 5.9: *Tempos de resposta fim a fim sem controle de admissão variando carga.*

É possível observar na Figura 5.9 que os tempos de resposta foram condizentes com os valores de qualidade almejados, ou seja, as classes de serviço privilegiadas foram atendidas com os menores tempos do sistema, mesmo com variações de carga no servidor. Porém apesar das políticas de controle de versão (precisa/imprecisa) do PCV atuar no sentido de reduzir a carga do servidor em situações de sobrecarga e, apesar do PCV conseguir manter a diferenciação de serviço, os tempos de resposta continuam a crescer rapidamente. Portanto, foi implementado um controle de admissão. O controle de admissão, nesse caso, é importante em casos de sobrecarga severa, onde as execuções imprecisas não conseguem reduzir suficientemente a carga.

5.5.2 Segundo Cenário

Outros experimentos foram efetuados, doravante com o mecanismo de controle de admissão ativado. Observou-se que os valores cumulativos obtidos no experimento utilizando o controle de admissão (Figura 5.10), convergiram rapidamente para o valor de qualidade (VQ) almejado para cada classe. O controle de admissão implementado aqui é estático e

descarta as requisições recém-chegadas ao servidor, em conformidade com o tamanho da fila de requisições do servidor (nesses experimentos utilizou-se uma fila com tamanho máximo de 600 requisições). Novamente na Tabela 5.3 pode ser observado os valores obtidos após análise de intervalo de confiança, (distribuição t de *Student*) com nível de confiança de 95%.

Tabela 5.3: *Análise de confiabilidade dos dados da Figura 5.10.*

Classes	Média	Int. Confiança	Desv. Média
Ouro	0,59	0,00412	0,69%
Prata	0,30	0,00281	0,92%
Bronze	0,10	0,00229	2,25%

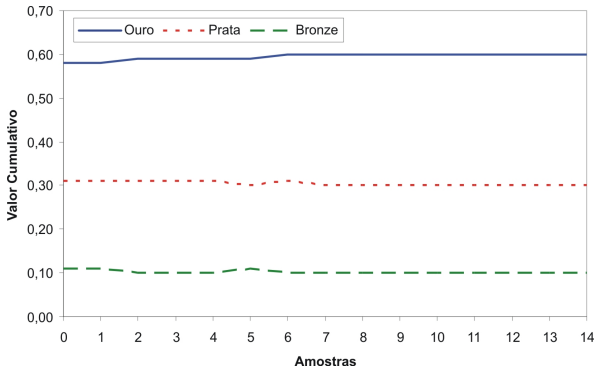


Figura 5.10: *Valores Cumulativos do PCV com controle de admissão.*

Novamente realizaram-se experimentos coletando amostras consecutivas com intervalo de um segundo, todas obtidas a partir da mesma carga no servidor. Os valores dos tempos de resposta obtidos são mostrados na Figura 5.11 e os descartes provenientes deste experimento são apresentados na Figura 5.12. A Tabela 5.4 apresenta a análise dos dados amostrados (intervalo de confiança, com 95% de confiança) e apresentados graficamente na Figura 5.11.

Tabela 5.4: *Análise de confiabilidade dos dados da Figura 5.11.*

Classes	Média (ms)	Int. Confiança	Desv. Média
Ouro	8509	342	4,02%
Prata	16109	386	2,40%
Bronze	21718	255	1,18%

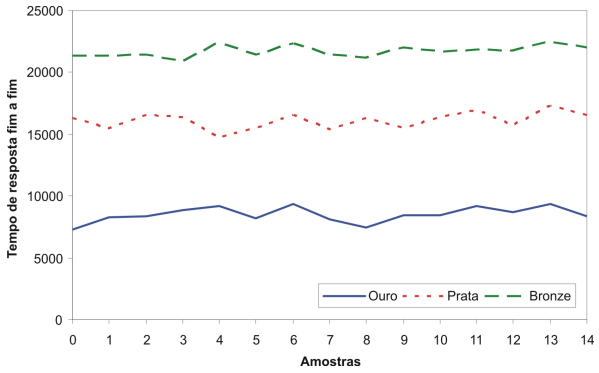


Figura 5.11: *Tempos de resposta fim a fim com controle de admissão.*

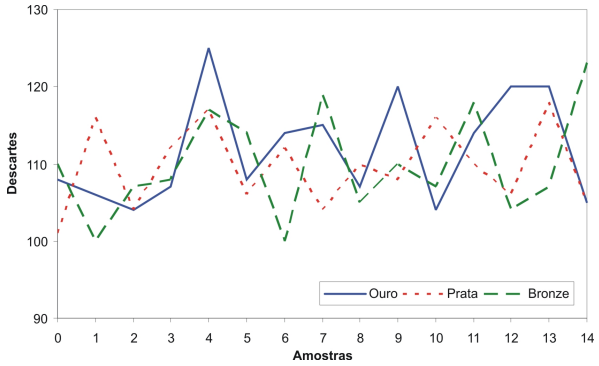


Figura 5.12: *Descartes provenientes do experimento da Figura 5.11.*

Interessante observar que os tempos de resposta para mesma carga apresentada na Figura 5.13 (com controle de admissão) comparada com os da Figura 5.9 (sem controle de admissão), são muito menores.

