Alan Massaru Nakai

“Novas Técnicas de Distribuição de Carga para Servidores Web Geograficamente Distribuídos”

CAMPINAS
2012

Universidade Estadual de Campinas
Instituto de Computação

Alan Massaru Nakai

“Novas Técnicas de Distribuição de Carga para Servidores Web Geograficamente Distribuídos”

Orientador(a): **Prof. Dr. Edmundo Roberto Mauro Madeira**
Co-Orientador(a): **Prof. Dr. Luiz Eduardo Buzato**

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de PhD em Ciência da Computação.

Este exemplar corresponde à versão final da Tese defendida por Alan Massaru Nakai, sob orientação de Prof. Dr. Edmundo Roberto Mauro Madeira.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Nakai, Alan Massaru, 1979-
N145n Novas técnicas de distribuição de carga para servidores Web geograficamente distribuídos / Alan Massaru Nakai. – Campinas, SP : [s.n.], 2012.

Orientador: Edmundo Roberto Mauro Madeira.
Coorientador: Luiz Eduardo Buzato.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. World Wide Web - Servidores. 2. Redes de computadores - Protocolos. 3. Sistemas distribuídos. 4. Serviços na Web. 5. Problema de balanceamento de carga. I. Madeira, Edmundo Roberto Mauro,1958-. II. Buzato, Luiz Eduardo,1961-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em inglês: New techniques for load distribution for geographically distributed Web servers
Palavras-chave em inglês:
Web servers
Computer network protocols
Distributed systems
Web services
Load balancing problem
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Edmundo Roberto Mauro Madeira [Orientador]
Marcos José Santana
Irineu Sotoma
Maria Beatriz Felgar de Toledo
Luiz Fernando Bittencourt
Data de defesa: 14-09-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 14 de Setembro de 2012, pela Banca examinadora composta pelos Professores Doutores:

Irineu Sotoma

Prof. Dr. Irineu Sotoma
FACOM - UFMS

Prof. Dr. Marcos José Santana
ICMC - USP

Maria Beatriz Felgar de Toledo

Profª. Drª. Maria Beatriz Felgar de Toledo
IC - UNICAMP

Luiz Fernando Bittencourt

Prof. Dr. Luiz Fernando Bittencourt
IC- UNICAMP

Edmundo Madeira

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Novas Técnicas de Distribuição de Carga para Servidores Web Geograficamente Distribuídos

Alan Massaru Nakai[1]

Setembro de 2012

Banca Examinadora:

- Prof. Dr. Edmundo Roberto Mauro Madeira (Orientador)
- Prof. Dr. Marcos José Santana
  ICMC - USP
- Prof. Dr. Irineu Sotoma
  Faculdade de Computação - UFMS
- Profa. Dra. Maria Beatriz Felgar de Toledo
  IC - UNICAMP
- Prof. Dr. Luiz Fernando Bittencourt
  IC - UNICAMP
- Prof. Dr. Gustavo Maciel Dias Vieira (Suplente)
  UFSCAR
- Profa. Islene Calciolari Garciaz (Suplente)
  IC - UNICAMP
- Profa. Dra. Claudia Bauzer Medeiros (Suplente)
  IC - UNICAMP

[1] Suporte financeiro de: Capes, Fapesp (07/56423-6)

Resumo

A distribuição de carga é um problema intrínseco a sistemas distribuídos. Esta tese aborda este problema no contexto de servidores web geograficamente distribuídos. A replicação de servidores web em *datacenters* distribuídos geograficamente provê tolerância a falhas e a possibilidade de fornecer melhores tempos de resposta aos clientes. Uma questão chave em cenários como este é a eficiência da solução de distribuição de carga empregada para dividir a carga do sistema entre as réplicas do servidor. A distribuição de carga permite que os provedores façam melhor uso dos seus recursos, amenizando a necessidade de provisão extra e ajudando a tolerar picos de carga até que o sistema seja ajustado. O objetivo deste trabalho foi estudar e propor novas soluções de distribuição de carga para servidores web geograficamente distribuídos. Para isso, foram implementadas duas ferramentas para apoiar a análise e o desenvolvimento de novas soluções, uma plataforma de testes construída sobre a implementação real de um serviço web e um software de simulação baseado em um modelo realístico de geração de carga para web. As principais contribuições desta tese são as propostas de quatro novas soluções de distribuição de carga que abrangem três diferentes tipos: soluções baseadas em DNS, baseadas em clientes e baseadas em servidores.

Abstract

Load balancing is a problem that is intrinsic to distributed systems. In this thesis, we study this problem in the context of geographically distributed web servers. The replication of web servers on geographically distributed datacenters allows the service provider to tolerate failures and to improve the response times perceived by clients. A key issue for achieving good performance in such a deployment is the efficiency of the load balancing solution used to distribute client requests among the replicated servers. The load balancing allows providers to make better use of their resources, soften the need for over-provision, and help tolerate abrupt load peaks until the system can be adjusted. The objective of this work was to study and propose load balancing solutions for geographically distributed web servers. In order to accomplish this objective, we have implemented two tools that support the analysis and development of load balancing solutions, a testbed that was built on top of a real web service implementation, and a simulation software that is based on a realistic model for web load generation. The main contributions of this thesis are the proposals of four new load balancing solutions that comprehend three types: DNS-based, client-based, and server-based.

Agradecimentos

Aos meus orientadores, Profs. Edmundo e Buzato, pela extrema dedicação e paciência.
À minha esposa Andréia, incondicionalmente compreensiva e amorosa.
Aos meus pais e à minha irmã, pela confiança e apoio.
Aos meus sogros, que sempre torceram por mim.
À Profa. Claudia, praticamente uma orientadora honorária.
À Carla, por toda ajuda e motivação.
À Edna e ao Vagner, meus amigos e consultores.
Aos colegas do LIS, pelo companheirismo e colaboração.
Aos colegas da Embrapa, em especial ao Assad, por todo o apoio.
Ao IC, pela oportunidade, e a todos os seus funcionários, que sempre foram prestativos.
Finalmente, agradeço as agências de fomento FAPESP (07/56423-6), CAPES e CNPq.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Esta tese aborda um problema intrínseco aos sistemas distribuídos: a distribuição de carga entre processadores paralelos. Aqui, este problema é estudado no contexto de serviços acessados por meio da web, cuja importância é sabidamente crescente. Cada vez mais companhias realizam porções significativas de seus negócios *online*. Cresce também a oferta de serviços pela web e, mais importante que isso, aumenta o número de consumidores para esses serviços.

Neste contexto, aumenta a demanda por soluções para prover alta disponibilidade de serviços web. Um motivo evidente para isto é o fator financeiro [97]. A indisponibilidade dos serviços pode gerar descontentamento de clientes, influenciando negativamente na imagem da companhia e, consequentemente, gerando perdas de receita. Além disso, a falta de disponibilidade do sistema em aplicações críticas, como aplicações militares e da medicina, pode ter consequências catastróficas, levando risco ao bem estar humano.

Uma solução comum para o aumento da disponibilidade de um serviço web é a adição de redundância ao sistema, via replicação dos servidores. Esta solução aumenta a probabilidade de que um cliente possa conectar-se a um servidor mesmo na presença de falhas parciais. Além disso, a replicação potencializa a escalabilidade do serviço web, diminuindo a chance de sobrecargas. Existem várias formas para replicar uma aplicação web pela internet, sendo por meio da replicação total da base de dados [41, 43, 78] ou por mecanismos de *caching* [73, 84]. A melhor solução depende da carga de trabalho da aplicação e do nível de consistência necessário entre as réplicas. Atualmente, serviços de CDN (*Content Delivery Network*) permitem que companhias repliquem seu conteúdo em centenas de servidores web espalhados pelo mundo. Mesmo pequenas e médias empresas podem implantar seus serviços em *datacenters* geograficamente distribuídos contratando serviços de computação em nuvem [44], como Amazon, Windows Azure e Google Apps.

Independente da forma como a aplicação é replicada, um problema essencial para este tipo de abordagem é a distribuição da carga entre as réplicas do servidor web [47]. A solução de distribuição de carga permite que os provedores de serviços façam melhor uso de seus recursos de hardware e diminuam a necessidade de provisão extra de recursos. Além disso, mesmo quando os recursos de hardware são elásticos, a distribuição de carga pode auxiliar a tolerar picos de carga até que o sistema seja ajustado. Como reflexo da importância deste tema, uma grande quantidade de artigos acadêmicos foram publicados sobre o assunto (ver Seção 2.1) e grandes companhias que atuam na internet, como Microsoft, IBM e Akamai, têm patenteado diversas soluções para distribuição de carga [10, 21, 29, 49, 52, 87].

De forma geral, as técnicas para distribuição de carga para servidores web replicados encontradas na literatura podem ser divididas em duas categorias: técnicas para servidores web geograficamente distribuídos e técnicas para servidores web replicados e hospedados em aglomerados. Incluem-se na primeira categoria as soluções aplicáveis a servidores web distribuídos nos quais todo nó replicado possui um endereço IP visível aos clientes. A segunda categoria engloba as abordagens aplicáveis aos servidores web distribuídos cujos nós compõem um aglomerado, apresentando um único endereço IP visível, geralmente atribuído ao roteador de borda do aglomerado. Esta tese foca a primeira categoria.

O problema da distribuição de carga entre servidores web geograficamente distribuídos consiste em ligar dinamicamente o cliente do serviço web com a réplica mais apropriada do servidor. É importante notar que, no contexto deste problema, é indiferente se a réplica é um único computador ou um *datacenter* com milhares de servidores. A réplica mais apropriada pode ser escolhida de acordo com a necessidade de distribuir a carga total do sistema entre as réplicas disponíveis e de prover menor tempo de resposta para o cliente [47]. A grande dificuldade em tratar este problema é lidar com as altas latências encontradas na internet, que podem afetar os tempos de resposta percebidos pelos clientes e dificultam a troca de informações de controle. A partir daqui, para facilitar a leitura, o termo “distribuição de carga” será utilizado para referenciar o problema da distribuição de carga para servidores web geograficamente distribuídos.

De acordo com Sivasubramanian et al. [85], uma solução para o problema de distribuição de carga é composta por uma política de distribuição de carga e por um mecanismo de distribuição de carga. A política de distribuição de carga define como selecionar uma réplica do servidor web para a qual atribuir uma requisição do cliente. A política é basicamente um algoritmo invocado quando o cliente realiza uma requisição ao serviço web. Um exemplo simples de política de distribuição é a estratégia de rotação (*Round Robin* - RR), no qual o DNS autoridade retorna um endereço IP, de uma lista de rotativa, a cada requisição recebida [51].

A política de distribuição de carga pode utilizar diferentes informações na tomada de decisões, como [24, 85]:

- Informações dos servidores: tamanho de fila de requisições, histórico de utilização, medidas de carga (p.ex. uso de CPU, memória, disco e rede) e capacidade de processamento;

- Informações dos clientes: distância entre clientes e servidores, carga gerada pelos clientes e latência sentida pelos clientes.

O mecanismo de distribuição de carga é o meio pelo qual os clientes são dinamicamente ligados à réplica selecionada pela política de distribuição de carga. Os mecanismos de distribuição de carga podem ser divididos em quatro tipos [16], dependendo em que parte do sistema a política é executada: (i) mecanismos de distribuição via DNS, (ii) via despachador, (iii) via servidor e (iv) via cliente. Na distribuição de carga via DNS, o DNS autoridade (ADNS - do inglês *Authoritative DNS*) realiza o papel de escalonador e responde às requisições de resolução de nomes com o endereço IP de um servidor apropriado, de acordo com a política adotada. No caso da distribuição de carga via despachador, um dispositivo (o despachador) recebe todas as requisições e as distribui entre as réplicas do servidor web. Nas soluções baseadas no servidor, os servidores web executam a política de distribuição de carga e redirecionam o excesso de carga para os servidores menos carregados. Nos mecanismos baseados nos clientes, as aplicações clientes executam a política de distribuição de carga e selecionam o servidor apropriado.

O objetivo desta tese de doutorado foi estudar e propor novas soluções de distribuição de carga para servidores web geograficamente distribuídos. A tese foca em três dos quatro tipos de soluções existentes: distribuição via DNS, via clientes e via servidor. Os mecanismos baseadas em despachador não foram abordados por serem mais apropriados para ambientes de aglomerados. O primeiro passo desta pesquisa foi a construção de uma plataforma de testes para apoiar a análise e o desenvolvimento de soluções de distribuição de carga. Esta etapa da pesquisa envolveu a implementação de um serviço web real, baseado em um *benchmark* para aplicações de comércio eletrônico e gerou a publicação [65]. A plataforma de testes apoiou o segundo passo da pesquisa, que consistiu em estudar soluções baseadas em DNS. A partir deste estudo, foi proposta uma nova solução de distribuição de carga via DNS que combina informações provenientes tanto de servidores quanto de clientes nas tomadas de decisões. Este resultado foi publicado em [62].

Embora a plataforma de testes fornecesse facilidades para desenvolver e analisar novas soluções de distribuição de carga, a falta de infraestrutura de hardware para executar os experimentos em maior quantidade e escala tornou-se um empecilho. Desta forma, decidiu-se utilizar simulações para os próximos passos da pesquisa. Para isso, desenvolveu-se um simulador baseado em modelos realísticos de geração de carga e internet.

O terceiro passo da pesquisa focou em soluções de distribuição de carga via servidores. Como resultado, foi proposta uma nova solução baseada em servidores que limita a quantidade de carga que cada servidor sobrecarregado pode redirecionar para os outros, levando em consideração a demanda e a oferta global de recursos e a latência entre os servidores. Esta solução foi publicada em [64].

O quarto passo da pesquisa foi propor uma nova solução baseada em clientes. Nesta solução, ao invés de selecionar o melhor servidor de forma gulosa, assim como na maioria das propostas encontradas na literatura, as aplicações clientes dividem sua carga entre diversos servidores e alteram dinamicamente a quantidade de carga enviada para cada um, de acordo com os tempos de resposta percebidos. Os resultados obtidos por esta solução foram publicados em [63].

O último passo da pesquisa consistiu em estender a solução apresentada em [64], tratando a divisão dos recursos disponíveis em servidores remotos como um problema de otimização. A nova solução foi analisada a partir de um número maior de simulações que englobaram aspectos de parametrização da política, escalabilidade e sensibilidade a mudanças bruscas de carga de trabalho. Os novos resultados foram submetidos para [61].

Em resumo, as contribuições desta tese são:

1. Uma plataforma de testes e um software de simulação para apoiar o desenvolvimento e avaliação de soluções de distribuição de carga;

2. Uma nova solução de distribuição de carga via DNS, que combina informações de clientes e servidores;

3. Uma nova solução de distribuição de carga via clientes, que adapta a quantidade de carga enviada para cada servidor dinamicamente;

4. Um middleware para compartilhamento de recursos entre as réplicas dos servidores web;

5. Duas novas soluções de distribuição de carga via servidor. A primeira, baseia-se em uma estratégia que limita a quantidade de carga que cada servidor sobrecarregado pode submeter para servidores remotos. A segunda estende a primeira, tratando a divisão de recursos como um problema de otimização.

Esta tese corresponde a uma coletânea de artigos e está organizada da seguinte forma:

- **Capítulo 2**: Revisão bibliográfica envolvendo trabalhos correlatos e outros trabalhos que influenciaram o desenvolvimento da tese.

- **Capítulo 3**: Corresponde ao trabalho [65]. Descreve o projeto e a implementação de uma plataforma de testes para apoiar a análise e o desenvolvimento de soluções de distribuição de carga para serviços web geograficamente distribuídos. O trabalho também apresenta uma comparação de soluções representativas dos diferentes tipos existentes: baseado em DNS, baseado em despachador, baseado no servidor e baseado no cliente.

- **Capítulo 4**: Corresponde ao trabalho [62]. Apresenta o desenvolvimento e a avaliação de uma nova solução de distribuição de carga via DNS, que combina informações de clientes e servidores visando amenizar os efeitos negativos do sistema de *caching* do DNS sobre a distribuição de carga.

- **Capítulo 5**: Corresponde ao trabalho [63]. Introduz o software de simulação para soluções de distribuição de carga. Também descreve o desenvolvimento e a avaliação de uma nova solução de distribuição de carga via clientes, que altera a quantidade de carga que cada cliente submete para cada servidor de forma adaptativa, buscando evitar a sobrecarga dos servidores e melhorar os tempos de resposta;

- **Capítulo 6**: Corresponde ao trabalho [64]. Apresenta o desenvolvimento e a avaliação de uma solução de distribuição de carga via servidores, que limita a quantidade de carga que cada servidor sobrecarregado pode redirecionar para servidores remotos levando em consideração a demanda e a oferta global de recursos;

- **Capítulo 7**: Corresponde a um trabalho submetido para [61]. Apresenta uma extensão da solução apresentada no Capítulo 6. Esta nova proposta utiliza programação linear para otimizar a quantidade de carga que cada servidor sobrecarregado pode redirecionar para servidores remotos.

- **Capítulo 8**: Conclusões do trabalho e possíveis extensões.

- **Apêndice A**: Apresenta análises estatísticas que complementam os resultados apresentados nos capítulos 4, 5 e 6.

Capítulo 2

Conceitos e Revisão Bibliográfica

Este capítulo apresenta conceitos relacionados à tese e trabalhos correlatos que influenciaram direta ou indiretamente no seu desenvolvimento. A Seção 2.1 descreve os diferentes tipos de mecanismos de distribuição de carga para servidores web distribuídos e apresenta trabalhos que abordaram cada um deles. A Seção 2.2 apresenta trabalhos relacionados à distribuição de carga em áreas afins. A Seção 2.3 aborda trabalhos relacionados às técnicas de replicação e *caching* para serviços web e a Seção 2.4 apresenta trabalhos que descrevem plataformas de testes para sistemas distribuídos.

2.1 Distribuição de Carga para Servidores Web Distribuídos

Nesta seção, são apresentados os tipos de solução de distribuição de carga para servidores web distribuídos e trabalhos acadêmicos que propõem soluções de cada tipo.

2.1.1 Distribuição de Carga via DNS

O principal propósito do DNS é permitir que usuários traduzam nomes para endereços IP de forma eficiente. A informação do DNS é distribuída em uma hierarquia de domínios e subdomínios, sendo cada um deles administrado independentemente por um servidor de nomes autoridade (ADNS) [59].

Os servidores de nomes são capazes de responder a dois tipos de consultas: iterativas e recursivas. No primeiro tipo, o servidor de nomes retorna a resposta para a consulta utilizando as informações de seu próprio banco de dados ou uma referência para outro servidor de nomes que pode ser capaz de responder a consulta. No segundo tipo, o servidor de nomes consulta todos os servidores de nomes necessários para obter a resposta e retorná-la para o usuário.

Para tornar a resolução de nomes mais eficiente, os servidores de nomes podem armazenar mapeamentos de nome-para-IP em uma *cache* local, evitando consultar o ADNS para cada resolução de nome. O tempo em que um mapeamento pode ser armazenado em *cache* é definido por um valor TTL (*Time to Live* – Tempo de Vida), que é definido pelo ADNS de cada domínio.

Normalmente, os servidores de nomes dos níveis mais altos da hierarquia são configurados para enviar e receber apenas consultas iterativas. Quando um programa necessita acessar uma URL, um resolvedor de nomes do sistema operacional envia uma consulta recursiva para o servidor de nomes local. Então o servidor de nomes local realiza consultas iterativas ao longo da hierarquia do DNS até ser capaz de responder a consulta do usuário.

Nas soluções para distribuição de carga baseadas em DNS, o ADNS do servidor Web distribuído desempenha a função de escalonador de clientes. Neste tipo de abordagem, a URL do serviço fornece uma interface única para o cliente, para quem a distribuição é transparente. A distribuição da carga é realizada no momento da resolução da URL requisitada pelo cliente. Quando o ADNS recebe uma requisição para resolução de URL, responde com o endereço IP de uma das réplicas do servidor web, de acordo com alguma política de distribuição preestabelecida. A Figura 2.1 ilustra o funcionamento dos mecanismos baseados em DNS.

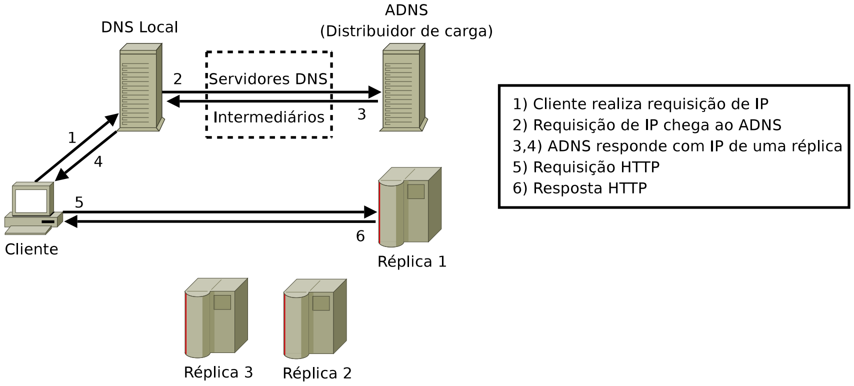


Figura 2.1: Mecanismo de distribuição de carga via DNS.

A principal limitação das soluções para distribuição de carga baseadas em DNS é que os mecanismos da internet para *caching* de endereços diminuem o controle do ADNS. Entre o cliente e o ADNS, muitos servidores de nomes armazenam mapeamentos de endereço para reduzir tráfego da rede, de forma que menos requisições de resolução de nomes alcançam o ADNS. Uma maneira de aumentar o controle do ADNS é diminuir o TTL das atribuições. Entretanto, TTLs muito pequenos aumentam o tráfego na rede e podem fazer com que os servidores de nomes intermediários tornem-se gargalos do sistema. Por esta razão, servidores de nomes intermediários podem recusar TTLs pequenos. Este problema motivou a criação de estratégias de distribuição de carga que utilizam TTLs dinâmicos.

Colajanni, Yu e Dias [25] apresentam e comparam políticas de distribuição de carga baseadas em DNS. Os autores dividem as políticas em três grupos, dependendo do tipo de informações que utilizam:

- Políticas que utilizam informações dos domínios dos clientes. Um exemplo destas políticas é o *2-Tier Round Robin*, no qual o ADNS divide os clientes em dois grupos, dependendo da quantidade de carga gerada, e aplica a estratégia rotativa separadamente em cada grupo.

- Políticas que utilizam informações dos servidores. Um exemplo destas políticas é a *Least Utilized Node* (LUN), na qual o ADNS seleciona o servidor menos utilizado.

- Políticas que utilizam informações dos clientes e dos servidores. Um exemplo destas políticas é a *Minimum Residual Load* (MRL), na qual o ADNS utiliza informações de carga dos clientes e da capacidade dos servidores para estimar o estado global do sistema.

Os resultados apresentados pelos autores mostram que, dentre os algoritmos abordados, os mais promissores combinam um alarme assíncrono que avisa que os servidores tornaram-se sobrecarregados com informações do domínio do cliente (divisão de classes). Informações detalhadas da carga dos servidores não se mostraram vantajosas.

Em outro trabalho, Colajanni e Yu [24] apresentam um estudo sobre políticas para distribuição de carga via DNS com TTL (*time-to-live*) variável. O uso de TTL variável visa aumentar o controle do DNS, que é limitado pelo mecanismo de *caching*. Os autores definem dois tipos de políticas baseadas em TTL variável:

- **TTL variável**: diminui o TTL conforme o número de servidores sobrecarregados aumenta;

- **TTL Adaptativo**: em um primeiro momento, o ADNS seleciona um servidor (conforme uma das políticas já mencionadas); em seguida, define um valor de TTL de acordo com a capacidade do servidor. Por exemplo, a estratégia TTL/k divide os domínios dos clientes em k classes, de acordo com a quantidade de carga gerada pelo domínio, e define TTLs diferentes para cada classe. Quanto maior a carga gerada pelo domínio, menor é o TTL. Isto faz com que domínios que geram mais carga troquem de servidor com maior frequência, amenizando a diferença entre as cargas de trabalho dos servidores.

Moon e Kim [60] propõem uma solução baseada em *Round Robin* que utiliza três tipos de informação para remover um servidor sobrecarregado da lista do ADNS: utilização da CPU, utilização da rede e utilização da memória. A política de remoção e adição de servidores da lista combina estas três informações com diferentes pesos, que podem ser alterados de acordo com a natureza da aplicação Web. A arquitetura proposta não exige alteração do servidor de DNS. Um módulo independente coleta as informações dos servidores e altera a lista do DNS, que por sua vez utiliza a estratégia rotativa.

Pan, Hou e Li [70] descrevem o sistema de seleção de servidores via DNS utilizado pela Akamai, uma grande rede de distribuição de conteúdo (CDN) que replica o conteúdo a ser distribuído em escala global. O DNS autoridade do *site* replicado retorna a URL de um servidor DNS da rede de distribuição. Este servidor DNS retorna o endereço da réplica mais próxima ao DNS local do cliente. Esta técnica pressupõe que o cliente está localizado próximo ao seu DNS local.

Shaikh, Tewari e Agrawal [83] apresentam um estudo sobre duas questões importantes relacionadas a mecanismos de distribuição de carga baseados em DNS: (i) os efeitos negativos do uso de TTLs pequenos ou nulos; e (ii) a pressuposição de que servidores DNS dão indicativos da localização e do desempenho dos clientes. Os estudos mostraram que o não uso de cache (TTL=0) pode causar uma sobrecarga na resolução de nomes de até 2 ordens de grandeza. Os autores também afirmam que a pressuposição de que a localização dos clientes é próxima à localização dos servidores DNS é violada com frequência. Além disso, a medida de latência do DNS local não é um bom indicativo da latência dos clientes. Os autores também sugerem a adição de um novo registro de recursos ao DNS para identificar o cliente que originou a consulta de resolução de nomes, visando aumentar a acurácia da seleção de servidores quando a proximidade do cliente é um fator decisivo.

Uma das contribuições desta tese é a proposta de uma nova solução de distribuição de carga via DNS (Capítulo 4). Esta solução utiliza informações de clientes, como a MRL [25], mas combina estas informações com informações provenientes dos servidores para diminuir o efeito negativo do sistema de *caching* do DNS sobre este tipo de solução.

2.1.2 Distribuição de Carga via Servidores

Nos mecanismos de distribuição de carga via servidores, após o primeiro nível de distribuição de carga, feito na resolução de endereços, qualquer réplica do servidor distribuído pode redirecionar requisições quando sobrecarregado. Uma desvantagem deste tipo de mecanismo é que o uso de redirecionamento pode aumentar o tempo de resposta para o cliente. Por outro lado, pode compensar o baixo controle do ADNS e contribuir para uma melhor distribuição da carga. A Figura 2.2 ilustra o funcionamento deste tipo de mecanismo.

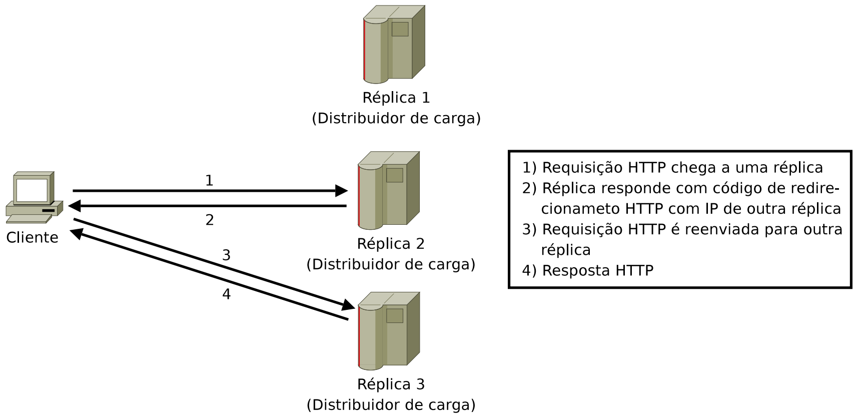

Réplica 1
(Distribuidor de carga)
1
2
Cliente
3
4
Réplica 2
(Distribuidor de carga)
Réplica 3
(Distribuidor de carga)
1) Requisição HTTP chega a uma réplica
2) Réplica responde com código de redire-cionameto HTTP com IP de outra réplica
3) Requisição HTTP é reenviada para outra réplica
4) Resposta HTTP

Figura 2.2: Mecanismo de distribuição de carga via servidores.

Cardellini [17] identifica três fases para políticas de distribuição de carga via servidores:

- Política de ativação do redirecionamento: determina quais nós do servidor web podem atuar como redirecionadores. As políticas de ativação de redirecionamento podem ser centralizadas ou distribuídas. Nas estratégias centralizadas, um único nó decide quando ativar o redirecionamento e realiza um *broadcast* da decisão. Neste tipo de estratégia, a decisão é tomada com base em informações globais, ou seja, envolvendo todos os nós servidores. Já no esquema distribuído, cada nó servidor decide quando deve ativar o redirecionamento. Estas estratégias podem considerar somente informações locais, do próprio nó, ou informações globais. Assim como as estratégias baseadas em DNS, a política de ativação de redirecionamento pode considerar diferentes tipos de informação, como a taxa de requisições por domínio e a carga dos nós servidores;

- Política de seleção de requisições: determina as requisições que podem ser redirecionadas, desde que o redirecionamento esteja ativo. Exemplos simples são: redirecionar qualquer requisição ou utilizar escolha aleatória. Políticas mais elaboradas podem levar em consideração informações do cliente, como sua localização, ou informações mais detalhadas, como o conteúdo da requisição HTTP;

- Política de localização de nó: determina o nó apropriado para o qual uma requisição pode ser redirecionada. De maneira semelhante à política de ativação de redirecionamento, a política de localização de nó pode variar de acordo com a centralização (ou não) da decisão e quanto ao nível de informação considerado.

A arquitetura básica ilustrada na Figura 2.2 pode variar, de acordo com a técnica utilizada para encaminhar as requisições para outra réplica do servidor. Dentre estas técnicas, pode-se citar: triangularização [9], redirecionamento HTTP [37, 89, 98, 46], reescrita de URL [53] e reescrita de pacotes [3]. Na triangularização, o cliente continua enviando pacotes para o primeiro servidor contactado, mesmo que o serviço esteja sendo servido por outro servidor. O primeiro servidor encapsula os datagramas recebidos em outros datagramas e encaminha para o servidor destino. Este último, por sua vez, responde diretamente para o cliente. O redirecionamento HTTP baseia-se na funcionalidade do protocolo HTTP que permite ao servidor web responder uma requisição com os códigos de estado 301 ou 302 no cabeçalho da resposta. Estes códigos instruem o cliente a ressubmeter a requisição para outro servidor. No mecanismo de reescrita de URL, o servidor redirecionador reescreve dinamicamente as URLs dos objetos contidos na página de resposta, de forma que apontem para outros servidores. O mecanismo de reescrita de pacotes baseia-se em NAT (Tradução de Endereço de Rede - *Network Address Translation*) [86]. Neste mecanismo, todo servidor possui um IP visível aos clientes e um IP privado. Quando um servidor necessita redirecionar uma requisição, reescreve o IP fonte do pacote e o encaminha para o IP privado de outro servidor. Este último, por sua vez, responde para o mesmo servidor que encaminhou a requisição, que substitui o IP fonte da resposta e a encaminha para o cliente.

Chatterjee et al [20] apresentam uma solução na qual o ADNS atribui as requisições com base nas seguintes informações dos nós servidores: capacidade de processamento, carga de trabalho (CPU, memória e I/O) e o tamanho dos documentos requisitados (calculados a partir do tempo que o nó servidor necessita para servir o documento). Os domínios clientes são divididos em categorias, para cada qual é atribuído um valor TTL diferente. O ADNS reporta para todas as réplicas a informação global sobre os estados das mesmas. As réplicas utilizam essa informação para redirecionar as requisições para a réplica menos carregada, via redirecionamento HTTP.

Cardellini e Colajanni [17] compararam o desempenho de diferentes configurações de políticas de distribuição de carga via servidores. O estudo apresentado pelos autores inclui soluções totalmente centralizadas (ativação e localização centralizada), totalmente distribuídas e híbridas. Os resultados apresentados mostram que, embora a solução distribuída apresente um desempenho pior, são mais fáceis de implementar e impõem menor sobrecarga computacional e de comunicação.

Aarag e Jennings [3] propõem um sistema de reescrita de pacotes distribuído baseado em uma função *hash* adaptativa. Neste sistema, uma função *hash* adaptativa define se um servidor deve processar um pacote recebido ou encaminhá-lo para outro servidor. Se o servidor deve encaminhar um pacote, usa NAT (*Network Address Translation*) para reescrever os endereços fonte e destino do pacote e envia para o servidor apropriado. Os pacotes de resposta também tem seus endereços reescritos e são enviados para o cliente.

Hong et al [46] apresentam uma técnica que tem o objetivo de diminuir o número de mensagens necessárias para coletar informação de carga dos servidores Web replicados. Nesta técnica, os servidores são arranjados em anéis lógicos e trocam informação de carga ao longo do anel com o objetivo de identificar os nós com mais e menos carga. Usa-se redirecionamento HTTP para redirecionar as requisições dos clientes para o servidor apropriado.

Andreolini et al. [6] comparam várias soluções de distribuição de carga que variam de acordo com suas compatibilidades com quatro propriedades de sistemas autônomos: descentralização de controle, coleta e utilização de informações, adaptação a mudanças e colaboração fracamente acoplada. Os resultados apresentados mostram que a integração desses conceitos à solução de distribuição de carga melhoram a estabilidade e a robustez do sistema.

A solução proposta por Ranjan e Knightly [74] é composta por um conjunto de algoritmos que consideram a carga das CPUs e as latências de rede para reduzir o tempo de resposta e minimizar a utilização de recursos. Um algoritmo (QuID - *Quality-of-Service for Infrastructure-on-Demand*) aloca dinamicamente servidores dentro do aglomerado para atender a demanda local, enquanto um outro (WARD - *Wide Area Redirection*) decide para qual aglomerado remoto deve redirecionar requisições em caso de sobrecarga. Neste segundo caso, o próprio aglomerado redireciona as requisições, já que possui um enlace rápido para outro aglomerado. Um terceiro algoritmo decide se novos servidores devem ser alocados localmente ou se as requisições devem ser redirecionadas.

Pathan, Vecchiola, and Buyya [71] propõem uma solução similar a aquela proposta por [74]. A principal diferença é que a primeira baseia-se em redirecionamento HTTP, ou seja, o cliente é induzido a ressubmeter sua requisição para outro servidor. Já na segunda, o servidor que redireciona a requisição intermedia a comunicação entre o cliente e o outro servidor.

Ardagna et al. [7] combina técnicas de alocação de capacidade em nuvens distribuídas geograficamente com um mecanismo de redirecionamento de carga para minimizar os custos para alocação de máquinas virtuais, garantindo restrições de qualidade de serviço. O mecanimo de distribuição de carga baseia-se em predição de carga e utiliza técnicas de otimização não linear para decidir a fração de carga que deve ser atendida localmente e a fração que deve ser redirecionada.

Garg and Juneja [39] propõem uma solução na qual servidores e clientes colaboram por meio de dois tipos de agentes: *server ants*, que são executados nos servidores, e *client ants*, que são executados nos clientes. Tais agentes comunicam-se entre si para decidir o melhor servidor para servir cada cliente. Também mantém informações históricas sobre decisões anteriores que são levadas em consideração nas decisões futuras. Nishant et al. [66] apresentam uma abordagem semelhante que utiliza agentes móveis (*ants*) para percorrer a topologia da rede e identificar servidores sobrecarregados e não sobrecarregados.

As informações coletadas pelos agentes são utilizadas na distribuição da carga entre os servidores.

Esta tese apresenta duas novas soluções de distribuição de carga via servidores (Capítulos 6 e 7). Na primeira, a distribuição de carga é condicionada de acordo com limites de redirecionamentos impostos aos servidores sobrecarregados. Estes limites são calculados com base na demanda e oferta global de recursos. A segunda solução estende a primeira, tratando a divisão de recursos entre os servidores como um problema de otimização.

2.1.3 Distribuição de Carga via Clientes

Nos mecanismos de distribuição de carga via clientes, a política de distribuição de carga é executada na aplicação cliente (Figura 2.3). A principal desvantagem deste tipo de mecanismo é a falta de transparência para a aplicação cliente, que normalmente é um navegador web.

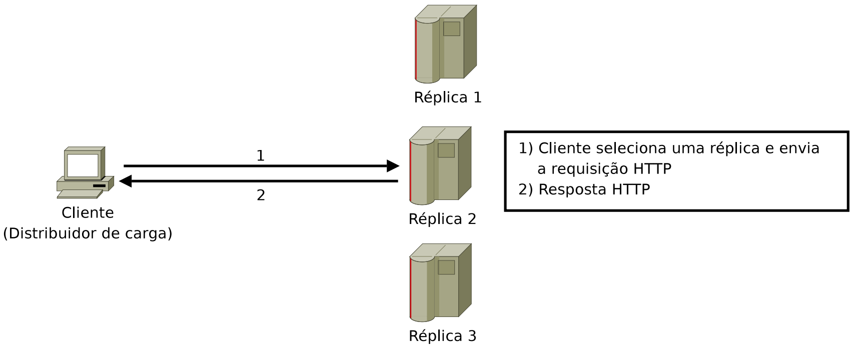


Figura 2.3: Mecanismo de distribuição de carga baseado no *Cliente*.

Um exemplo simples deste tipo de mecanismo de distribuição de carga é o uso de espelhos (*mirrors*). Nesta solução, uma página web é apresentada ao usuário com opções de diferentes URLs das réplicas dos servidores. O próprio usuário escolhe uma réplica da qual acessará o serviço.

