Goedson Teixeira Paixão

Escalabilidade em servidores cache WWW

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

Resumo

O grande crescimento em popularidade da *World Wide Web* tem motivado várias pesquisas com o objetivo de reduzir a latência observada pelos usuários. Os servidores cache têm-se mostrado uma ferramenta muito importante na busca desse objetivo.

Embora a utilização de servidores cache tenha contribuído para diminuir o tráfego na Internet, as estratégias de cooperação utilizadas na composição de grupos (*clusters*) de caches normalmente trazem uma degradação de desempenho aos servidores não sendo, por isso, escaláveis o suficiente para acompanhar o crescimento atual da WWW.

Nesta dissertação, propomos uma nova forma de cooperação entre servidores cache que não causa um impacto tão grande no seu desempenho, permitindo, assim, a criação de grupos de servidores cache que sejam capazes de crescer junto com a demanda dos seus usuários.

Abstract

The increasing popularity of the World Wide Web has led to significant research targetting the reduction of the latency perceived by users. Cache servers proved to be an important resource in accomplishing this goal.

Although the use of cache servers have reduced the data traffic in the Internet, the cooperation strategies employed while building clusters of those servers scale poorly and are not able to sustain their throughput under increasing demands.

In this work, we present a new cooperation strategy that does not affect significantly the server performance. The use of this strategy allows clusters of cache servers to scale well with their usage growth.

Agradecimentos

Gostaria de agradecer a todos que contribuiram para a realização deste trabalho:

Ao Wagner Meira Jr., não só pela orientação, mas, principalmente, pela amizade, paciência e ajuda durante a realização deste trabalho.

Aos meus colegas que proporcionaram um ambiente de trabalho agradável e de constante busca de novos conhecimentos durante esse curso.

Ao João Paulo Kitajima que, muito mais que um professor, foi um grande amigo durante a sua estada na UFMG.

Aos meus pais e irmãos que, mesmo a quilômetros de distância, sempre me incentivaram a continuar na busca dos meus ideais.

Finalmente, gostaria de agradecer à Juliene, pelo carinho, paciência e incentivo a continuar o meu trabalho, mesmo que isso às vezes significasse ser trocada pelo computador.

Ah! Não posso esquecer da CAPES, o apoio financeiro que tarda mas não falha.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A *World Wide Web* (referenciada neste trabalho como WWW ou Web) foi idealizada por Tim Berners-Lee, em março de 1989, como um meio de divulgação de pesquisas e novas idéias pelo CERN, Centro Europeu de Pesquisa em Física de Partículas, localizado em Genebra, na Suíça [4]. Foram desenvolvidos no CERN o primeiro servidor, o primeiro navegador WWW, a *Hypertext Markup Language* (HTML), uma linguagem para descrição de documentos para a Web, o *Hypertext Transfer Protocol* (HTTP), um protocolo de comunicação entre clientes e servidores e a forma de endereçamento de objetos[1] (URL, *Uniform Resource Locator*). Este desenvolvimento inicial foi divulgado pelo CERN em 1991.

Em 1993 foi criado no NCSA, *National Center for Supercomputing Applications*, o Mosaic, primeiro navegador WWW com interface gráfica de fácil utilização e multiplataforma, desenvolvido por Marc Andreseen. Neste mesmo ano, Robert McCool, outro pesquisador do NCSA, desenvolveu um novo programa servidor, menor e mais eficiente que veio a se tornar a base para o servidor mais utilizado atualmente na WWW, o Apache[1].

Desde então, a Web vem sofrendo um crescimento fantástico tanto em número de usuários quanto em número de *sites* existentes. Para se ter uma idéia desse crescimento, o número de servidores Web em operação cresceu de 200 em 1993 para mais de 2 milhões em 1998[24]. Há várias justificativas para esse crescimento de utilização: desenvolvimento de novos protocolos e linguagens que permitem o uso de dados multimídia, desenvolvimento de interfaces que permitem manipular esses dados, desenvolvimento da tecnologia de redes permitindo a interligação de computadores em escala global e, principalmente, a facilidade de compartilhar informações a baixo custo.

[1] um objeto pode ser uma página de texto, uma imagem, um arquivo etc.

1.1 Motivação

Desde a popularização da WWW com o surgimento dos primeiros navegadores gráficos em 1993, muito esforço tem sido feito para reduzir a latência percebida pelos seus usuários. O acesso à WWW pode se tornar lento por várias razões. Os sistemas servidores se tornam lentos quando submetidos a uma grande carga de requisições. Pode também haver congestionamento nos canais responsáveis pelo tráfego dos dados entre o servidor e o usuário.

Uma estratégia comum, apesar do alto custo, de aliviar este problema é aumentar a capacidade do recurso sobrecarregado: comprar um servidor mais rápido ou um *link* de comunicação com maior largura de banda. Este método, além de não ser economicamente viável na maioria dos casos, ignora a diversidade das fontes de degradação de desempenho a serem consideradas, mesmo em uma simples transação na Web. Além do usuário e do servidor acessado, há provavelmente um grande número de provedores de serviço que participam do processo de prover a informação requisitada pelo usuário. Claramente, de nada adiantaria aumentar a capacidade do servidor e dos seus *links* externos, se entre ele e o usuário existem canais com largura de banda que limita o número de requisições atendidas. Por exemplo, a Netscape[20] poderia ter milhares de servidores e milhares de canais de comunicação que um usuário ainda precisaria de várias horas para copiar a última versão do "Navigator" se o seu provedor utilizar apenas um *link* de 19,2Kbps para todas as suas conexões.

Os servidores *proxy*, desenvolvidos inicialmente para permitir o acesso à WWW por usuários protegidos por *firewalls*, mostraram ser de grande utilidade, também, na redução da latência percebida por estes usuários no acesso à Web, quando usados como servidores proxy-cache (ou simplesmente cache).

A utilização de servidores cache provou ser uma técnica útil na redução da latência percebida pelo usuário final. O conceito fundamental no qual se baseia o funcionamento desses servidores é a replicação de objetos populares da Web em um ponto mais próximo do usuário. Os benefícios do uso de caches advêm da repetição de acessos a um determinado objeto. Se uma cópia do objeto é armazenada no servidor cache na primeira vez em que ele é acessado, os próximos acessos podem ser satisfeitos pela cópia local, não sendo necessário recorrer ao servidor original. Por exemplo, a maioria dos programas de navegação na Web possuem caches locais em disco porque, em geral, um usuário acessa um pequeno conjunto de objetos freqüentemente. Da mesma forma, há repetição de objetos acessados pelos membros de um dado grupo de usuários. Estes usuários poderiam beneficiar-se da replicação desses objetos em um cache compartilhado em algum ponto da rede próximo deles (seu provedor de acesso à Internet, por exemplo).

Os caches WWW são implementados como agentes intermediários entre o usuário final e os servidores Web. Estes agentes armazenam localmente os objetos acessados mais recentemente para que não seja necessário recorrer ao servidor original em uma próxima requisição. Embora a utilização de caches tenha contribuído para diminuir o tráfego na Internet, as estratégias de cooperação utilizadas na composição de grupos (*clusters*) de

caches normalmente trazem uma degradação de desempenho aos servidores [14, 15] não sendo, por isso, escaláveis o suficiente para acompanhar o crescimento atual da WWW.

Neste trabalho, apresentamos um método de comunicação entre servidores cache que desenvolvemos[22] com o objetivo de minimizar o custo de processamento associado à cooperação entre eles, permitindo, assim, a criação de grupos de servidores que sejam capazes de crescer junto com a demanda de seus usuários.

Essa dissertação está dividida em seis capítulos. No capítulo 2 fazemos a revisão de estratégias de cooperação entre servidores cache já existentes, no capítulo 3 descrevemos o servidor cache utilizado nesse trabalho, no capítulo 4 descrevemos a nova estratégia de cooperação desenvolvida, no capítulo 5 descrevemos os experimentos realizados e discutimos os resultados desses experimentos e no capítulo 6 apresentamos as principais conclusões e contribuições obtidas neste trabalho e perspectivas de trabalhos futuros.

Capítulo 2

Trabalhos relacionados

O crescimento exponencial da Web e os problemas de desempenho causados por ele têm incentivado diversas pesquisas que resultaram na criação de servidores cache e estratégias de cooperação entre eles que sejam eficientes e escaláveis. Neste capítulo, fazemos uma revisão das estratégias de cooperação entre servidores cache existentes atualmente, analisando as suas vantagens e seus problemas de escalabilidade.

2.1 Arquitetura da WWW

Clientes e servidores WWW comunicam-se através do protocolo HTTP. Este protocolo funciona da seguinte forma (veja a Figura 2.1):

1. O cliente abre uma conexão com o servidor Web onde se encontra o objeto que deseja recuperar;
2. O cliente envia para o servidor o nome (URL) do objeto desejado e quaisquer especificações extras da transação (por exemplo, recuperar o arquivo somente se tiver sido modificado após determinada data);
3. O servidor envia o objeto para o cliente e fecha a conexão.

Quando utilizamos um servidor proxy, a comunicação é feita da seguinte forma (veja a Figura 2.2):

1. O cliente abre uma conexão com o servidor proxy;
2. O cliente envia para o servidor proxy a URL do objeto que deseja;
3. O servidor proxy abre uma conexão com o servidor Web;
4. O servidor proxy envia para o servidor Web a URL do objeto desejado;
5. O servidor Web envia o objeto para o servidor proxy e fecha a conexão;

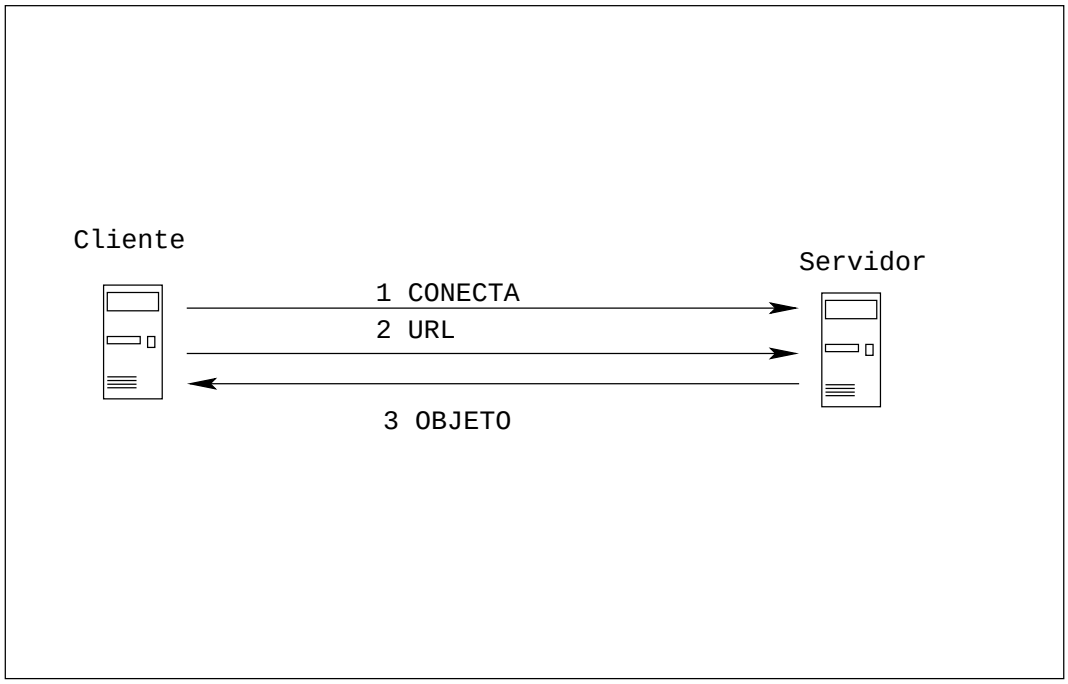

Figura 2.1: Transação HTTP.