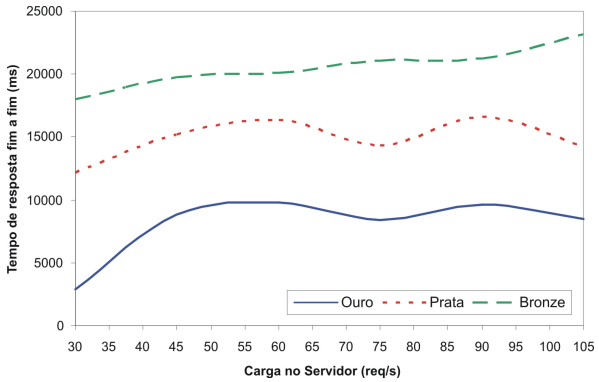


Figura 5.13: *Tempos de resposta fim a fim com controle de admissão.*

A Tabela 5.5 apresenta o total das execuções precisas e imprecisas por classe de serviço. Pode-se observar que a classe Ouro executou menos versões imprecisas, enquanto a classe Bronze foi mais penalizada.

Tabela 5.5: *Execuções precisas e imprecisas*

Classes	Exec. Precisas	Exec. Imprecisas
Ouro	299	101
Prata	229	171
Bronze	51	349

Um outro dado importante levantado foram os descartes ocorridos em cada classe durante esses experimentos com variação de carga no servidor e controle de admissão. Os descartes para cada classe de serviço podem ser observados na Figura 5.14.

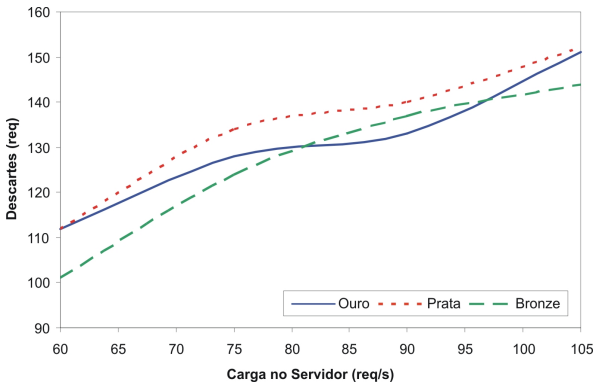


Figura 5.14: *Descartes provenientes do controle de admissão.*

O mecanismo de controle de admissão é acionado com a indicação de uma sobrecarga no servidor, nota-se, na Figura 5.14 que a partir de uma carga de 60 req/s o servidor passou a descartar requisições. É possível observar também que a quantidade de descartes no servidor aumenta à medida que a carga do servidor aumenta. O controle de admissão atua desconsiderando as classes de serviço. Isso pode ser verificado nas figuras **??**, onde os descartes ocorrem de forma aleatória, conforme as chegadas das requisições.

Vários valores de tamanhos de fila foram verificados para a escolha do valor indicado nesse trabalho. Esses valores podem ser verificados na Figura 5.15. Os tamanhos de fila variaram de 300, onde muitas requisições foram descartadas, porém obtendo tempos de resposta baixos, até 800, onde poucas requisições foram descartadas, porém obtiveram os tempos de

resposta elevados. É necessário sintonizar um valor ideal onde possa ser atendido o máximo de requisições possível dentro de tempos de resposta aceitáveis.

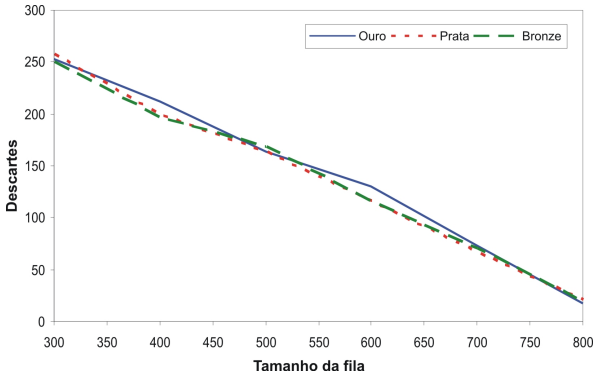


Figura 5.15: *Descartes vs tamanho da fila de requisições no servidor.*

A Figura 5.16 apresenta a relação entre os tempos de resposta e o tamanho da fila de requisições utilizadas no controle de admissão. Pode-se observar que os tempos de resposta tendem a aumentar à medida que os tamanhos das filas de descarte crescem. Consequentemente, quanto maior o tamanho da fila, maior o número de requisições aguardando por atendimento.

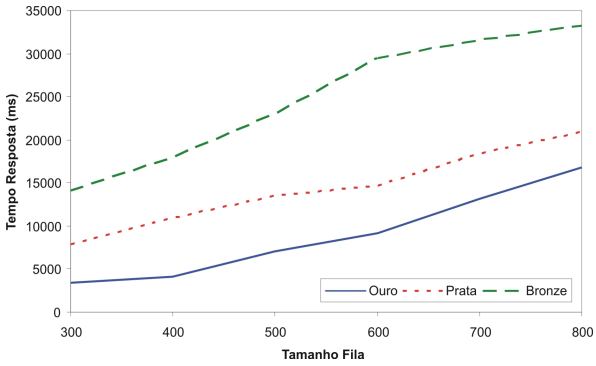


Figura 5.16: *Tempos de resposta vs tamanho da fila de requisições.*

5.5.3 Terceiro Cenário

Com o intuito de verificar a capacidade de reação do sistema, novos experimentos foram realizados, desta vez com a adição de um quarto cliente (cliente D).