Em soluções mais elaboradas, pode-se adicionar funcionalidades de distribuição de carga ao cliente por meio de *smart clients* carregados nas páginas web (ex. Java Applets) ou por meio de *proxies* localizados próximos aos clientes.

Dikes et al. [34] apresentam uma comparação de 6 políticas de seleção de servidores via clientes: *Random*, *Latency*, *BW*, *Probe*, *ProbeBW* e *ProbeBW2*. *Random* seleciona um servidor aleatoriamente. *Latency* e *BW* selecionam os servidores que oferecem a melhor latência de rede e banda de rede, respectivamente. *Probe* sonda todas as réplicas de servidores e seleciona aquela que responder primeiro. *ProbeBW* considera somente n réplicas com melhores bandas de rede e aplica *Probe* a elas. *ProbeBW2* sonda a latência de rede de todos os servidores e aplica BW aos n primeiros a responder. Em resumo, os resultados mostram que as políticas baseadas em sondagem apresentaram melhores desempenhos.

Conti, Gregori e Panzieri [28] propõem uma política de distribuição de carga via clientes que atribui uma nova requisição para a réplica com menor tempo de resposta. Para isso, antes de cada requisição, os clientes enviam mensagens para todos os servidores e medem o tempo das respostas. Em seguida, a requisição é enviada para o servidor que apresenta o menor tempo de resposta. Em um trabalho relacionado [26], os autores comparam esta política com uma política na qual o cliente divide a requisição em blocos menores e faz o *download* desses blocos, em paralelo, de todas as réplicas disponíveis. Segundo os autores, a solução paralela gera sobrecarga, porém em alguns casos pode ser útil: (i) quando os documentos para *download* são grandes; (ii) quando as operações do cliente são apenas de leitura; (iii) quando os clientes desejam maior vazão; e (iv) quando o desempenho das réplicas não é o gargalo do sistema.

Mendonça et al [56] apresentam uma comparação de políticas de seleção de servidores para serviços web SOAP (*Simple Object Access Protocol*). Consideraram cinco políticas no estudo: *random selection*, *parallel invocation*, *HTTPPing*, *best last* and *best median*. *Random selection* seleciona um servidor aleatoriamente. *Parallel invocation* invoca todos os servidores em paralelo e aguada a resposta mais rápida. *HTTPPing* envia uma pequena requisição HTTP para todos os servidores em paralelo. O primeiro a responder é invocado. *Best last* invoca o servidor com o melhor desempenho recente e *best median* invoca o servidor com o melhor que apresenta o melhor tempo de resposta mediano entre as últimas k invocações.

Esta tese propõe uma nova solução de distribuição de carga via clientes (Capítulo 5) na qual os clientes dividem sua carga entre diversos servidores. A fração de carga destinada a cada servidor é adaptada dinamicamente, buscando melhores tempos de resposta e evitando a sobrecarga dos servidores.

2.1.4 Distribuição de Carga via Despachador

Nos mecanismos baseados em despachador, um dispositivo trabalha como despachador do sistema, recebendo todas as requisições dos clientes e encaminhando para uma das réplicas do servidor web, de acordo com a política de distribuição de carga utilizada. O despachador possui o único endereço IP visível para os clientes e o encaminhamento da requisição é transparente. A Figura 2.4 ilustra o funcionamento deste tipo de mecanismo.

A principal vantagem dos mecanismos baseados em despachador é o controle total destes mecanismos sobre a distribuição das requisições dos clientes, já que todas as requisições passam obrigatoriamente pelo despachador. Por outro lado, uma desvantagem

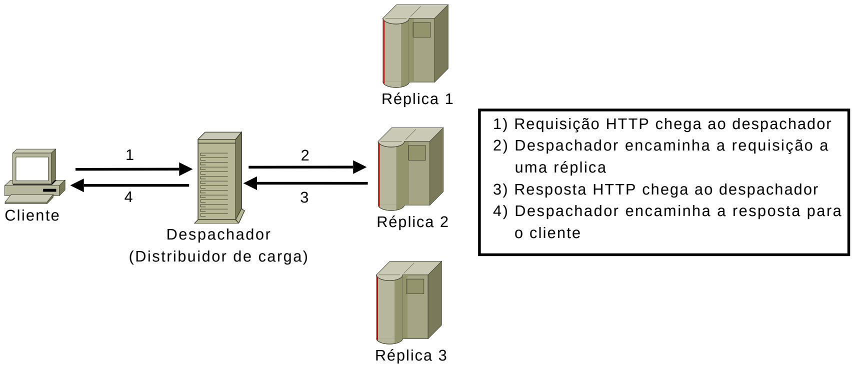

Réplica 1
Réplica 2
Réplica 3
Cliente
Despachador
(Distribuidor de carga)
1
2
3
4
1) Requisição HTTP chega ao despachador
2) Despachador encaminha a requisição a uma réplica
3) Resposta HTTP chega ao despachador
4) Despachador encaminha a resposta para o cliente

Figura 2.4: Mecanismo de distribuição de carga baseados em despachador.

é que o despachador é um ponto único de falha e pode se tornar o gargalo do sistema.

As técnicas utilizadas para encaminhar as requisições para as réplicas podem ser divididas em duas categorias, de acordo com a camada da rede na qual são aplicadas: camada de transporte ou camada de aplicação. Nas técnicas aplicadas à camada de transporte (TCP/IP), o servidor é determinado durante o estabelecimento da conexão TCP. Possíveis mecanismos de roteamento de requisições para estas técnicas são [19]:

- Reescrita dupla de pacotes (*Packet Double-Rewriting*): mecanismo baseado em NAT [86], semelhante ao mecanismo de reescrita de pacote descrito anteriormente. A diferença para o mecanismo anterior é que, neste caso, a tradução de endereço é realizada de forma centralizada pelo despachador;

- Reescrita simples de pacotes (*Packet Single-Rewriting*): o despachador altera o IP destino dos pacotes enviados pelos clientes e os encaminha para o nó servidor apropriado. O nó servidor altera os IPs fonte para o IP visível e os encaminha diretamente para o cliente [32];

- Tunelamento (*Tunneling*): o despachador empacota os pacotes enviados pelos clientes em outros pacotes e envia estes novos pacotes para o nó servidor apropriado. O nó servidor desempacota e processa o pacote original e responde diretamente para o cliente [72];

- Encaminhamento de pacotes (*Packet Forwarding*): o despachador compartilha o mesmo IP com os nós servidores. Os pacotes enviados pelos clientes são encaminhados para os endereços MAC privados dos nós servidores, que respondem diretamente para o cliente.

Nas técnicas aplicadas à camada de aplicação, o despachador primeiro estabelece a conexão TCP com o cliente, examina a requisição HTTP e então determina o nó servidor para o qual encaminhar a requisição. Este tipo de técnica pode ser menos eficiente do que aquelas aplicadas à camada de transporte, porém permite políticas de distribuição mais sofisticadas. A análise do conteúdo da requisição possibilita, por exemplo, melhorar o uso das *caches* dos servidores, prover serviços diferenciados em servidores especializados, aumentar o compartilhamento de carga e explorar afinidade de clientes. Dentre os mecanismos para roteamento das requisições, pode-se citar [19]:

- *TCP gateway*: um *proxy* no despachador intermedia a comunicação entre clientes e servidores;

- *TCP splicing*: o despachador mantém uma conexão TCP permanente com cada servidor. Uma vez que a conexão entre um cliente e o despachador é estabelecida e uma conexão com um servidor é escolhida, as duas conexões são conectadas de forma que um pacote é encaminhado de um *endpoint* para outros sem atravessar a camada de transporte até a camada de aplicação [23];

- *Hand off*: uma vez que a conexão TCP com o cliente é estabelecida e um servidor é escolhido, o despachador transfere seu *endpoint* da conexão TCP para o servidor, que passa a comunicar-se diretamente com o cliente [69].

Liu e Lu [54] propõem uma solução para distribuição de carga na qual um servidor centralizado realiza o papel de redirecionador de requisições. Este servidor recebe as requisições de todos os clientes e, de acordo com a política de distribuição de carga, redireciona as requisições (via redirecionamento HTTP) para um servidor apropriado. Além de utilizar o estado dos servidores na política de distribuição, a solução de Liu e Lu utiliza um controle de admissão que dá prioridade para requisições com maior recompensa. Dentre os nós servidores que admitem receber a requisição, aquele que apresenta menor tempo de resposta é escolhido.

Xiong et al. [94] propõem uma solução baseada em múltiplos despachadores. Nesta solução, chamada de MDDR (*Multiple Dispatchers with Direct Routing*), vários despachadores compartilham o mesmo endereço e recebem todas as requisições. Cada um utiliza uma função sobre o número da porta do cliente para decidir se atende ou descarta a requisição. Uma desvantagem desta solução é que não pode ser aplicada em servidores distribuídos geograficamente.

Wang et al. [93] descrevem uma solução que utiliza o padrão OpenFlow para distribuir a carga entre os servidores web. Nesta solução, um distribuidor de carga centralizado instala regras do tipo *wildcards* nos *switches* para direcionar as requisições de grupos de clientes para diferentes servidores, de acordo com suas capacidades.

O trabalho [40] apresenta um levantamento de diversas técnicas para distribuição de carga via despachador. Apesar dessas técnicas serem mais apropriadas para distribuição de carga entre computadores de um aglomerado, muitas das técnicas têm aplicação também no caso de servidores geograficamente distribuídos.

Apesar de muitas das idéias e técnicas relacionadas às soluções baseadas em despachadores serem aplicáveis à distribuição de carga para servidores geograficamente distribuídos, estas soluções são mais apropriadas para ambientes de *clusters* e não foram abordadas nesta tese.

2.2 Distribuição de Carga em Outras Áreas

Cheung e Jacobsen [22] propõem uma solução para distribuição de carga para sistemas *publish/subscribe*. Nesta solução os clientes (*subscribers*) interagem com agentes (*brokers*) que executam políticas de distribuição de carga. Quando um agente detecta sobrecarga, ele negocia a transferência de carga com uma lista de outros agentes. Quando um outro agente aceita receber a carga excedente, os dois iniciam uma sessão de distribuição de carga. Então, o agente sobrecarregado seleciona um conjunto de assinantes e solicita que eles migrem para o outro agente. Um agente nunca se envolve em mais uma sessão de distribuição de carga simultaneamente.

Dobber et al. [33] comparam três soluções de distribuição de carga para grades computacionais. A primeira é uma solução simples que divide a carga igualmente entre os nós processadores. A segunda, chamada de *Dynamic Load Balancing* (DLB) utiliza informações históricas para estimar o tempo de resposta dos nós e tenta calcular uma distribuição ótima. A última, chamada *Job Replication* (JR), replica as tarefas em n diferentes nós e aceita a primeira resposta. Segundo os autores, DLB e JR apresentam melhores resultados que a distribuição equitativa. Eles propõem uma solução que combina as duas abordagens e alterna entre elas de acordo com determinadas métricas.

Quang et al. [92] apresentam uma solução de distribuição de carga para sistemas P2P no qual cada *peer* mantém um histograma com a carga global do sistema. O histograma armazena a carga média de grupos de nós sem sobreposição. Esse histograma é utilizado nas decisões da política de distribuição de carga e é mantido por meio da comunicação entre *peers* vizinhos que disseminam toda informação que acumulam.

Galloway et al. [38] propõem uma política de distribuição para arquiteturas em nuvem que considera o consumo de energia para distribuir máquinas virtuais entre as máquinas físicas da nuvem. Segundo os autores, esta política proporciona economia de energia quando comparada a uma política rotativa, como aquela utilizada no Eucalyptus, um conhecido software para fornecer infraestrutura como serviço (IaaS – *Infrastructure as a Service*).

Ren et al [76] apresentam uma solução para distribuição de carga para servidores replicados em uma nuvem. Nesta solução, a política de distribuição utiliza dados históricos de requisições e de capacidade dos servidores para estimar a carga dos servidores caso sejam selecionados para atender uma nova requisição. O servidor que apresentar a menor carga estimada é selecionado.

2.3 Replicação e Caching na Web

As soluções para replicação e *caching* de aplicações web podem influenciar diretamente a distribuição da carga. Por exemplo, em soluções que utilizam replicação parcial de dados, a carga gerada para manter os dados consistentes pode variar de uma réplica para outra. Esta seção apresenta diversos trabalhos que propõem soluções para replicação e *caching* de aplicações web com dados dinâmicos. Alguns desses trabalhos influenciaram o projeto e o desenvolvimento da plataforma de testes.

Groothuyse et al [42] propõem uma técnica de replicação parcial dos dados. Nesta técnica, as tabelas do banco de dados são agrupadas de acordo com a necessidade de *templates* de acesso. São os chamados *clusters* de tabelas. Múltiplos servidores armazenam réplicas dos *clusters* de tabelas e um roteador de consultas realiza a distribuição de carga entre os servidores. Apesar de todas as réplicas de tabelas terem que ser atualizadas a cada modificação, nem todos os servidores precisam executar todas elas. Uma desvantagem é que o sistema proposto não fornece apoio a transações, considerando cada acesso ao banco de dados como uma atividade separada.

Os trabalhos [95], [81] e [79] apresentam *middlewares* semelhantes para replicação ativa de serviços web SOAP. De forma geral, nas soluções apresentadas os clientes acessam o serviço web por meio de um conjunto de *proxies* que enviam as requisições para todas as réplicas do serviço de forma totalmente ordenada. Como todas as réplicas executam as operações na mesma ordem, a consistência dos dados é mantida.

Os trabalhos [36] e [68] apresentam *middlewares* para replicação passiva de serviços web SOAP. Nestas soluções, uma réplica primária processa todas as operações e reporta as atualizações para um conjunto de réplicas secundárias. Caso a réplica primária falhe, um detector de falhas determina outra réplica para assumir a função da réplica primária.

Amiri et al. [4] propõem um mecanismo para *caching* de dados dinâmicos para aplicações web. Na abordagem proposta, os servidores de borda (*edge servers*) são instrumentados com um mecanismo chamado *DBProxy*. Este mecanismo gerencia a conexão remota entre a aplicação web e o banco de dados principal (*back-end database*). O *DBProxy* intercepta chamadas SQL disparadas pela aplicação e determina se elas são satisfeitas pela *cache* local. Caso não sejam, as chamadas são encaminhadas ao banco de dados principal. Políticas de gerência da *cache* determinam se uma chamada externa é armazenada ou não. A *cache* é implementada como visões materializadas que correspondem a subconjuntos horizontais e verticais das tabelas originais.

Olston et al. [67] desenvolveram um sistema de *caching* via *proxy* escalável capaz de distribuir conteúdo dinâmico para um amplo número de usuários. Neste sistema, os usuários acessam as aplicações indiretamente, por meio de *proxys*. Todas as atualizações são encaminhadas para servidores de dados. Os servidores *proxy* notificam as atualizações uns aos outros e possuem um módulo de invalidação que remove os dados potencialmente inconsistentes da *cache*.

Tolia e Satyanarayanan [90] apresentam um *middleware* chamado Ganesh, cujo objetivo é diminuir o volume de dados transmitidos entre o banco de dados principal e os servidores de *cache*. Os dados são fragmentados e "resumidos" utilizando-se *hashing* (SHA-1).

Amza et al [5] propõem um sistema de *caching* transparente para conteúdo Web dinâmico. A solução utiliza um mecanismo de particionamento do esquema do banco de dados que permite a realização de consultas parciais. Os resultados das consultas parciais são utilizados para construir a resposta da consulta original. A consistência das *caches* é mantida por meio da invalidação de resultados de consultas prévias, via *multicast* (comunicação em grupo confiável).

Bouchenak et al [13] apresentam um sistema de *cache* para conteúdo Web dinâmico chamado AutoWebCache. Este sistema armazena as páginas geradas dinamicamente indexadas pela URI das requisições dos clientes, incluindo os parâmetros da requisição. Uma tabela mantém detalhes das consultas (somente leitura) que geraram as páginas em *cache*. Quando ocorre uma escrita, um mecanismo analisa as consultas afetadas pela atualização e invalida as páginas relacionadas em *cache*.

Fagni et al [35] propõem um mecanismo de *cache* para buscadores web chamado SDC (*Static Dynamic Cache*). Este mecanismo divide a *cache* em dois níveis, uma porção estática e uma porção dinâmica. Na porção estática, são armazenados os resultados das consultas mais frequentes, obtidas a partir de históricos do buscador. As demais consultas são armazenadas na porção dinâmica, utilizando alguma política de atualização, como LRU (*Least Recently Used*).

Em princípio, está fora do escopo deste trabalho estudar a relação entre os mecanismos de replicação e de distribuição de carga. Neste trabalho, supõe-se que a carga gerada para replicação dos dados e sua variação no tempo são sempre similares em todas as réplicas, e, portanto, não têm influência no balanceamento de carga. Além disso, supõe-se que os servidores web não hospedam outras aplicações além da aplicação replicada. Desta forma, considera-se que não há outras fontes de carga influenciando o desempenho do servidor. Esta suposição é plausível pois, mesmo em situações nas quais recursos de hardware tem que ser compartilhados por diversas aplicações, soluções de virtualização permitem isolar a quantidade de recursos alocados para uma aplicação específica.

2.4 Plataformas de Testes

Esta seção apresenta trabalhos relacionados a plataformas de testes para sistemas distribuídos. Hong et al [45] descrevem a implementação de uma plataforma de testes (6PlanetLab) implementado na China, utilizando 50 nós espalhados por diversas universidades. Apesar de utilizar o arcabouço do PlanetLab, a versão apresentada é construída sobre um *backbone* IPv6 e, segundo os autores, acrescenta aspectos que melhoram a robustez e o custo/eficiência do *testbed*. Dentre eles: (i) um mecanismo semi-automático de recuperação a falhas; (ii) um middleware Keep-Alive Middleware para verificação da conexão dos nós com a Internet; e (iii) compartilhamento do sistema operacional maximizado.

O artigo [75] descreve o Ultrascience Net (USN), uma plataforma de testes para aplicações científicas de larga-escala. O USN é voltado para aplicações que requerem alta largura de banda para suportar transferência de grandes quantidades de dados. A rede interliga Atlanta, Georgia, Chicago, Illinois, Seattle, Washington e Sunnyvale por meio de quatro links de 10 Gb/s dedicados.

A motivação para utilização de plataformas de testes para redes de larga escala é a liberdade de implantar e testar *hardware*, *middleware* e *software*. No artigo [55], Martin et al descrevem a implementação do Data Transatlantic Grid (DataTAG), um *testbed* implantado em 2002, interligando o CERN, na Suíça, e o StarLight, em Chicago. O objetivo principal do DataTAG é apoiar a pesquisa da comunidade de grades computacionais sobre questões impostas por aplicações envolvendo grandes quantidades de dados sobre redes gigabit transoceânicas.

Miyachi, Chinen e Shinoda [57] argumentam que *softwares* de simulação não são suficientemente flexíveis (esquema e linguagem de modelagem próprios) para refletir o que será o produto final. Além disso, as simulações consomem muito tempo de execução. Afirmam que simular a Internet é praticamente impossível, graças a sua escala, complexidade e natureza dinâmica. Neste artigo, os autores apresentam um sistema para plataformas de testes configuráveis, que utilizam nós reais. Neste sistema, que supõe que todos os recursos estão no mesmo *site*, os usuários criam topologias experimentais virtuais (VLAN) sem alterar as conexões físicas da rede. Para realização dos experimentos, utilizaram o StarBED [58, 2] para testar o sistema. O StarBED é um testbed com 680 PCs interligados por *switches*.

Sanghi et al [80] apresentam uma plataforma de testes específica para avaliação de estratégias de distribuição de carga. A plataforma permite testar dois tipos de estratégias: baseadas em DNS e em despachador. As informações de carga dos servidores são monitoradas e periodicamente são informadas ao despachador, que dependendo da política adotada, utiliza essas informações para encaminhar as requisições HTTP. O despachador reporta informações do seu *cluster* para o DNS. Este último (BIND modificado) envia o endereço IP do cliente para uma aplicação paralela que decide para qual *cluster* encaminhar o cliente. As políticas implementadas são: aleatória, rotativa, baseada na capacidade do *cluster* e baseada no servidor com menos carga.

Esta tese apresenta duas ferramentas para apoiar a análise e o desenvolvimento de soluções de distribuição de carga. A primeira é uma plataforma de testes (Capítulo 3) construída sobre uma implementação real de um serviço web distribuído. A segunda é um software de simulação baseado em modelo realístico de geração de carga web (Capítulo 5).

Capítulo 3

Lab4WS: A Testbed for Web Services

3.1 Introduction

The Web has become the universal support for applications. Increasingly, heavy loaded applications, that place extra demand on servers and network resources, are been deployed in clusters located at geographically distributed datacenters linked via the Internet. In each datacenter, a cluster of servers hosts an application replica, that can itself be locally replicated, for the sake of fault-tolerance and performance.

Many factors contribute for successful deployments in these environments, such as the adopted replication methods and load balancing mechanisms. Ideally, before the actual deployment of such a complex service, engineers would like to have a way to quantitatively analyze aspects such as performance and availability achieved by the different solutions available. The result is an increasing need for tools and techniques that assist developers in understanding the behavior of these systems.

The tools usually employed to fulfill this need can be classified into modeling (e.g., queue modeling), simulation, and benchmarking a real deployment. Simulation and modeling are certainly powerful assessment tools and allow researchers to evaluate a variety of application deployments in a flexible way. However, using these approaches, many system details need to be simplified, leading sometimes to less accurate evaluations. From another standpoint, experiments on the Internet are very hard to build and to maintain. Besides, in a real environment, it is difficult to isolate variables of interest and to reproduce experiments with exactly the same conditions.

In the academy, many researches have studied topics related to the availability and the performance of distributed and replicated Web services. However, due to the experimental barriers, most of the studies made so far focus themselves on simulations, or rely on small scale environments to validate their solutions. Analyzing the literature, we realized that researchers who needed to perform large numbers of experiments, involving different techniques or mechanisms, have adopted simulations. Real environments, that are less flexible to be adapted to different techniques and mechanisms, have been used to validate specific implementations that need more realistic environments. Performing experiments with the flexibility of simulations and the level of details of real environments is a great challenge.

In order to address this challenge, this paper explores the feasibility of a fourth approach: building a software equipment that allows engineers to experiment with smaller scale deployments that preserve the properties of their real counterpart systems. Therefore, we describe the Lab4WS (Lab for Web Services), a testbed designed to ease the evaluation of techniques and mechanisms for improving Web services availability and performance – mainly replication and load balancing solutions. The testbed provides experimental conditions that are similar to those found in real systems and the flexibility of simulations to be adapted to new techniques and mechanisms. Lab4WS adopts the TPC-W, a well accepted e-commerce benchmark, as the basis for its workload generation. Alongside TPC-W, Lab4WS offers tools that help developers to set up and execute experiments to test their solutions.

This paper contributes to the reduction of the difficulties associated with testbed-based assessment of web applications presenting the requirements we have identified for the development of the Lab4WS Testbed and the architecture we developed to accomplish these requirements. Moreover, we illustrate the Lab4WS use presenting a case study in which we evaluate four WAN load balancing mechanisms (DNS-based, server-based, dispatcher-based, and client-based) for a replicated Web service.

The remainder of this text is organized as follows. In Section 3.2 we report the requirements that guided us to the development of the testbed. Section 3.3 describes the solutions we have adopted to fulfill the requirements and the architecture of the testbed. Section 3.4 presents a case study in which we evaluate four classes of WAN load balancing mechanisms. Section 3.5 presents related works and Section 3.6 concludes the paper with our final comments and future work.

3.2 Testbed Requirements

The design and implementation of Lab4WS have been guided by an initial set of requirements extracted from our on experiments and the analysis of the facilities used by related research (Section 3.5). Here we summarize the requirements we have considered so far.

- **Benchmark Application**: The initial application available in Lab4WS should be both widely available—to allow the reproduction of experiments by others—and should represent a benchmark workload for Web applications.

- **Internet Latencies and Bandwidths**: The testbed should allow the emulation of the latencies and bandwidths usually found in the Internet. This requirement is important because it affects the behaviour of the application in many ways. For example, the obvious implication of selecting higher latencies and low bandwidths is a sharp drop in the response time perceived by the client. Higher latencies also affect the delay for the propagation of state updates among replicas. Latency variations affect how control information flows through the various load balancing agents, thus it is crucial to the comparison of competing load balancing policies.

- **DNS System**: The DNS System is very relevant for our studies. Its hierarchical design directly affects the distribution of the load between the replicas of the Web service and this distribution is essential to reproduce realistic workloads. In most of the load balancing solutions proposed in the literature, the DNS has been chosen as a load balancing agent (Section 3.5).

- **Realistic Load Generation**: The distribution of clients across Internet domains is one of the important factors that define the workload that reaches the application. Since the TTL adopted by a DNS server for the resolution of the name of an application affects all clients contained in the domain served by the server, larger domains will tend to generate heavier loads than smaller domains. Therefore, if larger domains are directed to a given application server and smaller ones are redirected to another one, the former will receive a heavier load than the later. An important requirement for the testbed is to allow a precise assessement of the impact on the workload of different DNS and client setups. In Lab4WS we should be able to condition workloads by changing the number of clients per domain, the DNS TTL times, and the workload (web interactions per second) generated by each client.

- **Adaptability to New Mechanisms**: Many replication and load balancing solutions have been proposed by different authors and have been validated in different environments, simulated or not (see Section 3.5). An important requirement of the testbed was the facility of deploying new solutions, in order to allow the comparison of the large variety of existing mechanisms, and to support the development of new ones.

- **Experiment Management Facilities**: The kind of experiments we intended to perform might involve tens of machines and require the configuration of tens of parameters. So, another testbed requirement was to facilitate the control of the experiment execution and to allow the user to vary the parameters of the evaluated mechanisms in a friendly way.

3.3 Lab4WS Testbed

The main features of Lab4WS are: (i) its flexibility for building and testing different solutions for web services; and (ii) provision of a realistic experimental environment for the quantitative assessement of the solutions. These two characteristics are a direct consequence of our decisions on how to select mechanisms that fulfill the requirements elicited in the previous section.

3.3.1 Addressing the Requirements

This section describes the software architecture and mechanisms that form the current implementation of Lab4WS.

- **Benchmark Application**: Our intention was to evaluate the Web system solutions under the load of well known Web applications, such as E-Commerce applications. Therefore, we decided to implement a SOAP Web service based on the TPC-W benchmark [91], a transactional benchmark for E-Commerce Web sites that is well accepted by the research community. This benchmark specifies a set of Web interactions that represent recurrent E-Commerce transactions.

- **Internet Latencies and Bandwidths**: In order to fulfill the latency and bandwidth requirement, the software architecture of the testbed has been implemented independently of the actual network upon which it will be deployed. The user is free to deploy the testbed in any network environment. The testbed can emulate the latencies and bandwidths usually found in LANs and WANs for a wide range of setups. Although emulation does not reproduce the actual variability of the Internet latencies, it emulates latencies of the same magnitude of those found in the Internet.

- **Realistic Load Generation**: The TPC-W benchmark also specifies three kinds of workloads that vary according to the percentage of read and write operations: (95reads/5writes, 80reads/20writes, and 50reads/50writes). This workloads are defined by tables that specify the probabilities with which a client chooses one of the available interactions to continue its Web session. The load generators of the Lab4WS testbed were implemented in compliance with the TPC-W specification, ensuring that the generated load emulates a very realistic load.

Besides the realistic load, that emulates real users, the testbed is also able to generate constant request rates, which are useful for evaluating the Web system throughput and response times. The constant request rate is generated through a pool of threads that emulate clients.

In order to distribute the load generation into different domains, we adopted the solution used by [24]. In this paper, the authors propose to divide clients into domains according to the Zipf's distribution, where the probability of a client to belong to the ith domain is proportional to $1/i^x$. This solution was motivated by previous works that demonstrate that if one ranks the popularity of client domains by the frequency of their accesses to a Web site, the size of each domain is a function with a big head and a very long tail. In the Lab4WS Testbed, the user can define the total number of clients and vary the skewness of their distribution changing the exponent x of the Zipf's distribution.

- **DNS System**: As well as the other Internet features, the DNS caching system can not be easily modeled. In our testbed, the effect of DNS caching is abstracted. All clients strictly respect the DNS TTL before asking for a new name resolution. A limitation of this abstraction is that the behavior of DNS-based mechanisms in our testbed can be considered as an optimistic behavior, since in real environments the non cooperative DNS servers can degradate the control of the DNS.

- **Adaptability to New Mechanisms**: In order to fulfill this requirement we designed the software architecture of the testbed in such a way that the basic Web system elements (Servers, Dispatchers, and Proxies) are extensible by components that implement replication or load balancing mechanisms. These components present a standard interface that the Web system elements are able to access. Thus, testbed users can implement new components that implement new solutions and deploy them on the testbed.

- **Experiment Management Facilities**: To deal with this requirement, we designed the testbed elements with an RMI (Remote Method Invocation) interface, which is accessed through a centralized experiment manager that allows the testbed user to control the entire system.

3.3.2 Lab4WS: Software Architecture

The software architecture of the testbed is composed of three systems: (i) the TPCW Web Service – a Web Service that simulates the workload of an e-commerce Web site; (ii) the Control Infrastructure – the set of elements that control the experiments; and (iii) the Functionality Components – that implement the replication and load balancing mechanisms that are tested. Figure 3.1 illustrates the testbed architecture. These parts are detailed in the following.

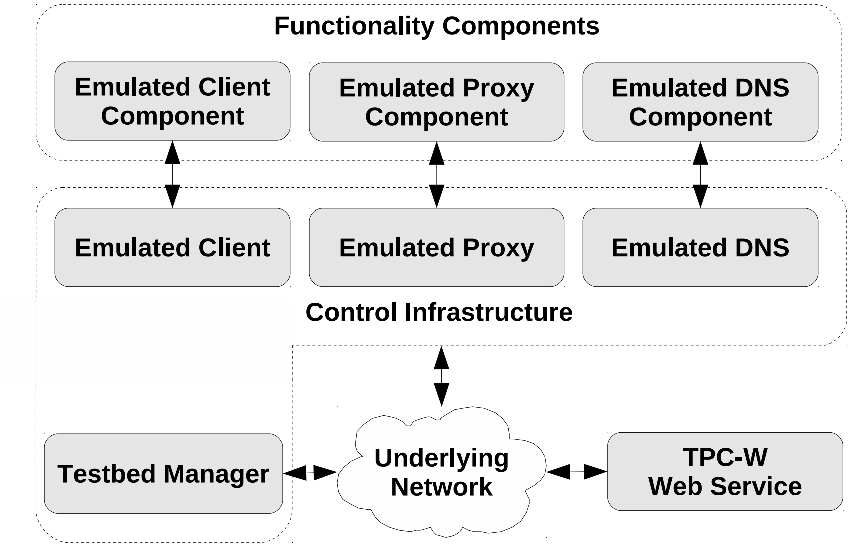


Figure 3.1: Testbed architecture.

TPCW Web Service

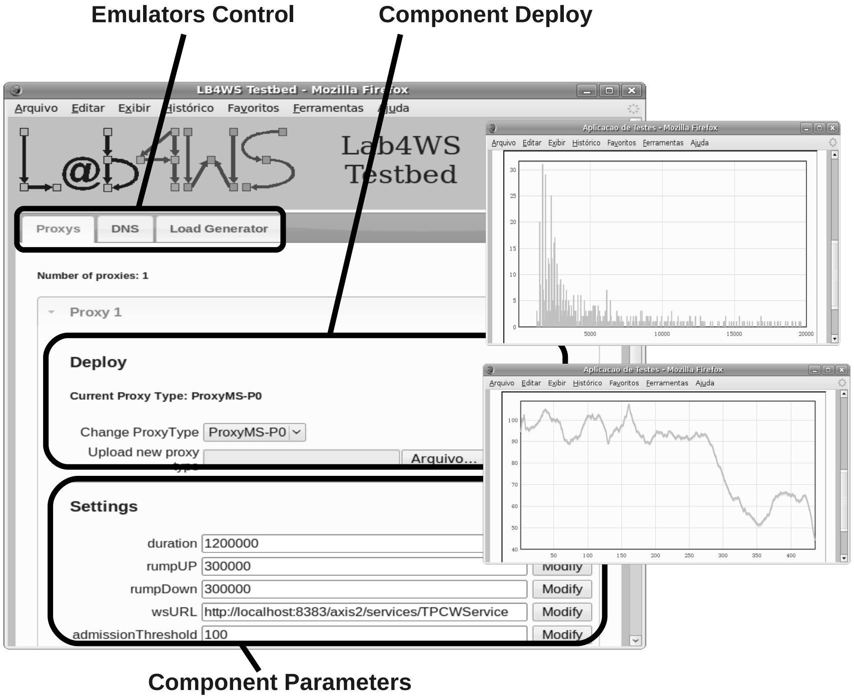
The TPCW Web Service is a Web Service based on the TPCW benchmark. We modified the implementation from the University of Wisconsin,[1] encapsulating the database operations as a Web service. The resulting service consists of a set of 20 operations that allow clients to search and order products.

Control Infrastructure

The Control Interface comprises a set of emulators that emulate devices in which mechanisms defined by the users may be deployed. Emulators are extended by Functionality Components, which implement the user mechanisms. There are three types of emulators: Emulated Clients, Emulated Proxies, and DNS Emulator. Emulated Clients are elements that generate load for the TPC-W Web Service. Emulated Proxies emulate devices that intercept messages between clients and service providers. Finally, the DNS Emulator is the element that emulates a DNS server.

The Control Infrastructure also comprises the Testbed Manager, which offers a Web interface that allows users to manage their experiments. The Testbed Manager controls all emulators through an RMI interface and provides the following functionalities: (i) experiment management; (ii) upload and deployment of Functionality Components; and (iii) on-line performance information. Figure 3.2 shows some snapshots of the Testbed Manager Web interface.

[1] http://mitglied.lycos.de/jankiefer/tpcw/index.html


Emulators Control
Component Deploy
LB4WS Testbed - Mozilla Firefox
Arquivo Editar Exibir Histórico Favoritos Ferramentas Ajuda
Lab4WS Testbed
Proxys
DNS
Load Generator
Aplicacao de Testes - Mozilla Firefox
Component Parameters

Number of proxies: 1

Proxy 1

Deploy

Current Proxy Type: ProxyMS-P0

Change ProxyType ProxyMS-P0

Upload new proxy type Arquivo...

Settings

duration 1200000

rumpUP 300000 Modify

rumpDown 300000 Modify

wsURL http://localhost:8383/axis2/services/TPCWService Modify

admissionThreshold 100 Modify

Figure 3.2: Manager Web interface

Functionality Components

The Functionality Components implement the mechanisms defined by the user that can be deployed on the Control Infrastructure emulators. There are three types of Functionality Components: (i) Emulated Client Components; (ii) Emulated Proxy Components; and (iii) DNS Emulators Components. As their names suggest, they aggregate functionalities to Emulated Clients, Emulated Proxies, and DNS Emulators, respectively. In our initial experiments, we have implemented 8 components:

Emulated Client Components

- DNS Cache Simulator: Simulates DNS caching at the client side.

- Client Best Last: A client-based load balancing mechanism. Clients store the response time of the last invocation of certain operation for all replicas and invoke the replica that presents the best last response time.

Emulated Proxy Components

- Passive Replication Middleware: This component allows the replication using a master-slave strategy. One of the replicas is chosen as the master replica, which processes all write and critical read operations.

- Active Replication Middleware: This component allows the replication using an active strategy. In this strategy, all write operations are broadcasted to all replicas and processed in the same order.

- HTTP Round Robin Redirector: This component allows the use of load balancing based on server-side HTTP redirection. The Emulated Proxies are placed in front of the Web servers and intercept all incoming requests. If the server utilization exceeds a given threshold, the proxy starts redirecting the requests to one of the other server replicas.

- HTTP Least Loaded Dispatcher: This component allows the use of a dispatcher-based load balancing mechanism. An Emulated Proxy is placed between the clients and the servers. It receives all requests and forwards them to the least loaded replica according to the server utilization information.

DNS Emulator Components

- DNS Round Robin: When a request for name resolution arrives, the DNS Emulator responds with the address of one of the replicas in a round robin way;

- DNS Least Loaded: The DNS Simulator receives utilization information from monitors of each Web Service replica. When a request for name resolution arrives, the DNS Simulator responds with the address of the less loaded replica.

3.3.3 Testbed Usability

The Lab4WS Testbed allows the assessment of replication and load balancing solutions in a variety of ways, such as: (i) comparison of replication solutions (e.g. active and passive approaches); (ii) comparison of load balancing mechanisms; (iii) evaluation of different settings of replication and load balancing solutions; and (iv) evaluation of replication or load balancing solutions under different conditions of network latencies and bandwidths.

3.4 Case Study: Comparing WAN Load Balancing Mechanisms in a SOA Scenario

The load balancing is a key issue for the performance of a replicated Web service. This problem consists on providing the dynamic binding between clients and Web service replicas, aiming to minimize the maximum load of a service replica, between all replicas, at a given time.

There are four classes of load balancing mechanisms for geographically distributed Web services in the literature [16]: DNS-based, Server-based, dispatcher-based and client-based. In the DNS-based solutions, the Authoritative DNS (ADNS) of the replicated Web server performs the role of the client request scheduler. When it receives a request for a URL resolution, it replies the IP address of one of the server nodes, according to some load distribution policy. In Server-based strategies, the load distribution policy runs in the server side. Any overloaded server replica can redirect requests to other replicas. In dispatcher-based strategies, a host placed between clients and server replicas receives all requests and forwards them to the appropriate replica. In the client-side strategies, the client runs the distribution load policy and decides to which server it sends the requests. All classes may vary in a number of parameters, such as: the distribution policy, the level of information used in decision making, and the periodicity of information sampling and dissemination.