6. O servidor proxy envia o objeto para o cliente e fecha a conexão.

No caso em que o servidor proxy faz, também, o serviço de cache, os passos de 3 a 5 podem não ocorrer caso o proxy já tenha o objeto desejado no seu cache.

No protocolo HTTP 1.0 [5], para cada objeto a ser recuperado, é repetido o ciclo descrito acima. Para recuperar uma página que contém 5 figuras, por exemplo, são utilizadas 6 conexões HTTP (1 para o texto da página e 1 para cada figura). O protocolo HTTP 1.1 [10] minimiza o número de conexões empregando conexões persistentes.

2.2 Cooperação entre servidores cache

Nesta seção, descrevemos as estratégias de cooperação entre servidores cache existentes atualmente, analisando as suas vantagens e seus problemas de escalabilidade.

Internet Cache Protocol

O ICP [28] é um protocolo baseado em UDP/IP projetado para permitir a criação de conjuntos de servidores cache que cooperam entre si. Este protocolo funciona da seguinte forma (veja a Figura 2.3):

1. O servidor, após receber uma requisição HTTP e verificar a falta do objeto no seu cache, envia para cada um dos seus parceiros uma mensagem `ICP_QUERY` contendo a URL do objeto desejado;

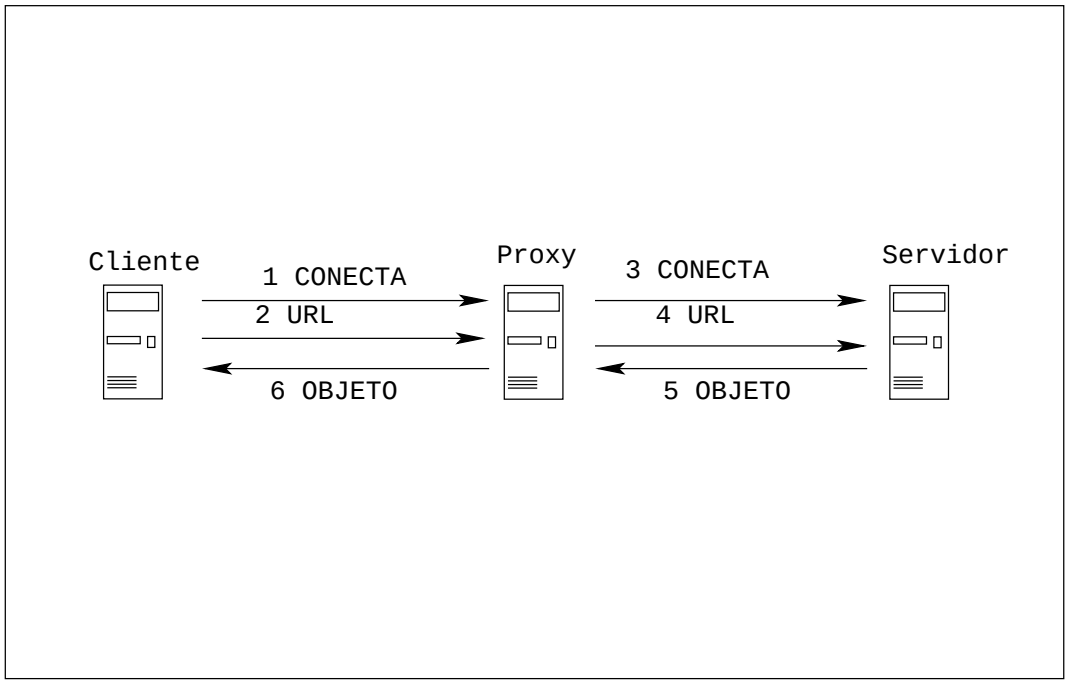

Figura 2.2: Transação HTTP com proxy.

2. Cada parceiro, ao receber a mensagem `ICP_QUERY`, consulta o seu cache local e responde com uma mensagem `ICP_MISS` caso esse objeto não esteja armazenado ou `ICP_HIT` caso esse objeto esteja armazenado;

3. O servidor que originou a pesquisa, ao receber a primeira resposta `ICP_HIT`, estabelece uma conexão HTTP com o parceiro que enviou esta resposta para recuperar o objeto;

4. O parceiro envia o objeto para o servidor que o solicitou;

5. O servidor envia o objeto para o cliente que originou a requisição.

Uma grande desvantagem desse protocolo é o atraso adicional introduzido pela troca de mensagens ICP no tempo de resposta ao cliente. Como os servidores cache podem processar as mensagens ICP de forma bastante rápida, esse atraso é, geralmente, pouco maior que o tempo de ida e volta de uma mensagem entre os servidores cooperantes.

Outro problema grave do ponto de vista de escalabilidade do servidor, é o fato de que o número de mensagens trocadas é proporcional ao produto entre o número de servidores no grupo e o número de requisições HTTP recebidas levando a uma rápida degradação no desempenho dos servidores quando submetidos a grandes cargas de trabalho com um número grande de parceiros [17].

Cache Digests

Na tentativa de eliminar o tempo adicional introduzido pela troca de perguntas e respostas no protocolo ICP, foi introduzida na versão 2.0 do servidor cache Squid[19] a consulta

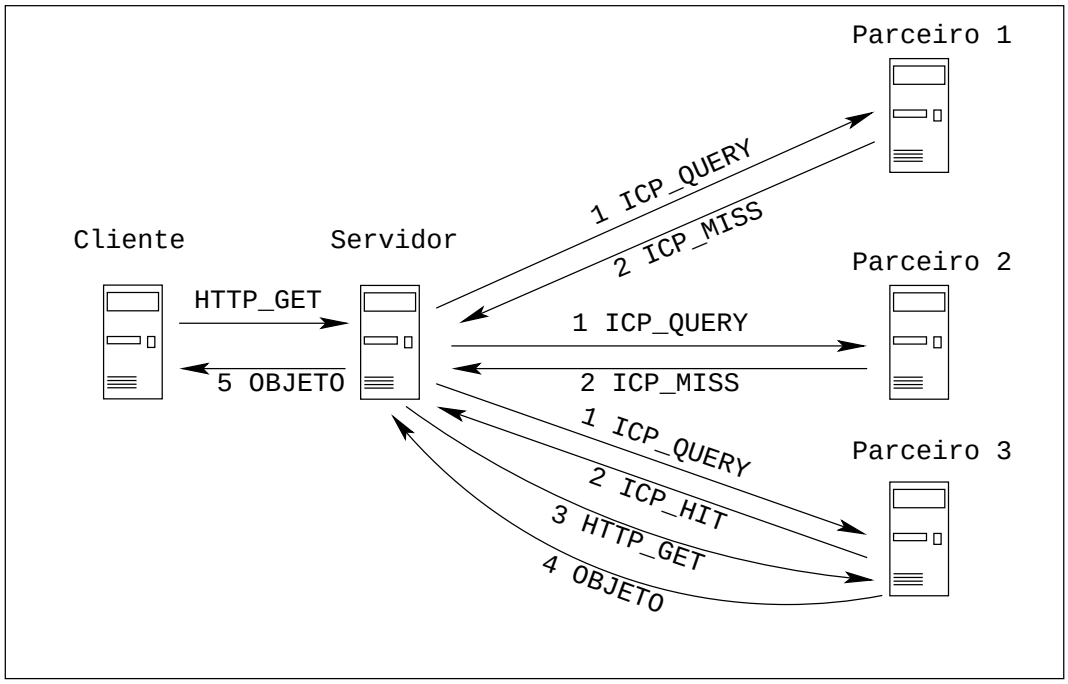

Figura 2.3: Transação HTTP com consulta ICP.

através de *Cache Digests*, que são resumos dos conteúdos dos caches dos servidores trocados periodicamente entre eles.

Um *Cache Digest* [26] consiste de um vetor de bits que podem estar ativados ou desativados. Para acrescentar uma entrada no *Cache Digest*, computamos um pequeno número de funções *hash* independentes para a chave dessa entrada. Os valores das funções indicam quais os bits devem ser ativados. Para verificar a presença de uma determinada entrada no *Cache Digest*, computamos as mesmas funções e verificamos os valores dos bits correspondentes, se todos estiverem ativados, há alguma probabilidade de que a entrada esteja presente no cache. A remoção de uma entrada do *Digest*, no entanto, não pode ser feita com a mesma facilidade porque não podemos simplesmente desativar os bits referentes à entrada removida, pois isso poderia resultar na desativação de bits referentes a entradas que ainda estão presentes. Por isso, essa operação não é implementada no Squid, sendo feita periodicamente uma reconstrução do *Digest* incluindo as entradas referentes aos objetos presentes no cache em um *Digest* que se encontra inicialmente vazio.

O tamanho do vetor de bits determina a probabilidade de uma pesquisa que encontra todos os bits ativados estar correta. A diminuição desse vetor resultará em um número maior de erros para o mesmo conjunto de dados.

A cooperação entre servidores Squid utilizando *Cache Digests* funciona da seguinte forma:

1. O Squid, no início de sua execução, solicita de cada um dos seus parceiros o *Cache Digest* local;

2. Ao receber um pedido de um objeto que não esteja presente no cache local, o servidor consulta os *Digests* dos seus parceiros;

3. Caso algum dos *Digests* indique a probabilidade de existência do objeto, o Squid solicita ao parceiro correspondente o objeto desejado.

A atualização periódica das cópias remotas do *Digest*, assim como a cópia inicial, é feita utilizando o protocolo HTTP. Este método soluciona o problema da troca de mensagens a cada requisição do protocolo ICP, tornando o sistema menos suscetível a uma degradação no desempenho provocada por excesso de comunicação. No entanto, ele traz alguns novos problemas:

1. O *Cache Digest* é uma estrutura de tamanho consideravelmente grande (geralmente entre 200KB e 2MB). Como cada servidor deve armazenar localmente o *Digest* de cada parceiro, há um aumento significativo no consumo de memória pelo Squid.

2. A atualização periódica dos *Digests* provoca tráfego em rajadas.

3. É possível que consultas ao *Digest* resultem em respostas erradas.

   Por exemplo, suponhamos um *Cache Digest* de 16 bits (representado na Figura 2.4) cujas funções *hash* são `h1` e `h2`. A presença do objeto `A` no cache, causa a ativação dos bits 3 e 7 do *Digest* enquanto a presença do objeto `B` ativa os bits 6 e 10.

