Avaliação de Mecanismos para Acesso a Bancos de Dados Heterogêneos através da *Web*

Edmar Edilton da Silva

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Avaliação de Mecanismos para Acesso a Bancos de Dados Heterogêneos através da *Web*

Edmar Edilton da Silva[1]

Setembro de 2001

Banca Examinadora:

- Prof. Dr. Célio Cardoso Guimarães
  Instituto de Computação - IC/Unicamp (Orientador)
- Prof. Dr. Ivan Luiz Marques Ricarte
  Faculdade de Engenharia Elétrica e Computação - FEEC/Unicamp
- Profa. Dra. Claudia Maria Bauzer Medeiros
  Instituto de Computação - IC/Unicamp
- Prof. Dr. Geovane Cayres Magalhães
  Instituto de Computação - IC/Unicamp

[1]Projeto "Acesso a Bancos de Dados Heterogêneos", financiado pela Compaq Brasil através do Termo Aditivo nº 7 do Convênio de Cooperação Tecno-Científica entre a Unicamp e a Compaq Brasil.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Silva, Edmar Edilton da

Si38a Avaliação de mecanismos para acesso a bancos de dados heterogêneos através da Web / -- Campinas, [S.P. :s.n.], 2001.

Orientador : Célio Cardoso Guimarães

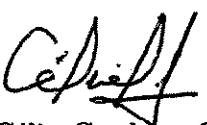
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Linguagem de programação (Computador). 2. World Wide Web (Sistema de recuperação da informação). 3. Banco de dados relacionais. 4. Java (Linguagem de programação). 5. Perl (Linguagem de programação de computador). I. Guimarães, Célio Cardoso. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Avaliação de Mecanismos para Acesso a Bancos de Dados Heterogêneos através da *Web*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Edmar Edilton da Silva e aprovada pela Banca Examinadora.

Campinas, 14 de Setembro de 2001.



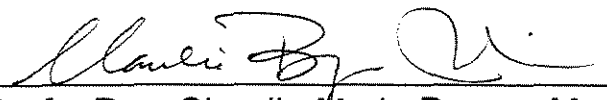
Prof. Dr. Célio Cardoso Guimarães
Instituto de Computação - IC/Unicamp
(Orientador)

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

TERMO DE APROVAÇÃO

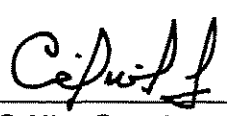
Tese defendida e aprovada em 14 de setembro de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Ivan Luiz Marques Ricarte
FEEC - UNICAMP



Profa. Dra. Claudia Maria Bauzer Medeiros
IC - UNICAMP

Prof. Dr. Geovane Cayres Magalhães
IC - UNICAMP



Prof. Dr. Célio Cardoso Guimarães
IC - UNICAMP

Resumo

A *World Wide Web* é atualmente considerada a plataforma tecnológica ideal para o desenvolvimento de aplicações na Internet e *intranets* corporativas. Na maioria dos casos, essas aplicações acessam SGBDRs. O padrão CGI existente é muito ineficiente em períodos de grande volume de carga no servidor *Web*. A natureza do acesso a SGBDs através da *WWW* é completamente diferente das aplicações de bancos de dados tradicionais. A diferença básica está no caráter sem estado (*stateless*) do protocolo HTTP. O conceito de sessão de usuário, encontrado em aplicações de bancos de dados tradicionais, não se aplica na *WWW*.

Esta dissertação compara algumas APIs de código aberto usadas para acessar bancos de dados heterogêneos. Essas APIs são muito usadas no desenvolvimento de aplicações de bancos de dados com preservação de estado (*stateful*) no contexto da *Web*. Nessas arquiteturas, um *pool* de conexões persistentes é usado para eliminar o custo associado à abertura de uma nova conexão com o SGBD. O ganho de desempenho é medido através de uma série de experimentos em que é simulado o tráfego gerado por vários clientes HTTP concorrentes. As APIs examinadas foram: Java Servlet (JDBC), Perl (DBI) e Python (*Database* API).