In this section, we illustrate the use of the Lab4WS Testbed presenting a case study in which we compare the four classes of WAN load balancing mechanisms in a SOA (Service Oriented Architecture) scenario. Figure 3.3 illustrates the scenario we presuppose in this paper. In this scenario, a set of retailers firm partnerships with a big E-Commerce enterprise to outsource the application logic of their e-store Web sites. Each e-store is visited by a great number of end customers, and accesses the E-Commerce enterprise services via Web Services. The E-Commerce enterprise needs to distribute the load incoming from the e-stores among its geographically distributed replicas of servers.

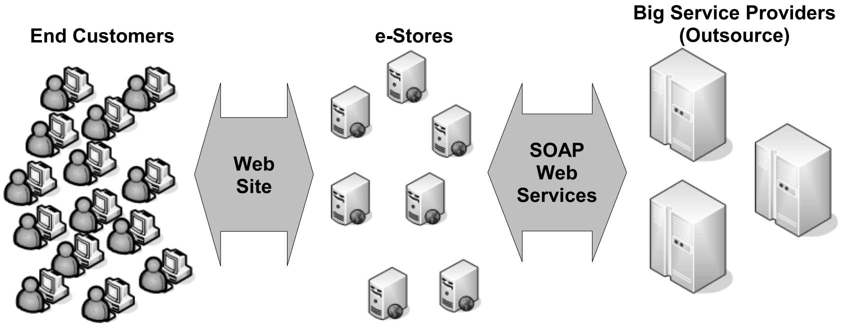


Figure 3.3: Presupposed scenario.

3.4.1 Methodology

For these experiments, our testbed was deployed on the Schooner[2] network testbed. Schooner is a user-configurable lab environment that allows users to model and emu-

[2] https://www.schooner.wail.wisc.edu/

late network topologies on a cluster, varying parameters such as latency and bandwidth. We ran the experiments using 4 TPC-W Web Service replicas and 9 machines running Emulated Clients with a load equivalent to 245 requests/second. The load was distributed among the Emulated Clients using the pure Zipf distribution ($x = 1.0$), that give us a high unbalanced workload. Figure 3.4 shows the topology of the emulated network, on which the testbed was deployed. All machines were Pentium 4, 3GHz, with 4GB of memory. A latency of 100 ms was introduced into the links between clients and proxies, to emulate a wide area network latency.

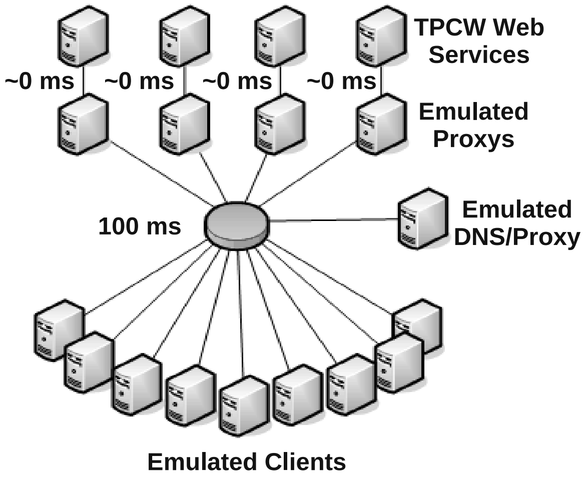


Figure 3.4: Experiment topology.

In order to evaluate the different load balancing mechanisms, we adopted three main metrics: the maximum system queue length, the response time observed by the clients, and the throughput, also observed by the clients. The maximum system queue length is the largest number of requests waiting to be responded on a service replica, observed at a given instant, among all service replicas. If one plots the cumulative frequency of the maximum system queue length, it is possible to infer the fraction of time in which at least one of the replicas had its queue length larger than a certain value.

If the incoming load exceeds the service replica capacity, the response rate becomes lower than the request rate and the incoming requests tend to accumulate at the server. Thus, the request queue grows. Since the requests take longer to be served, the response times observed by the client increase, and the system throughput goes down.

3.4.2 Experimental Results

In our experiments we compared four load balancing mechanisms (DNS Least Loaded, Dispatcher Least Loaded, Server-side HTTP Redirection using Round Robin, and Client

Best Last) with a uniform distribution, which represents the "ideal" load balancing. In these experiments we used the DNS TTL=60s and interval for load information dissemination of 1s (for Least Loaded mechanisms). In the HTTP Redirection mechanism, the adopted threshold was a queue length of 100. Load was generated using Zipf distribution with $x = 1.0$.

Figure 3.5 shows the cumulative frequency of the maximum system queue length for the experiments. The Server Redirection and Client Best Last mechanisms presented good performances, showing maximum system queue length close to the uniform distribution. The other two, which use the least loaded policy, were not effective, showing high maximum system queue lengths.

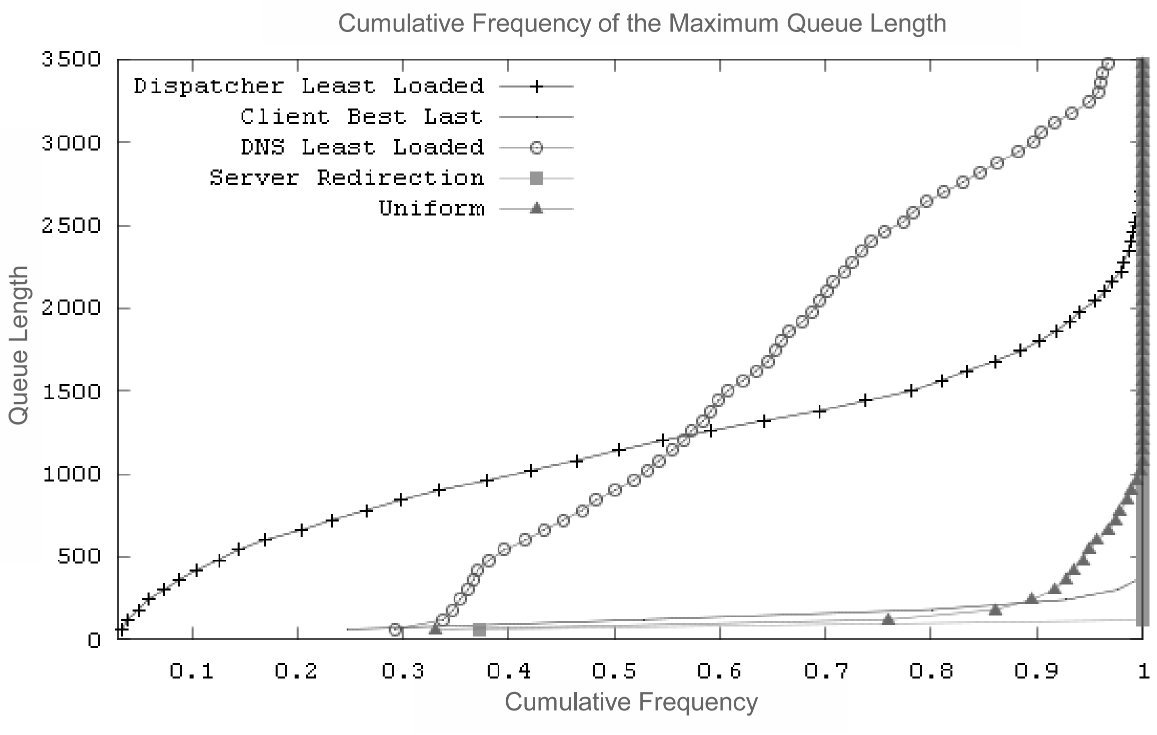


Figure 3.5: Cumulative frequency of the maximum system queue length.

The response times and throughputs obtained in experiments reflected the results of the maximum system queue length measurement. Figures 3.6 and 3.7 show the cumulative frequency of response times for *getBook* and *subjectSearch* operations, respectively. The former, returns the details of one item of the book store, and the later performs a search for items by the subject in the book store. The DNS Least Loaded and the Dispatcher Least Loaded mechanisms presented higher response times.

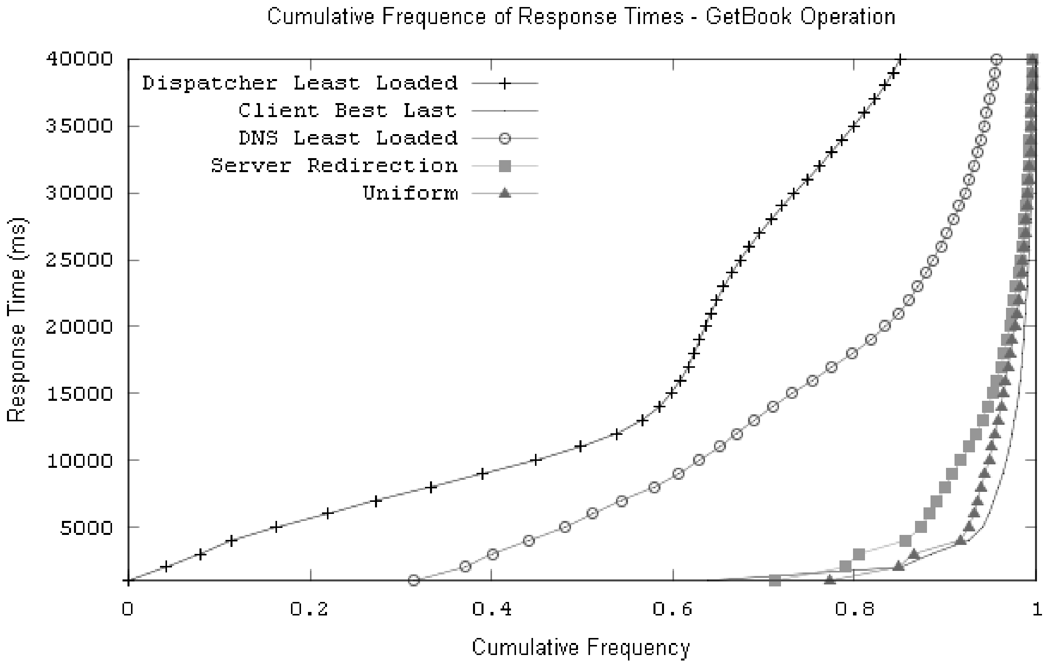


Figure 3.6: Cumulative frequency of response times (getBook operation).

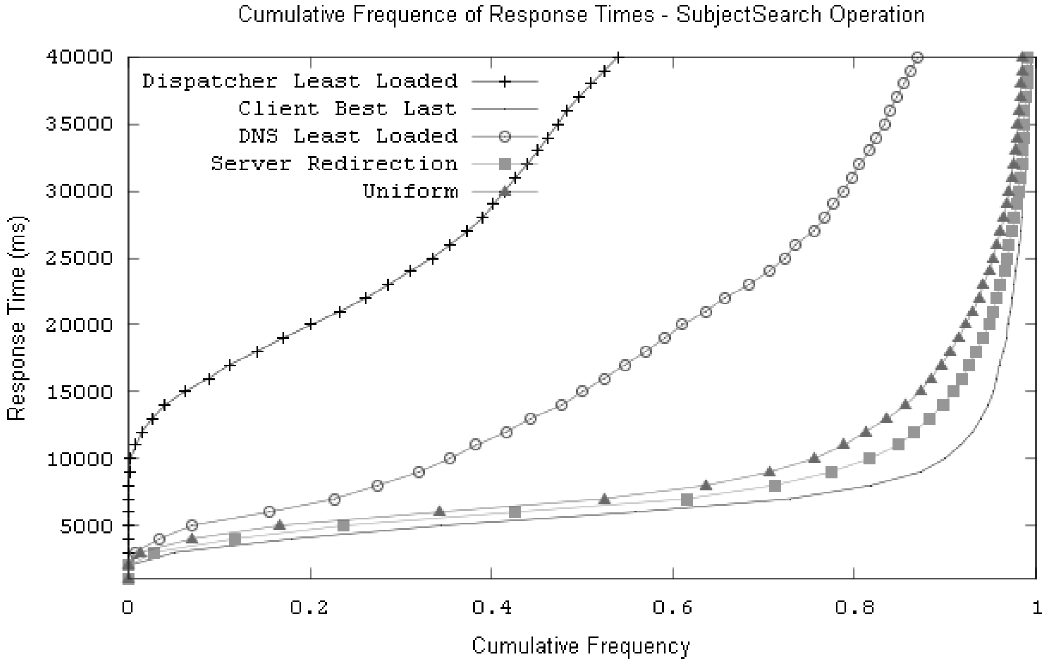


Figure 3.7: Cumulative frequency of response times (subjectSearch operation).

Figure 3.8 compares the throughput of the different mechanisms. Again, while the results obtained by the Server Redirection and the Client Best Last were close to the uniform distribution, the throughput of the DNS Least Loaded and the Dispatcher Least Loaded mechanisms presented a significant oscillation.

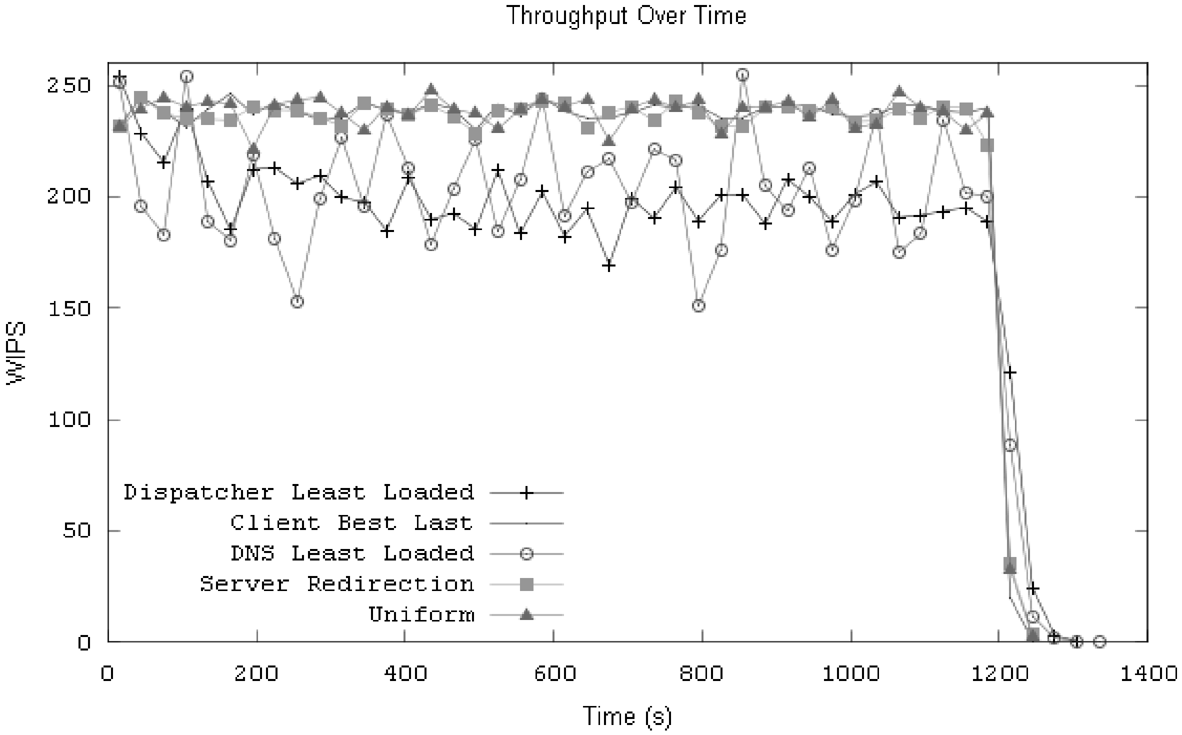


Figure 3.8: Throughput.

The experimental results show that, for the considered scenario and parameters settings, Client Best-Last and Server-side Redirection performed very well, presenting response times and throughput close to the ideal distribution. Otherwise, the two mechanisms that used the server utilization information (DNS-based Least Loaded and Dispatcher-based Least Loaded) presented the worst performances. We believe that a main reason for this result was the time interval between server utilization information updates, during which, the load balancing mechanism may select the same server replica using an outdated information. It is important to remember that the two mechanisms have intrinsic drawbacks: the low control of the DNS and the dispatcher bottleneck.

3.5 Related Work

Many works have addressed the problem of replication of Web services, e.g. [95, 79, 36, 68]. The papers [95], and [79] present frameworks based on active replication. In these frameworks, client requests are intercepted by a set of proxies that rely on a group communication middleware to deliver the requests to all replicas in the same order. Once all replicas process the requests in the same order, the consistence among them is preserved. The authors of [36] and [68] propose solutions based on passive replication. In these solutions, client requests are sent to a unique primary replica, which propagates the updates to the other replicas. If the primary replica crashes, one of the secondary ones becomes the new primary.

The load balancing for distributed Web servers has also been studied by several researchers, e.g. [24, 20, 17, 54, 26, 56]. The works [24] and [20] present DNS-based load balancing mechanisms. The considered policies include examples that use information of server utilization, load information from client domains, and the size of the requested documents, in decision making. The paper [17] explores the use of server-side redirection in the load balancing mechanisms. It presents a study about the combination of DNS-based policies and redirection schemes that uses centralized or distributed control on basis of global or local state information. Liu and Lu [54] propose a solution based on a centralized dispatcher that redirects client requests to the server with better response times. The papers [26] and [56] present comparisons of client-based load balancing mechanisms.

In general, most of works either have used simulations or have performed the experiments on LANs, not focusing on WAN issues. We have verified that the works that have addressed the replication of Web services did not focus on popular Web applications, such as E-Commerce applications. We also realized that most authors who have proposed or analyzed load balancing solutions, have focused on static content applications and have mainly focused their works on DNS-based, Server redirection, or client-based approaches.

Our testbed aims to facilitate the research on replicated/distributed Web services. Our users have the flexibility to adapt the scale of the experiments deploying the testbed on emulated networks or real WANs, according to their needs (and available resources). The architecture of our testbed was designed to accept new replication and/or load balancing solutions. The users can develop new Functionality Components (see Section 3.3.2) and deploy them on the testbed in a friendly way.

3.6 Conclusion

In this paper, we have presented Lab4WS a testbed for web services that contributes to the reduction of the difficulties associated with assessment of replication and load balancing solutions for Web applications. Indeed, Lab4WS can be used for the assessment of other Web application mechanisms, e.g., caching mechanisms. In contrast with previous work, our research focus on a benchmark Web application, an E-Commerce Web service.

To test the utility of our approach we present an experimental assessment of four classes of load balancing mechanisms. The results evidence the flexibility and usefulness of our testbed whose tools have eased the building, deployment, and execution of the load balancing experiments.

Future work includes the combined evaluation of different compositions of replication and load balancing mechanisms, the study of efficient solutions to measure and distribute load between *heterogeneous* Web server replicas, and the introduction of dependability measures.

Capítulo 4

DNS-based Load Balancing for Web Services

4.1 Introduction

With the increasing adoption of SOA (Service-Oriented Architecture), a new scenario arises, where highly accessed web applications are deployed as web services and clients are not web browsers accessing a web site, but other enterprises using the services of other providers.

This new kind of scenario may require higher levels of web service dependability because of the QoS contracted by the service consumers. Thus, providers replicate their applications in clusters geographically distributed linked via the Internet for the sake of fault-tolerance and performance. A key issue for good performance in these environments is the efficiency of the load balancing mechanism used to distribute client requests among the replicated services.

This work revisits the research on DNS-based load balancing mechanisms for geographically replicated web services. In this kind of load balancing solution, the Authoritative DNS (ADNS) of the distributed Web service performs the role of the client request scheduler, redirecting the clients to one of the server replicas, according to some load distribution policy. Differently from previous works, that considered the simple browser-server scenario, in our work we consider a SOA scenario.

It is known that large Internet corporations – e.g. Google [11] and Akamai [70, 88] – use DNS-based load balancing mechanisms. These mechanisms benefit from the existing DNS infrastructure, providing transparency for the Web clients. However, this kind of strategy has a main limitation: the low control of the ADNS over the load balancing caused by the DNS caching system, that prevents name resolution queries to reach the ADNS.

The main contribution of this paper is the proposal of a new DNS-based load balancing mechanism that uses client load information in order to better distribute the load among the replicated servers – the Current Relative Minimum Load (CRML). Besides, the mechanism reduces the negative effects of the DNS caching over the load balancing through the cooperation of the ADNS and the servers.

We also present the evaluation of our load balancing policy over an experimental testbed implemented on basis of the TPC-W [91], a well accepted E-Commerce benchmark. The experiments show that the CRML policy behaved as good as other policies in the scenario in which the ADNS had full control of the name resolution queries and behaved better than the others in a scenario where the ADNS had partial control.

The remainder of this text is organized as follows. Section 4.2 provides an overview of the DNS system and the DNS-based load balancing mechanisms. Section 4.3 presents related works. In Section 4.4 we describe our new load balancing mechanism. Section 4.5 presents the testbed used for the evaluation of our policy and Section 4.6 shows the experimental results. Section 4.7 concludes the paper with our final comments and future work.

4.2 Background

In the DNS-based load balancing mechanisms, the authoritative nameserver of the distributed Web server performs the role of the client request scheduler. When it receives a request for URL resolution, it replies the IP address of one of the server nodes, according to some load distribution policy.

The main advantage of this kind of load balancing mechanism is that it benefits from the existing DNS infrastructure. This makes these mechanisms immediately deployable in today's Internet [70].

Unfortunately, a limitation of DNS-based load balancing mechanisms is the weak control of the ADNS over the load balancing, caused by the DNS caching, that prevents a portion of DNS queries to reach the ADNS.

The simplest DNS-based load balancing mechanism is the *Round Robin* (RR) policy. In this policy, when a request for name resolution arrives at the ADNS, it responds with the address of the next replica of its list, in a rotative way. More sophisticated approaches apply information from server node utilization and/or client domain information to select a server replica.

4.3 Related Work

The use of information about server node utilization in DNS-based load balancing mechanisms is exemplified in [25, 96, 60]. In these works, an agent monitors the state of the servers and reports this information to the ADNS. When a name resolution query arrives, the ADNS uses the utilization information of the server nodes to assign one of the replicas to the client. Many kinds of information can be used in the ADNS decision, such as request queue length, CPU, network, or memory usage.

There are two types of client domain information that can be used in the load balancing: client proximity and client domain load. In the first case, the ADNS tries to assign the nearest server to the client [11, 70, 88]. A main concern in this kind of load balancing mechanism is to estimate the proximity between clients and servers. Since the ADNS does not have information about the client host, a solution for this problem is to assume that clients are located near to their local nameservers and estimate the distance between servers and local nameservers.

The capacity of estimating the load generated by client domains may be very useful for load balancing mechanisms. This information allows the ADNS to treat differently domains that generate high request rates (hot domains) from the others (cold domains). The works [25], [24], and [20] present promising results using information about client domain load for: (i) avoiding the assignment of hot domains to the same servers; (ii) estimating the real load of each server; and/or (iii) applying different TTLs for name resolutions addressed to hot and cold domains.

In this work, we present a new DNS-based load balancing mechanism that combines information from clients and servers to alleviate the effects of the DNS caching on the load balancing.

4.4 The CRML Policy

This section presents our proposed load balancing policy, the Current Relative Minimum Load (CRML).

4.4.1 Rationale

The idea for the algorithm was motivated by the analysis of the three load balancing policies described in [25, 24]. Here is a summary of them:

- ***Least Utilized Node*** **(LUN)**: the ADNS assigns a request to the less utilized node, based on the most recent server load information;

- ***Two Tier Round Robin*** **(RR2)**: the ADNS divides client domains into two groups, hot and cold domains. Hot domains are those that generate high number of requests. The policy applies the round robin strategy separately to each group, trying to avoid the assignment of requests proceeding from hot domains to the same server nodes;
- ***Minimum Residual Load*** **(MRL)**: the ADNS maintains a table containing all the assignments and their times of occurrence. Based on this table and on estimates of the request rate of each client domain, the policy calculates the load of each web server replica and assigns an incoming name resolution request to the least loaded one;

A deficiency of LUN is that the information used by the algorithm in decision making is often outdated. This happens because the algorithm considers only the last utilization information received from the servers, however, the state of the servers may have changed at the moment of a new assignment. In order to deal with this deficiency it should be necessary to make decisions based not only on the last utilization information but also considering the assignments performed after this information has arrived.

In our experiments, the RR2 policy presented a significative improvement in comparison to RR. This result shows us that to handle differently hot and cold domains is a good strategy. A deficiency of RR2 is that, even if a server is overloaded, the algorithm continues to assign new name resolution requests to that server because of the round robin strategy.

The MRL works quite fine because of its ability of estimating the total load of the servers based on previous name resolution assignments. Nevertheless, if the control of the ADNS decreases, by the reason of the caching of intermediary DNS servers, the MRL assignment table becomes incomplete and leads the ADNS to make wrong decisions. Moreover, if the number of clients is very high, it could be very expensive to maintain the assignment table.

4.4.2 Policy Description

In the CRML policy, we follow the hypothesis that the distribution of hot domains among the server replicas dictates the success of the load balancing mechanism. Thus, as well as RR2, the CRML policy divides the clients into two groups, hot and cold domains. Cold domains are distributed among the server replicas using the ordinary round robin strategy.

In order to better distribute the load generated by hot domains, the ADNS maintains an assignment table containing the assignments of servers to hot domains and their time of occurrence. Note that the set of assignments in this table is potentially incomplete, because many clients might have received name resolutions from intermediary DNS servers. Hence, the ADNS cooperates with the servers to compensate the effect of the DNS caching. Each server tracks the current set of hot domains from which it is receiving requests and report this information to the ADNS. Besides, servers also report estimates of the load (request rates) that hot domains are generating. Combining the information of the assignment table and the information proceeding from the servers, the ADNS can estimate the request rate each server is receiving from hot domains.

Let S be the set of web servers; let $l_i(a)$ be the load generated by the assignment a to the server i; let A_{Ki} be the set of assignments to the server i, known by the ADNS, and whose TTL has not expired; and let A_{Hi} be the assignments reported by the server i. When the ADNS receives a name resolution request from a hot client it computes:

$$CRML = \underset{i \in S}{min} \left\{ \sum_{a \in \{A_{Ki} \cup A_{Hi}\}} l_i(a) \right\}$$

and assigns it to the the corresponding server.

The efficiency of the CRML depends on how the set of assignments that the ADNS knows ($\{A_{Ki} \cup A_{Hi}\}$) is close to the reality. The ADNS knowledge is limited by the staleness of the information reported by the servers. The last sets of assignments (A_{Hi}) received from the servers may lack assignments that were established after the information was sent and may contain assignments that are not valid anymore.

In a certain limit, the use of the ADNS assignment table reduces the staleness of the server-side information. Another way to reduce the effects of stale server-side information is to exclude any assignment of the client that is requesting a name-resolution from the CRML calculation. Previous assignments related to this client is obviously not valid anymore, since it is requesting a new one.

Since the assignment table of the CRML does not store the state of all clients, only the states of hot clients, the cost for maintaining the table is smaller than using MRL.

4.4.3 Software Architecture

This section presents the software architecture that supports the proposed load balancing policy. The architecture is illustrated in Figure 4.1.

In this architecture, the server replicas are composed of two modules: (i) the web server module, that processes the incoming HTTP requests, and (ii) the monitor module, that collects information about hot client domains and reports it to the ADNS.

The ADNS is a DNS server extended with two modules, the domain mapper and the scheduler. The former is responsible for identifying if an incoming request comes from a hot or a cold domain. The latter uses the information reported by the server replicas and the information stored in the DNS assignment table to decide the better replica to which redirect a client.

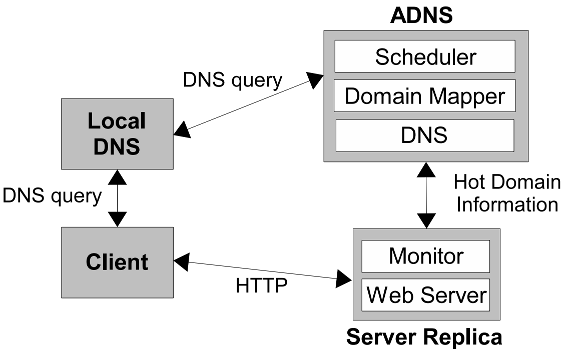


Figure 4.1: CRML Architecture.

4.5 Experimental Testbed

In order to evaluate our load balancing mechanism, we implemented an experimental testbed, the Lab4WS (Lab for Web Services). This section presents this testbed. More details about the Lab4WS Testbed can be found in [65].

The testbed includes the implementation of a SOAP Web service based on the TPC-W benchmark [91], a transactional benchmark for E-Commerce web sites that is well accepted by the research community. The service consists of a set of 20 operations that allow clients to search and buy products. A proxy placed in front of each service replica collects client information and reports to the DNS the list of hot clients that are accessing the replica and the estimative of load generated by those clients.

The load generation of our testbed is performed by a set of client emulators. Each load generator emulates the load of an E-Commerce Web site that serves an entire Web domain and generates requests to the TPC-W Web Service. The generated load follows the TPC-W specification, which specifies three kinds of workloads that vary according to the percentage of read and write operations. As in previous works – e.g. [24] – the load generation is divided between different clients according to the Zipf's distribution, where the probability of a client to belong to the ith domain is proportional to $1/i^x$. The testbed user can vary the skewness of the distribution changing the exponent x of the function. This solution was motivated by previous works that demonstrated that if one ranks the popularity of client domains by the frequency of their accesses to a Web site, the size of each domain is a function with a big head and a very long tail.

The testbed also contains a DNS emulator that materializes the ADNS of the web system. This emulator allows the testbed user to deploy new load balancing policies in a friendly way. The effect of the DNS caching is implemented as a mechanism that controls the percentage of name resolution requests that reach the ADNS. The testbed user can define this percentage. The client emulators randomly decide what name resolution requests are sent to the ADNS. When a name resolution request is not sent to the ADNS, the client emulator reuses the last name resolution it received, emulating a caching effect.

4.6 CRML Evaluation

In our experiments, we consider a scenario in which a set of retailers form partnerships with a large E-Commerce enterprise to outsource the application logic of their e-store Web sites. Each e-store is visited by a great number of end customers, and accesses the E-Commerce enterprise services via Web Services. The E-Commerce enterprise needs to distribute the load incoming from the e-stores among its geographically distributed replicas of servers.

4.6.1 Methodology

For these experiments, our testbed was deployed on the Emulab[1] network testbed. Emulab is a user-configurable lab environment that allows users to model and emulate network topologies on a cluster, varying parameters such as latency and bandwidth. We ran the experiments using 5 TPC-W Web Service replicas and 16 machines running Emulated Clients with a load equivalent to 300 requests/second (75 requests/second for each secondary replica). All machines were Pentium Xeon 64, 3GHz, with 2GB of memory. A latency of 100 ms was introduced into the links between the machines, to emulate a wide area network latency.

In order to evaluate the different load balancing mechanisms, we adopted two main metrics: the maximum system queue length and the response time observed by the clients. The maximum system queue length is the largest number of requests waiting to be answered on a service replica, observed at a given instant, among all service replicas.

If the incoming load exceeds the service replica capacity, the response rate becomes lower than the request rate and the incoming requests tend to accumulate at the server. Thus, the request queue grows. Since the requests take longer to be served, the response times observed by the client increase, and the system throughput goes down.

4.6.2 Experimental Results

We have evaluated the CRML policy on two scenarios. In the first, we compare five load balancing mechanisms (RR, LUN, MRL, RR2, and CRML) with a uniform distribution, which represents the “ideal” load balancing, assuming that the ADNS has total control over the name resolution requests. In the second scenario, we compare MRL, RR2, and

[1] https://www.emulab.net/

CRML, which presented the better performances in the first scenario, assuming that only 30% of the name resolution requests reach the ADNS.

In these experiments we used the DNS TTL=60s and interval for server information dissemination of 10s. The threshold adopted to identify hot clients was a load equal to 4 requests/second for MRL and CRML, and a load equal to 10 requests/second for RR2. Load was generated using Zipf distribution with $x = 1.0$.

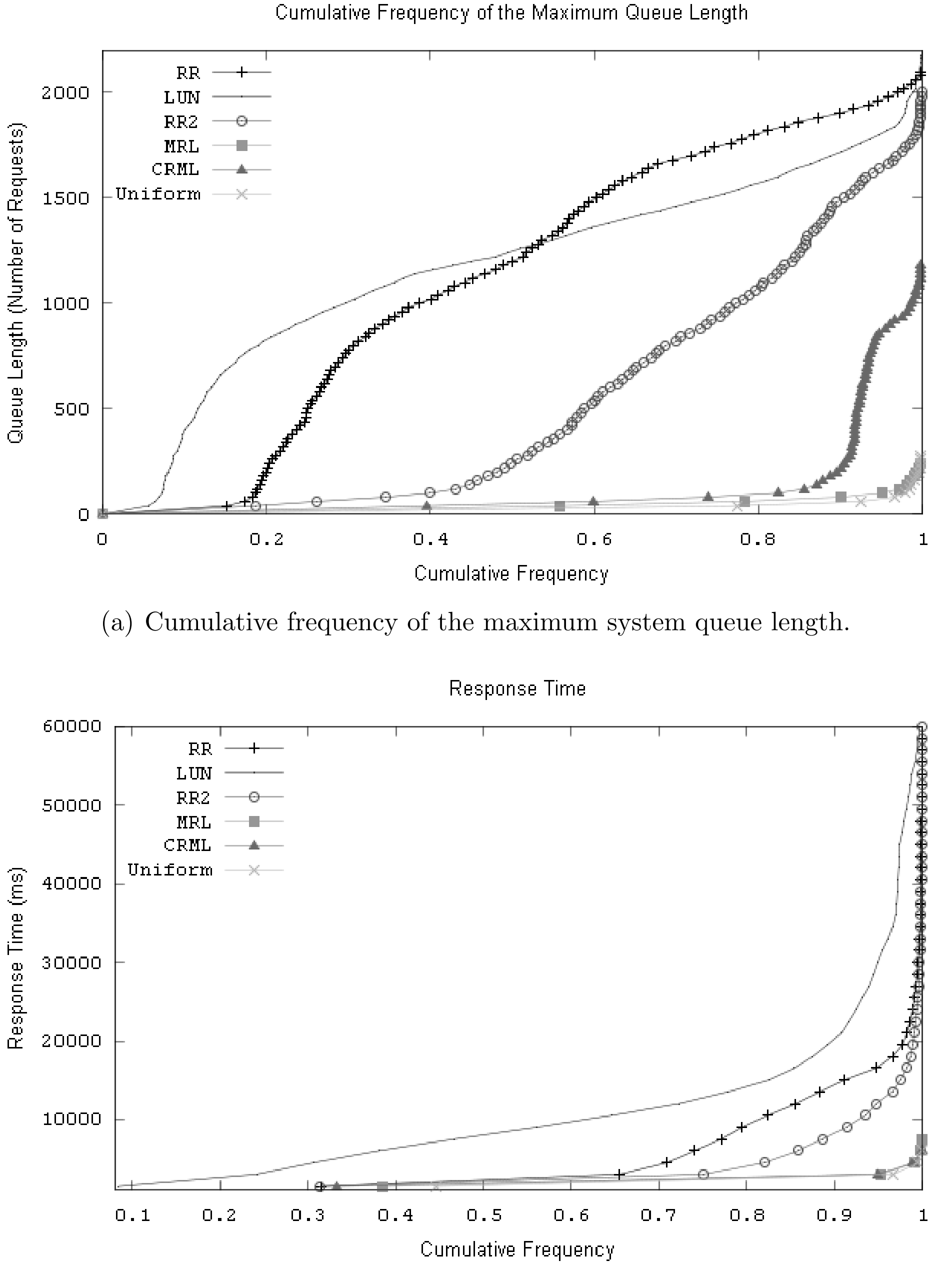
Figure 4.2(a) shows the cumulative frequency of the maximum system queue length for the first scenario. The MRL and CRML mechanisms presented good performances, showing maximum system queue lengths close to the uniform distribution. This result was expected because in this scenario, where the ADNS has full control over the name resolution requests, these two policies are able to compute the state of the servers with a high precision. The RR2 did not perform as good as CRML and MRL, but its performance was better than RR, showing that the use of client load information improved the round robin strategy. The LUN policy presented the worst performance, showing that the simple use of information about server utilization is not effective for load balancing.

The response times obtained in the experiments of the first scenario reflected the results of the maximum system queue length measurement. Figure 4.2(b) shows the cumulative frequency of response times for the *subjectSearch* operation, which performs a search for items by the subject in the book store. While, using CRML and MRL, 95% of the operation requests were answered in less then 3s, using RR2, about 25% of the requests were answered in more than 3s. Using RR, about 35% of the operation requests were answered in at least 3s, and, using LUN, more than 80% of the requests were answered in at least 3s.

The results obtained in the experiments of the second scenario are shown in Figure 4.3. As expected, due to the low control of the ADNS, the RR2 and MRL policies presented worse results than in the first scenario. In more than 50% of the time, these policies showed maximum system lengths larger than 1500 (Figure 4.3(a)). Differently from the others, CRML worked quite fine in the second scenario, presenting maximum system lengths smaller than 250 in 95% of the time.

The performance of the three policies is also perceived in the response time graph (Figure 4.3(b)). While the CRML presented response times lower than 3s in more than 95% of the requests, RR2 and MRL presented response times higher than 3s in 35% of the requests.