   Ao consultarmos esse *Digest*, procurando pelo objeto `C`, que ativaria os bits 10 e 3, concluiremos, erroneamente, que este objeto está presente no cache.

   A probabilidade de ocorrência de falsos acertos pode ser diminuída através do aumento do tamanho do *Digest*. A diminuição de falsos acertos, no entanto, pode não compensar a quantidade extra de dados que deverão ser transmitidos entre os servidores.

   Uma consulta a um *Digest* desatualizado também pode resultar em erros indicando a inexistência de objetos que foram incorporados recentemente ao cache do servidor.

   Ambas as situações podem resultar em desperdício de recursos.

4. A reconstrução periódica do *Digest* pode ser uma operação computacionalmente intensiva em caches com um grande volume de objetos armazenados.

CRISP

Pesquisadores da AT&T e da Duke University desenvolveram em 1997 o *Caching and Replication for Internet Service Performance* (CRISP) [12], uma estratégia de cooperação entre servidores cache baseada em diretórios.

Esta estratégia baseia-se na existência de uma máquina centralizadora dedicada à manutenção de um diretório das URLs dos objetos armazenados em todos os servidores cache

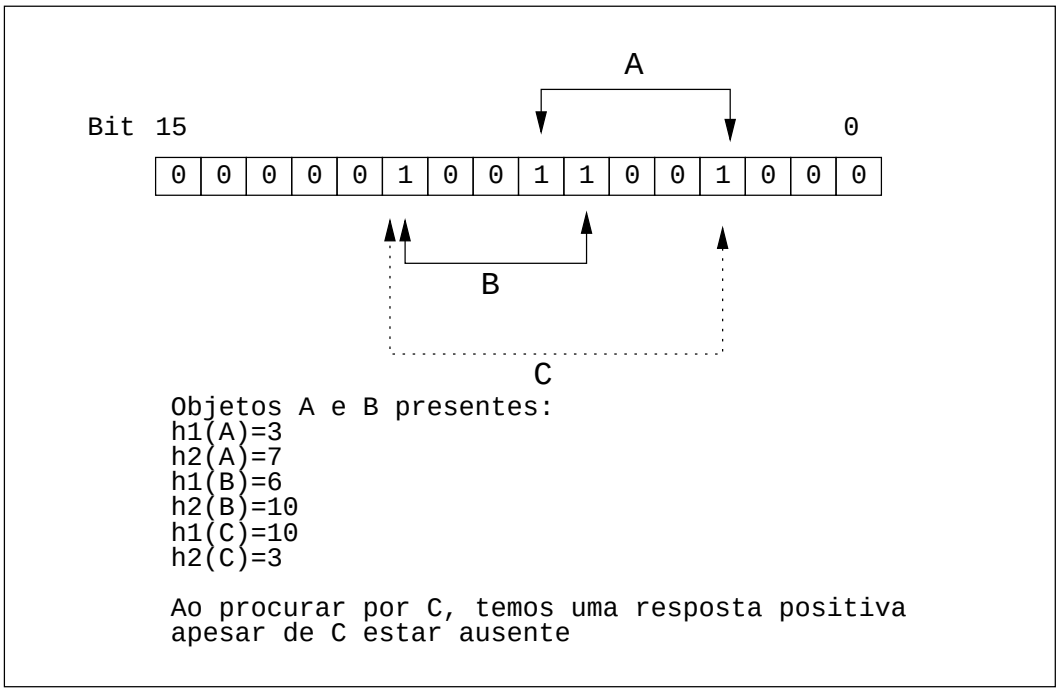

Figura 2.4: Falso acerto em um Cache Digest.

do grupo. Cada servidor cache, ao adicionar ou remover um objeto do seu cache local, deve enviar uma mensagem à máquina centralizadora informando a operação. Ao receber o pedido de um objeto que não está presente no seu cache, um servidor consulta o diretório na máquina centralizadora para saber em quais servidores esse objeto está armazenado (se estiver em algum deles). Recebendo a resposta da máquina centralizadora, o servidor estabelece uma conexão HTTP com o parceiro que possui o objeto e o recupera, repassando-o para o cliente.

Esta estratégia reduz o número de comunicações de consultas por requisição de N (no caso de ICP) para 1 sem o custo adicional da transmissão de resumos dos caches dos servidores. A existência de um servidor de diretório central, no entanto, apresenta os seguintes problemas:

- A indisponibilidade do servidor de diretório transforma o grupo de servidores cache em vários servidores que não cooperam entre si.
- A centralização de consultas no servidor de diretórios, pode levar a uma degradação no desempenho do sistema com o crescimento da demanda e do número de servidores atendidos por esse diretório.

Além disso, servidores que detêm objetos muito populares, podem acabar sobrecarregados por serem muito requisitados por clientes e outros servidores, aumentando o tempo de resposta aos usuários. Este problema poderia ser resolvido se um servidor, ao recuperar um objeto de um parceiro, armazenasse uma cópia no seu próprio cache. Essa medida, no

entanto, levaria a um desperdício de espaço de cache nos servidores devido à existência de cópias dos mesmos objetos nos vários servidores do grupo.

CARP

A Microsoft introduziu, em 1997, no Microsoft Proxy Server 2.0, o *Cache Array Routing Protocol*[17], um protocolo para cooperação entre servidores cache que permite, a qualquer servidor cache, descobrir, sem a necessidade de comunicação, qual servidor possivelmente possui um determinado objeto. Este protocolo funciona da seguinte forma:

1. Ao receber uma requisição de um cliente, o servidor calcula uma função *hash* para a URL do objeto desejado;

2. O valor da função *hash* para a URL é combinado com o valor de uma outra função *hash* para cada um dos servidores do grupo;

3. O servidor, cuja combinação servidor+URL tiver o maior valor, é o que deverá conter o objeto. O servidor que recebeu o pedido estabelece uma conexão HTTP com aquele que provavelmente possui o objeto e o recupera para o cliente.

Esta estratégia consegue eliminar totalmente a comunicação efetuada para consultar os servidores, sendo necessária apenas uma comunicação periódica entre eles para comunicar a sua disponibilidade. O espaço de cache é melhor aproveitado porque apenas um dos servidores do grupo possui uma cópia de cada objeto.

O fato de apenas um servidor possuir uma cópia de cada objeto, no entanto, pode causar uma degradação no desempenho do sistema, já que todas as requisições por um objeto muito popular terão que passar pelo servidor que o possui (mesmo que a requisição original tenha sido feita a um outro servidor) aumentando o número de requisições que esse servidor precisa tratar.

Cisco Cache Engine

O servidor cache da Cisco[8] pode ser utilizado de forma transparente para o usuário. Os roteadores utilizando o *Web Cache Control Protocol*, interceptam todas as requisições para o porto TCP do protocolo HTTP(80) e as redirecionam para um servidor cache que se encarrega de fazer o atendimento. Essa arquitetura permite a criação de grupos, particionando o conjunto de endereços IP entre os servidores do grupo.

Essa estratégia, como o CARP, tem a vantagem de não necessitar de consultas para determinar qual o servidor que possui um determinado objeto. Ela, no entanto, pode ser afetada pelo mesmo problema de degradação de desempenho para servidores que possuem objetos muito populares. Além disso, a indisponibilidade de um dos servidores, pode tornar impossível acessar os objetos pelos quais ele é responsável.

2.3 Limitações dos métodos de cooperação existentes

Os métodos de cooperação entre servidores cache descritos neste capítulo podem comprometer o desempenho dos servidores que os utilizam. Nesta seção iremos discutir problemas comuns a todos eles, que fazem com que estes métodos não utilizem eficientemente os recursos disponíveis.

Os métodos descritos utilizam sempre o protocolo TCP/IP [27] como base para a comunicação entre os servidores, ignorando a existência de subsistemas de comunicação mais eficientes e confiáveis como redes locais[7] e comunicação entre processos numa mesma máquina via memória compartilhada. Projetado para prover uma comunicação confiável em ambientes de rede não confiáveis e a longas distâncias, o protocolo TCP/IP inclui vários cabeçalhos e mecanismos para controle de erros desnecessários em ambientes mais confiáveis como as redes locais atuais. É notório que uma máquina dedicada à execução de um servidor cache tem uma carga baixa de processamento[25], o que significa que seria viável utilizar mais de um servidor cache ao mesmo tempo em uma máquina. O uso de mais de um processo servidor em uma mesma máquina torna este processamento extra ainda mais indesejável, pois o *kernel* precisa fazer todo um trabalho de divisão dos dados em pacotes, criação de cabeçalhos e cálculos de *checksums* no envio dos dados, para em seguida desfazer todo esse trabalho para entregar os dados ao processo receptor.

Um outro problema de desempenho dos métodos já citados está no fato de que para cada objeto recuperado de outro servidor é estabelecida uma nova conexão de rede[5] cujo estado deve ser verificado periodicamente. Esta verificação é feita, em geral, através da chamada de sistema `select`. Como esta chamada de sistema trabalha verificando, seqüencialmente, cada um dos descritores de arquivo passados para ela como parâmetro, o maior número de conexões a serem verifcadas implica um maior tempo gasto pelo processo dentro do *kernel* fazendo este trabalho[18]. Este aumento linear no tempo gasto para fazer a verificação do estado das conexões pode significar um grande aumento no tempo total de serviço quando o servidor está submetido a grandes cargas de trabalho, levando a uma degradação significante na sua capacidade de atendimento.

Estes métodos têm ainda a desvantagem de causar uma diminuição no desempenho não só do servidor que origina a requisição mas também no servidor que a atende, já que este último precisa dedicar parte do seu tempo de processamento e da sua capacidade de comunicação ao primeiro como a um outro cliente qualquer.

Ou seja, o custo do processamento de pacotes TCP/IP, aliado à serialização do *kernel*, tornam pouco escaláveis os métodos de cooperação entre servidores cache que utilizam conexões HTTP para o transporte de dados entre servidores.

Capítulo 3

O servidor cache Squid

O Squid [19] é um servidor cache derivado do *Harvest Project* [9], desenvolvido na Universidade do Colorado em Boulder. Atualmente o seu desenvolvimento é liderado por Duane Wessels do *National Laboratory for Applied Network Research* e financiado pela *National Science Foundation*. Ele atende a requisições nos protocolos HTTP, FTP e GOPHER, suportando conexões SSL (*Secure Sockets Layer*) e um controle de acesso que permite, por exemplo, especificar quais clientes podem fazer uso do servidor ou quais domínios este servidor pode acessar. Este é o servidor cache mais popular atualmente e, como o seu código fonte é disponível gratuitamente, modificações no seu funcionamento podem ser feitas facilmente. Por essas razões escolhemos este servidor para este trabalho. Nas próximas seções iremos descrever o seu funcionamento a fim de subsidiar a elaboração de uma nova estratégia de cooperação.