Abstract

The World Wide Web is currently considered the ideal technological platform for the development of applications on the Internet and corporate intranets. In the majority of the cases, such applications access relational DBMSs. The current CGI standard is very inefficient under periods of heavy workload on the Web server. The nature of the access to DBMSs through the WWW is quite different from the traditional database applications. The basic difference is in the stateless characteristic of the HTTP protocol. The user session concept, found in traditional database applications, does not apply on the WWW.

This dissertation compares some open source APIs used to access heterogeneous databases. These APIs have often been used for the development of stateful database applications in the context of the Web. In these architectures, a persistent connection pool is used in order to eliminate the cost of establishing new connections to the DBMSs. The performance gain is assessed through a series of measurements in which we emulate the traffic caused by several concurrent HTTP clients. The APIs assessed were: Java Servlet (JDBC), Perl (DBI) and Python (Database API).

Para ser grande, sê inteiro: nada
Teu exagera ou exclui.

Sê todo em cada coisa. Põe quanto és
No mínimo que fazes.

Assim em cada lago a lua toda
Brilha, porque alta vive.

Fernando Pessoa
Odes de Ricardo Reis, 1933

Agradecimentos

"Combati o bom combate, completei a carreira, guardei a fé."

(2º Timóteo 4:7)

- Agradeço primeiramente a Deus pelo dom da vida e por conduzir sempre os meus passos com amor supremo, sendo o motivo maior de toda minha felicidade e existência. Agradeço por ter me amparado nos momentos mais difíceis e por ter me tornado forte quando eu me achava fraco.
- Ao meu orientador, Prof. Dr. Célio Cardoso Guimarães, pela excelente orientação e acima de tudo pela amizade. Suas preciosas idéias, seu entusiasmo e sua dedicação foram fundamentais para a realização deste trabalho.
- Aos meus pais, José e Madalena, e a todos meus irmãos, Sandra, Vilson, Wilson e José Edson†, pelo incentivo e carinho em todos os momentos. Agradeço aos meus irmãos Marco Antônio e Edilson, que em especial foram responsáveis pela conclusão desta etapa de minha vida. Obrigado pela força e principalmente por me manterem financeiramente durante parte de minha graduação, sem essa ajuda, não seria possível a realização deste trabalho.
- À minha namorada, Letícia Purcino, pelo grande apoio que me deu durante os últimos meses deste trabalho e pela compreensão nos momentos mais difíceis.
- Aos meus amigos de república, minha família em Campinas, Maikol, Sandro, Arthur e Schubert, que fizeram deste tempo longe de casa uma acolhedora e divertida convivência. Lembro-me bem de todos os momentos que passamos juntos, dos estudos e da ralação, que com certeza me trarão saudade.
- Aos meus colegas de mestrado, professores e funcionários pela convivência, apoio e amizade, enfim, a toda a comunidade do Instituto de Computação/Unicamp, pelo ambiente agradável, organizado e propício à pesquisa científica.
- Ao CNPq e Compaq Brasil pelo apoio financeiro ao meu trabalho, porque sem ele, não haveria condições de realizá-lo.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A *WWW* (*World Wide Web*), popularmente conhecida como *Web*, é atualmente considerada a plataforma tecnológica ideal para o desenvolvimento de aplicações na Internet e intranets corporativas. Tais aplicações, em muitos casos, são aplicações que acessam SGBDs (Sistemas de Gerenciamento de Bancos de Dados) relacionais. Contudo, a natureza do acesso a bancos de dados através da *Web* é completamente diferente das aplicações para bancos de dados tradicionais. A diferença básica está no caráter sem estado (*stateless*) do protocolo HTTP. O conceito de sessão de usuário, encontrado em aplicações de bancos de dados tradicionais, não é aplicado na *Web*. Em vez disso, cada interação com o servidor de informação é tratada como uma interação totalmente independente da interação anterior.