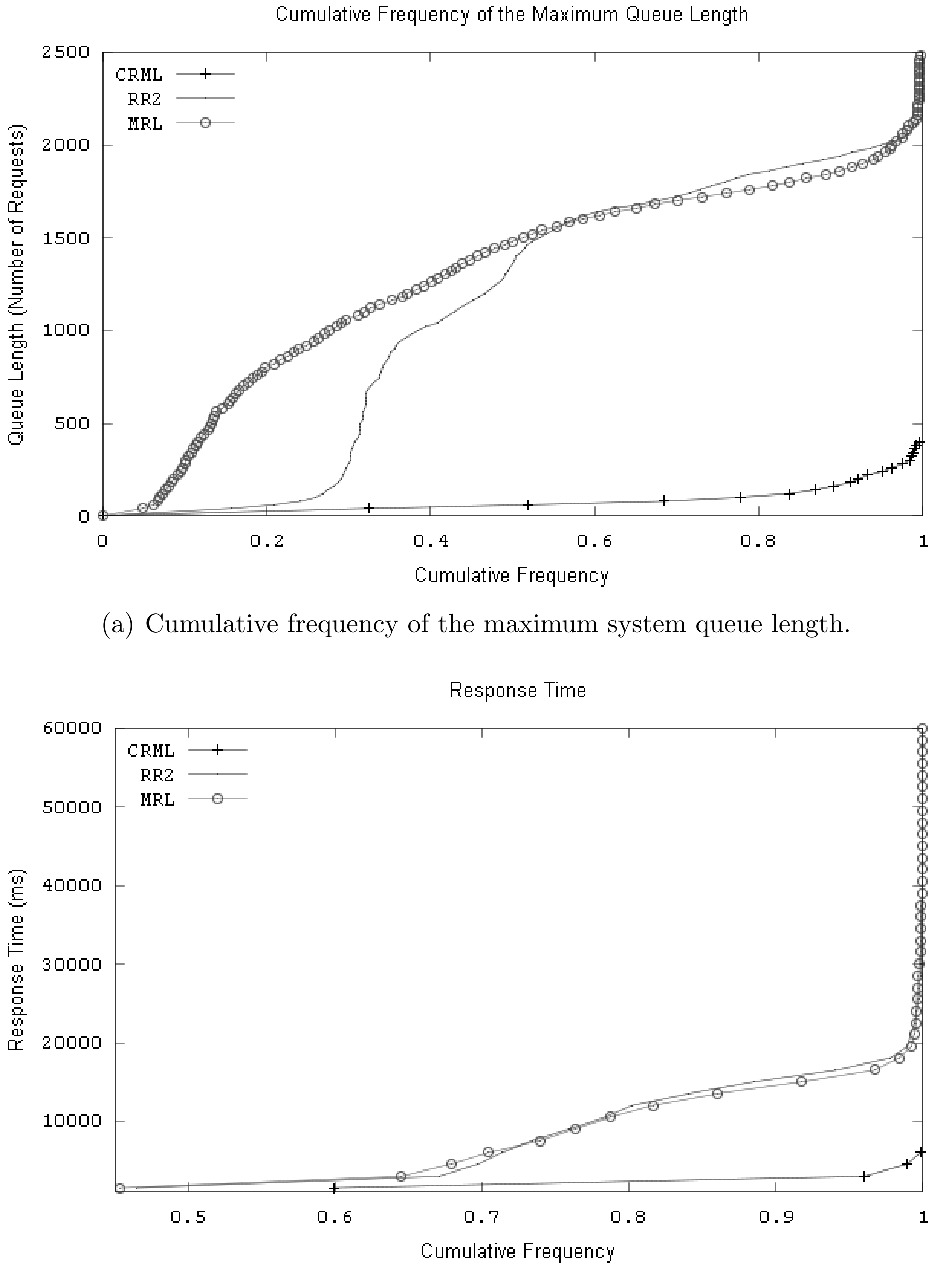
The experimental results show that, in the scenario where the ADNS had full control of the name resolution queries, CRML performed as well as the best policy (MRL) considered, presenting response times close to the ideal distribution. Moreover, CRML presented a better performance than RR2 and MRL in the scenario where the ADNS had partial control. This result shows that the cooperation between the ADNS and the servers compensated the effect of the DNS caching on the calculation of the server load states, allowing a good load balancing.



(a) Cumulative frequency of the maximum system queue length.

(b) Cumulative frequency of response times.

Figure 4.2: Experimental results: 100% of ADNS control.



(a) Cumulative frequency of the maximum system queue length.

(b) Cumulative frequency of response times.

Figure 4.3: Experimental results: 30% of ADNS control.

4.7 Conclusion

In this paper, we have presented a new DNS-based load balancing policy, the CRML. This policy combines information from clients and servers to alleviate the negative effect of the DNS caching over the load balancing mechanism.

The experimental results showed that our load balancing policy worked as good as other DNS-based load balancing policies in the scenario where the ADNS had full control over name resolution requests. Furthermore, CRML outperformed RR2 and MRL in the scenario where the ADNS control was limited.

Future work includes: (i) the evaluation of the sensitivity of our policy to different combinations of DNS TTLs, time interval of server information propagation, and the threshold for identifying hot domains; and (ii) the evaluation of the combination of CRML with server redirection mechanisms and dynamic TTL policies.

Capítulo 5

Improving the QoS of Web Services via Client-Based Load Distribution

5.1 Introduction

The replication of a web service over geographically distributed locations can improve the QoS perceived by its clients. An important issue in such a deployment is the efficiency of the policy applied to distribute client requests among the replicas.

Most of the load distribution solutions proposed so far are based on mechanisms that redirect client requests via DNS [25, 11, 70, 96, 60, 88, 62] or proxy web servers [17, 20, 74, 71]. Client-based solutions have been neglected because of their lack of transparency to the clients (a.k.a. web browsers). However, the web is becoming the universal support for applications that support previously unforeseen usage scenarios. For example, it is now common to find smart clients that can generate interactions in a programmatic way (e.g. SOA). In scenarios like this, where transparency is not an obstacle, client-based load distribution becomes an interesting alternative.

A main advantage of this kind of solution is the fact that the client can monitor end-to-end network latencies (response times) in a better way than the DNS and/or web servers. In some cases, a mechanism for client-side server selection may be the only option available. For example, a client may want to distribute their requests among a set of equivalent web services that do not belong to the same provider. In this case, server-side solutions are not applicable.

The client-side server selection techniques found in the literature can be divided into two groups: those that equitably distribute the requests among the distributed web services, and those that try to choose the best replica to which a client should send all requests. In the latter case, the focus has been the criteria used for decision making (e.g. latency, bandwidth, and best response time), and the way the criteria are applied (e.g.

using historical data and/or dynamic probing of servers).

Although previous work has studied the efficiency of the server selection mechanisms for a single client, they do not have assessed the effect of these mechanisms for the whole system. In this work, we have simulated scenarios where several clients, generating different workloads, access replicas of a web service distributed world wide. In these simulations, we assessed server selection policies that are representative examples of the two groups of solutions. Our experiments indicate that the two types of solutions led the system to load-unbalanced states and, consequently, to the worsening of the response times perceived by the clients.

With respect to this problem, we propose a new approach for client-based server selection that adaptively assigns different selection probabilities to each server regarding network latencies and end-to-end response times. Our results show that the proposed strategy can achieve better response times than algorithms that eagerly try to choose the best replica for each client.

The main contributions of this paper are: (i) the evaluation of client-based server selection policies in scenarios where several clients use the same policy; and (ii) the proposal of a new solution that outperforms existing ones by dynamically adapting the fraction of load each client submits to each server.

The remaining of the text is organized as follows. Section 5.2 discusses related work. Section 5.3 presents our policy. Section 5.4 describes our simulations and Section 5.5 shows the results. Finally, in Section 5.6 we present our final remarks and future work.

5.2 Related Work

Dikes et al. [34] present a comparison of 6 client-side server selection policies for the retrieval of objects via the Internet: **Random**, **Latency**, **BW**, **Probe**, **ProbeBW**, and **ProbeBW2**. **Random** selects a server randomly (equitably distributes the load). **Latency** and **BW** select the server that offers the best historical network latency and bandwidth, respectively. **Probe** probes the network latency of all servers (via ping) and selects the first to reply. **ProbeBW** considers only n servers with the fastest historical bandwidths and applies Probe to them. **ProbeBW2** probes the network latency of all servers and applies BW to the first n servers to reply. In summary, the results showed that the policies based on dynamic network probes presented best service response times. A similar result was presented by Conti, Gregori, and Panzieri in [28].

In the papers [26] and [27], Conti, Gregori and Lapenna compare a probe-based policy to a parallel solution. In the parallel solution, the total amount of data to be retrieved is divided in equal fixed-size blocks. Then, one block of data is requested to each service replica. The authors argue that this strategy can be useful when: (i) the objects to be downloaded are large; (ii) the client operations are read-only; (iii) the client wants better throughput; and (iv) the performance of the service replicas is not the bottleneck of the system.

Mendonça et al. [56] compared 5 client-side server selection policies for SOAP web services: **Random**, **HTTPPing**, **Parallel Invocation**, **Best Last**, and **Best Median**. Random and HTTPing are equivalent to Random and Probe from [34]. In Parallel Invocation, the client requests all servers in parallel and waits for the fastest response. Best Last selects the server with the best recent response time. Best Median selects the server with the best median response time. According to the authors, Best Last and Best Median are the best choice in most of cases.

The main difference between our work and the previous ones is that we are concerned with the effect of the server selection policy in scenarios where several clients use the same policy. We show that representative examples of the policies mentioned before present better or worse performances depending on the system condition. Therefore, we propose a new adaptive solution that outperforms the others by producing best response times.

5.3 Adaptive Server Selection

The server selection problem can be defined as follows. Let P_i be the probability of a client to send a request to the server i, with $1 \leq i \leq n$ and $\sum_{i=1}^{n} P_i = 1$. The problem is to find P_i that offers the best QoS. As we can see, all solutions mentioned before belong to one of the two types:

1. $P_i = 1/n$: Equitably distributes the load among n servers;

2. $\begin{cases} P_i = 1, \text{ if } i = k \\ P_i = 0, \text{ if } i \neq k \end{cases}$: Where k is selected according to some policy criterion (latency, bandwidth, response times, etc.).

Examples of server selection policies of the first type are: Round Robin, Random, and Parallel Invocation. The positive points of this type of policies are: (i) they are simple and do not require the client to gather server information; and (ii) they distribute client requests among all servers, thus reducing the possibility of server overload. Figure 5.1 (i) shows a scenario that is favorable to these policies. In this scenario, the equitable distribution helps not to overload any server. A negative point is that, since these policies do not consider any server information (e.g. latency, response times), they send requests to “worst” servers with the same probability as they send them to the “best” servers. Figure 5.1 (ii) illustrates an unfavorable scenario, where servers 3 and 4 do not have enough capacity to serve the incoming load and, thus, become overloaded.

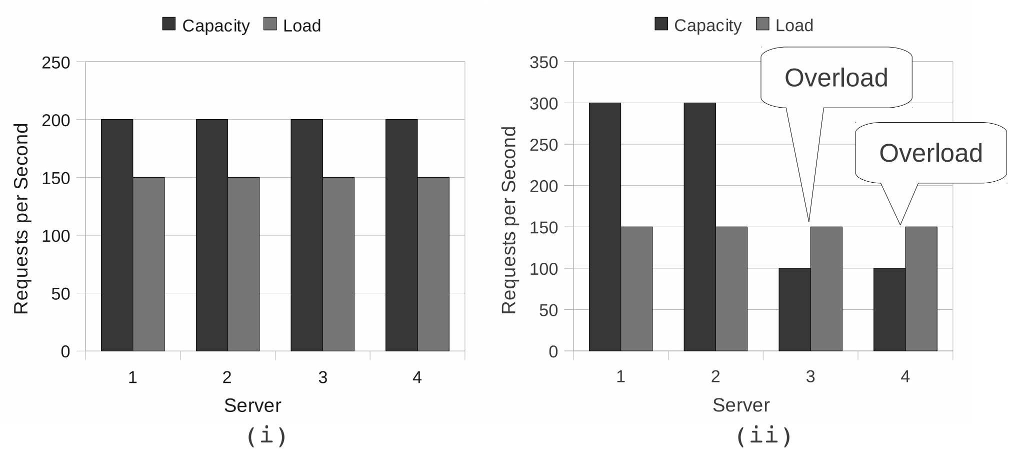


Figure 5.1: Equitably distribution server selection. (i) Favorable scenario. (ii) Unfavourable scenario.

The second type includes those server selection policies that use some criterion to select the best server. The advantage of this kind of solution is that it allows the client to perceive changes in the responsiveness of the selected server and react to these changes. These polices can offer good performance if the aggregated workload assigned to the "best server" does not exceed its capacity. Otherwise, the clients may start chasing for the best server and this leads another server to become overloaded. This situation is illustrated in Figure 5.2. In a time $t1$ (Figure 5.2(i)) , clients select server 1, that becomes overloaded. Once the clients perceive the changes in the responsiveness of server 1, they select server 2 (Figure 5.2(ii)), and then, it also becomes overloaded.

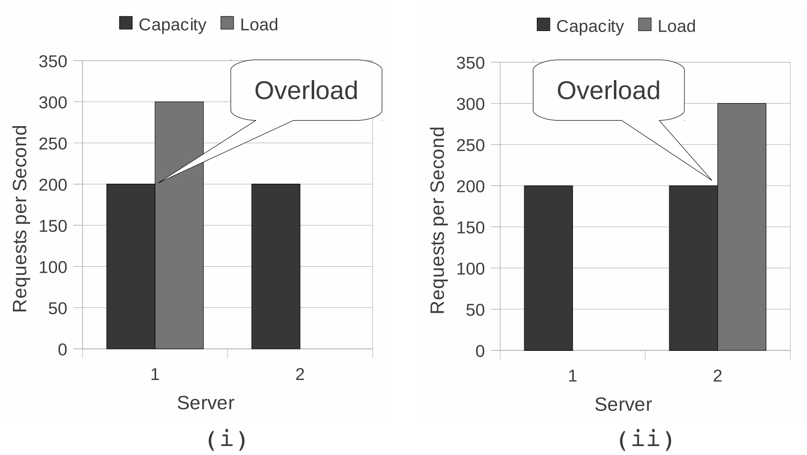


Figure 5.2: Best server. (i) t1: best server overloaded. (ii) t2: Chase for the best server effect.

Considering the advantages and problems of these two types of solution, we propose a new approach that dynamically adjusts Pi in order to minimize response times by avoiding the overloading of the servers. Our hypothesis is simple: if clients cooperatively balance the load of the system by distributing requests in a smart way, all of them can benefit from that.

In our solution, as well as in the Probe policy ([34]), the clients probe the network latencies of the servers in a predefined time interval. Each client maintains a list of servers that is sorted according to their network latencies. Here, we assume that, in normal conditions, the response times of nearby servers tend to be better than the response times of farther ones. The main idea of our solution is to find the highest selection probability the client can assign to each server, giving higher priorities to the nearest ones, without overloading them. In order to accomplish this objective, we propose the heuristic described in the following.

The pseudo-code in Figure 5.3 describes our heuristic. At the beginning of its execution, a client sets $Pi = 1/n$, as in the Random policy. It sends requests to the servers according to P_i and maintains a sliding-window mean of the response times of all servers (MRT) updated (lines 11 to 17). Another process monitors MRT (lines 20 to 22). Whenever the mean of server i (MRT_i) does not exceeds MRT_j, $i+1 \leq j \leq n$, the probability to select the server i is increased in a predefined amount INC (lines 27 to 30). In this case, INC is subtracted from P_j in a direct proportion to MRT_j. If $MRT_i > MRT_j$ or if a request to server i fails, then the client assumes that server i is overloaded and performs a contention action. It decreases P_i in a predefined amount DEC, which is added to P_j in an inverse proportion to MRT_j (lines 32 to 35). DEC should be very much higher than INC, in order to grant that, once a problem in the responsiveness of server i is detected, the aggregated load sent to server i is reduced to less than its capacity.

It is important to note that, in our approach, clients cooperate to maintain the load balancing of the system in a very indirect way. There are no interactions among clients or between clients and servers to exchange load and/or state information. The only information each client has, as in the other solutions, is the response times produced by the servers. Using this information, a client can avoid overloading the "best" servers by willingly sending fractions of its requests to other servers. This happens in isolation from the other clients. However, since all clients perform in the same way, the overall system load stands balanced. Considering that the clients send as much requests as possible to better servers and the servers are not overloaded, the system can provide very good response times.

```
Definitions:
S: set of servers, ordered by network latencies
Si: ith server
Pi: probability of selecting the server i
MRTi: mean response time of the server i
INC: probability increment
DEC: probability decrement
t_UPDATE: time between probability updates

#CLIENT
On each request:
  SELECT server Sk, according to Pi (1 <= i <= n)
  status <- SEND request to Sk
IF status == SUCCESS
  Update MRTk
ELSE
  Alarm[k] <- TRUE

#MONITOR
Continuously:
  IF MRTi (1 <= i <= n-1) > MRTj (i+1 <= j <= n)
    Alarm[i] <- TRUE

#PROBABILITY UPDATE
On each t_UPDATE seconds:
  FOR i<-1:(n-1)
    IF alarm[i] == FALSE
      Pi <- Pi + INC
      Subtract INC from Pj (i+1 <= j <= n)
                   in direct proportion to MRTj
    ELSE
      Pi <- Pi - DEC
      Add DEC to Pj (i+1 <= j <= n)
              in inverse proportion to MRTj
      Alarm[i] <- FALSE
```

Figure 5.3: Heuristic for adaptive server selection probabilities.

5.4 Methodology

In order to evaluate our solution, we have implemented a simulator using the CSIM for Java[1], a discrete event simulator framework. In the following, we describe how we have simulated the web services and their clients, the topology of the simulations, and the configuration parameters.

5.4.1 Load Generation and Web Servers

We used the PackMime Internet traffic model [15, 1] to generate HTTP traffic in our simulations. PackMime has been obtained from a large-scale empirical study of real web traffic and has been implemented in the *ns-2*[2], a well known network simulator. In order to use the model in our simulations, we have implemented a Java version of the PackMime.

PackMime allows the generation of both HTTP/1.0 and HTTP/1.1 traffic. The intensity of the traffic is controlled by the rate parameter, which is the average number of new connections started per second. The implementation provides a set of random variable generators that drive the traffic generation. Each random variable follows a specific distribution. The distribution families and the parameters used in PackMime are described in [15]. In our simulations, we used the following random variables:

- **PackMimeHTTPFlowArrive**: interarrival time between consecutive connections;
- **PackMimeHTTPNumberPages**: number of pages requested in the same connection (if using HTTP/1.1);
- **PackMimeHTTPObjsPerPage**: number of objects embedded in a page;
- **PackMimeHTTPFileSize**: sizes of files (pages and objects);
- **PackMimeHTTPTimeBtwnPages**: gap between page requests;
- **PackMimeHTTPTimeBtwnObjs**: gap between object requests;

We assumed that each geographically distributed replica of the web server is composed of a cluster of servers. Each server is simulated as a queueing system with fixed service time of 10 ms. Thus, a cluster with k servers provides a capacity of k*100 requests/second. The request interarrival time distribution is defined by the PackMime model [15, 1]. Figure 5.4 illustrates the workload generating 210 connections per second. Note that,

[1] http://www.mesquite.com/

[2] http://www.isi.edu/nsnam/ns/

since each connection generates more than one HTTP request, the number of requests per second reach peaks of 380. It is important to note that, although the service time is fixed, the response time perceived by the clients also includes the server queue time. Therefore, different loads do affect the entire response time.

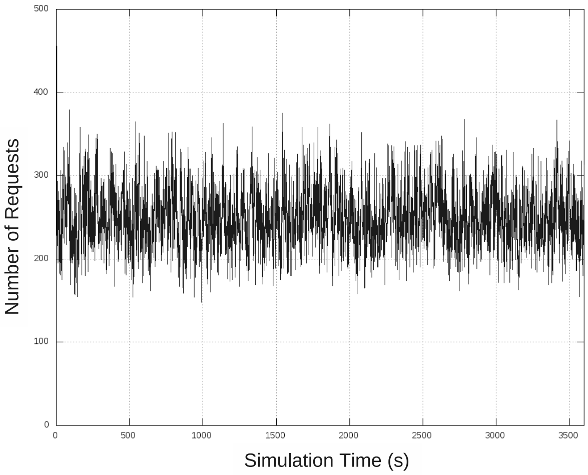


Figure 5.4: Example of workload generated by Packmime.

In order to divide the load between different clients, we adopted a solution similar to the one proposed by [24]. In this paper, the authors propose to divide clients (in this case, end customers) into domains according to the Zipf's distribution, where the probability of a client to belong to the ith domain is proportional to $1/i^x$. This solution was motivated by previous work that demonstrate that if one ranks the popularity of client domains by the frequency of their accesses to a Web site, the size of each domain is a function with a large head and a very long tail. In our simulation, the total aggregate load is divided between the clients according to the Zipf's distribution. We can also vary the skewness of the distribution by changing the exponent x of the function.

5.4.2 Internet Latencies

We have considered a scenario with six replicas of the web server that are world wide distributed (Figure 5.5): one in South America (S1), one in North America (N1), two in Europe (E1 and E2), and two in Asia (A1 and A2).

We used the average of the latencies (ping RTT/2) measured on real hosts of PlanetLab[3] in Brazil, USA, Belgium, Austria, Japan, and China, to simulate the latencies among the replicated web servers. Table 5.1 shows the measured RTTs. We also consider

[3] http://www.planet-lab.org

Table 5.1: Table of latencies among the replicas (in milliseconds).

	S1	N1	E1	E2	A1
N1	89				
E1	138	48			
E2	140	58	18		
A1	193	109	151	162	
A2	272	156	114	122	68

that each replica serves a region and that the latency between a replica and a client of its region is 10ms.

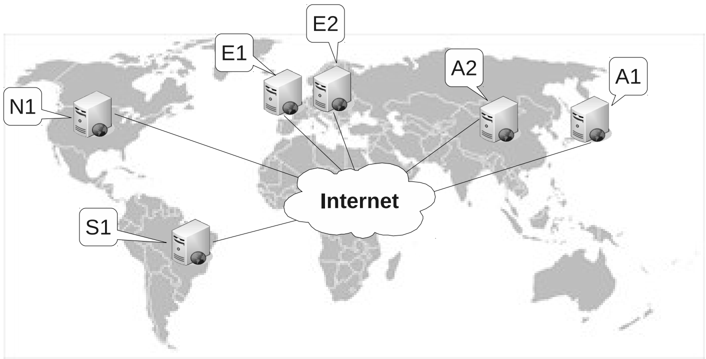


Figure 5.5: Web Server Replicas.

Although we used fixed latencies in our simulations, their magnitudes are real and we believe that small variations that do not affect the magnitude of the latencies would not affect our results. Since we consider the entire response time (network + service) to make decisions in the policies, from the client's perspective, it does not matter if the differences in response time are caused by network latency variations or due to variable server workload. Thus, we believe that, varying network latencies would not change the overall results.

In order to consider the latency of the TCP protocol, we adopted the analytic model proposed by Cardwell et al. ([18]). This work extended previous models for TCP steady-state by deriving models for two other aspects that can dominate TCP latency: the connection establishment three-way handshake and the TCP slow start [77]. Therefore, the model proposed by Cardwell et al. can predict the performance of both short and long TCP flows under varying rates of packet loss.

In practice, the TCP model estimates the time needed to transfer data from one endpoint to another in terms of: (i) the size of the data to be transferred; (ii) the network latency between the two endpoints; and (iii) the packet loss rate probability.

5.4.3 Configuration

We compared our solution (AD) with two other server selection policies:

- Round Robin (RR): Each client sends requests to all servers in a rotative way;
- Best Server (BS): Each client uses RR to probe all servers. The server that presents the best mean response time is selected. Next, the client keeps sending all requests to the selected server until its mean response time exceeds the mean response time of other server. In this case, the client starts probing again, in order to avoid using out-of-date mean response times.

In order to present the flexibility of our solution, we performed our experiments considering two scenarios, one that favors BS and another that favors RR. In the first, the total capacity of the servers was set to 1200 requests per second (rps) divided among the servers as follows: S1 = 100 rps, N1 = 300 rps, E1 = 200 rps, E2 = 300 rps, A1 = 200 rps, and A2 = 100 rps. The clients were configured to generate approximately 72% of the total capacity. In the second scenario, the total capacity was divided equitably among the servers and the aggregated load was set to approximately 90% of the total capacity.

The parameters used in the heuristic are shown in Table 5.2. Note that, since *DEC* needs to be sufficiently large to alleviate the load of an overloaded server, we adopted a *DEC* proportional to P_i.

Table 5.2: AD Simulation Parameters.

Parameter	**Value**	**Description**
INC	0.01	Probability increment.
DEC	$0.3P_i$	Probability decrement.
t_UPDATE	1s	Time between probability updates.
WSIZE	30 requests	Window size for response time slide mean.

5.5 Results

Figure 5.6 presents the mean response times for Scenario 1. This scenario is unfavorable for RR (Round Robin), that does not have any information about server capacity. Although the total capacity of the system is much higher than the load, the server capacities are heterogeneous. Two servers (A1 and E2) have capacities lower than the equitable fraction of load distributed by RR. This leads A1 and E2 to become overloaded, making their queues grow faster than others queues and, consequently, affecting the system response times.

The bad performance of RR is also the result of its lack of adaptability to the overloading of the servers. Also, RR does not benefit from the small network latencies offered by nearby servers, because it blindly distributes the requests among all servers.

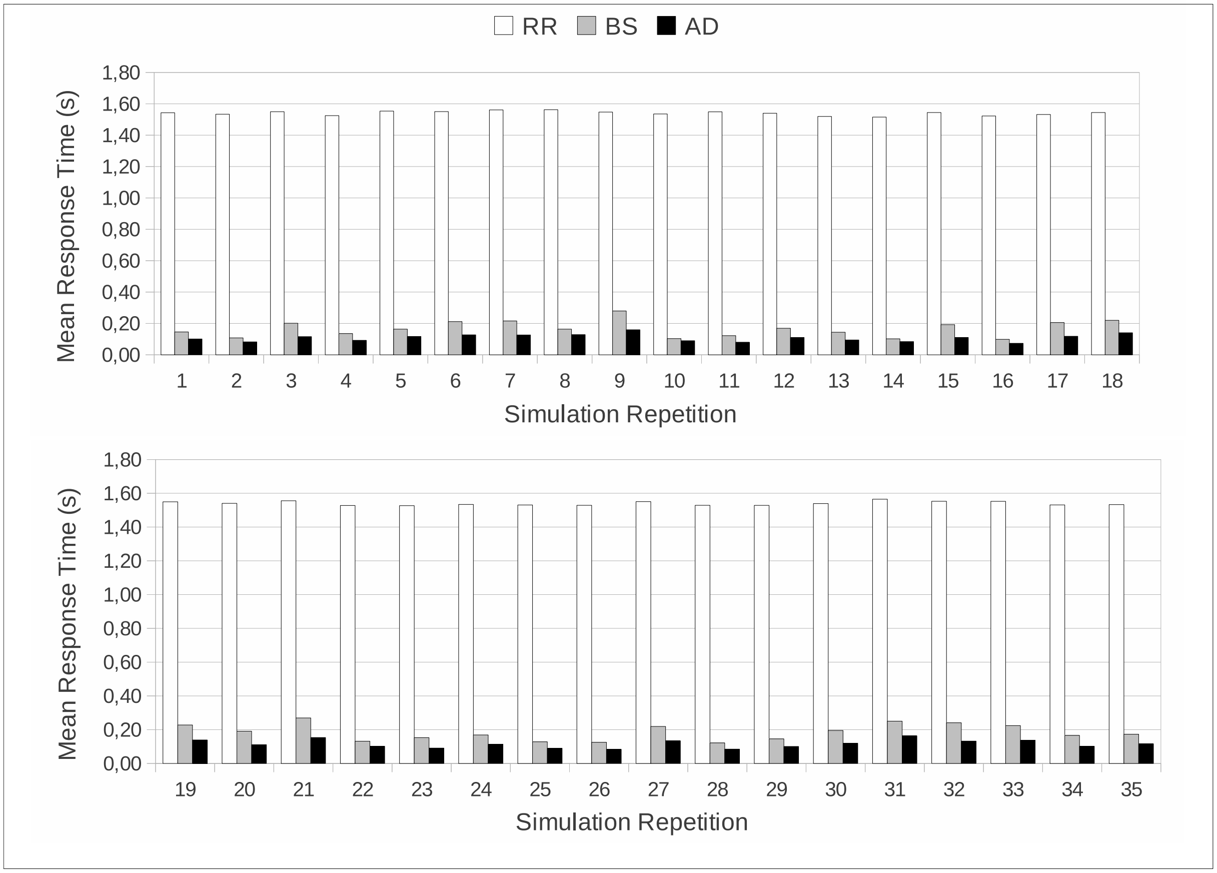


Figure 5.6: Mean response times for scenario 1.

For the sake of visualization, we replotted the mean response times of BS (Best Server) and AD (Adaptive) in Figure 5.7. As we can see, AD presented better response times in all simulations. In more than 85% of the simulations, AD presented mean response times more than 25% smaller than BS. In some cases, the difference was more than 40%.

This result was expected because, although BS tries to select the server with best response times, once the nearby server becomes overloaded, BS deviates the entire load of the client to farther servers that are lightly loaded. Although the best server decision is continuously re-evaluated by clients, the time it takes to change its decision is enough to overload the current chosen server. In the same situation, AD deviates only a fraction of the load, keeping as much requests as possible to be served by the nearby servers.

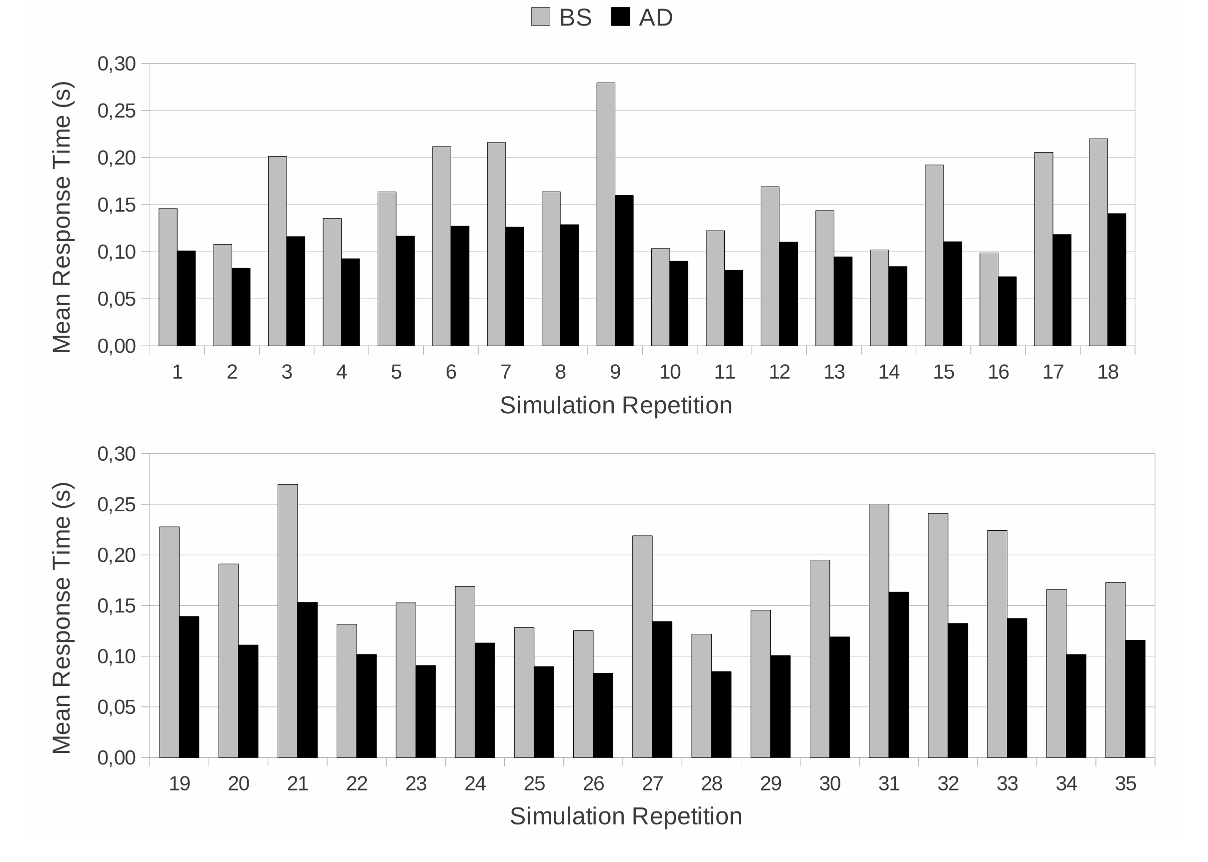


Figure 5.7: Mean response times for scenario 1.

The results obtained in scenario 1 may suggest that server selection policies that equitably distribute the load are useless if compared to policies that dynamically adapts to server condition changes. However, the simulations of scenario 2 show that it is not a general rule.

In scenario 2, the system is almost saturated – the load corresponds to 90% of the capacity. Nevertheless, since the servers have similar capacities, RR's equitable distribution grants that no server becomes overloaded. On the other hand, the higher load amplifies the effect of chasing the best server implemented by BS. For this reason, BS presented the worst performance in this scenario, with mean response times 18% larger than RR and 48% larger than AD on the average.

Even though RR does not cause server overloading in this scenario, it still does not benefit from the small network latencies offered by nearby servers. This makes AD to perform better than RR, presenting mean response times more than 25% smaller than

RR in more than 85% of the simulations.

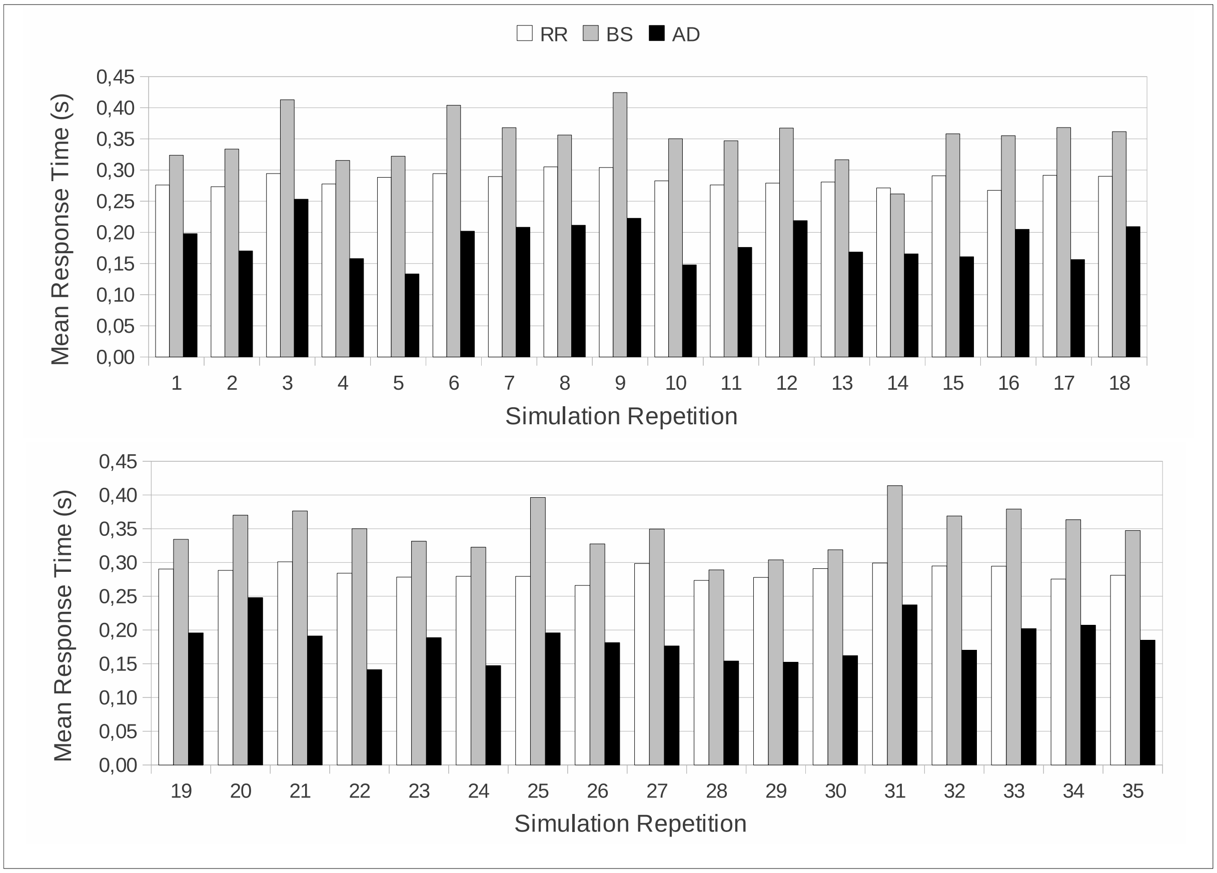


Figure 5.8: Mean response times for scenario 2.

While the first scenario is characterized by a lightly loaded system with heterogeneous servers, the second presents an almost saturated system with homogeneous servers. It is clear that, in the first case, due to its adaptability to server state changes, BS performed better than RR. In the second case, the equitable load distribution produced by RR outperformed BS's greedy strategy. Nevertheless, AD produced the best response times in both scenarios. This indicates that our solution successfully adapted to the system states while the other solutions did not. The results suggests that, in the considered scenarios, our hypothesis is valid.

5.6 Conclusion

A main advantage of client-side server selection policies is that clients can monitor end-to-end response times in a better way than server-side solutions. Besides, sometimes, client-side policies are the only option available. Most of the client-side policies proposed so far select one server to which the client should send all requests or equitably distribute the load among all of them.