3.1 Visão geral do Squid

O Squid utiliza um único processo para atender a todas as requisições que recebe. Para evitar que a espera pelo fim de uma operação demorada, como a leitura de dados de um arquivo ou o recebimento de dados pela rede, impeça a execução de outras operações, todo o processamento é orientado a eventos e utiliza operações assíncronas. O mecanismo de eventos no Squid funciona da seguinte forma:

- Ao iniciar uma operação assíncrona (uma leitura de arquivo, por exemplo), o Squid adiciona aquela operação à sua lista de eventos pendentes, indicando qual rotina deve ser chamada quando acontecer o evento esperado (por exemplo, o arquivo terminou de ser lido);
- Periodicamente o Squid verifica se ocorreu algum evento pelo qual ele está esperando (através da chamada de sistema `select`) e trata aqueles eventos que tenham ocorrido, chamando as rotinas especificadas no momento de início das operações.

O Squid é composto por três programas principais (`squid`, `dnsserver` e `unlinkd`). O `squid` é o responsável pela função principal do servidor, ou seja, atender às requisições feitas por clientes. O `dnsserver` e o `unlinkd` são responsáveis, respectivamente, pela tradução de nomes de domínios para números IP e pela remoção de arquivos quando solicitados pelo `squid`.

Esta separação em três programas diferentes foi feita devido ao fato de que a tradução de nomes de domínio para endereços IP não pode ser feita de forma assíncrona, a remoção de arquivos não precisa ser feita em tempo real e ambas podem ter um custo computacional alto. Como o Squid utiliza apenas um processo para atender a todas as requisições recebidas, a execução de uma dessas operações iria bloquear o atendimento de outras requisições, caso fosse feita pelo próprio processo `squid`. Para evitar esse bloqueio, o processo `squid` mantém um conjunto de processos `dnsserver`s e `unlinkd`s ativos e, quando necessita executar tais operações, envia uma requisição para um desses processos.

3.2 Armazenamento de dados

Para evitar a utilização de diretórios com um número muito grande de arquivos, o que resultaria em um tempo maior para as operações de abertura de arquivos (quanto maior o número de arquivos em um diretório, mais demorada é a operação de abertura de arquivos), o Squid organiza os objetos do seu cache em disco da seguinte forma:

- No arquivo de configuração, o administrador define um ou mais `cache_dir`s;
- Para cada `cache_dir`, ele define o caminho do diretório onde serão armazenados os seus dados, a quantidade máxima de dados a serem armazenados, o número de subdiretórios que devem ser criados no caminho especificado (diretórios de primeiro nível) e o número de diretórios a serem criados em cada diretório de primeiro nível (diretórios de segundo nível);
- Todos os arquivos são armazenados em algum dos diretórios de segundo nível. Cada objeto é identificado por um número que também indica o caminho exato do arquivo que o contém.

3.3 Estruturas de dados do Squid

O Squid foi projetado para ser um servidor cache eficiente, otimizando o máximo possível o tempo de resposta ao usuário. Para isso o Squid mantém em memória principal objetos em trânsito, objetos "quentes" (aqueles objetos que são acessados mais freqüentemente), dados sobre todos os objetos no cache (metadados do cache), o resultado das últimas consultas ao DNS e os últimos objetos que não puderam ser recuperados. Mantendo todas estas estruturas em memória, o Squid evita: (1) fazer chamadas desnecessárias

ao DNS, (2) tentar buscar objetos inexistentes e (3) fazer acessos a disco para objetos muito populares.

Duas estruturas de dados do Squid são diretamente relacionadas à cooperação entre servidores:

Metadados do cache: Um dos conjuntos de dados mais importantes mantidos pelo Squid é o conjunto de metadados do cache. Estes dados são armazenados em uma tabela *hash* de `StoreEntry`s. Uma `StoreEntry` contém a assinatura MD5 [23] da URL a que se refere o objeto, última vez que o objeto foi referenciado, quando foi modificado, prazo de expiração do objeto, tamanho do objeto e identificador do arquivo que contém esse objeto no cache além de vários outros dados sobre o estado do objeto. É através de uma consulta a essa tabela que o Squid pode determinar se um dado objeto está no cache e, em caso positivo, onde encontrá-lo.

Dados de configuração: O Squid mantém, em uma estrutura chamada `Config`, os dados referentes à configuração do servidor. Esta estrutura contém a definição de endereços nos quais o servidor atende a requisições, dados sobre outros servidores cache que podem ser consultados quando um objeto não é encontrado localmente, quantidade de memória a ser utilizada pelo servidor, entre outras informações. Para a implementação de uma estratégia de cooperação mais eficiente, será de grande importância as informações sobre quais os diretórios utilizados pelo servidor para armazenar os objetos do cache. Estas informações são mantidas no campo `cacheSwap`, na forma de um vetor de estruturas contendo o caminho, o número de diretórios no primeiro nível e o número de diretórios no segundo nível para cada diretório definido pela diretiva `cache_dir` no arquivo de configuração.

3.4 Funcionamento do Squid

Como dito anteriormente, o Squid utiliza um único processo para atender todas as requisições recebidas de clientes. Para que esse atendimento possa ser feito de forma eficiente, ele utiliza rotinas assíncronas de entrada e saída de forma que o atendimento a um cliente não seja interrompido enquanto se espera, por exemplo, o término de uma operação de leitura necessária para atender outro cliente.

Nesta seção, descreveremos o processo de atendimento de uma requisição HTTP (outros protocolos funcionam de forma semelhante). Devemos lembrar que esta sequência de operações não será encontrada de forma tão explícita no código fonte do Squid, já que a maioria das operações de comunicação e acessos a disco são feitos de forma assíncrona, através da instanciação de eventos a serem tratados quando a operação é completada.

A descrição que faremos nas seções a seguir não pretende ser uma descrição completa do funcionamento do servidor Squid, sendo omitidas, principalmente, o tratamento das condições de erro que podem ocorrer durante o processamento de uma requisição. O nosso

objetivo é permitir o entendimento da tarefa realizada sem explicar detalhes do código do servidor.

3.4.1 Recebimento e atendimento de uma conexão

Ao receber um pedido de conexão, o Squid aceita esse pedido e aguarda o envio da requisição HTTP. Após o recebimento da requisição, ele faz a análise do texto recebido para determinar o método utilizado (por exemplo, `PUT` ou `GET`) e a URL do objeto desejado, fazendo em seguida, uma consulta à tabela de metadados do cache para determinar se o objeto existe no cache local. Caso o objeto exista no cache e a cópia existente ainda seja válida, o servidor envia o objeto para o cliente. Caso o objeto não exista no cache local, o Squid pode recorrer a algum servidor cache parceiro ou ao servidor HTTP que possui o objeto. O processo de recuperação de arquivos do cache e de colaboração entre servidores cache é descrito com maiores detalhes nas próximas seções.

3.4.2 Recuperação de um objeto existente no cache

Quando o Squid encontra o objeto solicitado pelo cliente no seu cache, este objeto pode estar na memória principal (por ser um objeto "quente" ou por ainda estar em trânsito) ou pode estar disponível somente no disco.

No caso de o objeto estar disponível em memória principal, o processo de atendimento é bastante simples. Basta enviar os cabeçalhos da resposta HTTP e, em seguida, enviar partes do objeto até que todo o seu conteúdo tenha sido enviado.

No caso de o objeto estar disponível somente no disco, no entanto, antes que se possa enviar qualquer dado ao cliente, é necessário determinar qual o arquivo que contém esse objeto. Para isso, o campo `swap_file_number` da `StoreEntry` referente ao objeto requisitado é interpretado da seguinte forma:

1. Os oito bits mais significativos determinam qual `cache_dir` contém o arquivo;
2. Sendo `L1` o número de subdiretórios no primeiro nível do `cache_dir` indicado e `L2` o número de subdiretórios de segundo nível, `(swap_file_number/L2) mod L2` determina o diretório de segundo nível e `(swap_file_number/L2/L2) mod L1` determina o subdiretório de primeiro nível que contém o arquivo;
3. O nome do arquivo é a representação hexadecimal de `swap_file_number`;

Uma vez determinado o arquivo que contém o objeto desejado, o Squid pode abrir este arquivo e, em seguida, começar a enviá-lo para o cliente à medida em que o mesmo é lido do disco.

3.4.3 Recuperação de objeto em servidor parceiro

Quando o Squid não encontra o objeto desejado no seu próprio cache, ele pode consultar servidores parceiros sobre a existência desse objeto e, caso algum parceiro tenha o objeto, solicitar esse objeto através de uma requisição HTTP. Essa consulta pode ser feita utilizando métodos descritos no Capítulo 2: *Internet Cache Protocol (ICP)*, *Cache Digests* ou *Cache Array Routing Protocol*.

Como já dissemos anteriormente, esses métodos de cooperação não têm boa escalabilidade e não são capazes de tirar um bom proveito dos recursos de comunicação existentes em redes locais, já que baseiam toda a sua comunicação no protocolo TCP/IP. No próximo capítulo descrevemos uma nova estratégia de cooperação entre servidores cache desenvolvida com o objetivo de tirar o máximo proveito dos recursos de comunicação existentes, eliminando os custos de consulta e reduzindo a transferência de dados entre os servidores.

Capítulo 4

SHMSquid

Como dissemos na Seção 2.3, os métodos tradicionais de cooperação entre servidores cache podem não fazer um bom uso dos recursos disponíveis quando aplicados em instalações mais específicas como redes locais de alta velocidade ou máquinas com vários processadores. Neste capítulo, apresentamos a implementação de uma nova estratégia de cooperação entre servidores cache projetada com o objetivo de eliminar os custos associados à troca de informações entre os servidores. Essa estratégia utiliza as facilidades de comunicação disponíveis para processos que estejam executando em uma mesma máquina ou ambientes de memória compartilhada distribuída (DSM)[16, 6].

Na próxima seção, fazemos uma descrição do método implementado, discutindo as suas vantagens e desvantagens quando comparado aos métodos já existentes. Em seguida, explicamos mais detalhadamente a implementação deste método no servidor Squid, a que denominamos SHMSquid.

4.1 Cooperação entre caches usando memória compartilhada

A disponibilidade de máquinas com arquitetura SMP a baixo custo vem crescendo de maneira acelerada. Isto torna viável (e possivelmente mais barato devido ao compartilhamento de vários componentes) a substituição de um grupo de servidores cache composta por uma rede de máquinas de pequeno porte por uma máquina dotada de vários processadores semelhantes aos das máquinas originais. Esta substituição traria o benefício de uma comunicação mais eficiente entre servidores executando nesta nova configuração (em comparação com servidores executando em máquinas separadas na rede original). Entretanto, o servidor de cache Squid não é capaz de tirar proveito desta nova capacidade de comunicação porque, como já explicamos, os seus métodos de cooperação são baseados no protocolo TCP/IP, não importando o meio de comunicação utilizado e ignorando recursos que possam estar sendo compartilhados pelos processos.