A Figura 5.17 apresenta o resultado do experimento com o controle de admissão desativado (Uma parte do log utilizado neste experimento pode ser verificado no Apêndice B).

Pode-se observar que os valores cumulativos das classes se mantiveram estáveis até o momento da chegada das novas requisições (200 requisições) enviadas pelo cliente D (instante 25s). A chegada dessas novas requisições causou perturbação no sistema, porém pode-se observar no gráfico que o sistema reagiu a essa nova perturbação, convergindo a partir do instante 40s para os valores cumulativos almejados.

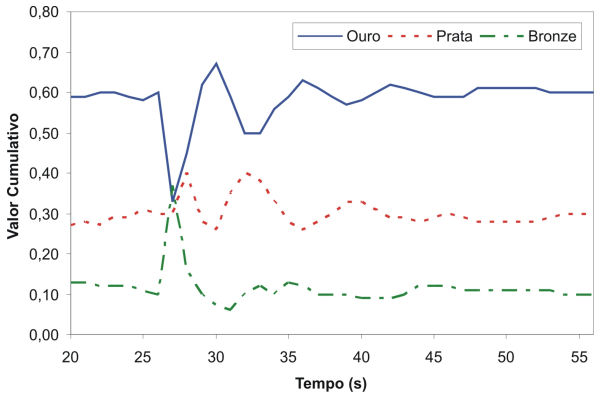


Figura 5.17: *Valores Cumulativos com perturbação e sem controle de admissão.*

O experimento realizado com o controle de admissão ativado (Figura 5.18) mostrou que a partir do instante 16s quando as requisições do cliente D passaram a chegar ao servidor, os valores cumulativos se mantiveram estáveis. É possível observar também o aumento no número de requisições admitidas pelo servidor (eixo y, lado direito), similarmente se observa o aumento do número de descartes das requisições, provenientes do controle de admissão.

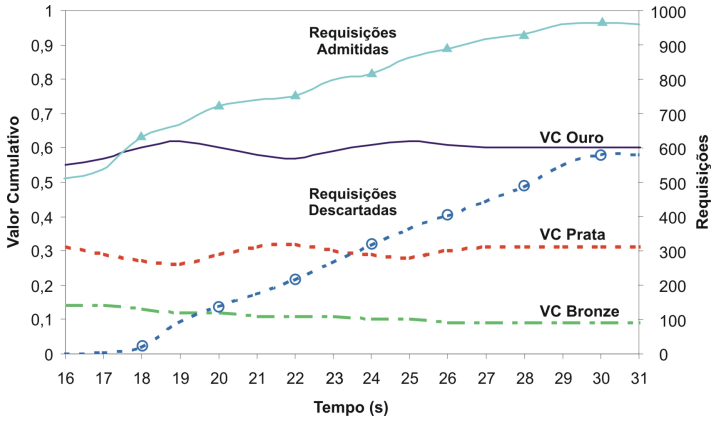


Figura 5.18: *Valores Cumulativos, requisições aceitas e descartes.*

5.6 Conclusões do Capítulo

Este capítulo apresentou os detalhes da implementação do modelo de QoS proposto no servidor *web* Apache, com base no modelo apresentado no Capítulo 4. Especificou também formas de implementar o modelo alterando o módulo *Worker* MPM do Apache 2 e criando um módulo DSO para exercer a função de gerador de conteúdo.

O MPM *Worker* responsável por gerenciar as portas de comunicação, aceitar conexões e alocar processos para atendimento das requisições, foi alterado e adicionado funcionalidades de classificação de requisições. Foram criadas 3 filas de requisições com níveis de prioridades diferentes. O mecanismo de controle de admissão também foi implementado nessa fase, onde em condições de sobrecarga as requisições recém-chegadas são descartadas.

O módulo responsável por processar efetivamente as requisições (gerador de conteúdo), aqui chamado de **meumodulo** foi desenvolvido utilizando as novas funcionalidades oferecidas pelo Apache e principalmente fazendo uso das API´s oferecidas nessas novas versões. Uma importante característica no desenvolvimento desses módulos dinâmicos é sua fácil inserção em novos servidores *web*, necessitando apenas que sejam alterados alguns parâmetros de configuração no arquivo de configuração do Apache.

O presente capítulo desenvolveu e discutiu também vários experimentos realizados sobre o modelo de qualidade de serviço para servidores *web* proposto nesse trabalho. Apontou a inviabilidade no uso de técnicas de qualidade de serviço com prioridades estáticas. Posteriormente, foram avaliados os mecanismos de diferenciação de serviços proporcional aqui propostos mostrando a eficácia do modelo e a necessidade de utilização de um mecanismo de controle de admissão, também avaliado.

Ao mesmo tempo, esse capítulo apresentou uma importante contribuição em direção a uma melhor compreensão do funcionamento de técnicas de implementação de mecanismos de qualidade de serviço em servidores *web*, sobretudo de sistemas que empregam mecanismos de classificação de requisições em diferentes classes de serviços, descarte de requisições, políticas de seleção de versões de páginas *web* e políticas de escalonamento diferenciadas em servidores *web*.

Resultados parciais deste trabalho foram publicados em [56]. Outros resultados investigando a adição de mecanismos de controle de admissão dinâmicos, com realimentação utilizando técnicas de teoria de controle estão sendo investigados.