Aplicações de bancos de dados tradicionais são construídas baseadas em técnicas de bancos de dados centralizados ou distribuídos. A maior parte dos bancos de dados distribuídos são heterogêneos e interligados por uma LAN (*Local Area Network*) e/ou por uma WAN (*Wide Area Network*). Embora as técnicas desenvolvidas na integração de bancos de dados heterogêneos permitam pessoas acessar sistemas de bancos de dados heterogêneos que podem estar geograficamente distribuídos, o desenvolvimento de tais aplicações ainda é complexo e custoso.

Com a Internet globalmente acessível e *browsers* disponíveis em várias plataformas, torna-se possível construir aplicações de bancos de dados baseadas na *Web*. Uma vez que o banco de dados está conectado à *Web*, ele é acessível de qualquer computador interligado à Internet no mundo. Além disso, devido aos *browsers* estarem disponíveis para a maioria das plataformas, eles eliminam a necessidade de se projetar diferentes interfaces através de diferentes plataformas. Na Internet é freqüente o caso em que um *site* atrai um grande número de usuários diariamente, o que normalmente não acontece em uma aplicação de bancos de dados tradicional. Além disso, um servidor de bancos de dados normalmente tem uma limitação no número de conexões concorrentes, devido aos recursos de *hardware*

ou de restrições de licença do *software.*

A maioria dos SGBDs não foi projetada para o ambiente da Internet. Embora esses sistemas possuam mecanismos de autorização para controlar o acesso aos dados armazenados no banco de dados, eles normalmente não fornecem um mecanismo de comunicação de rede seguro para a transmissão de dados. A maior parte desses sistemas usa *sockets* padrão ou RPC (*Remote Procedure Call*). Quando os sistemas de bancos de dados são interligados com a Internet, o protocolo de comunicação nativo do SGBD se torna vulnerável a *hackers* e curiosos que podem interceptar e até mesmo alterar os dados transmitidos. Esse problema não será tratado nesta dissertação.

1.1 Motivações e Objetivos da Dissertação

Um programa CGI, também conhecido como *gateway*, é um processo que é executado no servidor *Web* toda vez que ocorre uma requisição proveniente de um cliente (*browser*). Sempre que o *gateway* for realizar uma atividade no banco de dados ele abre uma conexão com a base de dados, retorna os dados recuperados e termina, não guardando nenhuma informação de estado da operação executada. Mais tarde se o mesmo cliente desejar realizar uma outra operação na mesma base de dados, o *gateway* deverá abrir uma outra conexão envolvendo uma nova autenticação do usuário e para isso consumirá novamente mais recursos do sistema, provocando assim um decréscimo no desempenho do mesmo.

Verificamos que o projeto da aplicação e as técnicas de bancos de dados tradicionais precisam ser reavaliadas no que diz respeito a alcançar uma alta escalabilidade e controle sobre o acesso de usuários aos bancos de dados através da *Web* [LF98, XLF99, LF99]. Esta dissertação apresenta uma arquitetura para o desenvolvimento de aplicações de bancos de dados com estado (*stateful*) no contexto da *Web*. Nessa arquitetura, informações de estado são mantidas pelo cliente *Web* (*browser*) e por um agente de acesso a bancos de dados especializado denominado Gerenciador de Aplicações de Bancos de Dados na *Web* (GABD*Web*), tornando a aplicação ciente de uma sessão de usuário. Com o uso do GABD*Web*, uma aplicação poderá melhorar o seu desempenho e escalabilidade reutilizando conexões entre várias requisições de um mesmo usuário. Pode-se também, restringir o conjunto de usuários que tem permissão de acesso ao sistema.

A Figura 1.1 mostra o agente GABD*Web* juntamente com a arquitetura de *gateways* tradicionais. O GABD*Web* tem a capacidade de preservar informações de estado entre consecutivos acessos a um mesmo banco de dados pelo mesmo cliente, eliminando assim algumas operações desnecessárias, tais como a re-autenticação de um mesmo usuário.