Our simulations have shown that in scenarios where several clients use the same server selection policy, these two types of solution can lead to load-unbalanced states and, consequently, to the worsening of response times.

In this paper, we argue that if clients collaborate in order to balance server load they can obtain better response times. Our solution adaptively changes the fraction of load each client sends to each server giving higher priorities to nearby servers. Although this less greedy strategy of sending fractions of the load to worser servers seems to be counterintuitive, our experiments have shown that our solution overcomes the two types of policies proposed so far, even in scenarios that favors one type or another.

Future work include: (i) the evaluation of the sensitivity of our solution through an even more comprehensive set of simulations; and (ii) the deployment and evaluation of our solution in a real testbed [65].

Capítulo 6

Load Balancing for Internet Distributed Services using Limited Redirection Rates

6.1 Introduction

The web has become the universal support for applications. Increasingly, heavy loaded applications, that place extra demand on servers and network resources, are being deployed in clusters located at geographically distributed datacenters linked via the internet. Content Delivery Networks (CDNs) and cloud computing make this reality even more evident.

A key issue for good performance in these environments is the efficiency of the load balancing mechanism used to distribute client requests among the replicated services. The load balancing solution allows service providers to make better use of resources and soften the need of over-provision. Besides, even when the provision of extra servers is possible, load balancing can help to tolerate server overload until the system can be adjusted. Server overload can be caused, for example, by abrupt load peaks or by partial failures that reduce the capacity of the web server hardware. Just to mention a concrete example, Google adopts specialized load balancing mechanisms at four different levels of the software (hardware) architecture that supports its search engine [11].

There are four classes of load balancing mechanisms for geographically distributed web services [16]: DNS-based ([24, 96, 60, 62]), server-based ([17, 20, 74, 71]), dispatcher-based [54], and client-based ([26, 56]). In the DNS-based solutions, the Authoritative DNS (ADNS) of the replicated web server performs the role of the client request scheduler. When it receives a request for a URL resolution, it replies the IP address of one of the server nodes, according to some load balancing policy. In server-based solution, the load balancing policy runs in the server side. Any overloaded server replica can redirect requests to other replicas. In dispatcher-based solutions, a host placed between clients and server replicas receives all requests and forwards them to the appropriate replica. In the client-side solutions, the client runs the distribution load policy and decides to which server it sends the requests. An excellent discussion on load management strategies for large scale web services can be found in [14].

In this paper we focus on server-based load balancing solutions. So far, most of proposals found in the literature aim to minimize the web service response times by redirecting requests to an optimal remote server that is chosen in terms of communication latency and workload. To the best knowledge of the authors, none of the existing proposals focus on techniques that prevent redirected requests to overload the remote server. This situation can happen if various overloaded servers redirect requests to the same server or if the load of a remote server suffers an abrupt change. Our solution avoids this problem by combining a new strategy based on limited rates of request redirection and a heuristic that helps web servers to tolerate overload caused by abrupt load peaks and/or partial failures.

The main contributions of our work are:

- A protocol for limited redirection rates that avoids the overloading of the remote servers;
- A middleware that supports this protocol and minimizes the response times of redirected requests;
- A heuristic based on the protocol that tolerates abrupt load changes.

We have evaluated our solution through a set of simulations that covered a variety of scenarios. In order to perform the simulations, we have built a simulator based on realistic internet models ([15, 1, 18]).

The remainder of this text is organized as follows. Section 6.2 presents related works. In Section 6.3, we present our new load balancing solution. Section 6.4 describes our simulations and shows the results. Section 6.5 concludes the paper with our final comments and future work.

6.2 Related Work

The server-based load balancing for distributed web services has been studied by many researchers. Cardellini and Colajanni [17] compare the performance of centralized versus distributed control algorithms for the activation of the load balancing mechanisms, for the localization of the destination servers, and for the selection of the requests to be redirected.

The study concludes that there is a trade-off between the slightly better performance of centralized algorithms and the lower complexity of distributed ones.

Chatterjee et al. [20] present a load balancing solution that redirects requests based on server capacity, server load, and size of the requested documents. The status of all servers is centralized by a single entity. This global information is used by overloaded replicas to redirect requests to the lightly loaded one.

The solution proposed by Ranjan and Knightly [74] considers both CPU load and network latencies to reduce the system response time. If a request arrives at web server k, then the objective is to dispatch the request to a web server j satisfying $min(2\Delta_{kj} + T_j)$, where Δ_{kj} is the network latency between k and j and T_j is the estimate of service time in j.

Pathan, Vecchiola, and Buyya [71] propose a solution similar to the one proposed by Ranjan and Knightly [74]. The main difference is that, in the former, when a request is redirected, the end client needs to resend the request to the other server replica. In the latter, the redirecting replica intermediates the communication between the end client and the other replica by forwarding the request. Therefore, the solution proposed in [71] needs to estimate the network latency among end clients and server replicas, which is much more difficult than estimate the network latency among server replicas.

Our solution also tries to minimize the system response time considering the load status of the web servers and network latencies, like [74] and [71]. The innovation of or work is twofold: (i) we propose a remote resource reservation mechanism that avoids overloaded web servers to overload lightly loaded web servers, and (ii) a heuristic based on this mechanism that tolerates abrupt load peaks.

6.3 Limited Redirection Rates

6.3.1 Overview

The response time of a request redirected by a web server to a remote one is affected by two factors: the latency between the web servers, and the time the remote web server takes to process the request. Intuitively, the best choice would be to redirect requests to the closest lightly loaded web server. However, this intuitive policy may be very inefficient if the redirections overload the remote web server. For example, consider the situation illustrated in Figure 6.1. In this situation there are three web servers with the same capacity, let us say, 100 rps (requests per second). The gray bars represent the average load incoming to each server over the time. In $t1$, the web server A becomes overloaded and needs to redirect an average of 40 rps. Note that, although, in this example, the overload is caused by an abrupt load peak, it could also have been caused by a partial failure of server A, which would reduce its capacity.

Let us consider that B is the closest lightly loaded web server. In time $t2$, A starts redirecting 40 rps to B, which becomes overloaded and multicasts an alarm message informing that it is overloaded ($t3$). Once A receives this alarm, it stops redirecting requests to B and starts redirecting to the next closest lightly loaded web server ($t4$), i.e., C. Then, C becomes overloaded.

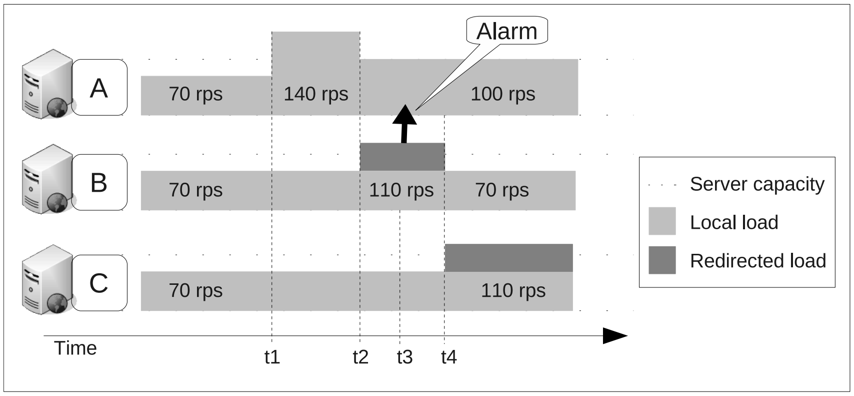


Figure 6.1: Intuitive policy.

The situation illustrated in Figure 6.1 happens because the web servers share only the binary information "overloaded or not". This policy can be improved if the web servers also inform the capacity they can lend to the others. In this case, the web server A could split its exceeding load between B and C. However, even this improved policy may be inefficient in some cases. For example, let us consider the situation in Figure 6.2. In $t1$, the servers inform the capacity they can lend. Next, in $t2$, A becomes overloaded. It knows the capacity of B and C and appropriately split its exceeding load between them ($t3$). When D becomes overloaded ($t4$), it also knows the capacity of B and C, but it does not know if another web server is already redirecting requests to B neither how much redirected load B might be already receiving. Thus, since D's exceeding load fits to the last B's capacity information, D starts redirecting to B ($t5$), which becomes overloaded.

Clearly, there is a trade-off between redirecting exceeding load to closest remote servers, aiming better network latencies, and sparing requests to various servers, aiming to avoid overload them. Our solution tries to balance this trade-off to minimize the service response times. In our solution, an overloaded web server redirects its exceeding load to the set of closest lightly loaded web servers that, together, are able to serve its demand. The main idea of our solution is a protocol that imposes a maximum limit of requests that each overloaded server is allowed to redirect to each lightly loaded web server. The limits are defined considering information about the exceeding capacity of lightly loaded web servers and about the network latencies among the servers. This protocol, which we

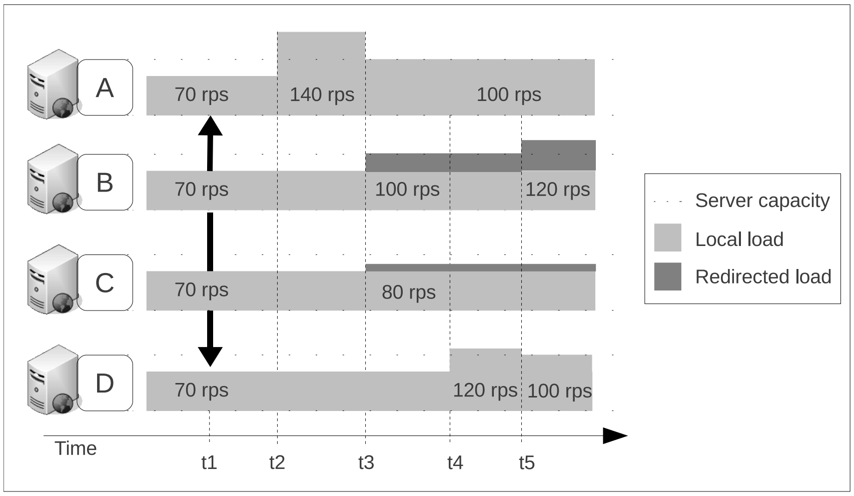


Figure 6.2: Improved intuitive policy.

call LRR (Limited Redirection Rate), avoids the situation illustrated in Figure 6.2.

Figure 6.3 exemplifies the operation of our load balancing solution. Again, let us consider that A and D are overloaded web servers and B is the closest lightly loaded web server for both. Supposing that A becomes overloaded first, the load balancing happens in four steps. First, A successfully obtains permission to redirect up to 30 rps requests to B (step 1). Since the capacity of B is not enough to its demand, A also obtains permission to redirect up to 10 rps to C (step 2). When D becomes overloaded, it tries to obtain permission to redirect requests to B, but B has already compromised its entire capacity (with A) and refuses D's request (step 3). Thus, D obtains permission to redirect up to 20 rps to C (step 4). The large arrows indicate the average load, in requests per second, each web server is receiving or redirecting after step 4.

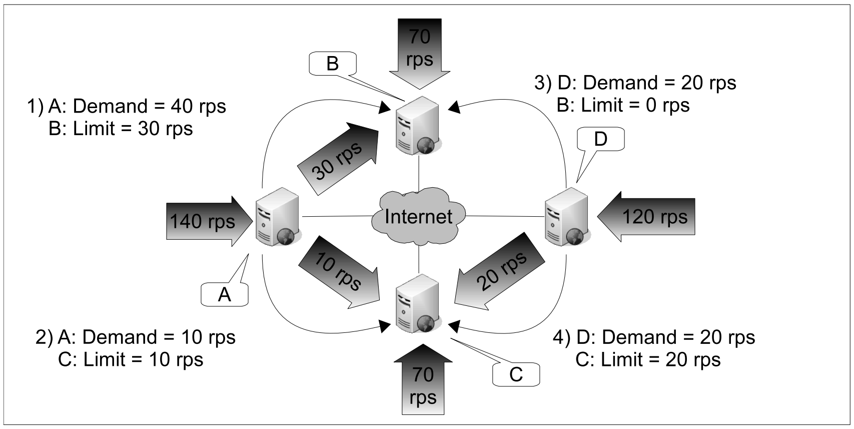


Figure 6.3: Limited Redirection Rate.

Our load balancing solution is composed of two parts, a middleware for the management that supports the protocol (Section 6.3.2), and a load balancing heuristic (Section 6.3.3).

6.3.2 Virtual Web Resource

In this paper we propose the Virtual Web Resource (VWR), a middleware that allows web servers to share their resources. Figure 6.4 illustrates the VWR concept.

In VWR, web servers can assume two different states: providers and consumers. Providers are lightly loaded web servers that can share resources with other web servers. Consumers are overloaded web servers that consume resources shared by providers.

The VWR abstracts the location of the providers to the consumers, which see the resources shared by the providers as a single pool of remote resources. Consumers can allocate the portion of the VWR they need and then use it to serve the load they are not able to serve using local resources.

The VWR manages the amount of resources provided and consumed by remote web servers. The load submitted to the VWR by a consumer is divided among the minimum number of closest providers (supposedly) able to serve the demand, aiming to minimize the network latencies.

The middleware that implements VWR is composed of two software elements, the VWRMaster and the VWRNodes. The VWRMaster is the core of the middleware. It is responsible for allocating virtual resources to the consumers and also for reallocating virtual resources when the status of providers changes. In order to perform this, it maintains information about: the status of the web servers, the network latencies among the web servers, the amount of resources shared by each provider, the amount of virtual resources allocated to each consumer, and the mapping between virtual resources and physical resources (providers).

It is important to emphasize that, in the VWR, when a consumer *allocates* an amount of virtual resources, there are no guarantee that the providers will serve the redirected requests in the expected time. The redirection rate limits are defined on basis of estimates of the provider's free capacity. Since the web workload is very dynamic, the provider condition may abruptly change. In this case, the redirections arriving to the provider will be served in a best effort way. Although this lack of guarantee seems worthless, the middleware ensures that the aggregate redirected load does not exceed the estimate free capacity of the providers, and, as we will show, this fact allows us to achieve good results.

The VWRNodes are software agents that run on every web server. They monitor the current load of the web server and communicate with the VWRMaster to offer local resources or to allocate virtual resources. The VWRNode also monitors the network

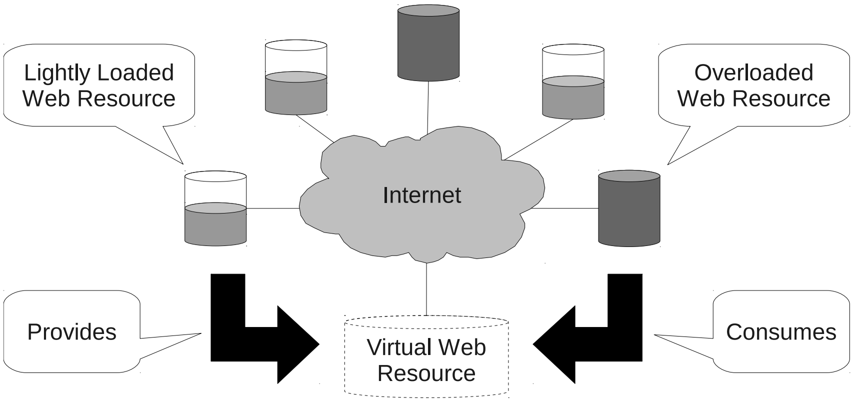


Figure 6.4: Virtual web Resource.

latencies between the local web server and all remote web servers and reports this information to the VWRMaster. The VWRMaster abstraction can be implemented as a simple replicated table using a total-order broadcast middleware [30].

Figure 6.5 shows the interaction between a consumer and the VWR middleware. The consumer web server communicates with the local VWRNode to allocate virtual resources and to submit load to them. In order to allocate the virtual resources, the VWRNode interacts with the VWRMaster through the following messages:

- **ReportLatencyTable**: sent by VWRNodes to report the estimate of latencies among the local web server and the others;

- **AcquireVirtualResources**: sent by VWRNodes to request the allocation of an amount of virtual resources (in requests per second) or to modify the amount already allocated. If the invoking VWRNode is a provider, the VWRMaster sets the state of the VWRNode as "consumer" and reallocates virtual resources for all consumers that were consuming resources of this provider. Otherwise, if the invoking VWRNode is a consumer updating its value amount of virtual resources, the VWRMaster reallocates the virtual resources to it;

- **ProviderList**: sent by VWRMaster to inform the list of providers that will provide the virtual resources to the VWRNode. This list also informs the maximum load that each provider is supposed to serve. The list is chosen regarding the latencies from providers to the consumer and is ordered by the network latency. In order to minimize response times, the VWRNode should use all capacity of the closest provider before submit load to the next one.

At the consumer side, the VWRNode also acts as a flow controller, not allowing the consumer to submit to the VWR more requests than the maximum limit.

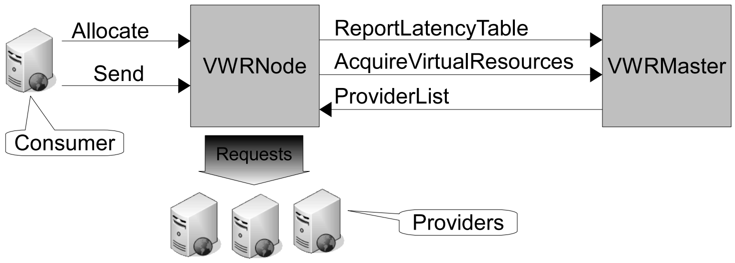


Figure 6.5: Interaction between a consumer and the middleware.

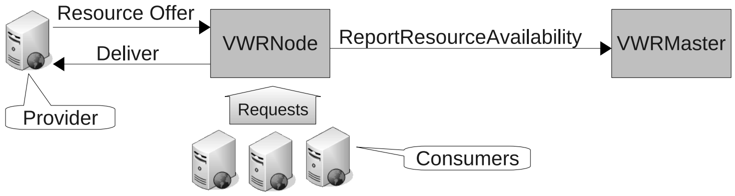


Figure 6.6: Interaction between a provider and the middleware.

Figure 6.6 shows the interaction between a provider and the VWR middleware. The provider web server communicates with the local VWRNode to announce resources that can be shared. The VWRNode receives requests incoming from consumers and delivers them to the web server. In order to offer resources to be shared, the VWRnode interacts with the VWRMaster through the following message:

- **ReportResourceAvailability**: sent by VWRNodes to announce the amount of resources (in requests per second) they are able to share or to update the value amount already shared. If the invoking VWRNode is a consumer, the VWRMaster sets the state of the VWRNode as “provider” and disposes any virtual resource it has allocated. Otherwise, if the invoking VWRNode is a provider reducing the amount of resources it is sharing, the VWRMaster reallocates virtual resources for all consumers that share resources of this provider;

The message cost of our solution varies from 2 to N messages per workload threshold violating change (N = number of geographically distributed web servers). Once the resources are assigned to the consumer, subsequent redirections do not involve any other control messages. This cost does not differ substantially from the cost of any other load distribution solution that relies on global state information. In short, our message overhead is low.

```
IF Queue <= th_queue, THEN
  Deliver request
  RETURN
ELSE
  Try to send the request to VWR
  IF request was sent to VWR, RETURN
ENDIF

Deliver request
RETURN
```

Figure 6.7: Load Balancing Heuristic.

6.3.3 Load Balancing Heuristics

The VWR middleware solves the problem of choosing the remote web servers to which redirect requests. It needs to be combined with some policy that decides when to redirect requests. Figure 6.7 presents a pseudo-code of the heuristic we adopted to make this decision. The basic idea of this heuristic is trying to keep the local request queue shorter than a threshold, without ever overloading the remote web servers. The reason for this heuristic is simple. If a request needs to wait in a queue, it should not pay the price of a redirection latency.

When the web server receives a request, it checks its queue. If the queue is shorter than the threshold (**th_queue**), the web server processes the request locally. If not, the web server tries to submit the request to the VWR. If the request could not be sent to the VWR, it is enqueued locally.

Another decision problem is related to the amount of remote resources that an overloaded web server should allocate. This decision depends on the request rate the web server receives. In our solution, we use a sliding average of the request rate to detect significant load changes. The remote resource allocation is dynamically adjusted on the basis of the average request rate.

An important issue regarding the remote resource allocation is the wide area network latencies that delay the communication between VWRNodes and the VWRMaster. This causes server state infomation staleness and makes the reaction to abrupt request rate changes not fast enough.

In order to deal with this problem, we propose the use of a capacity safety margin. Each web server must have spare resources (CPU, memory, disk, etc.) beyond the amount needed to support its average request rate. This safety margin aims to tolerate abrupt

```
Definitions:
C: web server capacity
L: Last significant average load
   change
A: Last average load sample
S: Safety margin size
ST: Load sample time
SAWS: window size for load sliding
      average
th_inf: Threshold for significant
load decrease
th_sup: Threshold for significant load
increase

On each ST seconds:
  A = average request rate of SAWS seconds
  IF (A < (L-th_inf)) or
     (A >= (L+th_sup)), THEN
    L = A
  ENDIF
  IF (L+S)>C, THEN
    Offer (C-(L+S))
  ELSE
    Allocate ((L+S)-C)
  ENDIF
  RETURN
```

Figure 6.8: Remote resource allocation heuristic.

peaks of load until the remote resource allocation can be adjusted. For example, suppose that a web server with capacity C rps and capacity safety margin S rps is receiving an average load of L rps. If $(L+S) < C$, the web server can share $(C-(L+S))$ rps. If not, the web server has to allocate $((L+S)-C)$ from remote resources. The pseudo-code in Figure 6.8 describes the heuristic used to manage the remote resource allocation.

6.4 Evaluation

In order to evaluate our solution, we have implemented a simulator using the CSIM for Java[1], a discrete event simulator framework. Sections 6.4.1 to 6.4.3 detail our simulations

[1] http://www.mesquite.com/

and Section 6.4.4 presents the results.

6.4.1 Load Generation and Web Servers

We used the PackMime Internet traffic model [15, 1] to generate HTTP traffic in our simulations. PackMime has been designed from a large-scale empirical study of real web traffic and has been implemented in the *ns-2*[2], a well known network simulator. In order to use the model in our simulations, we have implemented a Java version of the PackMime.

PackMime allows the generation of both HTTP/1.0 and HTTP/1.1 traffic. The intensity of the traffic is controlled by the rate parameter, which is the average number of new connections started per second. The implementation provides a set of random variable generators that drive the traffic generation. Each random variable follows a specific distribution. The distribution families and the parameters used in PackMime are described in [15]. In our simulations, we used the following random variables:

- **PackMimeHTTPFlowArrive**: interarrival time between consecutive connections;
- **PackMimeHTTPNumberPages**: number of pages requested in the same connection (if using HTTP/1.1);
- **PackMimeHTTPObjsPerPage**: number of objects embedded in a page;
- **PackMimeHTTPFileSize**: sizes of files (pages and objects);
- **PackMimeHTTPTimeBtwnPages**: gap between page requests;
- **PackMimeHTTPTimeBtwnObjs**: gap between object requests;

We assumed that each geographically distributed replica of the web server is composed of a cluster of servers. Each server is simulated as a queue system with fixed service time of 10 ms. Thus, a cluster with k servers provides a capacity of k*100 requests/second. A cluster may have a partial failure, losing up to $k - 1$ servers. In that case, the capacity of the cluster is reduced in a direct proportion to the number of failures. For the sake of simplicity, we will refer to a cluster of servers simply as a web server and represent the partial failures as a capacity reduction. The request interarrival time distribution is defined by the PackMime model [15, 1].

[2] http://www.isi.edu/nsnam/ns/

6.4.2 Internet Latencies

We have considered a scenario with six replicas of the web server that are world wide distributed (Figure 6.9): one in South America (S1), one in North America (N1), two in Europe (E1 and E2), and two in Asia (A1 and A2).

We used the average of the latencies measured on real hosts of PlanetLab[3] in Brazil, USA, Belgium, Austria, Japan, and China, to simulate the latencies among the replicated web servers. Table 6.1 shows these latencies. We also consider that each replica serves a region and that the latency between a replica and a client of its region is 10ms.

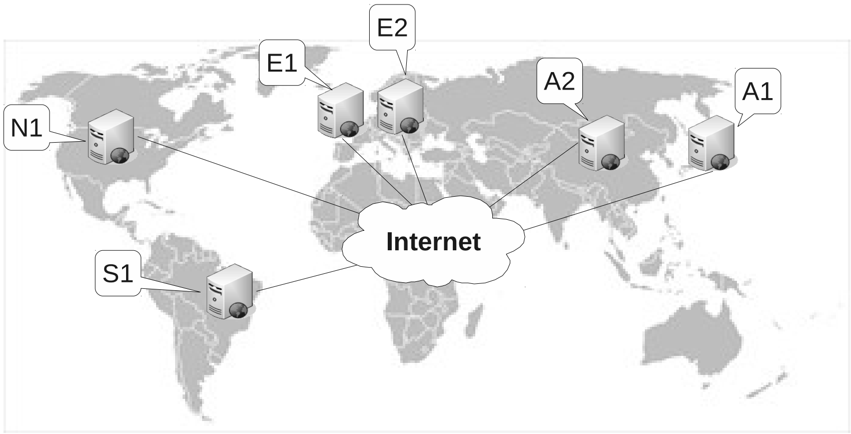


Figure 6.9: Web Server Replicas.

Table 6.1: Table of latencies among the replicas (in milliseconds).

	S1	N1	E1	E2	A1
N1	89				
E1	138	48			
E2	140	58	18		
A1	193	109	151	162	
A2	272	156	114	122	68

In order to consider the latency of the TCP protocol, we adopted the analytic model proposed by Cardwell et al. ([18]). This work extended previous models for TCP steady-state by deriving models for two other aspects that can dominate TCP latency: the connection establishment three-way handshake and the TCP slow start [77]. Therefore, the model proposed by Cardwell et al can predict the performance of both short and long TCP flows under varying rates of packet loss.

[3] http://www.planet-lab.org

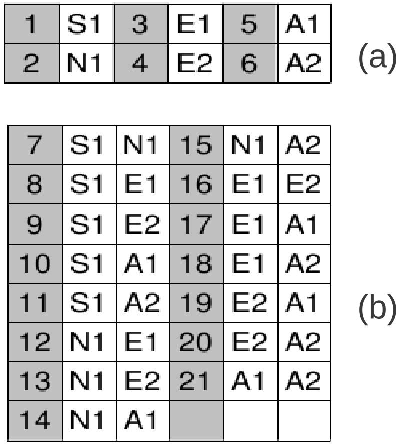
In practice, the TCP model estimates the time needed to transfer data from one endpoint to another in terms of: (i) the size of the data to be transfered; (ii) the network latency between the two endpoints; and (iii) the packet loss rate probability.

6.4.3 Configuration

We compared our solution with two other load balancing policies:

- Round Robin with Asynchronous Alarm (RR): Improves the simple round robin policy by considering the load status of remote servers. Overloaded servers broadcast an alarm informing their status. This way, all overloaded servers can remove each other from their rotative list;
- Smallest Latency (SL): Overloaded servers redirect exceeding requests to the closest lightly loaded remote server. Overloaded servers use asynchronous alarm to inform their status.

We run the simulations on 21 variations of the scenario depicted by Figure 6.9: all combinations of 1 overloaded server and 5 lightly loaded servers, and all combinations of 2 overloaded servers and 4 lightly loaded servers. Figure 6.10 shows the overloaded replicas of each scenario.



1	S1	3	E1	5	A1
2	N1	4	E2	6	A2

(a)

7	S1	N1	15	N1	A2
8	S1	E1	16	E1	E2
9	S1	E2	17	E1	A1
10	S1	A1	18	E1	A2
11	S1	A2	19	E2	A1
12	N1	E1	20	E2	A2
13	N1	E2	21	A1	A2
14	N1	A1			

(b)

Figure 6.10: Simulation scenarios: (a) 1 overloaded server. (b) 2 overloaded servers.

The load generators (PackMime) were configured to generate 210 connections per second. The overload was induced through the simulation of a partial failure that reduced the capacity of the servers to 200 rps. Figure 6.11 illustrates the workload generated by one of our load generators to one of the web servers. The parameters used in the heuristics (Figures 6.7 and 6.8) are annotated in Table 6.2.

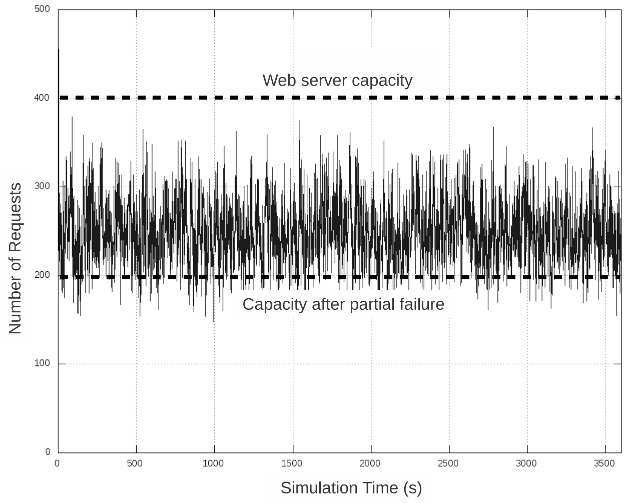


Figure 6.11: Example of workload generated by Packmime.

Table 6.2: LRR Simulation Parameters.

Parameter	**Value**	**Description**
th_queue	1 request	Threshold (queue size) for redirecting a request to the VWR.
S	0.25*A rps	Safety margin size.
ST	1s	Load sample time.
SAWS	30s	Window size for load sliding average.
th_inf	0.125*A rps	Threshold for significant load decrease.
th_sup	0.125*A rps	Threshold for significant load increase.

A: load average.

6.4.4 Results

In this section, we present the evaluation of our load balancing solution. Figure 6.12 summarizes the results of the entire set of simulations. Since the histograms of response times for all simulations presented the same distribution, we believe that the mean response times can be used to compare the performance of the different solutions.

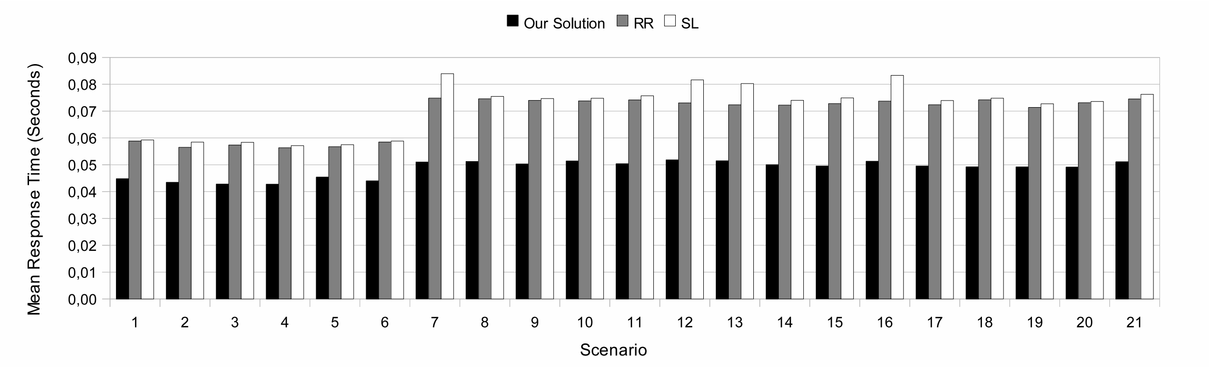


Figure 6.12: Mean response times for all scenarios.

Each bar in the chart represents the mean of the mean response times of one of the load balancing policies, in one scenario. In general terms, we can see that LRR presented mean response times 29% smaller than RR and 31% smaller than SL. In the following we explain important aspects of our solution by analyzing two scenarios (8 and 13 - Figure 6.10) in detail.

We first analyse the results obtained through a simulation instance of the scenario 8, where servers S1 and E1 suffer partial fails and become overloaded. Figures 6.13, 6.14, and 6.15 present the histograms of the response times perceived by the clients for the three load balancing policies. As we can see, a frequency analysis shows that while 95% of the requests were responded in less than 0.14 s using LRR (Limited Redirection Rate), using RR and SL, the same percentage of requests were responded in 0.28 and 0.27 s, respectively. This indicates that LRR has provided better response times.

The meaning of the histograms of Figures 6.13, 6.14, and 6.15 is summarized by the chart of Figure 6.16 that shows the mean response times presented by each load balancing policy. According to the figure, the mean response time presented by LRR is 32% smaller than SL's and 31% smaller than RR's. This corroborates the histogram results, indicating that LRR presented better response times.

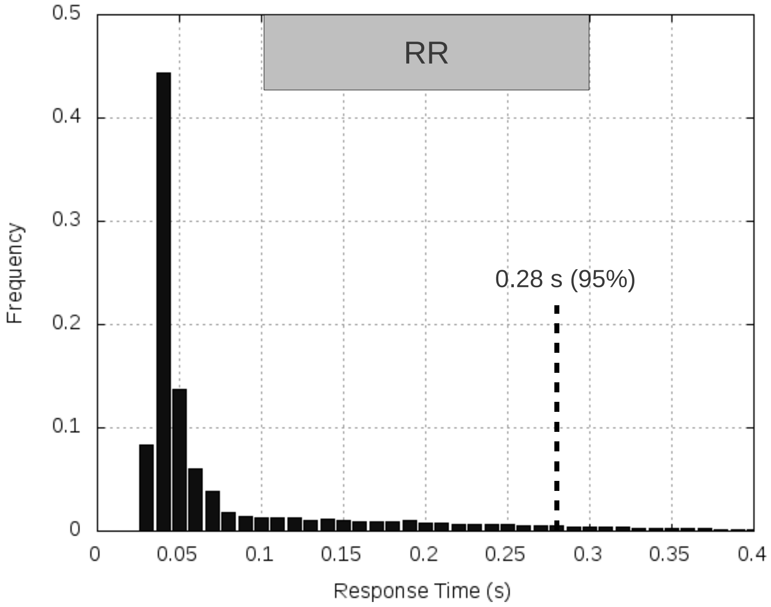


Figure 6.13: Histogram of response times for scenario 8, using RR.

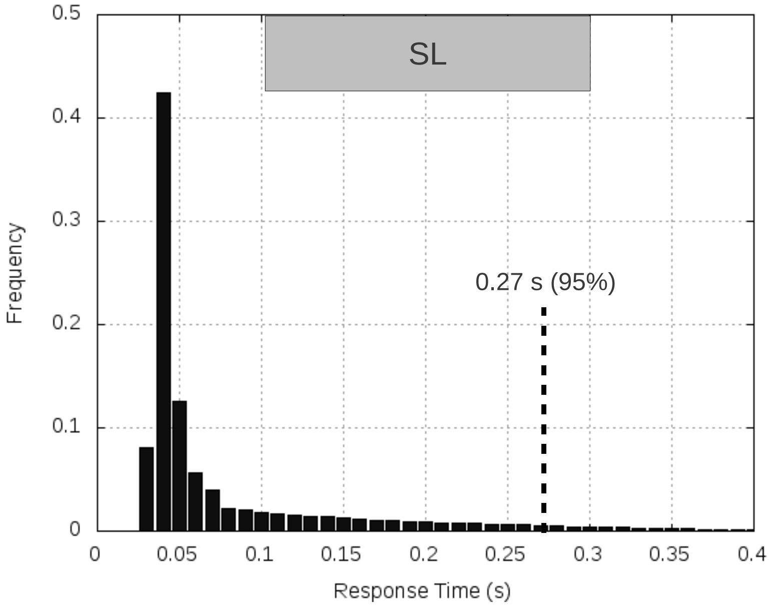


Figure 6.14: Histogram of response times for scenario 8, using SL.

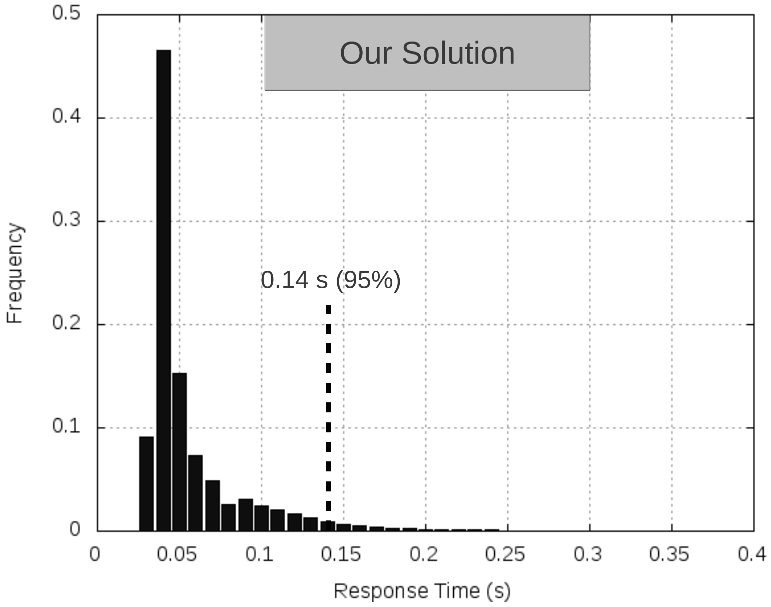


Figure 6.15: Histogram of response times for scenario 8, using our solution.

Let us analyse the aspects that made LRR to behave better than the others in scenario 8. Table 6.3 shows the percentage of requests redirected to each server by each policy. As we can see, RR reduces the probability of redirections to overload the remote servers, because exceeding requests are divided among various remote servers. However, RR is not aware of the network latencies, making far remote servers receive the same amount of redirected requests than closer ones. This affects RR's request response times.