Capítulo 6

Conclusões

O presente capítulo conclui esta dissertação começando com a revisão dos objetivos, que foram apresentados no capítulo introdutório. Apresenta-se também uma visão geral do trabalho seguida pelas contribuições oferecidas pelo mesmo e encerrando com a exposição de alguns trabalhos futuros.

6.1 Revisão das Motivações e Objetivos

Com o aumento do número de serviços oferecidos através da Internet tornou-se necessário o desenvolvimento de suportes e técnicas adequadas que dêem suportes a esses novos serviços. Porém, clientes desses novos serviços têm de conviver com clientes de aplicações tradicionais da Internet, tais como o correio eletrônico, por exemplo, cujas necessidades por serviços são bem menos exigentes.

O paradigma de melhor esforço adotado pela Internet atual faz necessária a adoção de soluções imediatas que forneçam qualidade de serviço diferenciada entre os clientes dessas aplicações. A IETF propôs dois padrões abertos para provimento de QoS conhecidos como IntServ e DiffServ. Entretanto, pouco adianta o emprego de soluções na infraestrutura da rede, caso as aplicações nos pontos finais também não estejam preparadas para essa nova realidade. Um exemplo é o fato que as aplicações servidoras existentes (ex. servidores *web*) adotarem a política de atendimento de requisições segundo uma política FIFO *(First-in, First-out).*

Recentemente, vários trabalhos que buscam oferecer qualidade de serviço em servidores *web* têm surgido. Esses trabalhos atuam de forma complementar a outros que atuam na infraestrutura da rede. A impossibilidade de delimitar a carga desses servidores leva a situações de sobrecargas transientes, nas quais os tempos de resposta das requisições ultrapassam valores aceitáveis para usuários humanos. Nesses casos, quando os agentes de usuário (ex. navegadores *web*) implementam valores de *timeout*, todos os recursos do servidor (espaço em fila, processador, memória, acesso a disco, etc.) envolvidos na requisição são desperdiçados.

Grande parte das pesquisas encontradas na literatura adota o uso de mecanismos de controle de admissão. O objetivo do controle de admissão é reduzir a carga do servidor em situações de sobrecarga, evitando assim o desperdício sucessivo de recursos ocasionado pelo cancelamento seqüencial de requisições.

Outras abordagens propõem diferenciação de serviços nos atendimentos às requisições. Na impossibilidade de se oferecer um serviço diferenciado individual, visando a escalabilidade, essa abordagem sacrifica a qualidade de serviço individual de cada requisição, lidando com um número conhecido de classes de serviço (ex. ouro, prata, bronze). Para cada requisição submetida ao servidor esta é classificada e agrupada em sua classe de serviço. O mecanismo utilizado para diferenciar os serviços pode ser realizado através da atribuição de prioridades, do controle do tamanho da fila de requisições, do número de threads ou de processos que executam as requisições, etc.

Além do mecanismo de controle de admissão, outras técnicas como a do escalonamento adaptativo por computação imprecisa podem ser adotadas com a finalidade de controlar a carga dos servidores *web*. Nesse trabalho esse mecanismo foi implementado através de duas versões de páginas *web*: uma versão precisa (completa) e outra apenas com texto (imprecisa). Em uma situação de sobrecarga temporária, o servidor, implementado segundo esse modelo, responde a requisições de forma parcial (página imprecisa). Mesmo com conteúdo reduzido, o cliente recebe uma resposta à sua requisição. Dessa forma, os resultados gerados pelos servidores implementados de acordo com o modelo proposto dependem diretamente da carga que é submetida ao servidor *web*.

6.2 Visão Geral do Trabalho

Esta dissertação iniciou com uma descrição de um estudo sobre qualidade de serviço e apresentação dos principais trabalhos relacionados encontrados na literatura. Em seguida

nossos esforços se concentraram no desenvolvimento de um modelo que ofereça qualidade de serviço no contexto de servidores *web*.

Este trabalho descreveu um modelo de adaptação para servidores *web* baseado em computação imprecisa e escalonamento por prioridades dinâmicas, objetivando conseguir uma diferenciação proporcional de serviços.

O uso da técnica de computação através de versões imprecisas utilizada nesse trabalho permite responder às requisições dos clientes, mesmo com qualidade reduzida, e evitar que esses usuários esperem por longos períodos de tempo ou até mesmo recebam negação de serviço. Em situações onde o servidor se encontra demasiadamente sobrecarregado, as execuções imprecisas não conseguem mais reduzir a carga do sistema, sendo melhor descartar as novas ativações que chegam ao servidor e atender as que já estão sendo processadas. Por esse motivo, um controle de admissão foi adicionado ao modelo.

O modelo foi avaliado através de medições realizadas a partir do protótipo desenvolvido em um servidor *web* Apache versão 2.2. Os experimentos efetuados no servidor *web* mostraram alguns dados interessantes, difíceis de capturar em um ambiente simulado. Houve, por exemplo, a possibilidade de medir os tempos de resposta fim a fim e avaliar alguns *overheads* que ocorrem no servidor Apache.

Acredita-se que o *overhead* adicional, proveniente da abordagem proposta nesse trabalho, seja compensado pelos benefícios oferecidos. Considerando que servidores *web*, frequentemente, apresentam diversos outros *overheads* no momento da chegada das requisições, como, por exemplo, módulos de controle de acesso e redirecionamento de determinados tipos de requisições, conforme apresentado no Capítulo 3.