O tema da dissertação é direcionado a bancos de dados heterogêneos porque muitas empresas de grande porte, por razões históricas, possuem SGBDs de fornecedores distintos, cada um com uma API (*Application Programming Interface*) distinta do outro,

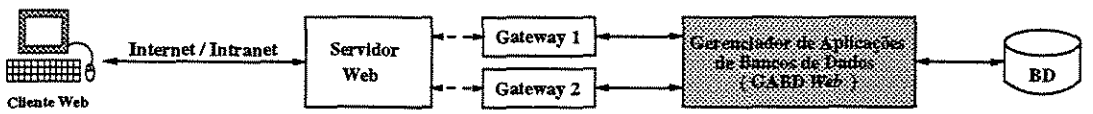


Figura 1.1: Arquitetura Tradicional com o Agente GABD *Web*.

dificultando o desenvolvimento de aplicações corporativas que necessitem de informações contidas nos diversos SGBDs. Uma API genérica e independente do SGBD vem de encontro a essas necessidades.

Os objetivos a serem alcançados nesta dissertação são:

- Estudo comparativo de diversas propostas abertas para o desenvolvimento de aplicações de bancos de dados baseadas na *Web*;
- Desenvolvimento de um mecanismo de acesso genérico e persistente a bancos de dados (GABD *Web*), que proverá as funções básicas para acesso persistente a bancos de dados heterogêneos (isto é, de fornecedores distintos), utilizando APIs de código aberto tais como JDBC, DBI e Python Database API;
- Análise e comparação da arquitetura GABD *Web* com *gateways* tradicionais de acesso a SGBDs relacionais via *Web*.

1.1.1 Sistemas Alvo

Os sistemas alvo da arquitetura apresentada são aplicações possivelmente complexas de bancos de dados, em três níveis (*three-tier*), tendo como *host* um único servidor de aplicação, podendo acessar simultaneamente diversos bancos de dados em servidores distintos.

A Figura 1.2 mostra um cenário possível para os sistemas alvo considerados. Existe um servidor de aplicação, um ou mais servidores de bancos de dados e vários clientes. Normalmente, os clientes estão executando em alguma versão do Microsoft Windows (como por exemplo, NT Workstation), mas outros tipos de sistemas operacionais como Unix e MacOS também são comuns. Os clientes interagem com o sistema via algum *browser*, sendo o Netscape Navigator e o Microsoft Internet Explorer os mais utilizados. Normalmente, existe no lado dos servidores um sistema operacional tal como Unix ou Windows NT. Os servidores de bancos de dados proprietários mais comuns são: Oracle, Microsoft SQL Server, Informix, Sybase, DB2, dentre outros.

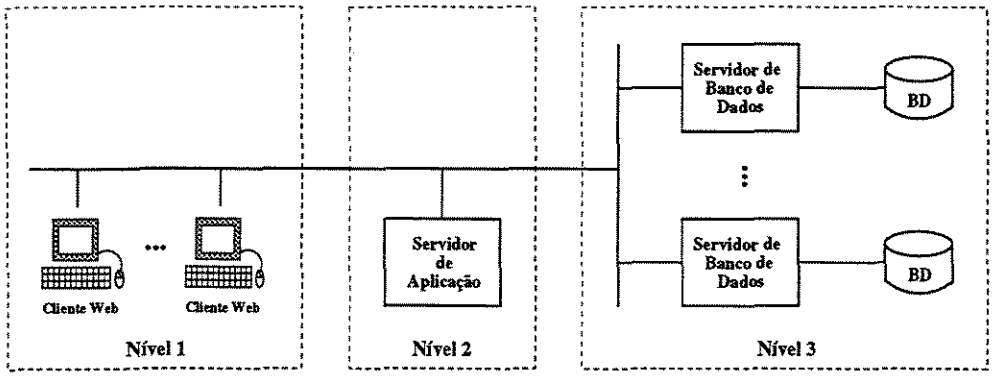


Figura 1.2: Cenário Possível dos Sistemas Alvo.