A fim de aproveitar as facilidades de comunicação disponíveis para processos que estejam executando em uma máquina SMP, implementamos no Squid um novo método de cooperação que permite que um servidor recupere dados existentes no cache de um parceiro sem interferir no processamento deste último. A nova estratégia funciona da seguinte forma:

1. Cada servidor cria um espaço de memória compartilhado no qual coloca a sua tabela de `StoreEntry`s;

2. Ao receber uma requisição, o servidor procura na sua própria tabela de `StoreEntry`s pela existência do objeto desejado (como já é feito normalmente);

3. Caso não encontre o objeto desejado, o servidor procura na tabela de cada um de seus parceiros pela existência do objeto;

4. Se o objeto é encontrado na tabela de algum dos parceiros, o processamento da requisição continua como descrito na Seção 3.4.2, diferindo apenas na localização do arquivo, que agora é feita com base na configuração do parceiro em lugar da própria configuração do servidor.

Fazendo a recuperação de objetos existentes no cache de parceiros desta forma, evitamos os custos de comunicação e eliminamos as várias chamadas de sistema feitas para troca de dados entre os servidores tornando o custo deste atendimento igual ao de um atendimento satisfeito pelo cache local.

A principal desvantagem deste método, com relação aos métodos já existentes, vem do fato de o parceiro não participar na transação e, portanto, não contabilizar o acesso feito. Como o acesso feito via memória compartilhada não é contabilizado pelo servidor que controla aquele objeto, a sua lista de objetos utilizados mais recentemente não fica corretamente atualizada podendo acontecer de ele eliminar do seu cache um objeto que foi acessado recentemente pelos seus parceiros (mas não por ele próprio) e conservar objetos menos acessados ocasionando uma taxa de acerto global mais baixa. Este problema, no entanto, pode ser resolvido pela implementação de um mecanismo de comunicação entre os servidores que torne possível informar ao parceiro sobre a utilização de objetos do seu cache.

4.2 Comunicação através de memória compartilhada

A variação System V dos sistemas operacionais UNIX [2] introduziu três mecanismos de comunicação entre processos que estão atualmente disponíveis na maiorias das versões de UNIX disponíveis no mercado. Entre esses mecanismos de comunicação está o uso de memória compartilhada, que descreveremos nesta seção.

A implementação de memória compartilhada no System V permite que um processo compartilhe uma região do seu espaço de endereçamento e atribua a essa região permissões de acesso para leitura e/ou escrita, discriminando entre processos do mesmo usuário, processos de usuários do mesmo grupo ou processos de usuários de outros grupos. Um outro processo pode, então, anexar essa região de memória ao seu próprio espaço de endereçamento e acessar os dados nela contidos utilizando as mesmas instruções que usaria para acessar qualquer outra parte da sua memória (obedecendo às restrições de acesso estabelecidas pelo processo criador). Para manipular regiões de memória compartilhada, um processo deve utilizar as seguintes chamadas de sistema:

- `int shmget(key, size, flags)` - Esta chamada de sistema é utilizada para criar um novo segmento de memória compartilhada ou obter o identificador de um segmento já existente. O sistema procura na tabela de memória pela chave `key`: se a encontra e o controle de acesso atribuído à região permite o acesso, retorna o identificador da região (que pode ser usado para anexar a região de memória ao espaço de endereçamento do processo). Caso não seja encontrada uma entrada associada à chave `key` e a opção `IPC_CREAT` tenha sido especificada no parâmetro `flags`, o sistema verifica que o tamanho especificado esteja entre o tamanho mínimo e máximo definidos para regiões compartilhadas e aloca a região com o tamanho especificado e retorna o seu identificador.

- `void* shmat(id,addr,flags)` - Esta chamada é usada para acoplar uma região de memória compartilhada ao espaço de endereçamento do processo. O parâmetro `id` é o valor retornado por `shmget` e identifica a região de memória a ser acoplada, `addr` é o endereço onde o usuário quer acoplar a memória compartilhada e `flags` indica se a memória vai ser usada somente para leitura (`SHM_RDONLY`) e se o sistema deve fazer alinhamento do endereço (`SHM_RND`) passado pelo usuário. O valor de retorno é o endereço onde a região foi realmente acoplada que pode ser diferente do endereço fornecido. Se o endereço fornecido na chamada a `shmat` é igual a 0, o sistema procura por um endereço onde a região de memória possa ser acoplada.

- `int shmdt(addr)` - Esta chamada remove a região de memória compartilhada que foi acoplada em `addr` do espaço de endereçamento do processo e retorna o identificador da região especificada.

- `shmctl(id,cmd,buf)` - Esta chamada é utilizada para ler ou alterar várias informações relacionadas à região de memória compartilhada identificada por `id`. O parâmetro `cmd` pode assumir os valores `IPC_STAT` (para ler informações sobre a região de memória), `IPC_SET` (para alterar informações sobre a memória compartilhada) ou `IPC_RMID` (para marcar a região para ser removida quando não houver mais processos que a estejam utilizando). O parâmetro `buf` deve ser um apontador para uma estrutura do tipo `shmid_ds`, que contém várias informações sobre a região de memória

compartilhada, como, por exemplo, permissões de acesso, última vez que foi acoplada por um processo, última vez que foi removida do espaço de endereçamento de um processo, número de referências, identificador do usuário que criou etc.

4.3 O módulo `shmlib`

Para facilitar a utilização de memória compartilhada no Squid implementamos um módulo que nos permite criar um segmento de memória compartilhada e gerenciar esse espaço fornecendo uma interface para alocação e liberação de memória semelhante às funções `calloc` e `free` da biblioteca C padrão. Esse módulo fornece as seguintes funções para manipulação da memória compartilhada:

- `int shm_init(key, size)` - Esta função cria um segmento de memória associado à chave `key` com tamanho suficiente para conter `size` bytes mais um apontador, ou seja, o tamanho mínimo do bloco é `size + sizeof(void*)`. O número de bytes disponíveis para alocação no segmento de memória inicializado por `shm_init` é igual à menor potência de 2 que seja maior ou igual ao valor `size`. O valor de retorno é igual a 0 em caso de sucesso e o endereço no qual o segmento de memória compartilhada foi acoplado é colocado na primeira palavra do segmento para permitir que outros processos traduzam apontadores que possam estar contidos nesse segmento para o seu próprio espaço de endereçamento. Esta função cria também as listas de blocos de memória livres e alocados utilizadas no gerenciamento do espaço de memória compartilhada.

- `void *shm_connect(key)` - Esta função acopla o segmento de memória associado com a chave `key` ao espaço de endereçamento do processo em modo de somente leitura e retorna o apontador para a área onde o segmento foi acoplado.

- `shm_disconnect(mem)` - Esta função retira do espaço de endereçamento do processo o segmento de memória apontado por `mem`. Caso o segmento tenha sido marcado para remoção e não haja mais processos o utilizando, ele será liberado pelo sistema.

- `shm_clear(mem)` - Esta função marca o segmento de memória apontado por `mem` para ser removido quando não houver mais processos que o estejam utilizando.

- `SHARED_PTR(base, ptr)` - Esta macro permite a um processo traduzir endereços contidos no segmento de memória compartilhada do espaço de endereçamento do processo criador do segmento para o seu próprio espaço de endereçamento de forma a poder referenciar a memória apontada por ele. O parâmetro `base` é o endereço onde o segmento foi acoplado no espaço de endereçamento do processo e `ptr` é o apontador a ser traduzido. Esta macro utiliza o fato de o segmento de memória conter em sua primeira posição o endereço inicial na memória do processo criador.

A função `shm_calloc(n, size)` aloca um bloco de tamanho suficiente para conter $n \times size$ bytes na região de memória compartilhada inicializada por `shm_init`. O algoritmo utilizado na alocação é o seguinte:

1. Fazemos `block_size` igual à menor potência de 2 que seja maior ou igual a $n \times size$;
2. Procuramos por um bloco livre de tamanho igual a `block_size` e retornamos o endereço desse bloco caso ele exista adicionando-o à tabela de blocos alocados;
3. Caso não existam blocos livres de tamanho `block_size`, procuramos por um bloco de tamanho maior e, se encontrarmos, dividimos esse bloco em partes menores até conseguirmos um bloco de tamanho `block_size`, o endereço do bloco encontrado é então retornado pela função e adicionamos esse bloco à tabela de blocos alocados;
4. Se não conseguimos encontrar um bloco de tamanho suficiente nos passos 2 e 3, a função retorna `NULL`.

A função `shm_free(addr)` é usada para liberar o bloco de memória iniciado em `addr` que foi alocado com a função `shm_calloc`. O algoritmo que implementamos na função `shm_free` é o seguinte:

1. Procuramos o endereço `addr` na tabela de blocos alocados. Se não encontramos o endereço, ele não foi alocado usando `shm_calloc` e acusamos um erro no programa;
2. Se encontramos o bloco correspondente a `addr`, passamos o seu descritor para a tabela de blocos livres;
3. Se o bloco sendo liberado é contíguo a um outro do mesmo tamanho, nós os unimos em um único bloco e o inserimos na lista de blocos com tamanho igual ao dobro do anterior. Este passo é repetido até que o bloco seja inserido em uma lista onde não haja blocos contíguos a ele.

4.4 Implementação da cooperação usando memória compartilhada no Squid

Neste trabalho, fizemos a implementação de um mecanismo de colaboração entre servidores cache que permite que um processo recupere objetos existentes no cache de um outro processo sem a intervenção deste último, buscando reduzir o custo de colaboração entre servidores que estejam sendo executados em uma mesma máquina.

Nesta seção descrevemos os pontos principais da implementação deste mecanismo de colaboração.

4.4.1 Configuração e compartilhamento dos dados

Para permitir o compartilhamento de dados entre os servidores Squid através de memória compartilhada, adicionamos os seguintes parâmentros ao arquivo de configuração:

- `shm_key key` - Este é um parâmetro obrigatório no SHMSquid. Ele determina que `key` será a chave utilizada para criação do segmento de memória compartilhada. Este valor será usado pelos parceiros existentes na mesma máquina para acessar os dados compartilhados pelo servidor. O parâmetro `key` deve ser um valor inteiro.

- `shm_size size` - Este parâmetro determina o tamanho do segmento de memória compartilhada a ser utilizada pelo servidor. Este valor deve ser grande o suficiente para todas as `StoreEntry`s que serão criadas durante o seu funcionamento. Este parâmetro pode ser omitido, sendo assumido o valor padrão de 8 MB.

- `shm_peer key` - Este parâmetro é utilizado para indicar a chave utilizada por outro servidor para alocar a sua memória compartilhada, permitindo ao Squid acessar os dados que esse servidor colocou nessa área. Deve haver uma entrada `shm_peer` para cada servidor com o qual queiramos colaborar via memória compartilhada e o valor `key` deve ser igual ao valor especificado por `shm_key` na configuração dos outros servidores.

A estrutura de configuração `Config` foi modificada para incluir os novos dados de configuração sendo que o campo `shm_peers` possui uma lista de parceiros contendo, para cada um deles, um apontador para a sua tabela de `StoreEntry`s, a sua configuração de diretórios de cache e o endereço onde se inicia a memória compartilhada daquele servidor.