6.3 Contribuições e Escopo do Trabalho

Dentro dos objetivos almejados para esse trabalho e das atividades desenvolvidas durante sua concepção, pode-se enumerar os seguintes pontos de destaque dessa dissertação:

- A principal contribuição deste trabalho foi a proposição de um modelo e o desenvolvimento de um protótipo para provimento de qualidade de serviço (QoS) nos elementos finais da rede (servidores *web*). Este modelo trabalha de forma complementar aos mecanismos de QoS na infraestrutura da rede já existentes.

- Diferentemente de outros trabalhos que utilizam tempo de resposta como métrica, neste trabalho adotou-se o valor cumulativo, pois este consegue mensurar a diferença de qualidade obtida por execuções precisas, imprecisas e descartes de requisições. Apesar disso, neste trabalho também se procurou medir individualmente os tempos de resposta das requisições.

- Uma das contribuições desse trabalho foi a avaliação do modelo proposto através de medições realizadas com a implementação detalhada do modelo no servidor *web* Apache. A implementação do protótipo em um ambiente real considera perturbações inexistentes em ambientes simulados, tornando assim os resultados obtidos próximos da realidade.

- Desenvolvimento de uma plataforma de experimentos de abordagens de qualidade de serviço (QoS) em servidores *web* utilizando Apache.

6.4 Perspectivas Futuras

Considerando o estágio atual deste trabalho, algumas possibilidades para sua continuidade são vislumbradas e propostas a seguir:

- Como trabalho futuro, propõe-se a inclusão do modelo proposto dentro do contexto de páginas de conteúdo dinâmico.

- Apesar dos resultados obtidos com o controle de admissão empregado nesse trabalho serem satisfatórios, existe o interesse de se implementar um controle de admissão dinâmico utilizando técnicas da teoria de controle interligado diretamente com o controle de versões (precisas/imprecisas) .

- Outra questão que está sendo investigada é a alteração do modelo proposto, considerando descarte seletivo entre as requisições que chegam ao servidor, através da adição de uma nova classe de serviços (ex. classe diamante) e mapeamento da tabela de precedência de descartes do PHB AF (Seção 2.4.4) para o PCV.

Espera-se que esta dissertação possa servir de base para propostas de modelos futuros, que busquem atender às novas perspectivas que estão para surgir na área de qualidade de serviço, principalmente em servidores *web*.

Apêndice A

Versões Precisas e Imprecisas de uma Página *Web*

Abaixo são apresentadas as versões precisas (Figura A.1) e imprecisas (Figura A.2) das páginas *web* utilizadas na adaptação de conteúdo deste trabalho.