1.2 Organização da Dissertação

Esta dissertação está organizada da seguinte forma:

- No Capítulo 2 são apresentados os fundamentos necessários à compreensão do resto da dissertação, principalmente algumas formas de integração de bancos de dados e tecnologias *Web*;
- No Capítulo 3 são detalhados trabalhos relacionados com o assunto da dissertação (desenvolvimento de aplicações de bancos de dados baseadas na *Web*);
- No Capítulo 4 são apresentadas algumas ferramentas de código aberto que podem ser usadas no desenvolvimento de aplicações de bancos de dados heterogêneos para a *Web*;
- No Capítulo 5 é apresentada a arquitetura GABD *Web* construída a partir das ferramentas de código aberto apresentadas no Capítulo 4; são apresentados os detalhes de implementação da arquitetura para o ambiente Java, Perl e Python;
- No Capítulo 6 são apresentados alguns experimentos para comparar o desempenho da arquitetura GABD *Web* com as arquiteturas de *gateways* tradicionais;
- No Capítulo 7 é apresentada uma conclusão das avaliações das arquiteturas de acesso a bancos de dados no contexto da *Web*, um resumo das contribuições e possíveis direções futuras;
- No Apêndice A é feita a validação das técnicas apresentadas na dissertação através da implementação de uma aplicação real, denominada agência de viagens *Travel*.

Capítulo 2

Fundamentos

Este capítulo apresenta conceitos e terminologias necessários à compreensão da dissertação. O capítulo está organizado da seguinte forma: a seção 2.1 apresenta o conceito de sistemas de código aberto, a seção 2.2 apresenta uma descrição de bancos de dados heterogêneos e a necessidade de integração desses sistemas, a seção 2.3 apresenta os conceitos de gerenciamento de conexão e sessão. Por último, a seção 2.4 apresenta as principais abordagens utilizadas na integração de bancos de dados e tecnologias *Web*.

2.1 Sistemas de Código Aberto

A idéia básica atrás do *software* de código aberto (*open source software* [Ray]) é simples: quando um programador pode ler, redistribuir e modificar parte do código fonte de um *software*, esse se desenvolverá mais rapidamente. Grupos de usuários ajudam no desenvolvimento do *software*, detectando e consertando problemas. Uma vez, que o código fonte está exposto aos usuários, os problemas são detectados mais facilmente e resolvidos mais rapidamente, aumentando assim a segurança do *software*.

A filosofia ou modelo de código aberto é considerada por muitos ser mais eficaz do que o modelo tradicional fechado (*software* proprietário), onde poucos programadores têm acesso ao código fonte do *software*. Uma prova de que essa filosofia funciona é a grande aceitação do sistema operacional Linux, das linguagens de programação Java, Perl e Python e do servidor *Web* Apache.

O conceito de código aberto não significa somente acessar o código fonte de um *software*. Os termos de distribuição devem obedecer aos seguintes critérios [Ray]:

- Livre distribuição;
- Deve incluir o código fonte;

- Deve permitir alterações no código fonte;
- Não deve discriminar pessoas ou grupos;
- Não deve discriminar áreas de possíveis utilização do *software*.

Do ponto de vista do usuário é excelente trabalhar com *softwares* de código aberto. Além da segurança e qualidade que são adicionados ao *software*, o modelo de código aberto tem outras vantagens, o cliente não fica na dependência do fornecedor do *software* e se o suporte técnico do fornecedor se tornar oneroso, pode-se contratar outra empresa.

2.2 Bancos de Dados Heterogêneos

Organizações privadas, governamentais e acadêmicas movimentam um grande volume de informações na execução de suas atividades diárias. A autonomia das atividades organizacionais e as necessidades particulares de cada uma levaram à coexistência de diversos SGBDs em uma mesma organização.

Esses múltiplos SGBDs podem ser baseados em diferentes modelos (como relacional, hierárquico ou orientado a objetos) com linguagens e representações diversas, muitas vezes para informações similares. Assim a coexistência de diferentes sistemas de bancos de dados apresenta-se como natural e inevitável no contexto das corporações.