Após a leitura do arquivo de configuração, o Squid cria o segmento de memória compartilhada definido no arquivo de configuração (usando a função `shm_init`) e aloca espaço no início desse segmento para uma estrutura contendo dois apontadores: um para a cópia das suas configurações de diretórios cache colocada na memória compartilhada e outro para a tabela *hash* contendo as suas `StoreEntry`s (que passam a ser todas alocadas através da função `shm_calloc`).

Para que um processo possa distinguir as suas próprias `StoreEntry`s das criadas por outros processos, acrescentamos à estrutura da `StoreEntry` o campo `owner_key` onde o criador da entrada armazena a chave definida por `shm_key` no arquivo de configuração. Desta forma, o servidor é capaz não só de saber que aquela entrada pertence a outro processo como também identificar na sua configuração os dados necessários para recuperar o objeto a que a entrada se refere.

4.4.2 Recuperação de um objeto em cache de outro servidor

Nesta seção iremos descrever o mecanismo de recuperação de objetos que se encontram no cache de servidores que estão configurados para colaborar através da utilização de memória compartilhada.

O servidor SHMSquid coloca todas as suas `StoreEntry`s na região de memória compartilhada, de forma que um outro processo possa consultar o conteúdo do seu cache sem que, para isso, seja necessário interferir no seu processamento. Para evitar condições de corrida que poderiam surgir no caso de haver um compartilhamento total desses dados (com processos podendo modificar as `StoreEntry`s de outros processos), decidimos que um servidor acessaria a memória de outros em modo de somente leitura.

Como mencionado na Seção 4.1, a maior parte do atendimento feito utilizando dados obtidos via memória compartilhada é semelhante ao atendimento de uma requisição usando dados do próprio servidor. Aqui iremos descrever os pontos em que estes atendimentos diferem.

Durante o atendimento normal, o Squid cria uma estrutura `MemObject` associando-a à `StoreEntry` encontrada. Esta estrutura contém dados como o tamanho do objeto sendo transmitido, quanto está em memória (já foi lido do disco) e quanto já foi enviado, servindo para o Squid controlar o envio do objeto. Mas, para fazer o atendimento utilizando os dados de outro processo, não podemos modificar a `StoreEntry` e, por isso, não podemos fazer a associação do `MemObject` como o Squid faz normalmente. Desta maneira, precisamos criar uma nova estrutura chamada `ShmHitCallbackData` que utilizamos durante o atendimento. Essa estrutura contém apontadores para a `StoreEntry` encontrada, o `MemObject`, a requisição sendo atendida e a configuração de diretórios do servidor que controla o objeto desejado.

Como a maior parte das rotinas utilizadas no atendimento de uma requisição recebe uma `StoreEntry` como parâmetro e acessa os outros dados necessários (`MemObject`, por exemplo) através dela, precisamos implementar novas rotinas que recebem, como parâmetro, `ShmHitCallBackData` e fazem acessos aos dados necessários através dessa estrutura. Estas rotinas, no entanto, foram implementadas seguindo a mesma filosofia de atendimento utilizada no tratamento de um *hit*. Desta forma, apesar de o atendimento a um *hit* local e a um *hit* em memória compartilhada serem realizados por conjuntos diferentes de rotinas, eles são feitos de maneira semelhante.

Como esta nova estratégia não traz custos adicionais devidos à cooperação entre os servidores cache, esperamos que ela venha a ser uma opção mais eficiente na implementação de servidores cache escaláveis, ou seja, servidores que possam ser utilizados para a formação de grupos maiores sem haver degradação no seu desempenho.

Capítulo 5

Análise de desempenho do SHMSquid

Neste capítulo descrevemos os experimentos realizados a fim de verificar a eficiência da estratégia de cooperação proposta no último capítulo e discutimos os resultados obtidos. Na próxima seção definimos as métricas de desempenho que utilizamos para comparar os resultados dos experimentos. Em seguida definimos os critérios de análise que utilizamos durante o trabalho e que serviram para definir os experimentos realizados. Finalmente, apresentamos os resultados obtidos e discutimos estes resultados, nas duas últimas seções.

5.1 Métricas de desempenho

A análise de desempenho de servidores cache aqui apresentada se baseará nas seguintes métricas:

Latência: é o tempo necessário para o servidor atender a uma requisição de um cliente. Esta métrica é composta por três medidas:

- **cliente** - Esta medida é a média do tempo de resposta aos clientes que acessam os servidores. O tempo de resposta de cada requisição é calculado sob a perspectiva do cliente, sendo o tempo decorrido entre o pedido de conexão e o final do recebimento do objeto pedido.
- **hit** - Esta medida equivale à mediana do tempo de serviço para prover objetos constantes do cache local do servidor. Este tempo de serviço é calculado pelo servidor como o tempo decorrido entre o recebimento do pedido de conexão e a conclusão do envio do objeto pedido.
- **miss** - Esta medida equivale à mediana do tempo de serviço para objetos que não foram encontrados no cache local do servidor. O cálculo deste valor é feito pelo servidor da mesma forma que o cálculo do valor `hit`.

Eficiência: é a parcela das requisições recebidas pelo servidor que resulta em um acerto no cache (i.e., o objeto está no cache do servidor). Essa métrica é composta de duas medidas:

- **local** - Esta medida equivale à parcela das requisições recebidas que o servidor responde com objetos do seu próprio cache.
- **cooperação** - Esta medida equivale à parcela das requisições recebidas que o servidor responde com objetos encontrados no cache de seus parceiros (i.e., outros servidores cache com os quais ele coopera).

Taxa de serviço: Através desta métrica, poderemos verificar a quantidade de trabalho realizado por um servidor por unidade de tempo. Esta métrica é baseada nas seguintes medidas feitas pelo servidor Squid:

- **requisições** - Esta medida equivale ao número médio de requisições HTTP atendidas pelo servidor por segundo.
- **disco** - Esta medida equivale ao número médio de operações de disco feitas pelo servidor por segundo.
- **rede** - Esta medida equivale ao número médio de operações de rede feitas pelo servidor por segundo.

Utilizando essas métricas, é possível comparar o desempenho das várias configurações utilizadas, identificando os ganhos e perdas devidos às estratégias utilizadas.

5.2 Critérios de análise de desempenho

Nesta seção, descrevemos os quatro critérios em que nos baseamos para definir os experimentos a serem realizados e discutimos os possíveis ganhos e perdas que a adoção ou não de cada um deles em uma configuração podem trazer. Nesta discussão consideramos a utilização de dois servidores Squid, apesar de esses critérios serem aplicáveis para grupos maiores de servidores. Em [21] analisamos o comportamento do SHMSquid em configurações com maior número de processadores.

Cooperação - A utilização de cooperação entre servidores pode trazer um ganho significativo em taxa de acerto no cache. Por outro lado, esta mesma cooperação pode causar um aumento no tempo de serviço de requisição. Comparando experimentos onde há cooperação entre os servidores com outros onde essa cooperação não existe, poderemos determinar os ganhos e perdas criados pela cooperação.

Multiprocessamento - A utilização de uma mesma máquina para executar os dois servidores pode trazer o benefício de uma comunicação mais eficiente entre eles, minimizando, desta forma, o aumento no tempo de serviço causado pela cooperação.

Por outro lado, o compartilhamento dos recursos da máquina entre os dois servidores pode diminuir ou até anular os benefícios de uma comunicação mais eficiente. Comparando experimentos onde há multiprocessamento com experimentos nos quais os servidores são executados em máquinas separadas, poderemos avaliar os custos e benefícios da adição do multiprocessamento.

Conectividade - Ao utilizarmos uma mesma máquina para executar os dois servidores, podemos configurar os dois para acessarem a rede através de uma mesma interface ou, se dispusermos de mais de uma interface de rede, configurá-los para acessarem a rede através de interfaces diferentes. Ao compartilharem a mesma interface de rede, os servidores podem ter o seu desempenho reduzido devido à disputa pelo uso simultâneo desse recurso. Comparando experimentos em que os servidores compartilham a mesma interface de rede com experimentos nos quais eles utilizam interfaces diferentes para acessar a rede, poderemos determinar o ganho de termos interfaces distintas para cada um dos servidores.

Compartilhamento - O compartilhamento de memória entre os servidores Squid pode proporcionar um aumento na eficiência da cooperação entre os servidores, já que, desta forma, a troca de dados entre os dois pode ser feita de uma forma mais rápida do que a comunicação através de mensagens TCP/IP utilizadas normalmente. Comparando configurações que utilizam o compartilhamento de memória com configurações que não a utilizam, poderemos determinar o ganho advindo deste compartilhamento e possíveis perdas relacionadas com a nova estratégia de cooperação.

Tomando como base os critérios descritos, definimos os experimentos a realizar a fim de verificar o desempenho do servidor Squid em diferentes ambientes e, principalmente, analisar o desempenho da nova estratégia de cooperação que desenvolvemos. A seguir, descrevemos mais detalhadamente cada uma das configurações utilizadas e a motivação para elas.

- **double** - esta configuração corresponde ao modo mais usual de utilização do servidor Squid na composição de grupos de servidores cache. Ela consiste de duas máquinas executando, cada uma delas, um servidor Squid, esses servidores cooperam entre si através de ICP e *Cache Digests*. Esta configuração, por ser a forma mais comum de composição de um grupo de servidores, servirá como base de comparação.

- **dual** - nesta configuração, utilizamos, como servidora de cache, uma máquina dotada de dois processadores e duas interfaces de rede. Essa máquina executa dois servidores Squid, cada um deles fazendo acesso à rede através de uma das interfaces. A comparação entre os resultados obtidos com esta configuração e os resultados da configuração `double` provê informações sobre os ganhos de uma comunicação mais eficiente entre os dois servidores (já que eles não precisam mais utilizar a rede para as suas comunicações). Por outro lado, pode haver uma perda de desempenho por causa

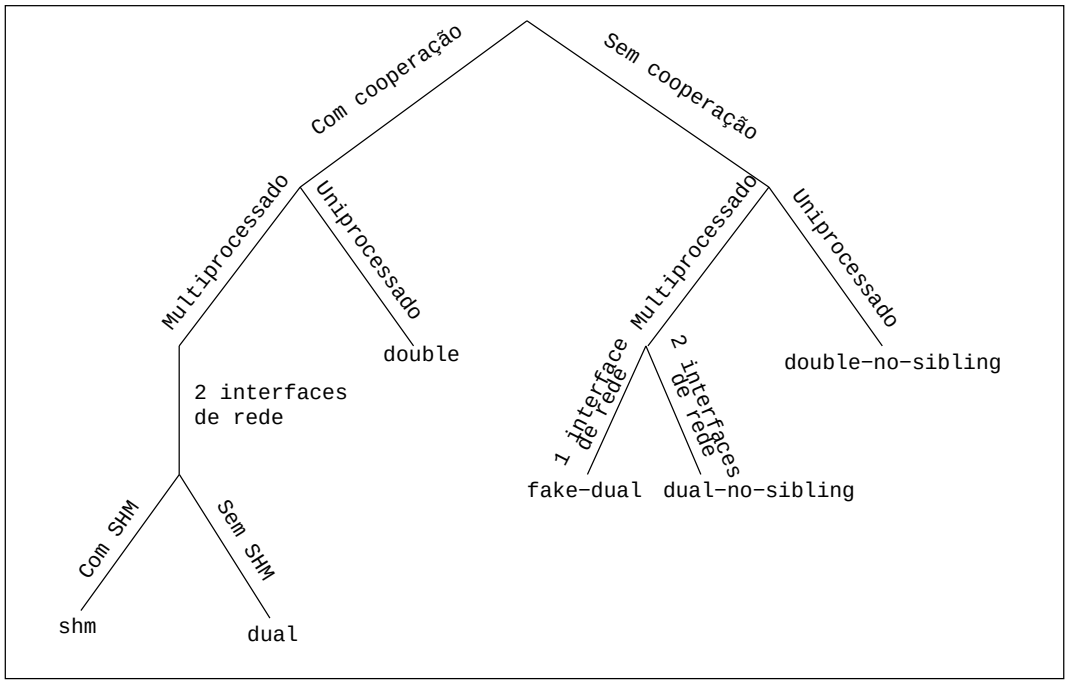

Figura 5.1: Relação entre os critérios de análise e as configurações utilizadas.

do compartilhamento de recursos do sistema, devido ao grande número de chamadas de sistema.