In scenario 8, the closest lightly loaded servers for S1 and E1 are distinct servers (N1 and E2 respectively), thus, this scenario prevents SL from choosing the same remote server for S1 and E1. As Table 6.3 shows, SL divided their redirections between N1 and E2, tending to make their queues grow up. Since our protocol limits the maximum redirection rate to each provider, LRR also sent more than 32% of redirections to A2. Thus, it avoids overloading the remote servers.

An interesting aspect to be pointed out is that LRR has redirected a number of requests more than twice the number of redirections achieved by the others (see Table 6.3). This is a consequence of the heuristic that aggressively tries to maintain the server queues short. The intuition behind this heuristic is simple: sometimes, it is better to pay the price of the network latency than to wait in the local queue.

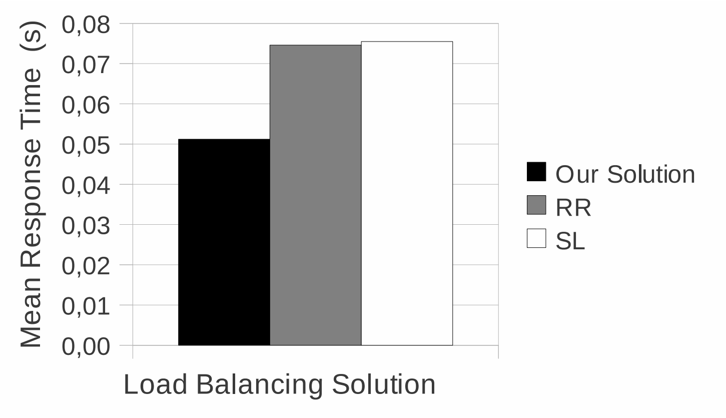


Figure 6.16: Mean response times for scenario 8.

Table 6.3: Scenario 8 - Percentage of redirections.

	Redirection Percentage						Total
Policy	S1	N1	E1	E2	A1	A2	Redir.
RR	0%	24,41%	2,35%	24,41%	24,41%	24,41%	214587
SL	0%	54,88%	0%	45,12%	0%	0%	214689
LRR	0%	42,64%	2,34%	21,88%	0,67%	32,46%	434996

Figure 6.17 shows the mean response times for scenario 13, where N1 and E2 have partial fails and become overloaded. Again, we can see that LRR has presented response times smaller than the other policies (36% smaller than SL's and 29% smaller than RR's).

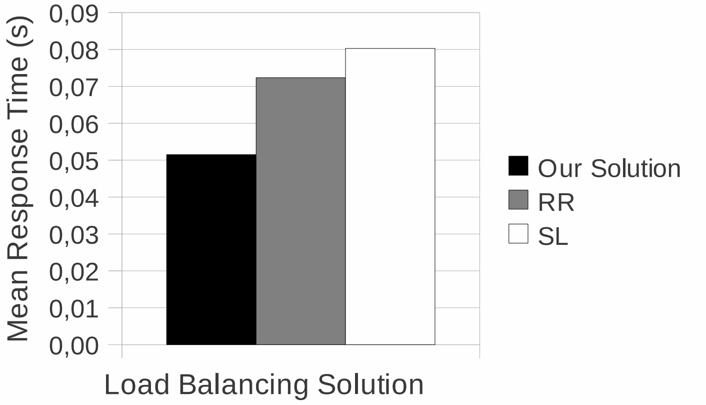


Figure 6.17: Mean response times for scenario 13.

Different from scenario 8, in scenario 13, the two overloaded servers share the same closest lightly loaded server (E1). This fact is indifferent for RR and it works the same way. Since LRR and SL are aware of network latencies, this scenario tends to affect their performance.

Table 6.4 shows the percentage of requests redirected to each server by each policy in scenario 13. As we can see, SL concentrated the redirections to E1. Eventually E1 became overladed, making SL to redirect a small amount of requests to S1 and A2. Thanks to the VWR protocol, LRR redirected E1, S1, and A2. This conservative mechanism prevents overloading E1.

Table 6.4: Scenario 13 - Percentage of redirections.

	Redirection Percentage						Total
Policy	S1	N1	E1	E2	A1	A2	Redir.
RR	24,42%	0%	24,42%	2,33%	24,42%	24,42%	210804
SL	8,14%	0%	84,05%	0%	0%	7,82%	210789
LRR	18,85%	0%	38,73%	3,76%	0%	38,65%	430581

6.5 Conclusion

Load balancing techniques can help geographically distributed web services to tolerate server overloads. The overload can be caused by abrupt load peaks and/or partial failures that reduce server capacity. In this paper, we have presented a new server-based load balancing policy for worldwide distributed web servers. To the best knowledge of the authors, none of the previous proposals have focused on techniques that prevent redirected requests from overloading remote servers. Our solution was designed on the basis of a protocol that limits the redirection rates, preventing such a problem and reducing response times.

In order to evaluate our solution, we have implemented a simulator based on realistic internet models and real internet latencies. The results obtained through a set of simulations show the benefits of our solution in comparison to two other well known policies, and justify our design decisions.

Future work includes: (i) the evaluation of more complex strategies for resource allocation in the VWR middleware; (ii) the evaluation of the sensitivity of our solution through an even more comprehensive set of simulations; (iii) the deployment and evaluation of our solution in a real testbed [65]; and (iv) the development of a new version of the middleware that uses linear programing to optimize the division of resources among consumers.

Capítulo 7

Analysis of Resource Reservation for Web Services Load Balancing

7.1 Introduction

Applications that place extra demand on servers and network resources are increasingly being deployed in clusters located at geographically distributed data centers linked via the internet; the data center software and hardware is a *cloud*. Clouds can offer services ranging from a very basic computing infrastructure to universal cloud-transparent applications or services.

A key issue for the performance of cloud-based applications is the efficiency of the load balancing mechanism used to distribute client requests among the data centers where the distributed application has been deployed. An efficient load balancing solution allows clouds to make better use of resources and can potentially reduce the need for resource overprovisioning. Besides, even when the provision of extra servers is possible, load balancing can help tolerate workload peaks until the system can be adjusted [8]. For example, Google adopts specialized load balancing mechanisms at four different levels of the software (hardware) architecture used to run its indexing and search engine [11]. Amazon, Microsoft and other important service providers also implement their own load balancing solutions [8].

From now on the term *web server* is used as a synonym for any class of geographically distributed computing facility addressable as a single entity and where a web service has been deployed. Thus, in the paper, a *server* can be anything from a single computer, to a cluster of computers within a data center, or even to a whole data center (cloud). It is useful to consider servers at different levels of abstraction because our solution can then be combined with any network—traffic engineering—setup devised to interconnect the data centers where the servers (applications) reside.

In this paper we focus on server-based load balancing solutions. This kind of solution provides a fine-grained level of control over the assignment of requests to the servers, since all requests reach, at least, one of them. This is not the case of DNS-based[1] approaches, in which the ADNS[2] does not have control over all the clients, due to the DNS caching system. Other clear advantage of server-based solutions is that they are transparent to the clients, unlike client-based solutions, in which the clients have to be instrumented to implement the load balancing. A third main feature of server-based solutions is the facility to use information about the state of the servers. Although, many solutions propose the exchange of control information among servers and the ADNS or dispatchers, this kind of communication is much easier to implement when it is restricted to the servers.

So far, most of the proposals found in the literature try to minimize the web service response times by redirecting requests to an optimal remote server that is chosen solely in terms of communication latency and workload. A problem with this kind of load balancing solution is that they distribute the redirected load among the lightly loaded servers through a reactive strategy. When a lightly loaded server becomes overloaded, because the load balancing mechanism has sent to it some extra workload, it may react to the workload increase with the creation of a workload bypass to yet another lightly loaded server. If this reaction chain to workload surges is not fast enough, the response times are negatively affected. Our solution avoids this problem by adopting a preventive approach. It combines a new strategy based on reservation of remote resources with a heuristic that helps web servers better balance their workload and tolerate load peaks.

The main contributions of our work are:

- A protocol for remote resource reservation that avoids the overloading of remote servers;
- A heuristic associated with the remote resource reservation protocol that enables the protocol to tolerate abrupt load changes;
- A middleware that supports the remote resource reservation protocol.

We have evaluated our solution through a large set of simulations that encompass an exhaustive set of workload scenarios based on realistic internet traffic models [1, 15, 18]. In these simulations, we investigate and compare the performance of our solution to other well-known approaches under varying workloads and scales. There is a set of experiments designed specifically to assess the suitability of the proposed solution to abrupt workload changes.

[1] **D**omain **N**ame **S**ystem.

[2] **A**uthoritative **D**omain **N**ame **S**erver.

The remainder of the text is organized as follows. Section 7.2 presents related work. In Section 7.3, we introduce our new load balancing solution. Section 7.4 describes our simulations setup and Section 7.5 presents the results. Section 7.6 concludes the paper with our final comments and future work.

7.2 Related Work

There are four classes of load balancing solutions for geographically distributed web services [16]: DNS-based, server-based, dispatcher-based [54], and client-based. In the DNS-based solutions, ADNS of the replicated web server performs the role of the client request scheduler. When it receives a request for a URL resolution, it replies with the IP address of one of the server nodes, according to some load balancing policy. In server-based solutions, as already mentioned, the load balancing policy is implemented at the server tier. Any overloaded server replica can redirect requests to other replicas. In dispatcher-based solutions, a host placed between clients and server replicas receives all requests and forwards them to the appropriate replica. In the client-side solutions, the client runs the distribution load policy and decides to which server it can redirect requests.

The papers [25, 60, 96] present DNS-based load balancing mechanisms that use information about server utilization in the server selection. In these works, an agent monitors the states of the servers and reports this information to the ADNS. When a name resolution query arrives, the ADNS uses the utilization information of the server nodes to assign one of the replicas to the client. Many kinds of information can be used in the ADNS decision, such as the length of the request queue, CPU, memory usage, or network traffic.

Other DNS-based proposals use information about clients in server selection. There are two types of client domain information that can be used in the load balancing: client proximity (latency) and client domain load. In the first case, the ADNS tries to assign the nearest server to the client [11, 70, 88]. A main concern in this kind of load balancing mechanism is to estimate the proximity between clients and servers. Since the ADNS does not have information about the client host, a solution for this problem is to assume that clients are located near to their local nameservers and estimate the distance between servers and local nameservers. Shaikh et al. [83] discuss the effectiveness of this solution and propose another solution that adds a new type of resource record to the DNS, in order to allow the ADNS to identify the client host.

The variability of request rates generated by client domains is an important issue for load balancing mechanisms. A name resolution proceeding from a large domain will cause a stronger impact on the Web system than a name resolution proceeding from a small domain. Hence, the estimation of the load generated by client domains may be very useful for the load balancing mechanisms. This information allows the ADNS to treat differently domains that generate high request rates (hot domains) from the others (cold domains). The works [24], [20] and [25] present promising results using information about client domain load for: (i) avoiding the assignment of hot domains to the same servers; (ii) estimating the real load of each server; and/or (iii) applying different TTLs for name resolutions destined to hot and cold domains. Following this line, [62] proposes a new policy that combines client load information and server load information in order to alleviate the negative effects of the DNS caching over the load balancing.

These mechanisms assume that servers collaborate with the ADNS, identifying and reporting the hot domains. In order to perform this task, it may be necessary to cluster the Web clients that access the same local nameserver. A name resolution affects the whole domain under the control of the same local nameserver. Techniques for clustering of Web clients include using prefix matching of IP addresses and name lookups to identify the domain name of the IP addresses. Another solutions for this problem are presented in [12, 50].

An approach used to deal with the low control of the ADNS is to complement the DNS-based mechanism with a second level of load balancing based on server-side redirection. In this approach, overloaded Web servers redirect incoming requests to other replicas. The redirection approach is adopted in [17, 46, 74].

Dikes et al. [34] present a comparison of 6 client-side server selection policies for the retrieval of objects via the Internet: *Random*, *Latency*, *BW*, *Probe*, *ProbeBW*, and *ProbeBW2*. *Random* selects a server randomly (equitably distributes the load). *Latency* and *BW* select the server that offers the best historical network latency and bandwidth, respectively. *Probe* probes the network latency of all servers (via ping) and selects the first to reply. *ProbeBW* considers only n servers with the fastest historical bandwidths and applies *Probe* to them. *ProbeBW2* probes the network latency of all servers and applies *BW* to the first n servers to reply. In summary, the results showed that the policies based on dynamic network probes presented best service response times. A similar result was presented by Conti, Gregori, and Panzieri in [28].

In the papers [26, 27], Conti, Gregori and Lapenna compare a probe-based policy to a parallel solution. In the parallel solution, the total amount of data to be retrieved is divided in equal fixed-size blocks. Then, one block of data is requested to each service replica. The authors argue that this strategy can be useful when: (i) the objects to be downloaded are large; (ii) the client operations are read-only; (iii) the client wants better throughput; and (iv) the performance of the service replicas is not the bottleneck of the system.

Mendonça et al. [56] compare 5 client-side server selection policies for SOAP web services: *Random*, *HTTPPing*, *Parallel Invocation*, *Best Last*, and *Best Median*. *Random* and *HTTPPing* are equivalent to *Random* and *Probe* from [34]. In *Parallel Invocation*, the client requests all servers in parallel and waits for the fastest response. *Best Last* selects the server with the best recent response time. *Best Median* selects the server with the best median response time. According to the authors, *Best Last* and *Best Median* are the best choice in most of cases.

Garg and Juneja [39] propose a solution in which clients and servers collaborate through a collection of agents that they call *client ants* and *server ants*. Client and server ants collaborates to decide the best server to serve the clients. The ants maintain historical information about previous requests and this information is used in future decisions. The ant behavior is also explored by Nishant et al. [66]. This work proposes the use of mobile agents (ants) that traverse the network topology to identify overloaded and underloaded servers and redistribute the load.

In [63] we have proposed an approach for client-based load distribution that adaptively changes the fraction of load each client submits to each service replica to try to minimize overall response times.

Liu e Lu [54] present a solution based on a centralized dispatcher that uses the state of the servers to redirect the client requests through HTTP redirection. Besides, the dispatcher also performs an admission control that gives priority to more rewardable requests. A survey of dispatcher-based solutions can be found in [40]. Although theses solutions are more suitable to load balancing inside the cluster, many techniques can be applied in the case of geographically distributed web servers.

Wang et al. [93] describe a solution that uses the OpenFlow standard to balance the load among replicated servers. In this solution, a centralized load balancer installs *wildcards* rules in the switches to direct requests for groups of clients to different servers according to their capacities.

Server-based load balancing for distributed web services, that is the focus of this paper, has been studied by many researchers. Cardellini and Colajanni [17] compare the performance of centralized versus distributed control algorithms for the activation of the load balancing mechanisms, for the localization of the destination servers, and for the selection of the requests to be redirected. The study concludes that there is a trade-off between the slightly better performance of centralized algorithms and the lower complexity of distributed ones.

Chatterjee et al. [20] present a load balancing solution that redirects requests based on server capacity, server load, and size of the requested documents. The status of all servers is centralized by a single entity. This global information is used by overloaded replicas to redirect requests to the lightly loaded one.

The solution proposed by Ranjan and Knightly [74] considers both CPU load and network latencies to reduce the system response time. If a request arrives at web server k, then the objective is to dispatch the request to a web server j satisfying $min(2\Delta_{kj} + T_j)$, where Δ_{kj} is the network latency between k and j and T_j is the estimate of service time in j.

Andreolini et al. [6] compare a number of load balancing solutions that vary according to their compliance with four properties of autonomic systems: decentralization of control, information collection and reflection, adaptation to a changing environment, and loosely-coupled collaboration. Their results show that the integration of these concepts into the load balancing solution leads to the improvement of the stability and robustness of the system.

Pathan, Vecchiola, and Buyya [71] propose a solution similar to the one proposed by Ranjan and Knightly [74]. The main difference is that, in the former, when a request is redirected, the original client needs to resend the request to the other server replica. In the latter, the redirecting replica intermediates the communication between the end client and the other replica by forwarding the request. Therefore, the solution proposed in [71] needs to estimate the network latency among end clients and server replicas, which is much more difficult than estimate the network latency among server replicas.

Ardagna et al. [7] combine capacity allocation techniques for geographically distributed cloud sites with a load redirection mechanism to minimize the costs of allocated virtual machines, while guaranteeing QoS constraints. The load redirection mechanism is based on workload prediction and uses optimization techniques to determine the amount of load that should be served locally and the amount of load that should be redirected to other sites.

Our solution minimizes the system response times considering the load status of the web servers and network latencies, as [71] or [74] but, in contrast to related work, the novelty in our work can be summarized as: (i) the proposition of a remote resource reservation mechanism that precludes overloaded servers from overloading lightly loaded ones, and (ii) the use of a heuristic based on this mechanism that allows servers to tolerate abrupt load peaks.

7.3 Remote Resource Reservation

In this section we introduce our load balancing solution and motivate our design decisions. Next, we describe the middleware and the heuristics that support the remote resource reservation.

7.3.1 Overview

The response time of a request redirected from a server to another is affected by two factors: the latency between them, and the time the remote server takes to process the request. Intuitively, the best choice would be to redirect requests to the closest lightly loaded server (in terms of latency, not physical distance). However, this intuitive policy may be very inefficient if the redirections end up overloading the remote server. For example, consider the situation illustrated in Figure 7.1. There are three servers with the same capacity, let us say, 100 requests per second (r/s). The gray bars represent the average load incoming over the time to each server. In $t1$, server A becomes overloaded and needs to redirect elsewhere 40r/s. Let us consider that B is the closest lightly loaded server. In time $t2$, A starts redirecting 40r/s to B that then becomes overloaded. As a consequence, it multicasts an alarm message to inform its peers that it is overloaded ($t3$). Once A receives this alarm, it stops redirecting requests to B and starts redirecting its excess workload to the next closest lightly loaded server C at $t4$. Now, C becomes overloaded.

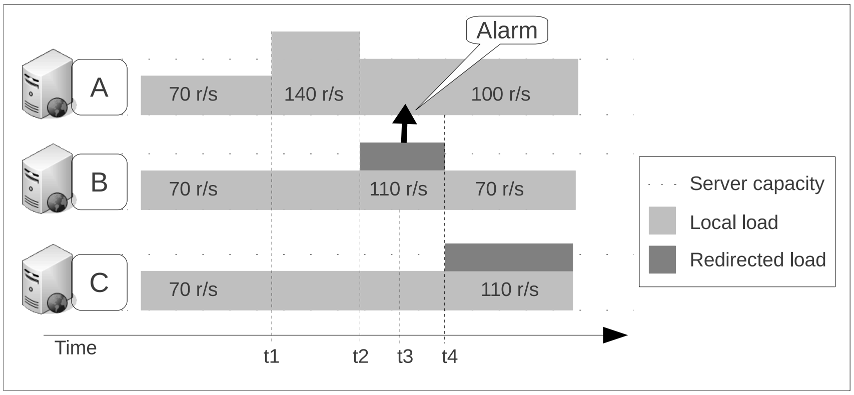


Figure 7.1: The intuitive policy not always gives the best result.

The situation just described (Figure 7.1) can happen because the servers share only a binary information “overloaded or not”. This policy can be improved if the servers also inform the capacity they can offer to the others. In this case, the server A could split its exceeding load between B and C. However, even this improved policy may be inefficient in some cases. For example, let us consider the situation in Figure 7.2. In $t1$, the servers inform the capacity they can lend. Next, in $t2$, A becomes overloaded. It knows the capacity of B and C and appropriately split its exceeding load between them ($t3$). When D becomes overloaded ($t4$), it also knows the capacity of B and C, but it does not know if another server is already redirecting requests to B neither how much redirected load B might be already receiving. Thus, since D's exceeding load fits to the last B's capacity information, D starts redirecting to B ($t5$), which then becomes overloaded.

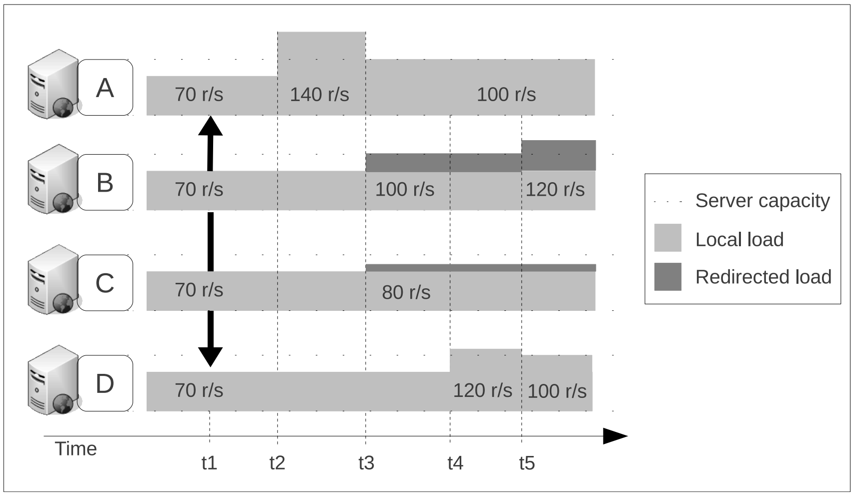


Figure 7.2: Improved intuitive policy.

Our load balancing policy tries to minimize the service response times by conditioning overloaded servers to redirect their excess load to the closest lightly loaded servers available *without overloading them.* The heuristic underpinning our solution is based on a reservation policy that allows overloaded servers to reserve capacity of remote servers before redirecting requests to them. The amount of capacity reserved for a server cannot be shared with any other set of servers. The implementation of this policy in our middleware, the Resource Broker (RB), allows servers to share their spare capacity among the set of servers that execute the application while avoiding the situation exemplified in Figure 7.2.

Figure 7.3 shows an example of the execution of our load balancing policy. Again, let us consider that A and D are overloaded servers and B is the closest lightly loaded server. Suppose that A becomes overloaded first, in $t1$, and then starts to negotiating resources with B ($t2$). Since B is not able to serve its extra load, A also negotiates resources with C. In $t3$, A starts to redirect 30r/s to B and 10r/s to C. Next, in $t4$, D becomes overloaded and starts negotiating resources with B ($t5$). Since B is not able to serve its extra load, D negotiates resources with C. Finally, in $t6$, D starts redirecting 20r/s to C.

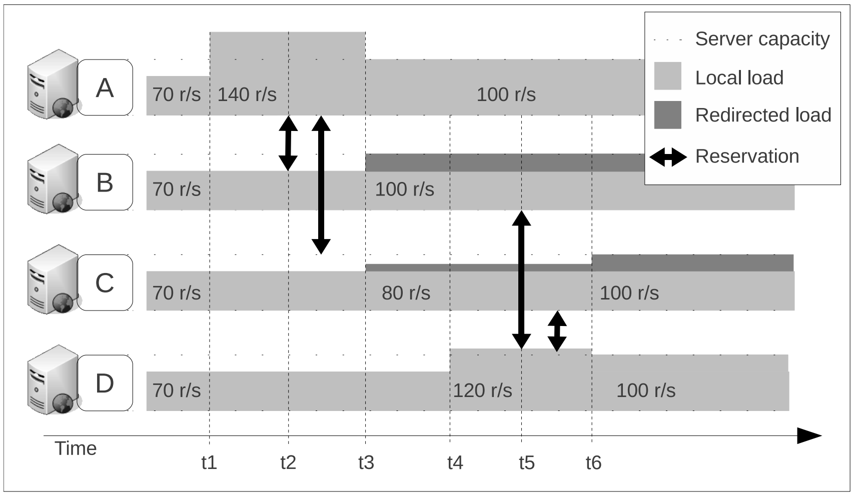


Figure 7.3: Remote Resource Reservation.

Our load balancing policy has been implemented as a heuristic (Section 7.3.3) that is performed by a resource broker middleware to manage the aggregate resource capacity available for the geographically distributed servers (Section 7.3.2).

7.3.2 The Resource Broker

The resource broker (RB) is the middleware we have created to manage the trade of resources among a set of servers. From the perspective of the RB, servers can play two different roles: providers and consumers (Figure 7.4). Providers are lightly loaded servers that can offer resources to other servers. Consumers are overloaded servers that consume resources offered by providers. Any server can act as a provider or as a consumer, according to its current workload. Hence, it can move from provider to consumer and vice-versa as many times as dictated by the changes in its workload.

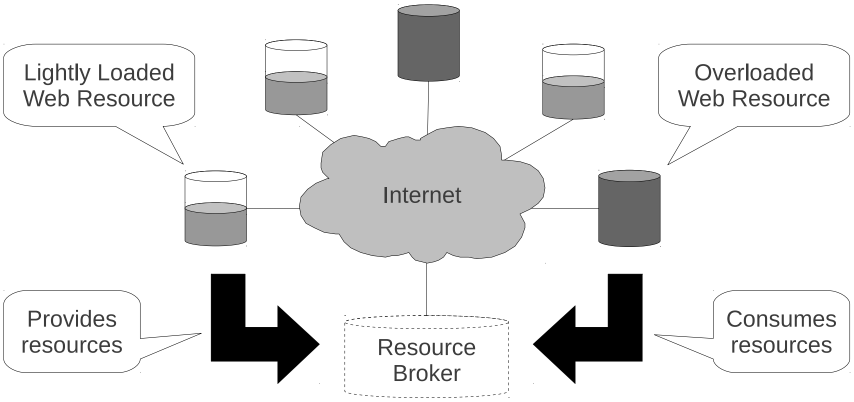


Figure 7.4: Resource Broker conceptual operation.

The RB makes the exchange of resources transparent to individual servers, that is, individual servers see the aggregated capacity of the system but do not know which server is actually a provider. Consumers can allocate resources through the RB to complement their local capacity and, thus, make them able to serve their workload.

The RB manages the amount of resources provided and consumed by the servers. The capacity requested to the RB by a consumer is equally divided among the minimum number of nearest providers available, in order to try to minimize network latencies.

The middleware that implements the RB is composed of two components: the RBMaster and the RBNodes. The RBMaster is the core of the middleware. It is responsible for allocating resources to the consumers and also for reallocating resources when the status of providers changes. In order to perform this, it maintains information about: the status of the servers, the network latencies among the servers, the amount of resources offered by each provider, the amount of resources allocated to each consumer, and the mapping between resources and actual resources providers. Since the RBMaster is a single point of failure, it should be replicated in order to guarantee its availability in the presence of partial failures.

The allocation of resources is modeled as an optimization problem that is solved by the RBMaster using linear programming. The linear model can be described as:

Objective function:

$$\text{Minimize:} \quad \sum_{j=1}^{p} \sum_{i=1}^{c} R_{ij} * L_{ij}$$

Constraints:

$$\text{Subject to:} \quad \sum_{j=1}^{p} R_{ij} \geq D_i \qquad , \quad 1 \leq i \leq c$$

$$\sum_{i=1}^{n} R_{ij} \leq O_j \qquad , \quad 1 \leq j \leq p$$

where, R_{ij} is the amount of resources, in requests per second, the consumer i should reserve from provider j; L_{ij} is the mean network latency between the consumer i and the provider j; p and c are the number of providers and consumers, respectively; D_i is the demand of consumer i (in r/s); and O_j is the amount of resources offered by provider j (in r/s).

The objective function is to find R_{ij}, with $1 \leq i \leq c$ and $1 \leq j \leq p$, that minimizes the mean latency of redirections. The first constraint ensures that the aggregate load redirected by a consumer i is served. The second constraint ensures that no provider will receive an aggregate redirected load higher than the capacity it offered. When the linear program does not converge to a solution, the requests for allocation of resources are refused and the redistribution of resources is postponed.

The RBNodes are software agents that run on every server. They monitor the server workload and communicate with the RBMaster to offer local resources or to allocate aggregated resources. The RBNode also monitors the network latencies between the local server and all remote servers and reports this information to the RBMaster.

Figure 7.5 shows the interaction between a consumer and the RB middleware. The consumer server communicates with the local RBNode to allocate resources and to submit load to them. In order to allocate the resources, the RBNode interacts with the RBMaster through the messages shown in Table 7.1.

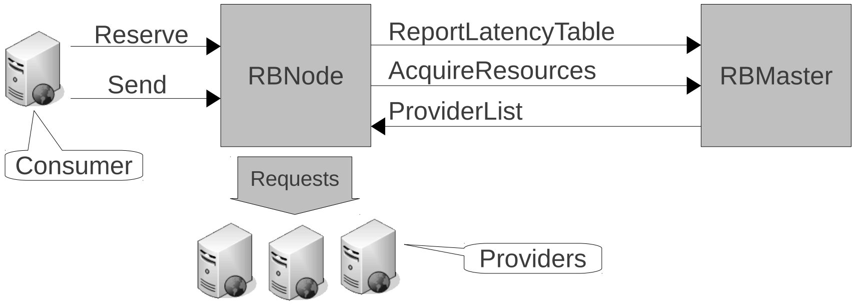


Figure 7.5: Interaction between a consumer and the middleware.

When a load request is submitted to the RBNode, it distributes the requests among the resource providers following a set of probabilities calculated as follows:

$$P_i = \frac{C_i}{C_{aggregated}}$$

where P_i is the probability of sending a request to the provider i; C_i is the capacity offered by server i; and $C_{aggregated}$ is the total capacity allocated in the RBMaster for the server. This probabilistic distribution helps to avoid abrupt accumulation of requests in the income queues of the providers.

Table 7.1: Messages for Resource Allocation

ReportLatencyTable	Sent by RBNodes to report the estimate of latencies among the local server and the others.
AcquireResources	Sent by RBNodes to request the allocation of an amount of virtual resources (in requests per second) or to modify the amount already allocated. If the invoking RBNode is a provider, the RBMaster sets it as consumer and reallocates virtual resources for all consumers that were consuming resources of this provider. Otherwise, if the invoking RBNode is a consumer updating its value amount of virtual resources, the RBMaster reallocates the virtual resources to it.
ProviderList	Sent by RBMaster to inform the list of providers that will provide virtual resources to the RBNode. This list also informs the maximum load that each provider is allowed to serve. The list is chosen regarding the latencies from providers to the consumer and is ordered by the network latency. In order to minimize response times, the RBNode should use all capacity of the closest provider before submitting load to the next one.

Figure 7.6 shows the interaction between a provider and the Request Broker. The provider communicates with the local RBNode to announce its spare capacity. The RBNode receives requests incoming from consumers and forwards them to the local server. In order to offer its spare capacity, the RBnode interacts with the RBMaster through the message described in Table 7.2.

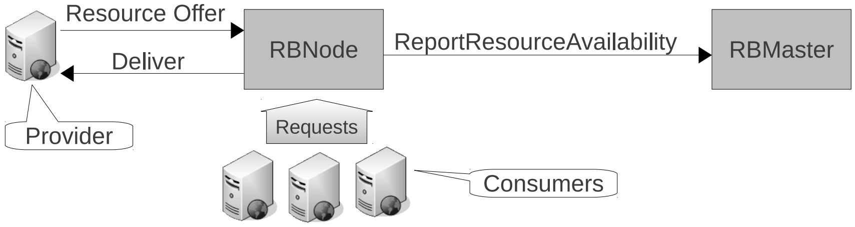


Figure 7.6: Interaction between a provider and the middleware.

Table 7.2: Message for Resource Sharing

ReportResourceAvailability	Sent by RBNodes to announce the amount of resources (in requests per second) they are able to offer or to update the value amount already offered. If the invoking RBNode is a consumer, the RBMaster sets it as provider and disposes any virtual resource it has allocated. Otherwise, if the invoking RBNode is a provider reducing the amount of resources it is sharing, the RBMaster reallocates virtual resources for all consumers that share resources of this provider.

Another function of the RBNode is to perform admission control for the provider. It guarantees that consumer redirections do not exceed the consumer reserved quota. If a consumer redirects load larger than its quota, the exceeding requests are enqueued. In order to avoid underutilization of resources, whenever free execution slots can not be filled with requests from not entirely consumed quotas, enqueued requests are executed in a round robin way.

7.3.3 Load Balancing Heuristics

The Resource Broker solves the problem of choosing the remote servers to which to redirect requests. The selection policy has to be combined with some load conditioning policy that dictates when to redirect requests. In our solution, the servers apply the following heuristic. First, the server calculates:

$$P_{local} = \frac{C_{local}}{C_{local} + C_{reserved}}$$

where P_{local} is the probability of executing an incoming request locally; C_{local} is the local capacity (in r/s); and $C_{reserved}$ is the capacity reserved with the Resource Broker.

Thus, according to P_{local}, the request is served locally or is sent to the Resource Broker. This probabilistic decision scatters the requests over local and remote resources rather than exhaust the local resources before redirecting. Such strategy avoids the accumulation of requests in the local queue.

Another decision problem is related to the amount of remote resources that an overloaded server should try to allocate. This decision depends on the workload of the server. In our solution, we use a sliding average of the request rate to detect significant load changes. The remote resource allocation is dynamically adjusted on the basis of the average request rate.

An important factor that can affect negatively the responsiveness of the servers to workload surges is the communication delay (latency) imposed by the wide area network on the message exchanges between the RBNodes and the RBMaster.

In order to deal with this problem, we propose the use of a capacity safety margin. Each server must have spare resources (CPU, memory, disk, etc) beyond the amount needed to support its average request rate. The safety margin is amount of capacity, in terms of requests per second, a server holds beyond its incoming average load to sustain its performance even in the presence of abrupt workload changes. It helps the server to tolerate peaks of load until the remote resource allocation can be adjusted. For example, suppose that a server with capacity Cr/s and capacity safety margin Sr/s is receiving an average load of Lr/s. If $L < (C - S)$, the server can offer $(C - (L + S))$r/s. If not,

```
Definitions:
C: Server capacity (in Requests per Second - r/s)
L: Last significant average load change (in r/s)
A: Last average load sample (in r/s)
S: Safety margin size (in r/s)
ST: Load sample time (in seconds)
SAWS: window size for load sliding  average
(in seconds)
th_inf: Threshold for significant load decrease
(in r/s)
th_sup: Threshold for significant load increase
  (in r/s)

On each ST seconds:
  A = average request rate of SAWS seconds
  IF (A < (L-th_inf)) or
     (A >= (L+th_sup)) THEN
    L = A
  ENDIF
  IF L < (C-S), THEN
    Offer (C-(L+S))
  ELSE
    Allocate ((L+S)-C)
  ENDIF
  RETURN
```

Figure 7.7: Remote resource allocation heuristic.

the server has to allocate $((L + S) - C)$ from the Resource Broker. The pseudo-code in Figure 7.7 describes the heuristic used to manage the remote resource allocation.

7.4 Simulation

In order to evaluate our solution, we have implemented a simulator using the CSIM for Java[3], a discrete event simulation framework. We have used the Gnu Linear Programming Kit (GLPK)[4] to implement the linear programming routine used to optimize the allocation of resources. The following sections detail our simulation setup.

7.4.1 Internet Latencies

We have considered a scenario with six servers distributed over the world (Figure 7.8): one in South America (S1), one in North America (N1), two in Europe (E1 and E2), and two in Asia (A1 and A2).

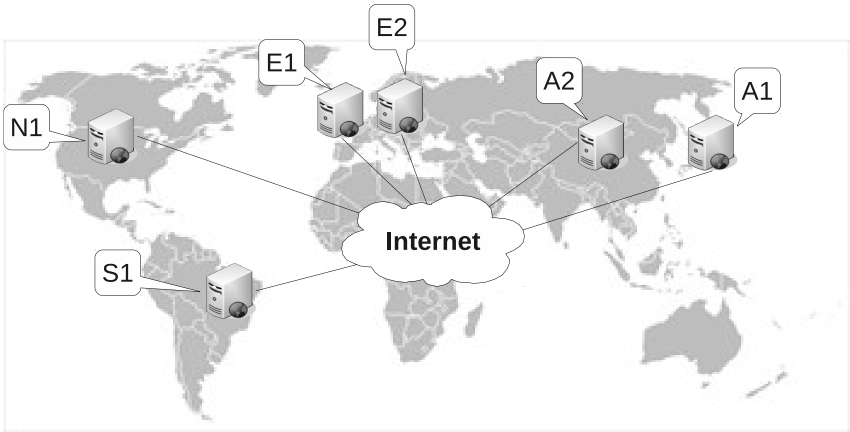


Figure 7.8: Geographical distribution of servers.

We adopted the average of the latencies (half of the round-trip time) $t \pm 5\% t$, measured on real hosts of PlanetLab[5] in Brazil, USA, Belgium, Austria, Japan, and China to simulate the latencies among the servers (Table 7.3). We also consider that servers of a given region execute primarily the requests of clients of that region and that the average latency between a client and its regional server is 10ms.

[3] http://www.mesquite.com/

[4] http://www.gnu.org/software/glpk/

[5] http://www.planet-lab.org

Table 7.3: Table of latencies among the replicas (ms).

	S1	N1	E1	E2	A1
N1	89				
E1	138	48			
E2	140	58	18		
A1	193	109	151	162	
A2	272	156	114	122	68

In order to take into account the latency of the TCP protocol, we adopted the analytic model proposed by Cardwell et al. [18]. This work extended previous models for TCP steady-state by deriving models for two other aspects that can dominate TCP latency: the connection establishment three-way handshake and the TCP slow start [77]. Therefore, the model proposed by Cardwell et al. can predict the performance of both short and long TCP flows under varying rates of packet loss.

In practice, the TCP model estimates the time needed to transfer data from one endpoint to another in terms of: (i) the size of the data to be transferred; (ii) the network latency between the two endpoints; and (iii) the packet loss probability.

7.4.2 Load Generation and Web Servers

We have used the PackMime Internet traffic model [1, 15] to generate HTTP traffic in our simulations. The PackMime has been designed from a large-scale empirical study of real web traffic and has been implemented in the *ns-2*[6], a well known network simulator. In order to use the model in our simulations, we have implemented it in Java.