A.1 Versão Precisa

Figura A.1: *Versão precisa.*

A.2 Versão Imprecisa

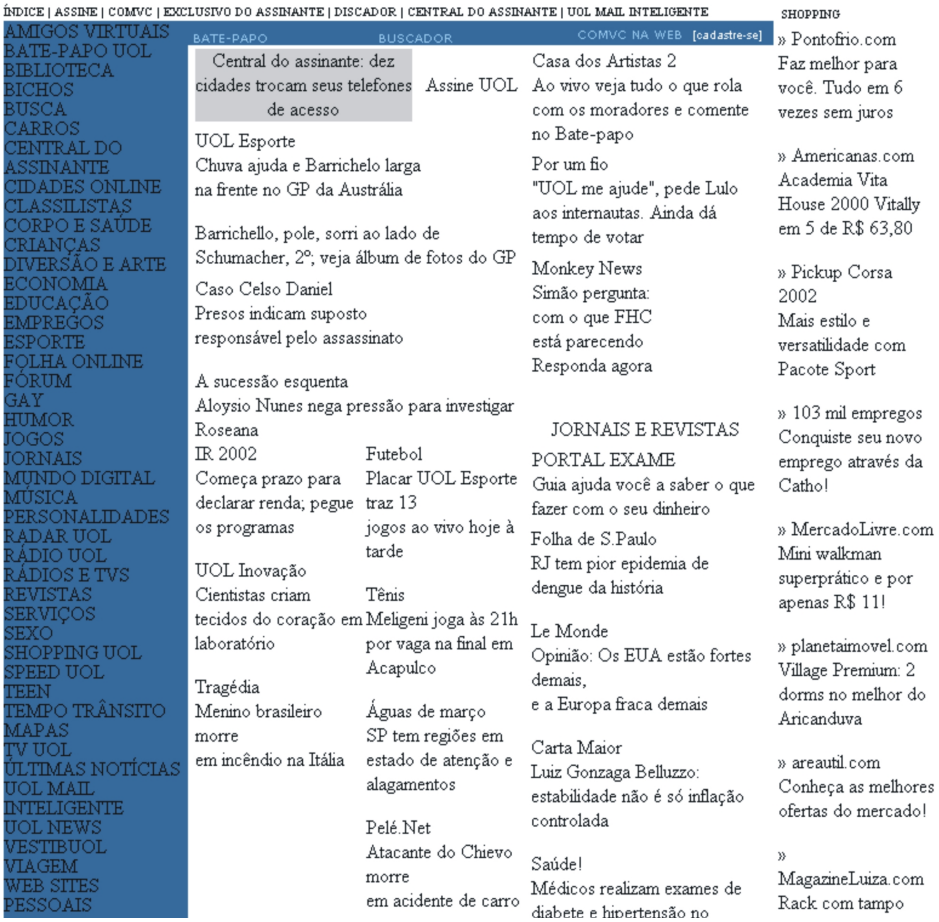


Figura A.2: *Versão imprecisa.*

Apêndice B

Log de Experimento sem Controle de Admissão

Abaixo são apresentados alguns dados coletados a partir do experimento realizado sem controle de admissão, com adição de um novo clinte (cliente D). Alguns dados interessantes podem ser verificados, como o tamanho da fila, quantidade de processos no sistema, valores cumulativos das classes, execuções precisas e imprecisas (até o momento) e o tempo de resposta das requisições (incluindo o tempo de espera na fila de requisições).

```
14:57:05  VAL CUMUL-> OURO: 0.59 PRATA: 0.30 BRONZE: 0.12
14:57:05  EXEC IMP-> OURO: 0 PRATA: 0 BRONZE: 0
14:57:05  EXEC PREC-> OURO: 173 PRATA: 140 BRONZE: 52
14:57:05  Tamanho da Fila 712
14:57:05  Classe Ouro: TEMPO RESPOSTA (TOTAL): 18540323
14:57:05  Classe Ouro: Processos no Sistema: 730
14:57:05  Tamanho da Fila 711
14:57:05  Classe Ouro: TEMPO RESPOSTA (TOTAL): 18540433
14:57:05  Tamanho da Fila 710
14:57:05  Classe Ouro: TEMPO RESPOSTA (TOTAL): 18540647
14:57:05  Tamanho da Fila 709
14:57:05  Classe Ouro: TEMPO RESPOSTA (TOTAL): 18552326
14:57:05  Classe Prata NOVA: Processos no Sistema: 729
14:57:05  Tamanho da Fila 708
14:57:05  Classe Ouro: TEMPO RESPOSTA (TOTAL): 18553881
14:57:05  VAL CUMUL-> OURO: 0.58 PRATA: 0.30 BRONZE: 0.12
14:57:05  EXEC IMP-> OURO: 0 PRATA: 0 BRONZE: 0
14:57:05  EXEC PREC-> OURO: 178 PRATA: 140 BRONZE: 52
14:57:05  Tamanho da Fila 707
14:57:05  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11836152
14:57:05  Classe Ouro: Processos no Sistema: 728
14:57:05  Tamanho da Fila 706
14:57:05  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11517760
```

```
14:57:05  Classe Ouro: Processos no Sistema: 727
14:57:05  Classe Ouro: Processos no Sistema: 726
14:57:06  Classe Ouro: Processos no Sistema: 725
14:57:06  Classe Ouro: Processos no Sistema: 724
14:57:06  Classe Ouro: Processos no Sistema: 723
14:57:06  Classe Ouro: Processos no Sistema: 722
14:57:06  Classe Ouro: Processos no Sistema: 721
14:57:06  Classe Ouro: Processos no Sistema: 720
14:57:06  Classe Ouro: Processos no Sistema: 719
14:57:06  Tamanho da Fila 705
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11805974
14:57:06  Classe Ouro: Processos no Sistema: 718
14:57:06  Classe Ouro: Processos no Sistema: 717
14:57:06  Classe Ouro: Processos no Sistema: 716
14:57:06  Classe Ouro: Processos no Sistema: 715
14:57:06  Tamanho da Fila 704
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 19042453
14:57:06  Tamanho da Fila 703
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11726227
14:57:06  VAL CUMUL-> OURO: 0.59 PRATA: 0.30 BRONZE: 0.12
14:57:06  EXEC IMP-> OURO: 0 PRATA: 0 BRONZE: 0
14:57:06  EXEC PREC-> OURO: 183 PRATA: 140 BRONZE: 52
14:57:06  Classe Ouro: Processos no Sistema: 714
14:57:06  Classe Ouro: Processos no Sistema: 713
14:57:06  Classe Ouro: Processos no Sistema: 712
14:57:06  Classe Ouro: Processos no Sistema: 711
14:57:06  Classe Ouro: Processos no Sistema: 710
14:57:06  Tamanho da Fila 702
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11758090
14:57:06  Classe Ouro: Processos no Sistema: 709
14:57:06  Tamanho da Fila 701
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11879600
14:57:06  Tamanho da Fila 700
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11771682
14:57:06  Tamanho da Fila 699
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11766887
14:57:06  Tamanho da Fila 698
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11761176
14:57:06  VAL CUMUL-> OURO: 0.59 PRATA: 0.29 BRONZE: 0.12
14:57:06  EXEC IMP-> OURO: 0 PRATA: 0 BRONZE: 0
14:57:06  EXEC PREC-> OURO: 188 PRATA: 140 BRONZE: 52
14:57:06  Tamanho da Fila 697
14:57:06  Classe Ouro: TEMPO RESPOSTA (TOTAL): 11764450
14:57:06  Tamanho da Fila 696
14:57:06  Classe Prata: TEMPO RESPOSTA (TOTAL): 11475690
14:57:06  Tamanho da Fila 695
14:57:06  Classe Prata: TEMPO RESPOSTA (TOTAL): 11470345
14:57:06  Tamanho da Fila 694
14:57:06  Classe Prata: TEMPO RESPOSTA (TOTAL): 11467178
14:57:06  Tamanho da Fila 693
14:57:06  Classe Prata: TEMPO RESPOSTA (TOTAL): 11466252
14:57:06  VAL CUMUL-> OURO: 0.60 PRATA: 0.30 BRONZE: 0.10
14:57:06  EXEC IMP-> OURO: 0 PRATA: 0 BRONZE: 0
14:57:06  EXEC PREC-> OURO: 189 PRATA: 144 BRONZE: 52
```

Referências Bibliográficas

[1] Abdelzaher, T. (1999). *QoS-Adaptation in Real-Time Systems*. Phd thesis, University of Michigan, Michigan.