- **double-no-sibling** - nesta configuração, utilizamos duas máquinas servidoras, cada uma delas executando um servidor Squid sem utilizar cooperação entre eles. Comparando os resultados dessa configuração com os resultados da configuração `double`, poderemos determinar os ganhos e prejuízos advindos da cooperação entre os servidores.

- **dual-no-sibling** - nesta configuração, utilizamos uma máquina com dois processadores e duas interfaces de rede para executar dois servidores Squid sem que eles cooperem entre eles. Através da comparação dos resultados desta configuração com os da configuração `double-no-sibling`, pretendemos descobrir qual a perda de desempenho devida ao fato de os dois servidores estarem compartilhando a mesma máquina. As perdas observadas nesta comparação devem ser originadas, principalmente, do fato de alguns segmentos de código do *kernel* não poderem ser executados simultaneamente por dois processadores em modo *kernel*.

- **fake-dual** - nesta configuração, utilizamos uma máquina com dois processadores executando dois servidores Squid que não cooperam entre si. Os dois servidores, no entanto, atendem requisições utilizando uma mesma interface de rede. Através da comparação entre os resultados desta configuração e os resultados da `dual-no-sibling`, pretendemos determinar o ganho proporcionado pela utilização de mais de uma in-

terface de rede.

- **shm** - nesta configuração, utilizamos uma máquina dotada de dois processadores e duas interfaces de rede para executar dois servidores Squid cooperando através do uso de memória compartilhada. Comparando os resultados dessa configuração com os resultados das configurações `double` e `dual`, poderemos determinar os ganhos e perdas no desempenho do servidor devidos à utilização dessa nova estratégia de cooperação.

Analisando os dados obtidos através destes experimentos, pretendemos verificar a eficiência da cooperação através de memória compartilhada na implementação de servidores cache escaláveis.

5.3 Ambiente experimental

Nos experimentos realizados durante este trabalho, utilizamos a metodologia para análise de desempenho de hierarquias de servidores cache sugerida em [11].

Utilizamos como servidores cache dois PCs multiprocessados, com processadores PentiumPro de 200MHz e 128Mb de memória cada um, executando o sistema operacional Solaris 7. Essas máquinas se comunicam através de um cabo serial, com velocidade de 38.400bps. Em todos os experimentos, os servidores Squid foram configurados para utilizar 20Mb de memória e 50Mb de espaço em disco.

Como servidores HTTP, utilizamos 3 estações SPARCstation SLC executando, cada uma delas, um processo que faz o atendimento de requisições HTTP gerando um arquivo baseado na URL pedida e introduzindo um atraso na resposta proporcional ao tamanho desse arquivo simulando, desta forma, os atrasos experimentados em acessos de longa distância através da Internet.

As requisições de objetos foram geradas por quatro estações SPARCstation SLC (duas acessando cada servidor cache), utilizando uma carga de trabalho baseada em *logs* do servidor cache do POP-MG[13]. Analisamos um *log* que contém 4.235.511 requisições para 1.079.044 objetos diferentes totalizando aproximadamente 12 Gb de dados. Para os nossos experimentos, geramos um conjunto de 80,000 requisições, conservando as características de popularidade de documentos e distribuição das requisições no tempo obtidas através dessa análise. Este conjunto de requisições é dividido entre os clientes, de forma que cada um deles gera uma seqüência de 20.000 requisições HTTP para os servidores cache.

Todas as estações SPARCstation SLC estão ligadas aos servidores cache através de uma rede Ethernet de 10Mbps chaveada.

5.4 Análise dos resultados

Nesta seção, faremos uma análise dos resultados obtidos nos experimentos que realizamos. As medidas são apresentadas nas Tabelas 5.1 (latência), 5.2 (eficiência) e 5.3 (taxa de serviço). Analisaremos esses resultados nas próximas seções, tomando como base os critérios apresentados na Seção 5.2.

Configuração	Cliente	Hit	Miss
double-no-sibling	7.3100	0.0028	3.2853
double	6.6830	0.0042	2.6625
dual-no-sibling	7.8090	0.0037	3.5638
dual	7.1829	0.0046	2.7939
fake-dual	7.8117	0.0037	3.5638
shm	6.3365	0.0028	2.5775

Tabela 5.1: Latência do atendimento de clientes (segundos).

Configuração	Local	Cooperação	Total
double-no-sibling	0.3097	-	0.3097
double	0.2750	0.1822	0.4567
dual-no-sibling	0.3127	-	0.3127
dual	0.2732	0.1927	0.4662
fake-dual	0.3150	-	0.3150
shm	0.2432	0.1410	0.3842

Tabela 5.2: Taxa de acerto no cache.

Configuração	Requisições	Disco	Rede
double-no-sibling	3.1475	22.8480	53.2459
double	3.8325	25.9467	74.0923
dual-no-sibling	3.1705	22.6536	52.1093
dual	3.9010	24.8328	70.0355
fake-dual	3.1644	22.6238	51.8748
shm	3.6044	25.4615	55.6723

Tabela 5.3: Taxa de serviço do servidor (operações por segundo).

5.4.1 Cooperação

Ao comparamos os dados de latência das configurações que utilizam cooperação com os dados das configurações em que os servidores não cooperam (comparando `double` com `double-no-sibling` e `dual` com `dual-no-sibling`) podemos verificar que a cooperação proporciona um tempo médio de resposta menor devido a uma maior taxa de acerto (aproximadamente 50% mais alta nas configurações que utilizam cooperação). Podemos ver,

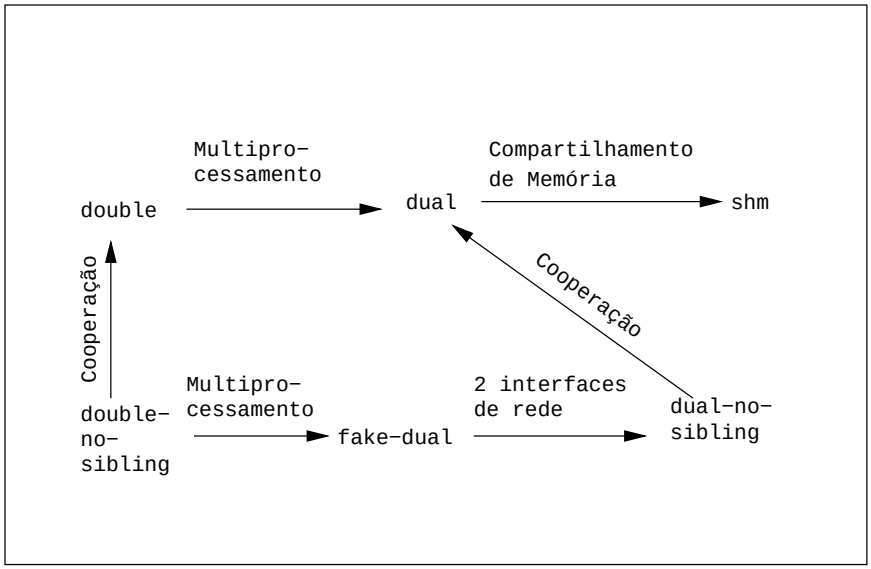

Figura 5.2: Diferenças entre as várias configurações dos testes executados

no entanto, que a latência para os *hits* é maior nas configurações em que há cooperação, isso se deve ao fato de que a cooperação entre os servidores impõe uma carga maior de trabalho, aumentando o número de conexões abertas simultaneamente em cada um deles (podemos observar na Tabela 5.3 que os servidores que cooperam têm uma atividade de rede aproximadamente 40% mais intensa do que os servidores que não cooperam) levando a um tempo maior na execução da maioria das operações. Para as requisições que resultam em *miss*, no entanto, esse aumento é compensado pelo fato de grande parte dessas requisições passarem a ser resolvidas consultando um outro servidor cache (em lugar de recorrer ao servidor HTTP que possui o objeto), resultando em um tempo menor de serviço para a maioria dessas conexões.

5.4.2 Multiprocessamento

Ao compararmos os dados de tempo de serviço das configurações que utilizam multiprocessamento com os dados das configurações que utilizam máquinas separadas para cada servidor (comparando `dual-no-sibling` com `double-no-sibling` e `dual` com `double`) podemos ver que a utilização de multiprocessamento causa um aumento no tempo médio de resposta dos servidores, apesar de as taxas de acerto permanecerem as mesmas. Como era esperado, esse menor desempenho dos servidores que utilizam multiprocessamento é devido à serialização de operações (principalmente execução de trechos do *kernel* que não podem ser paralelizados)[3]. Analisando a Tabela 5.3 podemos ver que as configurações multiprocessadas possuem atividade de disco e de rede ligeiramente menor que as configurações sem multiprocessamento, indicando a serialização das operações de entrada e saída de dados e comunicação como a causa do desempenho pior.

5.4.3 Conectividade

Comparando os dados de tempo de serviço da configuração que utiliza apenas uma interface de rede da máquina multiprocessada (`fake-dual`) com os da configuração equivalente que utiliza as duas interfaces (`dual-no-sibling`) podemos ver que a utilização de uma interface extra não proporcionou ganho de desempenho sob essa carga de trabalho. Entretanto, quando comparamos estas configurações trabalhando em situações de comunicação mais intensa (com a mesma carga de trabalho sem simular o atraso nos servidores Web), a configuração `fake-dual` apresentou um desempenho inferior ao da configuração `dual-no-sibling` (o tempo médio de resposta na configuração `fake-dual` foi aproximadamente 10% maior), mostrando que, mesmo que estejamos executando mais de um servidor na mesma máquina, o uso de mais de uma interface de rede traz benefícios apenas quando a quantidade de comunicação feita pelo servidor aproxima-se da capacidade máxima da interface.