PackMime allows the generation of both HTTP/1.0 and HTTP/1.1 traffic. The intensity of the traffic is controlled by the rate parameter, which is the average number of new connections started per second. The implementation provides a set of random variable generators that drive the traffic generator. Each random variable follows a specific distribution. The Java implementation of PackMime uses the same distribution families and the same parameters described in [15]; the random variables implemented are: (i) inter-arrival time between consecutive connections, (ii) number of pages requested in the same connection (if using HTTP/ 1.1), (iii) number of objects embedded in a page, (iv) sizes of files (pages and objects), (v) time gap between page requests, and (vi) the time gap between object requests. The random variable used to generate the transmission delays (RTT) was substituted by the model proposed by Cardwell et al [18]. The random variable used to model the execution time was substituted by a queueing computing

[6] http://www.isi.edu/nsnam/ns/

element of CSIM.

We defined five different workload profiles, shown in Table 7.4, with different rates of number of client sessions started per second. These profiles, or variations of them, were used in our experiment to assess the behavior of the different load balancing policies at different levels of system load.

Table 7.4: Workload profiles for testing different levels of system utilization.

Profile	Sessions/Second	Capacity Utilization
A	1400	$\approx 70\%$
B	1500	$\approx 75\%$
C	1600	$\approx 80\%$
D	1700	$\approx 85\%$
E	1800	$\approx 90\%$

In order to divide the load generation between different domains, we adopted a solution similar to the one proposed by Colajanni and Yu [24]. In the paper, the authors propose to divide clients into domains according to the Zipf's distribution, where the probability of a client to belong to the ith domain is proportional to $1/i^x$. This solution was motivated by previous work that demonstrate that if one ranks the popularity of client domains by the frequency of their accesses to a server, the size of each domain is a function with a big head and a very long tail. So, in our simulations, if we adopt the profile A, for example, 1400 (the total amount of sessions per second) is divided into fractions whose sizes follow a Zipf distribution, and then, these fractions are randomly assigned to one of the domains. The aggregated amount of sessions per second assigned to a domain defines the workload that is generated in that domain.

We have modeled each computing element used to assemble servers as a queueing system with a fixed service time of 10ms and a capacity of 100r/s. The maximum queue size is 500 requests. While the queue is full, the computing element drops the requests it receives from clients. The request inter-arrival time distribution is defined by the PackMime model. In our simulations, we have considered a scenario with six servers, each one comprised of a cluster of 4 computing elements that provide an aggregate capacity of 400r/s.

7.4.3 Measurements

A main effect of a poor load balancing is the accumulation of requests in the queues of the overloaded servers. As a consequence, there is an increase in the response times perceived by clients, and, in the worst case, the queues become full and servers start dropping requests. Considering this, we decided to evaluate the load balancing solutions in terms of the average of response times and the number of dropped requests.

In our experiments, each simulation was executed during 1 hour of simulation time and we started gathering statistics after the first 5 minutes (warm up) of simulation and stopped 5 minutes prior to the end of the simulation, obtaining simulations whose steady state lasted 50 simulation minutes. We have saved the average of the response times for the requests and the number of dropped requests for each steady period of simulation. Besides, we have also collected measurements of the utilization of the servers and the number of request redirections.

The number of simulations, 35 for each experimental setup, and their duration, 50 minutes of steady state, were designed to guarantee a 95% confidence level with different levels of accuracy for each experiment [48]. The specific accuracy and confidence level are indicated along with each experiment; they always guarantee that the conclusions drawn from the experiments are based upon statistically significant evidence. Each simulation used a different seed for the random number generators. The results are summarized as the average of the 35 averages of response times, with a 95% confidence interval. Similarly, we have also calculated the average of the number of dropped requests for the 35 simulations, with a 95% confidence interval. In the remaining of the text, unless explicitly mentioned, when we refer to the average of response times and to the average of dropped requests, we will be referring to the average of the 35 values gathered from the 35 independent simulations.

We have used one-way ANOVA [31, 82], with a 95% confidence level, to verify whether the differences among the average response times for distinct load balancing setups were significant; the results with a p-value < 0.001 are taken into consideration in the analysis of the results. Additionally, significance of the difference between two average response times where calculated using *bootstrapping* (10,000 resamplings) with a 95% confidence level [82].

7.5 Evaluation

In order to evaluate the effectiveness of the load balancing implemented by the RB, we have designed four sets of experiments. In the first set, we have investigated the effectiveness of the solution proposed using different safety margins. In the second set, we have compared our solution to other four well-known load balancing solutions. The third set of experiments has been designed to assess how our solution deals with sudden workload changes. The last set of experiments assessed the scalability of the RB.

7.5.1 Sensibility to Safety Margins

As explained before, the selection of a safety margin is an important parameter of our load balancing policy. So, to assess the sensibility of our solution to different safety margins, we run simulations with four different safety margins: 10% (RB-SM10), 15% (RB-SM15), 20% (RB-SM20), and 25% (RB-SM25) of the workload. These safety margin setups were simulated for each of the five workload profiles listed in Table 7.4.

Figure 7.9 shows the average response times for each combination of safety margin and workload. The one-way ANOVA and bootstrapping show that, for workload A, all safety margins did not present significant differences in the average response times; for the details see Annex A, Tables 7.7 and 7.8. This result is explained by the fact that all servers have enough processing capacity to deal with workload A regardless of the safety margin setup.

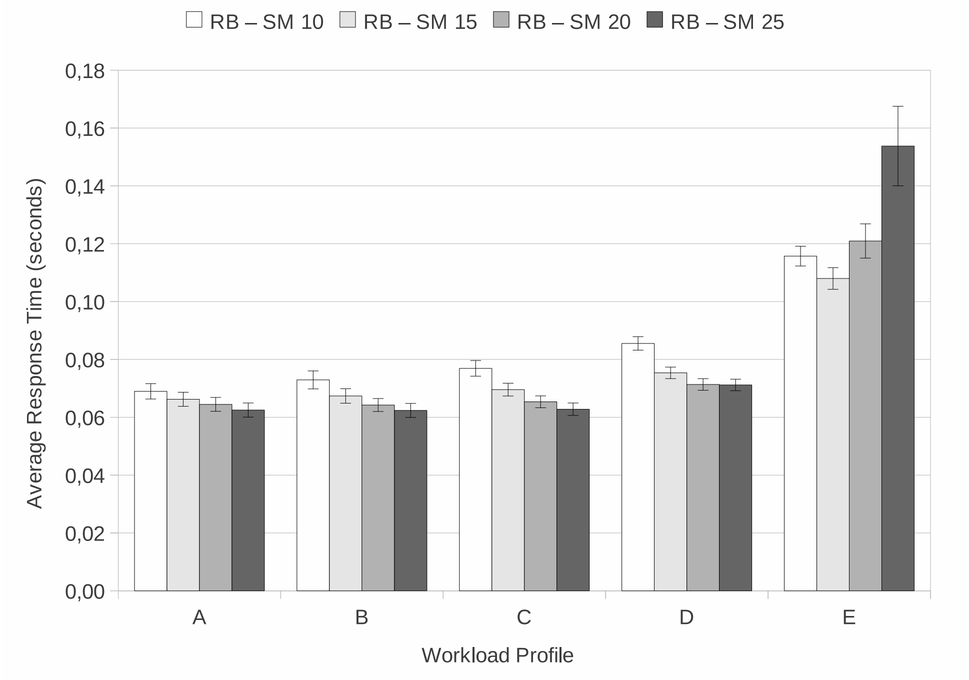


Figure 7.9: Average of response times for different safety margins.

For workloads B, C, and D (Figure 7.9), we can identify significant differences in the average response times as a function of the safety margins (Tables 7.7 and 7.8). As we can see, there is an inverse correlation between safety margin and the average response times; the larger the safety margins the smaller the average response times. The main reason for this result is related to the number of requests present in the local queue of the lightly loaded servers. Although the lower safety margin (RB-SM10) was enough to allow them to serve their average workload, they were not enough to reduce the effect of load peaks as they caused an increase in the number of requests waiting at the local queues of the servers, thus increasing their individual response times and with a negative effect upon the overall efficiency of the system. The larger safety margin prevents the increase of the local queues, thus allowing better response times. In the cases where the system has enough resources to provide to other servers, despite the maintenance of a higher safety margin, the overloaded servers are not substantially affected.

When the system is subject to a heavier load (Figure 7.9, Workload E), the tendency observed for lighter workloads is not observed. The largest safety margin (RB-SM25) gives the worst response times. The reason for this result is that when the exceeding aggregate capacity of the system is not enough to support the aggregate safety margin, the providers tend not to offer resources to consumers and there is a lack of resources to serve their exceeding load. As a result, there is an increase in the size of server queues, an increase in the response times, and also an increase in the number of dropped requests (Figure 7.10). The figure shows that with a safety margin of 25% of the workload (RB-SM25) the number of dropped requests is much higher than the number of dropped requests for the others. The number of dropped requests for the workloads A, B, C, and D (Table 7.4) is negligible, so the values have been omitted.

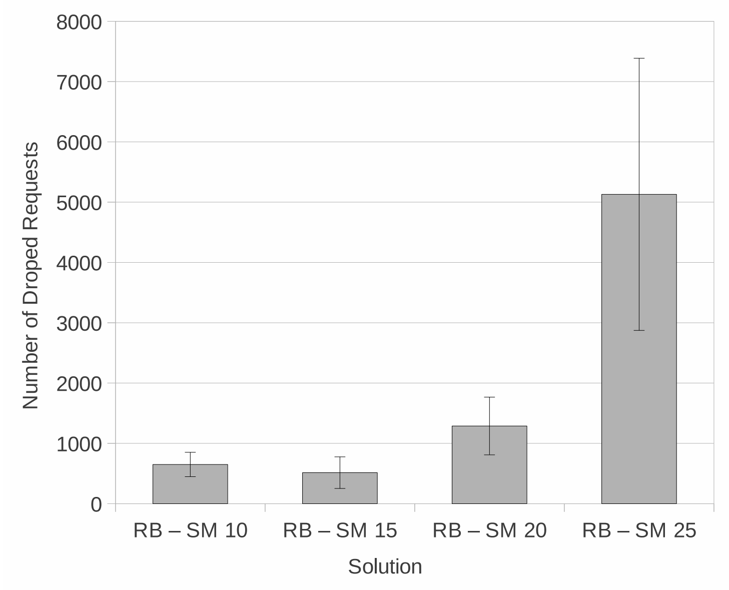


Figure 7.10: Number of requests dropped for different safety margins (Workload Profile E).

The results presented in Figure 7.9 indicate that the choice of the best safety margin becomes critical as the workload increases. Overall, it is possible to see that the value of the best safety margin varies according with the workload, as expected. The results for Workload E hint that there is an optimum value for the safety margin. For a given system setup, taking into account periodical or seasonal workload patterns, it is probably possible to devise an algorithm that uses the history of safety margins and response times for two purposes: (i) dynamically adjust the number of active servers as the workload varies, and (ii) offer a recommendation on when extra servers have to be added to data centers to meet demand without sacrificing the quality of the service (response times).

Since the 15% safety margin (SM15) has resulted in very similar response times for workloads A, B, C, and D, and has given the best result on workload E, we have adopted SM15 as the safety margin used in the setup of the experiments discussed in the next sections.

7.5.2 Comparative Study

We have compared our solution with four other load balancing solutions:

- Round Robin (RR): Overloaded servers redirect exceeding requests to one of the other servers, in a rotative way;

- Round Robin with Asynchronous Alarm (RR-AA): Improves the simple round robin policy by considering the load status of remote servers. Overloaded servers broadcast an alarm informing their status. This way, all overloaded servers can remove each other from their round-robin list;

- Smallest Latency (SL-AA): Overloaded servers redirect exceeding requests to the closest lightly loaded remote server. Overloaded servers use asynchronous alarm to inform their status;

- Least Used (LU): Every server periodically broadcasts a control message reporting its load. Overloaded servers redirect exceeding requests to the least used one.

The use of asynchronous alarms in RR-AA and SL-AA was motivated by [25] that has reported good performances of similar solutions. RR-AA and SL-AA were configured to accept redirected requests if their average load is smaller than 85% of the server capacity. The average load used in the decisions of RB, RR-AA, and SL-AA where calculated in a window of 30s. In LU, the servers were configured to report their current load every second.

Figure 7.11 presents the average of response times for each solution and workload; the details can be seen in the Annex, Tables 7.9 and 7.10. As the results show, LU presented the worst response times for all workloads. This happened because the periodicity of the load report does not provide information on the status of the servers fast enough to avoid the overloading of the selected server (the least used one). When overloaded servers perceive that the current least used server is another server, the previous one has already become overloaded, and then, despite the frequent communication of load reports, average response times increase.

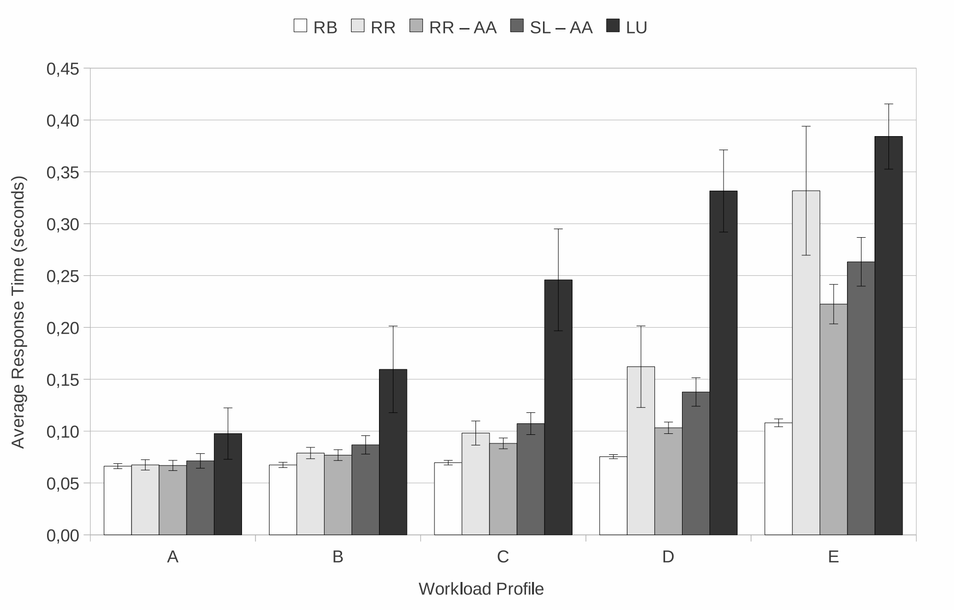


Figure 7.11: Average response time for different load balancing solutions.

For workload A, the use of policies RB, RR, $RR - AA$, and $SL - AA$ have given statistically identical average response times. This result indicates that for a given system setup if the workload is light enough, all four solutions present similar performances. However, as the load increases (workloads B to E), we can notice that RB, our solution, presents response times significantly lower than the others and that the difference between the response times returned by RB and those of the other solutions increases as the workload is increased.

In scenario E, where the capacity of the system is almost saturated, RR presented average response time between 0.163s and 0.286s higher than RB. SL-AA presented average response time between 0.132s and 0.179s higher than RB, and RR-AA presented average response time between 0.095s and 0.134s higher than RB. Considering that, in this scenario, the average response time of RB was 0.107s, the second best solution (RR-AA) presented a response time, at least, 88% higher.

The poor performance of RR was expected, since it ignores the state of remote servers when selecting a server to which to redirect its exceeding load. So, overloaded servers receive as much redirection as lightly loaded servers. Both RR-AA and SL-AA rely on a reactive strategy that raises an alarm when a server perceives a state change. Due to the internet latencies, the alarm does not provide a reaction in time to avoid the overloading of the servers. We can notice, however, that RR-AA presented a performance slightly better than SL-AA. The reason is that RR-AA tends to distribute the exceeding load more evenly than SL-AA. Different from the others, RB adopts a preventive strategy that distributes the exceeding load among the remote servers through a previous resource reservation.

Table 7.5 shows the utilization of each server for one simulation using workload profile *E*. As the coefficients of variation (COVs) indicate, the variance of server utilization for RB is lower than the variance for the others. This shows that RB has distributed the workload more efficiently among the different servers. A server utilization near the maximum suggests that the server had its queue occupation close to its limit during most of the experiment. This explains the increase in the response times and in the probability of dropping requests due to the occurrence of full queues. For the same workload profile, Figure 7.12 shows the average number of requests that were dropped by each load balancing solution. It is possible to see that the number of requests dropped by RB is negligible when compared to the number of requests dropped by the other solutions.

Table 7.5: Server utilization, Workload Profile E. $0,0 \leq$ utilization $\leq 1,0$.

	S1	N1	E1	E2	A1	A2	COV*
RB	0,91175	0,90625	0,87875	0,90675	0,8875	0,88375	0,016
RRAA	0,785	0,83275	0,96575	0,9905	0,8435	0,9565	0,095
SL	0,788	0,83175	0,9655	0,9895	0,8375	0,957	0,095
LU	0,99925	0,5555	0,9665	0,99075	0,7765	0,95675	0,202
RR	0,6245	0,7195	1,0	1,0	0,9355	1,0	0,188

*COV: Coefficient of Variation

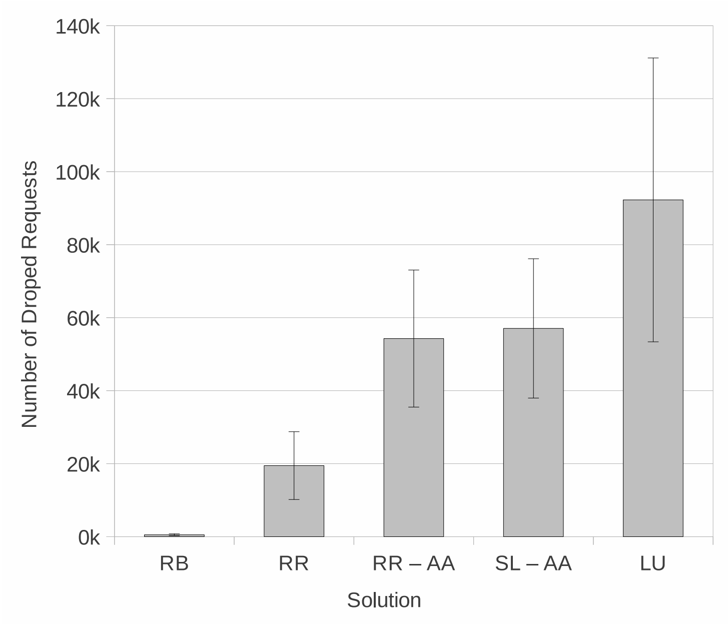


Figure 7.12: Number of requests dropped versus load balancing solutions (Workload Profile E).

7.5.3 Sensibility to Abrupt Load Changes

In order to assess our load balancing solution under abrupt workload changes we have adapted the workload profile E as explained next.

1. We have generated a regular workload of profile E, as described in Section 7.4.2;
2. We have selected the server with the highest local load, let us say Hr/s, and preset it to start the simulation with a load equal to the load of the server with lowest load, let us say Lr/s;
3. During the simulation, we increased the load of the preset server from Lr/s to Hr/s, in fixed steps, and at regular time intervals.

Two workload increment steps, 10% and 20%, were used in the experiments.

In the experiments carried out so far (Section 7.5.2), RR-AA has presented the second best average response times, so it has been selected as the benchmark for the experiments of this section. Figure 7.13 shows the average response times for each combination of load balancing solution and load increment step. In the figure, we can notice that a larger workload increment (peak) causes an increase in the average response times of both RB and RR-AA, nevertheless, RB has performed better in both the 10% and 20% abrupt increases.

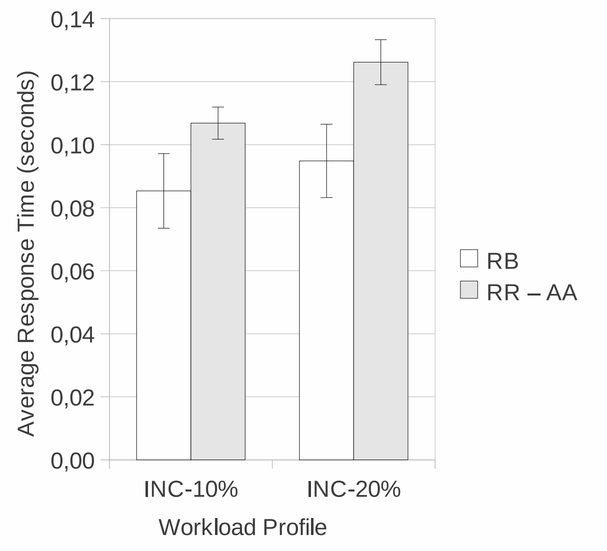


Figure 7.13: Average response time for different load increment steps.

The reason RB performs better (Figure 7.13) can be attributed to the fact that RB distributes the load more evenly among the servers than RR-AA, even when a server is subject to abrupt load increases. This fact is further evidenced by the utilization of the servers, shown in Table 7.6, for one simulation. For all servers, irrespective of increment value, the utilizations achieved in each server using RB are very similar, this is not true for RR-AA. The coefficients of variation summarize this fact very well. For example, the COV for INC-10% RB is less than half the COV for INC-20% RR-AA. As mentioned before, a server utilization near the maximum (1,0) indicates that a server had its incoming queue operating very close to its capacity during the simulation, that increases response times and the probability of requests being dropped due to the occurrence of full queues. Figure 7.14 shows the average number of dropped requests, it is possible to notice that RB dropped much less requests than RR-AA for both 10% and 20% workload increment steps.

Table 7.6: Server utilization for a simulation using different workload increment steps.

	S1	N1	E1	E2	A1	A2	COV*
INC-10% RB	0,81575	0,8195	0,70975	0,8705	0,864	0,7685	0,075
INC-10% RR-AA	0,59125	0,675	0,9665	0,82575	0,83225	0,95675	0,186
INC-20% RB	0,824	0,82875	0,72725	0,86825	0,867	0,77975	0,067
INC-20% RR-AA	0,60825	0,687	0,9665	0,84275	0,833	0,95675	0,176

*COV: Coefficient of Variation

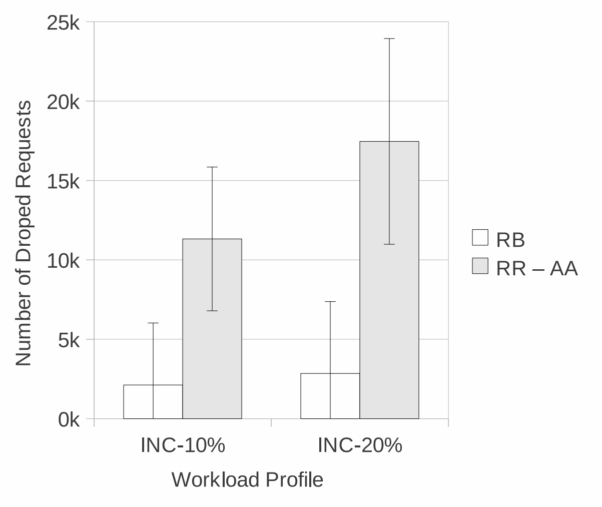


Figure 7.14: Number of requests dropped for different load increment steps.

7.5.4 Scalability

In order to assess the scalability of our solution, we compared RB to RR-AA at four different scales, besides the original 6 servers configuration, we have simulated system setups with 12, 18, and 24 servers. In the simulations, the latencies between each pair of servers were randomly selected from the set of latencies used for the six server scenario (Table 7.3).

The workload of the new scenarios was increased proportionally to the increase in the number of servers, using workload profile E, the heaviest one, as the basis. So, for example, the setup with 12 servers was simulated with a workload corresponding to 3600 (2 x 1800) sessions/second. Accordingly, the scenarios with 18 and 24 servers were simulated with workload profiles set to 5400 and 7200 sessions/second, respectively.

We have designed two variations of the workload profile E, according to the way sessions of the workload were assigned to the servers of the geographical regions. The rationale behind the creation of the variations is that one of them, the Scale-1 variation, should allow us to assess the effect of scale when, despite having overloaded servers, the workload distribution is flatter. The other, the Scale-2 variation, has been designed to concentrate load on a much smaller number of servers, making them very unequally loaded in relation to the other servers of the system. So, although the two workload variations generate the same overall workload, that is, the same number of sessions per second, the Scale-2 workload variation tends to generate scenarios with a smaller number of overloaded servers than the Scale-1 workload variation. However, the variance between the loads placed on servers of the Scale-2 variation is higher than the variance experienced by the servers of the Scale-1 workload variation.

To create the Scale-1 workload variation we have divided the sessions of the original workload E using the Zipf distribution, as described in Section 7.4.2. Each fraction of workload sessions produced by this distribution was replicated k times, where k is the scale factor, 2, 3, and 4, in relation to the original six server setup. Then, the replicated fractions of workload sessions were randomly assigned to the server regions. Differently, to generate the Scale-2 workload variation, the number of workload sessions of the original six server workload was first scaled up to match the scale of the servers (e.g., 3600 for the 12 servers setup). Then, the scaled up workload was partitioned according with the Zipf distribution and the workload partitions were then randomly assigned to the servers of each region.

Figure 7.15 presents the average response times for RB and RR-AA in simulations with the Scale-1 workload variation. This chart shows that RR-AA presented average response times significantly higher than RB in all scenarios. Besides, while RB presented similar average response times regardless of the number of servers, RR-AA presented an increase of response times as the number of servers increases (from 6 to 18 servers). Another evidence that RB performed better than RR-AA is the number of requests dropped due to full queues. As shown in Figure 7.16, RR-AA dropped a much higher number of requests than RB. When the Scale-2 workload variation is used in the simulations the results are similar, that is, RB presents better average response times than RR-AA (Figure 7.17) and also drops fewer requests than RR-AA (Figure 7.18).

An interesting result can be verified when comparisons are made between the results obtained with the Scale-1 and Scale-2 workload variations. In terms of the average response times (Figures 7.15 and 7.17), it is possible to see both RB and RR-AA had similar behavior. The average number of requests dropped tells that RR-AA drops much more requests when subject to the Scale-2 workload variation, so it deals poorly with more variable workloads. In contrast, RB exhibits the same behavior for both the Scale-1 and

Scale-2 workloads; this is further evidence RB's capacity to distribute more evenly the load among the servers due to its use of safety margins.

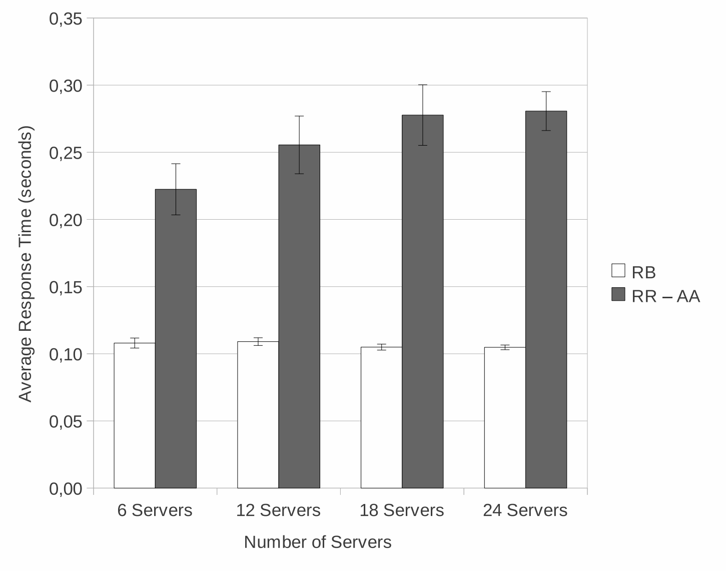


Figure 7.15: Average response time for different number of servers (workload: Scale-1).

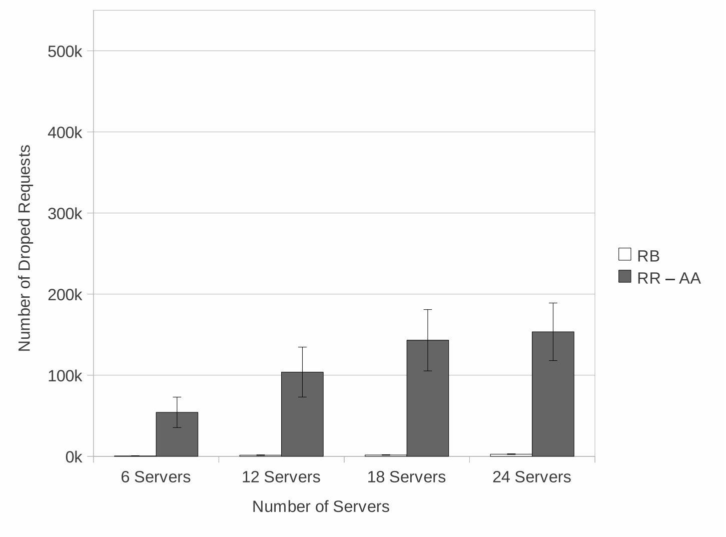


Figure 7.16: Number of dropped requests for different number of servers (workload: Scale-1).

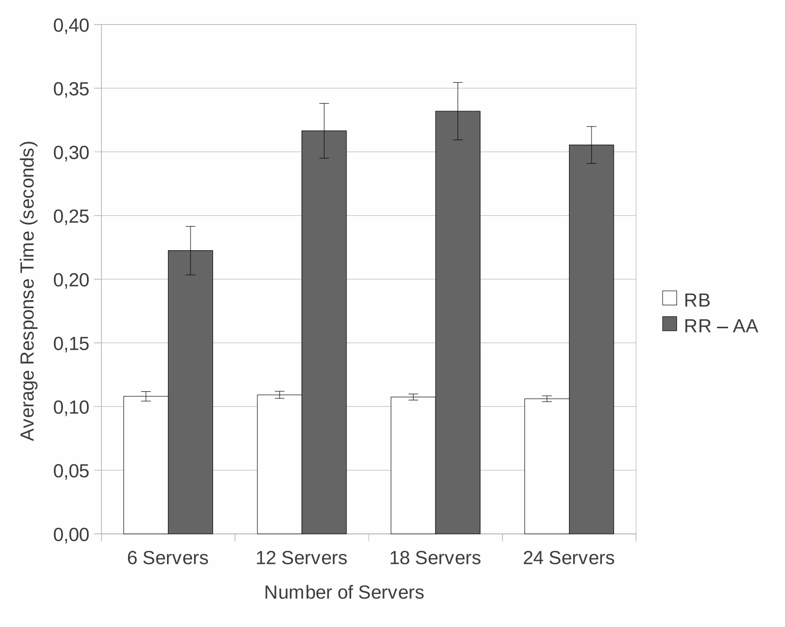


Figure 7.17: Average response time for different number of servers (workload: Scale-2).

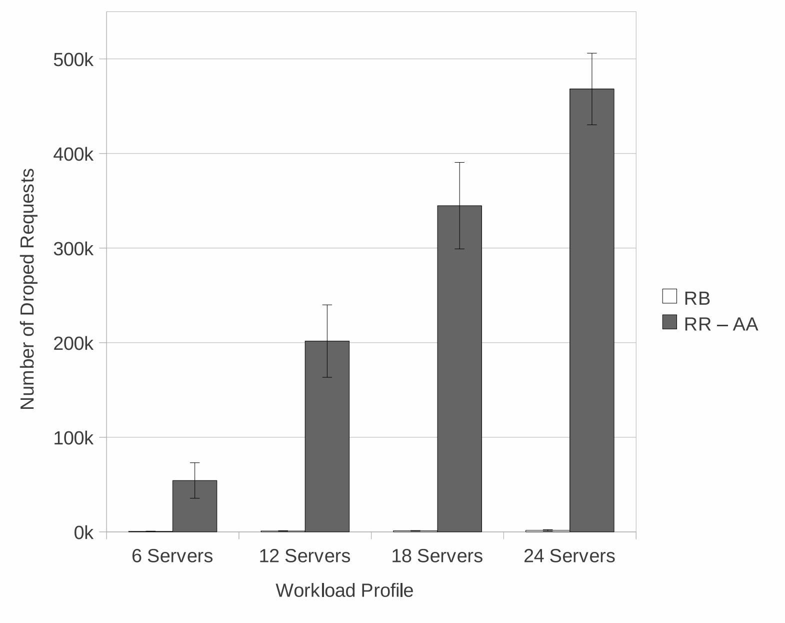


Figure 7.18: Number of different requests for different number of servers (workload: Scale-2).

7.5.5 Summary

Our first set of experiments (Section 7.5.1) has assessed the sensibility of our solution to four different safety margins. The results indicate that there is not a better choice for all situations and suggests that the use of a dynamic safety margin size would be an interesting improvement for RB. The second set of simulations (Section 7.5.2) has compared our approach to four well known solutions under different workloads. The results show that, in all scenarios, our solution presents similar or better performance than the others in terms of response times and number of dropped requests. The best of these four solutions (RR-AA) was selected as the benchmark for the remaining experiments.

The third set of experiments (Section 7.5.3) has assessed RB and RR-AA under abrupt load changes. For these experiments, we defined two scenarios with different load increment steps (10% and 20% of the current load). The results show that our solution performed significantly better than RR-AA in both scenarios. Finally, in the last set of experiments (Section 7.5.4), we have shown that our solution scales up better than RR-AA and that it is less sensitive to the variance of workload among the servers.

7.6 Conclusion

In this paper, we have introduced a new server-based load balancing policy for worldwide distributed web servers that was designed on the basis of a protocol for reservation of remote resources. This protocol and its implementation prevent the overload of remote servers by limiting the amount of load each server can redirect to the others. This strategy presented better results than strategies that react to the overloading of remote servers with simpler remedial actions.

In order to evaluate our solution, we have implemented a simulator based on realistic internet models and real internet latencies. The results obtained through simulation show that our solution is sensitive to the safety margin used to alleviate the effects of abrupt load changes. This suggests that a strategy based on dynamically adapting the safety margins would be an interesting improvement to our load balancing solution.

The simulations demonstrated that our solution presented better performance, both in terms of mean response times and the number of discarded requests, when compared to other well-known solutions, using different workload profiles. Additional experiments allowed us to conclude that our solution has better resilience to abrupt load changes and scales better than the best of the well-known solutions used in the experiments: a round-robin load balancer improved with asynchronous overload alarms (RR-AA).

Future work includes: (i) the implementation and evaluation of a dynamically adaptable, autonomous, safety margin mechanism; (ii) the evaluation of the sensitivity of our solution through an even more comprehensive set of simulations; (iii) the deployment and evaluation of our solution in a real testbed [65].

7.7 Annex A: One-way ANOVA and Bootstrapping

Table 7.7: One-way ANOVA: workload˜safety margins.

Workload	F Value	$Pr(> F)$
A	4.687	0.00379
B	12.203	4.034e-07
C	27.903	3.995e-14
D	39.919	<2.2e-16
E	24.988	6.114e-13

Table 7.8: Bootstrapping (95% confidence intervals for average response times): workload˜safety margins.

Workload	Groups	Inf. Lim.	Up. Lim
A	RB10-RB15	-0.0008	0.0062
	RB10-RB20	0.0009	0.0079
	RB10-RB25	0.0028	0.0100
	RB15-RB20	-0.0015	0.0051
	RB15-RB25	0.0003	0.0070
	RB20-RB25	-0.0014	0.0053
B	RB10-RB15	0.0016	0.0095
	RB10-RB20	0.0049	0.0124
	RB10-RB25	0.0067	0.0145
	RB15-RB20	-0.0001	0.0065
	RB15-RB25	0.0016	0.0084
	RB20-RB25	-0.0014	0.0051
C	RB10-RB15	0.0039	0.0109
	RB10-RB20	0.0083	0.0149
	RB10-RB25	0.0107	0.0175
	RB15-RB20	0.0013	0.0072
	RB15-RB25	0.0037	0.0099
	RB20-RB25	-0.0004	0.0054
D	RB10-RB15	0.0071	0.0132
	RB10-RB20	0.0112	0.0173
	RB10-RB25	0.0113	0.0174
	RB15-RB20	0.0013	0.0068
	RB15-RB25	0.0014	0.0070
	RB20-RB25	-0.0026	0.0030
E	RB10-RB15	0.0027	0.0128
	RB10-RB20	-0.0014	0.0121
	RB10-RB25	0.0243	0.0529
	RB15-RB20	0.0063	0.0199
	RB15-RB25	0.0324	0.0600
	RB20-RB25	0.0185	0.0483

Table 7.9: One-way ANOVA: workload˜load balancing solutions.

Workload	F Value	$Pr(>F)$
A	4.8055	0.00107
B	14.203	5.065e-10
C	35.698	<2.2e-16
D	57.95	<2.2e-16
E	37.402	<2.2e-16

Table 7.10: Bootstrapping (95% confidence intervals for average response times): workload˜load balancing solutions.