[2] Abdelzaher, T. e Lu, L. (2000). Modeling and performance control of internet servers. Proc. *Proc. Of 39th IEEE Conference on Decision and Control*, pp. 2234–2239, Sydney, NSW, Australia.

[3] Abdelzaher, T. F. e Bhatti, N. (1999). Web server qos management by adaptive content delivery. *IEEE Computer Networks Amsterdam, Netherlands*, Vol. 31, No. 11–16, pp. 1563–1577.

[4] Almeida, J., Dabu, M., A.Manikutty, e Cao, P. (1998). Providing differentiated levels of service in web content hosting. Proc. *First Workshop on Internet Server Performance*, Madison, Wisconsin.

[5] Amirijoo, M., Hansson, J., e Son, S. H. (2006). Specification and management of qos in imprecise real-time databases. *IEEE Transactions on Computers*, Vol. 55, No. 3, pp. 304–319.

[6] Andersson, M., Kihl, M., e Robertsson, A. (2003). Modelling and design of admission control mechanisms for web servers using non-linear control theory. Proc. *Proc. Proceedings of ITCom*.

[7] Apache (2006b). request rec structure reference.

[8] Apache (acessado em 06/2006a). The apache org.

[9] Arlitt, M. e Jim, T. (2000). A workload characterization of the 1998 world cup web site. *IEEE Network*, Vol. 14, No. 3, pp. 30–37.

[10] Aurrecoechea, C., Campbell, A. T., e Hauw, L. (1998). A survey of qos architectures. *Multimedia Systems*, Vol. 6, No. 3, pp. 138–151.

[11] Baker, F. (1995). Requirements for ip version 4 routers. *IETF RFC 1812*, Vol. .

[12] Banga, G. e Druschel, P. (1997). Measuring the capacity of a web server. Proc. *Symposium on Internet Technologies and Systems*, pp. 61–71.

[13] Barbetta, P. A., Bornia, A. C., e Reis, M. M. (2004). *Estatística para Cursos de Engenharia e Informática*. Editora Atlas, São Paulo, Br.

[14] Barford, P. e Crovella, M. (1998). Generating representative web workloads for network and server performance evaluation. Proc. *Measurement and Modeling of Computer Systems*, pp. 151–160.

[15] Baruah, S., Koren, G., Mao, D., Mishra, B., Raghunathan, A., Rosier, L., Shasha7, D., e Wang, F. (1992). On the competitiveness of on-line real-time task scheduling. *Journal Real-Time Systems*, Vol. 4, No. 2, pp. 1573–1383.

[16] Bhatti, N. e Friedrich, R. (1999). Web server support for tiered services. *IEEE Network*, Vol. 13, No. 5, pp. 64–71.

[17] Blair, G., Coulson, G., e Davies, N. (1994). System support for multimedia applications: An assessment of the state of the art.

[18] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., e Weiss, W. (1998). An architecture for differentiated services. *IETF RFC 2475*, Vol. .

[19] Braden, R., Clark, D., e Shenker, S. (1994). Integrated services in the internet architecture: an overview. *IETF RFC 1633*, Vol. .

[20] Bradner, S. (2005a). Ietf rights in contributions. *IETF RFC 3978*, Vol. .

[21] Bradner, S. (2005b). Intellectual property rights in ietf technology. *IETF RFC 3979*, Vol. .

[22] C Lu, T. F. Abdelzaher, J. A. S. e SO, S. H. (2001). A feedback control approache for guaranteeing relative delays in web servers. Proc. *Proc. Of 7th IEEE Real-Time Technology and Applications Symposium*, pp. 51–62.

[23] Cisco Systems, I. (2001). Diffserv – the scalable end-to-end qos model.

[24] Crovella, M. E. e Bestavros, A. (1997). Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE Trans. Networking*, Vol. 5, No. 6, pp. 835–845.

[25] Demers, A. e Srinivasan, S. (1989). Analysis and simulation of a fair queueing algotithm. Proc. *in Proceedings on Communications architectures and protocols*, Austin, Texas, USA.

[26] Dovrolis, C. e Ramanathan, P. (1999). A case for relative differentiated services and the proportional differentiation model. *IEEE Network*, Vol. 13, No. 5, pp. 26–34.

[27] Eggert, L. e Heidemann, J. (1999). Application-level differentiated services for web servers. *World Wide Web*, Vol. 2, No. 3, pp. 133–142.

[28] El-Gendy, M. A., Bose, A., e Shin, K. G. (2003). Evolution of the internet qos and support for soft real-time applications. Proc. *Proc. of the 2nd IEEE*.

[29] Fluckiger, F. (1995). *Understanding Networking Multimedia: Applications and Technology*. Prentice-Hall, Hertfordshire, UK, UK.

[30] Grone, B., Knopfel, A., Kugel, R., e Shimidt, O. (2004). *The Apache Modeling Project*. Free Software Foundation, CH.P.I. - Hasso Plattner Institute for Software Systems Engineering, Postdam, Germany.