5.4.4 Compartilhamento

Comparando os dados da configuração que utiliza compartilhamento de memória (`shm`) com os dados das configurações que não o fazem, podemos ver que o tempo médio de resposta desta configuração é o mais baixo de todos e o seu tempo de serviço de *hit* é igual ao tempo de serviço de *hit* da configuração `double-no-sibling`. Isso se deve ao fato de que a colaboração através de memória compartilhada não impõe um processamento de conexões extras ao servidor. Pelo mesmo motivo, o tempo de serviço de *miss* também é mais baixo.

Na Tabela 5.2, vemos que a taxa de acerto na configuração `shm` é mais baixa do que a das outras configurações em que há cooperação entre os servidores. Como mencionamos no Capítulo 4 isso se deve ao fato de que, como o servidor que possui o objeto não participa no processo de resolução de um *hit* através da memória compartilhada, a sua lista de objetos mais recentemente utilizados não é atualizada como no caso de uma cooperação normal, resultanto na eliminação de objetos que são muito acessados e, por isso, deveriam permanecer no cache.

Podemos ver na Tabela 5.3 que a configuração `shm`, quando comparada com a configuração `double`, tem um número menor de requisições atendidas por segundo, e menor atividade de disco e de rede (sendo a atividade de rede aproximadamente 40% mais alta na configuração `double`), apesar de o seu tempo de resposta a requisições ser menor. Isso se deve ao fato de que quase 20% das requisições recebidas de clientes (*hits* resolvidos pelo parceiro) são resolvidas na configuração `double` através da criação de uma conexão HTTP com outro servidor cache gerando, desta forma, um tráfego extra na rede.

Durante a execução de experimentos com o SHMSquid, uma vez encontrado o arquivo contendo o objeto desejado no cache do parceiro, pudemos detectar a ocorrência de alguns erros que obrigavam o servidor cache a ir buscar o objeto no servidor HTTP original. Os

tipos de erros ocorridos são os seguintes:

remoção do arquivo - como o servidor parceiro não sabe que um determinado objeto do seu cache está sendo acessado por outro servidor, ele pode vir a remover esse objeto enquanto o atendimento está sendo feito. Essa remoção pode ocorrer entre a localização do objeto na tabela de `StoreEntry`s e a tentativa de abertura do arquivo ou durante a leitura do arquivo.

objeto inválido no cache - o objeto encontrado no cache do parceiro pode não ter sido validado recentemente e por isso o servidor cache não pode utilizá-lo no atendimento da requisição por correr o risco de ele ter sido modificado desde a última vez em que foi recuperado do servidor original.

Verificamos, no entanto, que estes erros ocorrem com uma freqüência muito baixa (aproximadamente 1 vez para cada 4.000 requisições), tendo ocorrido 91 remoções de arquivo (63 durante a leitura e 28 antes da abertura do arquivo) e 13 acessos a objetos inválidos durante a execução de 400.000 requisições (5 execuções completas do experimento).

5.5 Discussão

Como esperado, constatamos que a cooperação entre servidores traz um ganho significativo na taxa de acertos e no tempo médio de resposta. Por outro lado, pudemos observar um aumento no tempo de serviço para os *hits* (que são as requisições de menor tempo de atendimento), o que sugere que um aumento no número de servidores cooperantes pode levar a uma degradação do sistema como um todo.

Observamos que, apesar de uma máquina multiprocessada oferecer um meio de comunicação mais rápido entre os processos servidores, as vantagens que poderiam surgir da utilização deste meio de comunicação mais rápido são superadas pelas desvantagens geradas pelo compartilhamento de recursos que não podem ser usados em paralelo, resultando em uma pequena perda de desempenho.

Pudemos perceber que a utilização de mais de uma interface de rede proporciona um ganho de desempenho em uma configuração multiprocessada. Este, no entanto, só pode ser observado quando os servidores estão sujeitos a uma carga de comunicação muito intensa.

Finalmente, verificamos que a nova estratégia proposta proporciona um tempo médio de resposta melhor que o das outras configurações sem que haja perda de eficiência no tratamento de *hits* (como acontece na cooperação normal entre servidores cache). Vimos que esse melhor desempenho é devido ao fato de um servidor não interferir no funcionamento do outro. Esta estratégia, no entanto, teve uma taxa de acerto mais baixa que as outras configurações onde havia cooperação entre os servidores porque o servidor que possui um objeto que é recuperado em um *hit* através de memória compartilhada não tem a sua lista de objetos acessados mais recentemente atualizada, resultando na retirada de objetos populares do cache.

Capítulo 6

Conclusões e trabalhos futuros

A taxa de crescimento da utilização da *World Wide Web* tem sido muito grande nos últimos anos, provocando um grande aumento no tráfego de dados na Internet. Embora o uso de servidores cache WWW tenha ajudado a reduzir esse tráfego de dados, as estratégias de cooperação entre servidores cache utilizadas atualmente podem resultar em uma degradação no seu desempenho, não sendo, portanto, escaláveis o suficiente para acompanhar o crescimento atual da WWW.

Neste trabalho, apresentamos uma nova estratégia de cooperação (uso de memória compartilhada) entre servidores cache projetada com o objetivo de permitir que os servidores cache de um grupo cooperem entre si sem que haja uma degradação no seu desempenho.

Pudemos verificar que a nova estratégia apresentada permite que um servidor se utilize de dados existentes no cache do seu parceiro sem interferir no desempenho deste último, alcançando, dessa forma, um desempenho melhor que o conseguido através de outros meios de cooperação e mostrando ser uma boa opção na implementação de servidores cache escaláveis.

Apesar de o uso de memória compartilhada ter proporcionado menores tempos de resposta aos clientes, pudemos observar que a taxa de acerto conseguida não foi tão boa quanto a que se consegue através das formas tradicionais de cooperação. Essa diferença observada é devida ao fato de as listas de objetos utilizados mais recentemente, usadas pelo servidor quando precisa decidir quais objetos retirar do seu cache, não são atualizadas quando um servidor acessa o cache do seu parceiro através de memória compartilhada, provocando uma distorção na política de reposição de objetos do cache.

Este problema de atualização das listas de utilização de objetos pode ser resolvido através da utilização de mecanismos de sincronização que permitam que um servidor informe ao seu parceiro quando fizer uso dos dados do seu cache. Implementando estes mecanismos de sincronização, esperamos obter taxas de acerto semelhantes às obtidas através dos meios normais de cooperação sem que haja uma queda no desempenho do servidor.

Apesar dos ganhos obtidos com a utilização do SHMSquid, sabemos que uma das grandes barreiras para o desenvolvimento de servidores escaláveis para a WWW é o suporte

inadequado dos sistemas operacionais[3]. Por isso, deve-se investigar novos mecanismos para dar suporte à execução de servidores Web, não só criando novas interfaces de programação que evitem os problemas já evidenciados, como também melhorando o suporte a multiprocessamento em sistemas correntes.

Bibliografia

[1] Apache. `http://www.apache.org/`.

[2] M. J. Bach. *The Design of the UNIX operating system*. Prentice-Hall Software Series. P T R Prentice-Hall, Inc., 1986.

[3] G. Banga, P. Druschel, and J. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance*, Madison, WI, June 1998.

[4] T. Berners-Lee, R. Cailliau, A. Luotonen, H. Nielsen, and A. Secret. The World Wide Web. In *Communications of the ACM*, volume 37, August 1994.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – http/1.0. RFC 1945, May 1996.

[6] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. Amorim. Hiding communication latency and coherence overhead in software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston,MA, October 1996.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit per second local area network. *IEEE-Micro*, 1995.

[8] Cisco Systems. *Cisco Cache Engine*. `http://www.cisco.com/univercd/cc/td/doc/product/webscale/webcache/`.

[9] P. B. Danzig, R. S. Hall, and M. F. Shwartz. A Case for Caching File Objects Inside Internetworks. Technical Report CU-CS-642-93, University of Colorado at Boulder, March 1993. `ftp://ftp.cs.colorado.edu/pub/cs/techreports/shwartz/FTP.CachingPS/Paper.ps.Z`.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2068, January 1997.

[11] E. Fonseca. Hierarquia de Servidores Proxy Cache WWW: Instrumentação e Análise de Desempenho. Master's thesis, Universidade Ferderal de Minas Gerais, 1999.

[12] S. Gade, M. Rabinovich, and J. Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. `http://www.cs.duke.edu/ari/cisi/crisp-recycle/`.

[13] W. Meira Jr., E. L. da Silva Fonseca, V. A. F. Almeida, and C. D. Murta. Performance Analysis of WWW Cache Proxy Hierarquies. *Journal of the Brazilian Computer Society*, 5(2), Novembro 1998.

[14] W. Meira Jr., E. Fonseca, V. Almeida, and C. Murta. Analyzing performance of cache server hierarchies. In *Proceedings of SCCC'98*, Antofagasta, Chile, November 1998.

[15] W. Meira Jr., E. Fonseca, V. Almeida, and C. Murta. Evaluating and configuring WWW caches. In *3rd International WWW Caching Workshop*, Manchester, UK, June 1998.

[16] W. Meira Jr., T. J. LeBlanc, N. Hardavellas, and C. Amorim. Understanding the performance of DSM applications. In *Proceedings of the Workshop on Communication and Architectural Support for Network-based Computing (CANPC)*, volume 1199 of *Lecture Notes in Computer Science*, pages 98–211, San Antonio, TX, February 1997. IEEE, Springer-Verlag.

[17] Microsoft Corporation. *Cache Array Routing Protocol and Microsoft Proxy Server 2.0*, 1998. `http://www.microsoft.com/proxy/documents/CarpWP.exe`.

[18] J. Mogul. Speedier Squid: A case study of an Internet Server Performance Problem. *;login: The USENIX Association Magezine*, February 1999.

[19] National Laboratory for Applied Network Research. *Squid Internet Object Cache*. http://squid.nlanr.net/Squid/.

[20] Netscape communications. `http://www.netscape.com/company/`.

[21] G. Paixão, W. Meira Jr., and F. Sanches. Servidores cache www em arquiteturas multiprocessadas. In *Anais do XI SBAC-PAD*, pages 319–323, Natal,RN, Setembro 1999. Sociedade Brasileira de Computação.

[22] G. Paixão, W. Meira Jr., and F. Sanches. Shmsquid: a scalable www cache server. In *Proceedings of the International Conference of the Chilean Computer Science Society*, pages 187–194, Talca, Chile, November 1999. Chilean Computer Science Society, IEEE Computer Science Press.

[23] R. Rivest. The MD5 Message-Digest Algorithm. Network Working Group RFC 1321, April 1992. `http://ds.internic.net/rfc/rfc1321.txt`.

[24] K. W. Ross. Distribution of Stored Information in the Web. `http://www.eurecom.fr/~ross/CacheTutorial/DistTutorial.html`, 1998.

[25] A. Rousskov. On Performance of Caching Proxies. `http://www.cs.ndsu.nodak.edu/~rousskov/research/cache/squid/profiling/papers/`, 1997.

[26] A. Rousskov and D. Wessels. Cache digests. `http://ircache.nlanr.net/Papers/cache-digests.ps.gz`, April 1998.

[27] W. R. Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[28] D. Wessels and K. Claffy. Internet Cache Protocol(ICP), version 2. Network Working Group RFC 2186, September 1997.