Workload	Groups	Inf. Lim.	Up. Lim
A	RRAA-RB	-0.0047	0.0061
	RR-RB	-0.0042	0.0067
	SL-RB	-0.0021	0.0128
	LU-RB	0.0100	0.0589
B	RRAA-RB	0.0040	0.0152
	RR-RB	0.0056	0.0175
	SL-RB	0.0106	0.0290
	LU-RB	0.0545	0.1366
C	RRAA-RB	0.0131	0.0243
	RR-RB	0.0186	0.0421
	SL-RB	0.0276	0.0488
	LU-RB	0.1294	0.2262
D	RRAA-RB	0.0220	0.0338
	RR-RB	0.0514	0.1287
	SL-RB	0.0492	0.0764
	LU-RB	0.2182	0.2964
E	RRAA-RB	0.0954	0.1336
	RR-RB	0.1639	0.2862
	SL-RB	0.1325	0.1789
	LU-RB	0.2460	0.3086

Capítulo 8

Conclusão

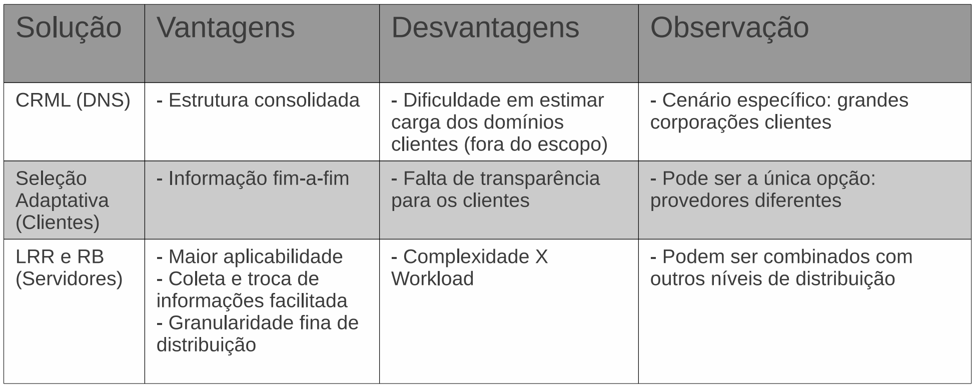
O tema desta tese é a distribuição de carga entre diversos processadores, um problema fundamental da área de sistemas distribuídos. A tese foca este problema no contexto de servidores web geograficamente distribuídos. Neste contexto, as aplicações clientes devem ser dinamicamente atribuídas a réplicas do servidor web que estão espalhadas pela internet. Durante a pesquisa, foram estudadas soluções de três tipos: soluções de distribuição de carga via DNS, via servidores e via clientes. Este estudo envolveu a implementação de uma plataforma de testes baseada em um serviço web real e de um software de simulação. Como resultado da pesquisa, foram propostas novas soluções dos três tipos. Demonstrou-se, por meio de experimentos e simulações, que as soluções propostas apresentam desempenho superior a soluções representativas de suas respectivas categorias.

A comparação entre as soluções propostas será alvo de investigações futuras. Entretanto, as características intrínsecas destas soluções permitem especular sobre suas aplicabilidades em diferentes cenários. As soluções de distribuição de carga via DNS, como a CRML (Capítulo 4), apresentam a vantagem de utilizar o sistema de resolução de nomes, uma estrutura já consolidada e amplamente utilizada. Talvez por esta razão, este tipo de abordagem seja utilizada como primeiro nível de distribuição de carga por grandes corporações, como Google e Akamai. No caso específico de soluções baseadas em DNS que utilizam informações dos clientes, como a CRML, a grande dificuldade é estimar a carga gerada pelos domínios clientes. Existem propostas para a solução deste problema, que não faz parte do escopo da tese, mas não são triviais. Apesar disto, pode-se vislumbrar cenários mais específicos onde a CRML poderia ser usada com sucesso. Por exemplo, um cenário onde um provedor fornecesse serviços para um grupo conhecido de grandes corporações clientes.

A seleção adaptativa de servidores via clientes (Capítulo 5), como outras soluções desta categoria, permite utilizar a informação de latência fim a fim. Esta informação é particularmente útil em soluções de distribuição de carga, já que reflete não só o estado dos servidores, como também da rede. Outra vantagem é que, em alguns casos, esta pode ser a única opção de distribuição de carga viável, como por exemplo, em cenários onde os provedores do serviço web não pertencem a mesma corporação, o que impede uma solução baseada em servidores. Apesar destes pontos positivos, o fato destas abordagens não serem transparentes para os clientes limita sua aplicabilidade.

Dentre as soluções propostas nesta tese, aquelas que realizam a distribuição de carga via servidor (Capítulos 6 e 7) são as que apresentam aplicabilidade mais abrangente. A grande vantagem deste tipo de solução é que elas permitem distribuição de carga com alta granularidade, afinal, todas as requisições chegam a algum dos servidores. Além disso, o provedor do serviço normalmente tem controle total sobre os servidores web, o que não acontece com servidores de nomes e clientes. Portanto, a troca de informações entre os servidores web é mais facilmente implementada do que entre servidores e outros dispositivos (DNS e clientes). Além disso, soluções baseadas em servidores são transparentes para os clientes e podem ser implementadas como um segundo nível de distribuição de carga, em combinação com soluções baseadas em DNS.

A Figura 8.1 sumariza as vantagens e as desvantagens das soluções propostas.



Solução	Vantagens	Desvantagens	Observação
CRML (DNS)	- Estrutura consolidada	- Dificuldade em estimar carga dos domínios clientes (fora do escopo)	- Cenário específico: grandes corporações clientes
Seleção Adaptativa (Clientes)	- Informação fim-a-fim	- Falta de transparência para os clientes	- Pode ser a única opção: provedores diferentes
LRR e RB (Servidores)	- Maior aplicabilidade - Coleta e troca de informações facilitada - Granularidade fina de distribuição	- Complexidade X Workload	- Podem ser combinados com outros níveis de distribuição

Figura 8.1: Vantagens e desvantagens das soluções propostas.

8.1 Contribuições

As principais contribuições desta tese são:

1. A especificação e o desenvolvimento do Lab4WS, uma plataforma que provê um ambiente de testes para soluções de distribuição de carga, preservando propriedades do ambiente real que são importantes para tais mecanismos. O Lab4WS foi construído com base em uma implementação real de um serviço web que segue a

especificação do TPC-W, um *benchmark* para comércio eletrônico bastante conhecido. A plataforma foi validada com implementações de soluções de distribuição de carga de quatro tipos e apoiou o desenvolvimento de uma nova solução baseada em DNS.

2. Um simulador Java para apoiar o desenvolvimento e a avaliação de soluções de distribuição de carga. Este simulador foi construído com base no modelo Packmime de geração de carga, que é utilizado no simulador ns-2, e em dados de latência de rede reais, coletados por meio da plataforma PlanetLab. Além disso, o simulador utiliza um modelo analítico, encontrado na literatura, que estima as latências do protocolo TCP. O simulador apoiou o desenvolvimento de 3 novas soluções para distribuição de carga.

3. Uma nova solução para distribuição de carga via DNS, a CRML, que utiliza informações de carga dos clientes para melhorar a distribuição entre os servidores. Esta solução reduz os efeitos negativos do mecanismo de *caching* do DNS sobre a distribuição de carga por meio da cooperação entre o DNS autoridade (ADNS) e os servidores. A CRML foi comparada a 4 outras soluções de distribuição via DNS e apresentou desempenhos semelhantes ou superiores em todos os experimentos, mostrando ganhos mais significativos em cenários onde a ação negativa do *caching* é maior.

4. Uma nova solução para distribuição de carga via clientes que atribui diferentes probabilidades de seleção para os servidores, de forma adaptativa, levando em consideração os tempos de resposta percebidos pelo cliente. A solução proposta foi comparada a duas outras que representam dois tipos: soluções que distribuem a carga igualmente entre todos os servidores e soluções que selecionam um único servidor de forma gulosa. As simulações mostram que a solução adaptativa apresenta melhores desempenhos tanto em cenários favoráveis ao primeiro tipo, quanto em cenários favoráveis ao segundo.

5. Uma nova solução para distribuição de carga via servidores baseada em limites de redirecionamento de carga – a LRR. Os limites são calculados dinamicamente de acordo com a demanda e a oferta global de recursos. Esta estratégia ajuda a prevenir a sobrecarga dos servidores remotos, refletindo nos tempos de resposta percebidos pelos clientes. A LRR foi avaliada em comparação a duas outras soluções e apresentou melhor desempenho em todos os cenários considerados.

6. Uma extensão da LRR que trata a divisão de recursos entre os servidores sobrecarregados como um problema de otimização e utiliza programação linear para resolvê-lo. Outro aperfeiçoamento foi a inclusão da reserva de recursos, que garante que a

demanda dos servidores sobrecarregados será atendida nos servidores remotos. A nova solução apresentou melhor desempenho quando comparada a 4 outras soluções. Além disso, mostrou melhor escalabilidade e melhor adaptação a mudanças bruscas de carga que a melhor das outras 4 soluções.

7. A especificação de um middleware para compartilhamento de recursos entre as réplicas dos servidores web. Este middleware foi inicialmente proposto como parte da LRR e depois foi estendido para a solução que utiliza reserva de recursos.

8.2 Trabalhos Futuros

Algumas possíveis extensões desta tese são:

1. **Implantação da plataforma Lab4WS em *datacenters* geograficamente distribuídos e em *hosts* do PlanetLab.** O Lab4WS foi validado e utilizado em aglomerados com enlaces que emulam as latências da internet. Entretanto, os componentes da plataforma poderiam ser implantados em *datacenters* distribuídos pela internet, utilizando serviços de nuvem, por exemplo, e em *hosts* do PlanetLab. Isto permitiria a avaliação de soluções de distribuição de carga em um ambiente de internet real. Além disso, abriria possibilidades para pesquisas em áreas correlatas, como replicação de servidores e soluções de *caching*.

2. **Implementação de uma versão distribuída do software de simulação.** A implementação do software de simulação limita a escala dos experimentos pela quantidade de memória física do computador que executa a simulação. Uma versão distribuída permitiria executar as simulações em *clusters*, possibilitando uma escala muito maior.

3. **Avaliação das soluções simuladas na plataforma de testes.** As soluções baseadas em clientes e em servidores propostas nesta tese foram avaliadas por meio de simulações. Um trabalho futuro é implementar estas soluções como módulos da plataforma Lab4WS e avaliá-las em um ambiente real.

4. **Comparação entre as soluções propostas.** As soluções propostas na tese foram avaliadas individualmente por meio de experimentos comparativos com outras soluções do mesmo tipo. Uma extensão da pesquisa seria comparar as soluções de diferentes tipos a fim de identificar os cenários em que cada uma é mais apropriada.

5. **Adaptação da seleção adaptativa de servidores via clientes para solução baseada em servidores.** Nas soluções de distribuição via servidores, os servidores

sobrecarregados podem ser vistos como clientes dos servidores para os quais redirecionam sua carga. Considerando que a seleção adaptativa via clientes, proposta nesta tese, apresentou bons resultados, a adaptação desta solução para servidores parece promissora.

6. **Extensões para soluções baseadas em servidores.** As soluções baseadas em servidores propostas nesta tese podem ser estendidas em diversos aspectos. Um destes aspectos é a diferenciação de requisições por conteúdo. Neste caso, os servidores sobrecarregados poderiam selecionar as requisições a redirecionar de acordo com o conteúdo esperado como resposta e priorizar determinadas classes de requisições. Outro aspecto que pode ser explorado é a granularidade dos redirecionamentos. Nas soluções propostas, considerou-se o redirecionamento no nível de requisição. Outra possibilidade seria o redirecionamento no nível de clientes. Neste caso, todas as requisições de um mesmo cliente são obrigatoriamente atendidas por um mesmo servidor. Isto seria particularmente importante em aplicações que utilizam sessões de clientes, porém, introduzirá outros problemas, como a necessidade de estimar a carga gerada pelos clientes. Um terceiro aspecto a ser estudado é o tipo de informação utilizada pelas políticas de distribuição de carga. Informações mais complexas podem melhorar o desempenho da distribuição de carga, porém, geram sobrecarga para coleta e disseminação dos dados.

7. **Soluções para aglomerados.** Uma continuação natural da pesquisa seria focar as soluções para distribuição de carga voltadas para aglomerados de servidores. Um estudo poderia analisar a viabilidade de utilizar as ideias propostas na tese com foco voltado para este outro contexto. Além disso, este tópico possui importantes aspectos de pesquisa a serem explorados, como por exemplo, a combinação da solução para distribuição de carga com estratégias para elasticidade horizontal do aglomerado e economia de energia.

Apêndice A

Análises Estatísticas Complementares

Este apêndice contém análises estatísticas que complementam os resultados apresentados nas seções 4, 5 e 6. As médias dos tempos de resposta apresentadas neste apêndice correspondem às médias das médias dos tempos de resposta de n experimentos, onde n varia de um caso para outro. Utilizou-se análise de variância (ANOVA) para determinar se a diferença entre as médias dos tempos de resposta é significativa. Apenas diferenças com nível de confiança menor que 0,001 foram consideradas significativas. Além disso, utilizou-se *bootstrapping* com 10.000 re-amostragens para estimar a diferença entre as médias dos tempos de resposta com confiança de 95%.

A.1 Tempo de Resposta para CRML

A Figura A.1 apresenta a média dos tempos de resposta de 8 experimentos para a operação *SubjectSearch*, no cenário com 30% de controle do ADNS, usando 3 soluções diferentes (CRML[1], MRL[2] e RR2[3]). Este cenário é descrito na Seção 4.6.2.

[1] Current Relative Minimum Load

[2] Minimum Residual Load

[3] Two Tier Round Robin

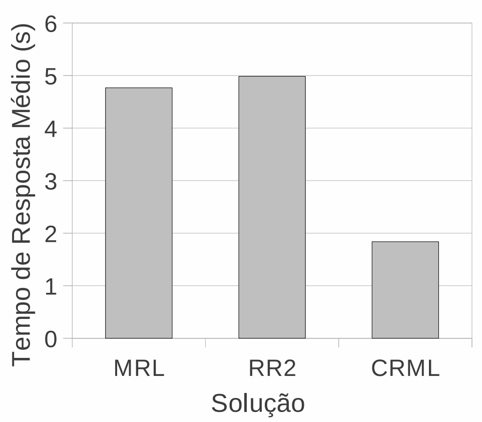


Figura A.1: Tempos de resposta médios para a operação *SubjectSearch*.

A Tabela A.1 apresenta o resultado da ANOVA e seu grau de confiança para as médias dos tempos de resposta da operação *SubjectSearch*, usando CRML, MRL e RR2. A ANOVA mostra que as médias dos tempos de resposta das diferentes soluções são significativamente diferentes.

Tabela A.1: ANOVA para tempo de resposta médio da operação *SubjectSearch*.

F Value	103,97
Pr(>F)	1,277e-11

A Tabela A.2 apresenta as diferenças entre as médias dos tempos de resposta da CRML e das outras soluções, calculadas a partir de *bootstrapping*. O resultado mostra que as médias dos tempos de resposta das soluções MRL e RR2 são significativamente maiores que as médias dos tempos de resposta da CRML.

Tabela A.2: Diferenças dos tempos de resposta médios da operação *SubjectSearch* entre CRML e outras soluções (Cenário com 30% de controle do ADNS).

	Limite Superior	Limite Inferior
MRL	2,591	3,237
RR2	2,647	3,582

A.2 Tempo de Resposta para Seleção Adaptativa de Servidores via Clientes

A Figura A.2 apresenta as médias dos tempos de resposta de 35 simulações que comparam diferentes soluções de seleção de servidores via clientes (RR[4], BS[5] e AD[6]) no cenário favorável à BS (Cenário 1 - Seção 5.5). As médias são apresentadas com intervalo de confiança com nível de confiança de 95%.

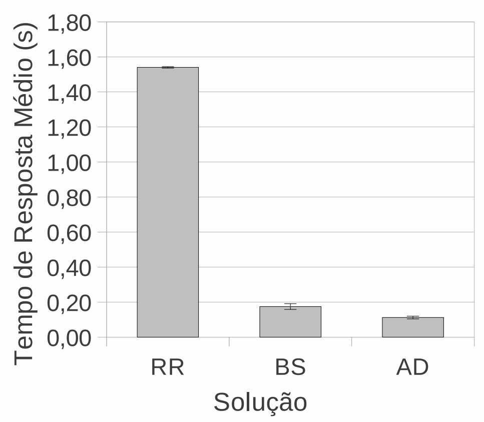


Figura A.2: Tempos de resposta médios para o cenário 1.

A Tabela A.3 mostra o resultado da ANOVA e o seu grau de confiança para as médias dos tempos de resposta da Figura A.2. A ANOVA mostra que as médias dos tempos de resposta das diferentes soluções são significativamente diferentes.

Tabela A.3: ANOVA para as médias dos tempos de resposta no cenário 1.

F Value	21683
Pr(>F)	< 2,2e-16

A Tabela A.4 apresenta as diferenças entre as médias dos tempos de resposta da AD e das outras soluções, calculadas por *bootstrapping*. Os resultados mostram que RR e BS apresentam médias dos tempos de resposta significativamente maiores que as médias dos tempos de resposta da seleção adaptativa (AD).

[4] Round Robin

[5] Best Server

[6] Adaptative Server Selection

Tabela A.4: Diferenças de tempos de resposta médios entre AD e outras soluções, para o cenário 1.

	Limite Superior	Limite Inferior
RR	1,419	1,436
BS	0,044	0,080

A Figura A.3 apresenta as média dos tempos de resposta para 35 simulações que comparam as diferentes soluções de seleção de servidores via clientes no cenário favorável à RR (Cenário 2 - Seção 5.5). As médias são apresentadas com intervalo de confiança com nível de confiança de 95%.

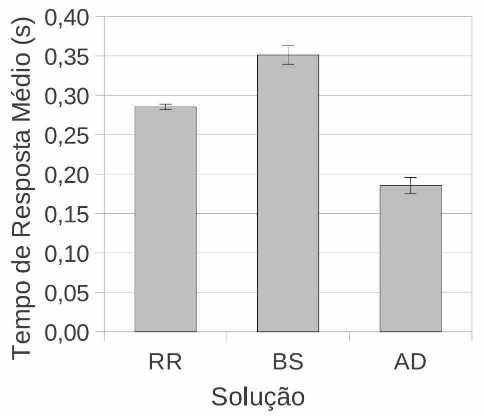


Figura A.3: Tempos de resposta médios para o cenário 2.

A Tabela A.5 mostra o resultado da ANOVA e o seu grau de confiança para as médias dos tempos de resposta da Figura A.3. A ANOVA mostra que as médias dos tempos de resposta das diferentes soluções são significativamente diferentes.

Tabela A.5: ANOVA para tempo de resposta médio no cenário 2.

F Value	320,03
Pr(>F)	< 2,2e-16

A Tabela A.6 apresenta as diferenças entre as médias dos tempos de resposta da AD e das outras soluções, calculadas por *bootstrapping*. Os resultados mostram que RR e BS apresentam médias dos tempos de resposta siginificativamente maiores que as médias dos tempos de resposta da seleção adaptativa (AD).

Tabela A.6: Diferenças de tempos de resposta médios entre AD e outras soluções, para o cenário 2.

	Limite Superior	Limite Inferior
RR	0,089	0,109
BS	0,150	0,180

A.3 Tempo de Resposta para LRR

A Figura A.4 apresenta as médias dos tempos de resposta para 6 simulações que comparam diferentes soluções de distribuição de carga (RR[7], SL[8] e LRR[9]) no cenário com um servidor sobrecarregado (Seção 6.4.3).

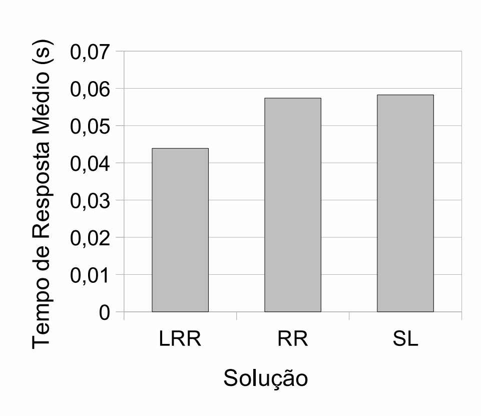


Figura A.4: Tempos de resposta médios para o cenário com um servidor sobrecarregado.

A Tabela A.7 mostra o resultado da ANOVA e o seu grau de confiança para as médias dos tempos de resposta da Figura A.4. A ANOVA mostra que as médias dos tempos de resposta das diferentes soluções são significativamente diferentes.

[7] Round Robin with Asynchronous Alarm

[8] Smallest Latency

[9] Limited Redirection Rate

Tabela A.7: ANOVA para tempo de resposta médio no cenário com um servidor sobrecarregado.

F Value	405,28
Pr(>F)	8,812e-14

A Tabela A.8 apresenta a diferença entre as médias dos tempos de resposta da LRR e das outras soluções, calculadas por *bootstrapping*. Os resultados mostram que RR e SL apresentam médias dos tempos de resposta siginificativamente maiores que as médias dos tempos de resposta da LRR.

Tabela A.8: Diferença de tempos de resposta médios entre LRR e outras soluções, para o cenário com um servidor sobrecarregado.

	Limite Superior	Limite Inferior
RR	0,012	0,014
SL	0,013	0,015

A Figura A.5 apresenta as médias dos tempos de resposta para 15 simulações que comparam as diferentes soluções de distribuição de carga no cenário com dois servidores sobrecarregados (Seção 6.4.3).

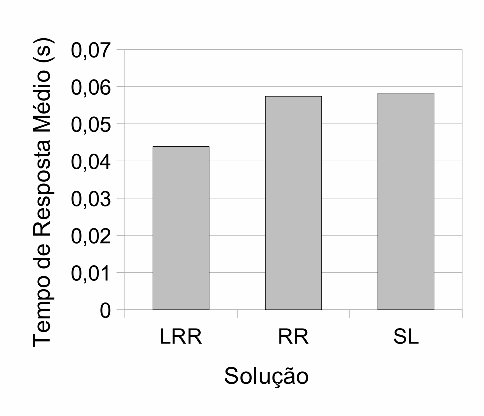


Figura A.5: Tempos de resposta médios para o cenário com dois servidores sobrecarregados.

A Tabela A.9 mostra o resultado da ANOVA e o seu grau de confiança para as médias dos tempos de resposta da Figura A.5. A ANOVA mostra que as médias dos tempos de resposta das diferentes soluções são significativamente diferentes.

Tabela A.9: ANOVA para tempo de resposta médio no cenário com dois servidores sobrecarregados.

F Value	588,75
Pr(>F)	<2,2e-16

A Tabela A.10 apresenta a diferença entre as médias dos tempos de resposta da LRR e das outras soluções, calculadas por *bootstrapping*. Os resultados mostram que RR e SL apresentam médias dos tempos de resposta siginificativamente maiores que as médias dos tempos de resposta da LRR.

Tabela A.10: Diferença de tempos de resposta médios entre LRR e outras soluções, para o cenário com dois servidores sobrecarregados.

	Limite Superior	Limite Inferior
RR	0,022	0,023
SL	0,024	0,028

Referências Bibliográficas

[1] Packmime-http: Web traffic generation in ns-2. http://www.cs.odu.edu/ mweigle/research/netsim/packmime-nsdoc.pdf.

[2] Starbed project. http://www.starbed.org/.

[3] H. E. Aarag and J. Jennings. Hashnat: A distributed packet rewriting system using adaptive hash functions. In *Proceedings of Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN 2005)*, pages 282–287. IEEE Computer Society, 2005.

[4] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: A dynamic data cache for web applications. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering (ICDE'2003)*, pages 821–831. IEEE Computer Society, 2003.

[5] C. Amza, G. Soundararajan, and E. Cecchet. Transparent caching with strong consistency in dynamic content web sites. In *Proceedings of the 19th annual international conference on Supercomputing (ICS'05)*, pages 264–273. ACM, 2005.

[6] M. Andreolini, S. Casolari, and M. Colajanni. Autonomic request management algorithms for geographically distributed internet-based systems. In Sven A. Brueckner, Paul Robertson, and Umesh Bellur, editors, *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'08).*, pages 171–180. IEEE Computer Society, 2008.

[7] D. Ardagna, S. Casolari, and B. Panicucci. Flexible distributed capacity allocation and load redirect algorithms for cloud systems. In *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD'2011)*, pages 163–170. IEEE, 2011.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[9] L. Aversa and A. Bestavros. Load balancing a cluster of web servers: using distributed packet rewriting. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference (IPCCC'00)*, pages 24–29, 2000.

[10] P. Bahl and F. Sun. System and method for performing client-centric load balancing of multiple globally-dispersed servers. Patente, 01127168.1, Microsoft Corporation, Redmond, Wa (US), 2009.

[11] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[12] A. Bestavros and S. Mehrotra. Dns-based internet client clustering and characterization. In *Proceedings of the IEEE International Workshop on Workload Characterization, 2001 (WWC 4)*, pages 159–168. IEEE Computer Society, 2001.

[13] S. Bouchenak, A. L. Cox, S. G. Dropsho, S. Mittal, and W. Zwaenepoel. Caching dynamic web content: Designing and analysing an aspect-oriented solution. In *Proceeding of Middleware 2006*, volume 4290 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2006.

[14] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[15] J. Cao, W. S. Yuan Gao Cleveland, K. Jeffay, F. D. Smith, and M. Weigle. Stochastic models for generating synthetic http source traffic. In *Proceedings of INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, 2004.

[16] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[17] V. Cardellini, M. Colajanni, and P. S. Yu. Request redirection algorithms for distributed web systems. *IEEE Transaction on Parallel and Distribributed Systems*, 14(4):355–368, 2003.

[18] N. Cardwell, S. Savage, and T. Anderson. Modeling tcp latency. In *Proceedings of the INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1742–1751, 2000.

[19] E. Casalicchio, V. Cardellini, and M. Colajanni. Content-aware dispatching algorithms for cluster-based web servers. *Cluster Computing*, 5(1):65–74, 2002.

[20] D. Chatterjee, Z. Tari, and A. Y. Zomaya. A task-based adaptive ttl approach for web server load balancing. In *Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC'2005)*, pages 877–884. IEEE Computer Society, 2005.

[21] P. Chauffour, E. Lebrun, and V. Mahe. Method for improving network server load balancing. Patente, US 7.908.355 B2, International Business Machines Corporation, Armonk, NY (US), 2011.

[22] A. K. Y. Cheung and H. Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Transactions on Computuer Systems*, 28(4):9:1–9:55, 2010.

[23] A. Cohen, S. Rangarajan, and H. Slye. On the performance of tcp splicing for url-aware redirection. In *Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems (USITS'99)*. USENIX Association, 1999.

[24] M. Colajanni and P. S. Yu. A performance study of robust load sharing strategies for distributed heterogeneous web server systems. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):398–414, 2002.

[25] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of task assignment policies in scalable distributed web-server systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):585–600, 1998.

[26] M. Conti, E. Gregori, and W. Lapenna. Client-side content delivery policies in replicated web services: parallel access versus single server approach. *Performance Evaluation*, 59(2+3):137–157, 2005.

[27] M. Conti, E. Gregori, and W. Lapenna. Content delivery policies in replicated web services: Client-side vs. server-side. *Cluster Computing*, 8(1):47–60, 2005.

[28] M. Conti, E. Gregori, and F. Panzieri. Qos-based architectures for geographically replicated web servers. *Cluster Computing*, 4(2):109–120, 2001.

[29] A. T. Davis, N. Kushman, J. G. Parikh, S. Pichai, D. Stodolsky, A. Tarafdar, and W. E. Weihl. Method of load balancing edge-enabled applications in a content delivery network. Patente, US 7.660.896 B1, Akamai Technologies Inc., Cambridge, MA (US), 2010.

[30] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[31] J. L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/-Cole, Cengage Learning, 7th edition, 2009.

[32] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available web server. In *Proceedings of the 41st IEEE International Computer Conference (COMPCON'96)*, page 85. IEEE Computer Society, 1996.

[33] M. Dobber, R. van der Mei, and G. Koole. Dynamic load balancing and job replication in a global-scale grid environment: A comparison. *IEEE Transactions on Parallel and Distributed Systems*, 20(2):207–218, 2009.

[34] S. G. Dykes, K. A. Robbins, and C. L. Jeffery. An empirical evaluation of client-side server selection algorithms. In *Proceeding of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, volume 3, pages 1361–1370, 2000.

[35] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems*, 24(1):51–78, 2006.

[36] C. Fang, D. Liang, F. Lin, and C. Lin. Fault tolerant web services. *Journal of Systems Architecture*, 53(1):21–38, 2007.

[37] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.

[38] J. Galloway, K. Smith, and J. Carver. An empirical study of power aware load balancing in local cloud architectures. In *Proceedings of the Ninth International Conference on Information Technology: New Generations (ITNG'2012)*, pages 232–236, 2012.

[39] A. Garg and D. Juneja. Collective intelligence based framework for load balancing of web servers. *International Journal of Advancements in Technology*, 3(1):64–70, 2012.

[40] K. Gilly, C. Juiz, and R. Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14:105–131, 2011.

[41] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Record*, 25(2):173–182, 1996.

[42] T. Groothuyse, S. Sivasubramanian, and G. Pierre. Globetp: template-based database replication for scalable web applications. In *Proceedings of the 16th international conference on World Wide Web (WWW'07)*, pages 301–310. ACM, 2007.

[43] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[44] B. Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

[45] D. Hong, Y. Wang, J. Lu, M. Chen, and X. Li. Building a robust and economical internet testbed: 6planetlab. In *Proceedings or the 15th IEEE International Conference on Networks (ICON 2007)*, pages 289–294, 2007.

[46] Y. S. Hong, J. K. No, and S. Y. Kim. Dns-based load balancing in distributed webserver systems. In *Proceedings of The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS 2006/WCCIA 2006)*, 2006.

[47] D. B. Ingham, S. K. S., and F. Panzieri. Constructing dependable web services. *IEEE Internet Computing*, 4(1):25–33, 2000.

[48] R. Jain. *The Art of Computer Systems Performance Analysis*, chapter 13. John Wiley & Sons, Inc., 1991.

[49] A. Josefsberg, J. D. Dunagan, M. D. Scheibel, and A. Wolman. Client-side load balancing. Patente, US 7.930.427 B2, Microsoft Corporation, Redmond, WA (US), 2011.

[50] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00)*, pages 97–110. ACM, 2000.

[51] T. T. Kwan, R. E. McGrath, and D. A. Reed. Ncsa's world wide web server: Design and performance. *Computer*, 28(11):68–74, 1995.

[52] F. T. Leighton, D. M. Lewin, R. Sundaram, R. S. Dhanidina, R. Kleinberg, M. Levine, A. M Soviani, B. Maggs, H. S. Rahul, S. Thirumalai, J. G. Parikh, and Y. O. Yerushalmi. Global load balancing across mirrored data centers. Patente, US 2010/0250742 A1, Akamai Technologies Inc., Cambridge, MA (US), 2010.

[53] Q. Li and B. Moon. Distributed cooperative apache web server. In *Proceedings of the 10th international conference on World Wide Web (WWW'01)*, pages 555–564. ACM Press, 2001.

[54] L. Liu and Y. Lu. Dynamic traffic controls for web-server networks. *Computer Networks*, 45(4):523–536, 2004.

[55] O. Martin, J. Martin-Flatin, E. Martelli, P. Moroni, H. Newman, S. Ravot, and D. Nae. The datatag transatlantic testbed. *Future Generation Computer Systems*, 21(4):443–456, 2005.

[56] N. C. Mendonça, J. A. F. Silva, and R. O. Anido. Client-side selection of replicated web services: An empirical assessment. *The Journal of Systems and Software*, 81(8):1346–1363, 2008.

[57] T. Miyachi, K. Chinen, and Y. Shinoda. Automatic configuration and execution of internet experiments on an actual node-based testbed. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the Development of NeTworks and Communities (TRIDENTCOM'05)*, pages 274–282. IEEE Computer Society, 2005.

[58] T. Miyachi, K. Chinen, and Y. Shinoda. Starbed and springos: large-scale general purpose network testbed and supporting software. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools (valuetools '06)*. ACM, 2006.

[59] P. Mockapetris. *RFC 1035 Domain Names - Implementation and Specification*. Internet Engineering Task Force, November 1987.

[60] J. Moon and M. H. Kim. Dynamic load balancing method based on dns for distributed web systems. In *E-Commerce and Web Technologies. EC-Web*, volume 3590 of *Lecture Notes in Computer Science*, pages 238–247. Springer, 2005.

[61] A. Nakai, E. Madeira, and L. E. Buzato. Analysis of resource reservation for web services load balancing. *Cluster Computing*. Submetido para publicação.

[62] A. Nakai, E. Madeira, and L. E. Buzato. DNS-based load balancing for web services. In *Proceedings of the 6th International Conference on Web Information Systems and Technologies (Webist'2010)*, 2010.

[63] A. Nakai, E. Madeira, and L. E. Buzato. Improving the qos of web services via client-based load distribution. In *Proceedings of the 29th Brazilian Symposium on Computer Networksand Distributed Systems (SBRC2011)*, 2011.

[64] A. Nakai, E. Madeira, and L. E. Buzato. Load balancing for internet distributed services using limited redirection rates. In *Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC'2011)*, 2011.

[65] A. M. Nakai, E. Madeira, and L. E. Buzato. Lab4WS: A testbed for web services. In *Proceedings of the 2nd IEEE International Workshop on Internet and Distributed Computing Systems (IDCS'09)*, 2009.

[66] K. Nishant, P. Sharma, V. Krishna, C. Gupta, K. Pratap Singh, N. Nitin, and R. Rastogi. Load balancing of nodes in cloud using ant colony optimization. In *Proceedings of the 14th International Conference onComputer Modelling and Simulation (UKSim'2012)*, pages 3–8, 2012.

[67] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'2005)*, pages 56–69, 2005.

[68] J. Osrael, L. Froihofer, M. Weghofer, and K. M. Göschka. Axis2-based replication middleware forweb services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2007)*, pages 591–598. IEEE Computer Society, 2007.

[69] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems (ASPLOS-VIII)*, pages 205–216, New York, NY, USA, 1998. ACM Press.

[70] J. Pan, Y. T. Hou, and B. Li. An overview of dns-based server selections in content distribution networks. *Computer Networks*, 43(6):695–711, 2003.

[71] M. Pathan and R. Buyya. Resource discovery and request-redirection for dynamic load sharing in multi-provider peering content delivery networks. *Journal of Network and Computer Applications*, 32(5):976–990, 2009.

[72] C. Perkins. Ip encapsulation within ip, 1996. RFC2003.

[73] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[74] S. Ranjan and E. Knightly. High-performance resource allocation and request redirection algorithms for web clusters. *IEEE Transactions on Parallel Distributed Systems*, 19(9):1186–1200, 2008.

[75] N. S. V. Rao, W. R. Wing, S. M. Carter, and Q. Wu. Ultrascience net: network testbed for large-scale science applications. *IEEE Communications Magazine*, 43:s12–s17, November 2005.

[76] X. Ren, R. Lin, and H. Zou. A dynamic load balancing strategy for cloud computing platform based on exponential smoothing forecast. In *Proceedings of the 2011 IEEE International Conference onCloud Computing and Intelligence Systems (CCIS'2011)*, pages 220–224, 2011.

[77] RFC793. Transmission Control Protocol, 1981.

[78] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[79] J. Salas, F. Perez-Sorrosal, M. Patiño-Martínez, and R. Jiménez-Peris. Ws-replication: a framework for highly available web services. In *Proceedings of the 15th international conference on World Wide Web (WWW'2006)*, pages 357–366. ACM, 2006.

[80] D. Sanghi, P. Jalote, P. Agarwal, N. Jain, and S. Bose. A testbed for performance evaluation of load-balancing strategies for web server systems. *Software: Practice and Experience*, 34(4):339–353, 2004.

[81] G. T. Santos, L. C. Lung, and C. Montez. Ftweb: A fault tolerant infrastructure for web services. In *Proceedings of the 2005 Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, pages 95–105. IEEE Computer Society, 2005.

[82] D. Sasha and M. Wilson. *Statistics is Easy*. Mogan and Claypool Publishers, 2nd edition, 2010.

[83] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of dns-based server selection. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'2001)*, pages 1801–1810, 2001.

[84] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66, 2007.

[85] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen. Replication for web hosting systems. *ACM Computing Surveys*, 36(3):291–334, 2004.

[86] P. Srisuresh and K. Egevang. Traditional ip network address translator (traditional nat), 2001. RFC3022.

[87] S. K. Srivastava and D. C. Tappan. Method and appartus providing highly scalable server load balancing. Patente, US 7.512.702 B1, Cisco Technology Inc., San Jose, CA (US), 2009.

[88] A. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind akamai (travelocity-based detouring). *SIGCOMM Computer Communication Review*, 36(4):435–446, 2006.

[89] W. Tang and M. W. Mutka. Load distribution via static scheduling and client redirection for replicated web servers. In *Proceedings of the 2000 International Workshop on Parallel Processing (ICPP'00)*, pages 127–133. IEEE Computer Society, 2000.

[90] N. Tolia and M. Satyanarayanan. Consistency-preserving caching of dynamic database content. In *Proceedings of the 16th international conference on World Wide Web (WWW'07)*, pages 311–320. ACM, 2007.

[91] TPC. TPC Benchmark W - Specification 1.8, February 2002. http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf.

[92] Q. H. Vu, B. C. Ooi, M. Rinard, and K. Tan. Histogram-based global load balancing in structured peer-to-peer systems. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):595–608, 2009.

[93] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Proceedings of Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-Ice'11)*, 2011.

[94] Z. Xiong, P. Yan, and C. Guo. Mddr: a solution for improving the scalability of dispatcher-based web server cluster. In *Proceedings of the IEEE International Conference on Communications (ICC'2005)*, volume 1, pages 38–42, 2005.

[95] X. Ye and Y. Shen. A middleware for replicated web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 631–638. IEEE Computer Society, 2005.

[96] H. Yokota, S. Kimura, and Y. Ebihara. A proposal of dns-based adaptive load balancing method for mirror server systems and its implementation. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications (AINA'2004)*, volume 2, pages 208–213, 29-31 2004.

[97] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP'01)*, pages 29–42. ACM Press, 2001.

[98] L. Zhuo, C. Wang, and F. C. M. Lau. Document replication and distribution in extensible geographically distributed web servers. *Journal of Parallel and Distributed Computing*, 63(10):927–944, 2003.