[31] Heinanen, J., Baker, F., Weiss, W., e Wroclawski, J. (1999). Assured forwarding phb group. *IETF RFC 2597*, Vol. .

[32] Inc, K. S. (acessado em 06/2006). Keynote systems performance indexes.

[33] Jacobson, V. (1995). Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, Vol. 25, No. 1, pp. 157–187.

[34] Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, NY, USA.

[35] Jensen, D. E. (1997). Eliminating the 'hard'/'soft' real-time dichotomy. *Computing and Control Engineering Journal*, Vol. 8, No. 1, pp. 15–19.

[36] Joutsensalo, J., Hämäläinen, T., Pääkkönen, M., e Sayenko, A. (2003). Adaptive weighted fair scheduling method for channel allocation. *Communications, 2003. ICC '03. IEEE International Conference*, Vol. 1pp. 228–232.

[37] Kang, K., Son, S. H., e Stankovic, J. (2003). Differentiated real-time data services for e-commerce applications. *Journal Electronic Commerce Research*, Vol. 3, No. 1–2, pp. 58–68.

[38] Kurose, J. F. e W.Ross, K. (2001). *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley, Hertfordshire, UK, UK, 2 edição.

[39] Liu, J., Kuan, W.-K., Lay, K.-J., Bettati, R., e Chung, J.-Y. (1994). Imprecise computation. *in Proceedings of the IEEE*, Vol. 82, No. 1, pp. 83–94.

[40] Liu, J., Lin, K. J., Shih, W. K., Yu, A. C., Chung, J. Y., , e Zhao, W. (1991). Algorithms for scheduling imprecise computations. *IEEE Computation*, Vol. 24, No. 5, pp. 58–68.

[41] Lu, C. (1999). *Feedback Control Real-Time Scheduling*. Phd thesis, University of Virginia, Virginia.

[42] Marucheck, M. e Strosnider, J. (1995). Some insight into the fault recovery properties of priority-driven schedulers. *Real-Time Systems Journal*, Vol. 9, No. 2, pp. 133–142.

[43] Mercer, C. W. (1992). An introduction to real-time operating systems: Scheduling theory.

[44] Monteiro, J. A. S., Sampaio, L., e Figueredo, M. (2002). Qualidade de serviço: Diagnóstico e alternativas. *RNP*, Vol. .

[45] Montez, C. e Fraga, J. (2002). Implementing quality of service in web servers. Proc. *IEEE SRDS 2002*, pp. 32–40, Osaka, Japan.

[46] Montez, C., Fraga, J., Oliveira, R. S., e Farines, J.-M. (1999). An adaptive scheduling approach in real-time corba. Proc. *2nd IEEE ISORC*, pp. 301, Saint-Malo, France.

[47] Mosberger, D. e Jim, T. (1998). httperf - a tool for measuring web server performance. *ACM - First Workshop on Internet Server Performance*, Vol. pp. 56–67.

[48] Murta, C. D. e Corlassoli, T. P. (2003). Fastest connection first: A new scheduling policy for web servers. Proc. *18th International Teletraffic Congress (ITC-18)*, Berlin, Germany.

[49] Nichols, K., Blake, S., Baker, F., e Black, D. (1998). Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. *IETF RFC 2474*, Vol. .

[50] Nielsen (acessado em 06/2006). Nielsen/net ratings.

[51] Pandey, R., Barnes, J. F., e Ollsson, R. (1998). Supporting quality of service in HTTP servers. Proc. *Symposium on Principles of Distributed Computing*, pp. 247–256, Puerto Callarta, Mexico.

[52] Peterson, L. e Davie, B. (2004). *Redes de Computadores - Uma Abordagem de Sistemas*. Editora Campus, São Paulo, Br.

[53] Postel, J. (1981). Internet protocol. *IETF RFC 791*, Vol. .

[54] Sabata, B., Chatterjee, S., Davis, M., Sydir, J. J., e Lawrence, T. F. (1997). Taxonomy for qos specifications. Proc. *Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*. IEEE Computer Society.

[55] Schroeder, B. e Balter, M. H. (2006). Web servers under overload: How scheduling can help. *ACM Trans. Inter. Tech.*, Vol. 6, No. 1, pp. 20–52.

[56] Semprebom, T., Oliveira, R., e Montez, C. (2006). Classes de serviço em servidores web apache através de escalonamento adaptativo e controle de admissão. Proc. *In: XII Simpósio Brasileiro de Sistemas Multimídia e Web, 2006 - Webmedia 2006*, pp. 273–282, Natal, RN.

[57] SPECweb96 (1996). A explanation of the specweb96 benchmark.

[58] Striegel, A. e Manimaran, G. (2000). Dynamic class-based queue management for scalable media servers. Proc. *Sixth IEEE RTAS*, pp. 228–236, Washington DC, USA.

[59] Trent, G. e Sakem, M. (1995). Webstone: The first generation in http server benchmarking.

[60] Vasiliou, N. e Lutfiyya, H. (2000). Providing a differentiated quality of service in a world wide web server. *SIGMETRICS Perform. Eval. Rev.*, Vol. 28, No. 2, pp. 22–28.

[61] Yoon, H., Kim, H., Oh, C., e Kim, K. (1999). A queue length-based scheduling scheme in atm networks. Proc. *Proceedings of the IEEE Region 10 Conference*, pp. 234–237, Cheju Island, South Korea.