2.3 Manutenção de Contexto na *Web*

Como definido em [Mal98], um *gateway* de um SGBD acessível pela *Web* possui duas interfaces, uma com o cliente *Web* através do servidor *Web* e a outra com o sistema de bancos de dados. A manutenção do contexto em relação ao sistema de bancos de dados é chamado gerenciamento de conexão, e a manutenção do contexto em relação ao cliente é chamado gerenciamento de sessão:

- **Gerenciamento de Conexão**: sistemas de bancos de dados supõem que requisições consecutivas de um único *thread* ou processo, com uma conexão de banco de dados e com o mesmo identificador de usuário, são emitidas pelo mesmo usuário e portanto não requerem re-autenticação do usuário. *Gateways* podem tirar vantagem desse aspecto pela criação de um *pool* de conexões. Essa técnica reduz a taxa de abertura de novas conexões com o banco de dados;
- **Gerenciamento de Sessão**: usuários esperam o suporte de sessão e não querem digitar a mesma informação várias vezes. Para suportar sessões, o *gateway* tem que

ser capaz de identificar quando uma sessão começa e quando termina, ser capaz de transmitir um identificador de sessão entre um cliente e o servidor *Web*, decidir onde armazenar o estado da sessão (cliente ou servidor), e saber como manipular erros de sessão não esperados.

2.3.1 Gerenciamento de Conexão

Nesta subseção, apresentaremos a manutenção de contexto do *gateway* em relação ao SGBD e um outro tipo de manutenção de contexto entre o cliente *Web* e o servidor *Web*.

Gerenciamento de Conexão - *Gateway*/SGBD

Com o objetivo de aumentar a escalabilidade de bancos de dados abertos a usuários na Internet, é descrita uma estratégia em que as conexões com os bancos de dados são compartilhadas entre os usuários. Os clientes *Web* podem manter suas conexões lógicas com um banco de dados através de um *pool* de conexões físicas compartilhadas. Com isso, não apenas um grande número de usuários podem ser beneficiados por uma quantidade de recursos físicos limitados, mas também o custo de comunicação de cada cliente será reduzido. O processo de conexão a um banco de dados é uma operação cara. Além disso, o número de conexões físicas a um banco de dados é normalmente limitado, por razões de *hardware* ou limite no número de licenças do SGBD. Em aplicações de bancos de dados na Internet, o número de usuários concorrentes normalmente é muito maior que o número de conexões físicas disponíveis. Para atender a um grande número de requisições com um tempo de resposta razoável, a técnica de *pool* de conexões compartilhadas é fortemente indicada para essas situações.

Quando um cliente *Web* requisita uma conexão, não será criada uma nova conexão física com o banco de dados; em vez disso, será retornada uma conexão já aberta do *pool* de conexões compartilhadas. Isto é, quando o cliente submeter uma requisição, uma conexão ociosa será recuperada do *pool* de conexões e atribuída a esse cliente. Depois que a requisição completar sua execução, a conexão alocada retornará ao *pool* de conexões. Desta forma, o cliente pode logicamente manter a conexão aberta para as próximas requisições. Se não existir conexões disponíveis quando uma requisição do cliente chegar, e não puder ser aberta uma nova conexão devido ao número máximo de conexões abertas já ter sido alcançado, a requisição será adicionada dentro de uma lista de espera. Quando uma conexão for liberada, ela será alocada para a requisição com a maior prioridade na lista de espera.

Gerenciamento de Conexão - Cliente *Web*/Servidor *Web*

Os usuários se preocupam com o tempo de latência da *Web*. A latência percebida por um usuário pode vir de várias fontes. Um servidor *Web* pode estar demorando um longo tempo para processar uma requisição. Isso ocorre se o servidor estiver sobrecarregado ou se possuir um acesso a disco lento. Clientes *Web* também podem adicionar um retardo se eles não puderem verificar rapidamente no documento HTML recuperado que outros objetos também deverão ser recuperados e mostrados ao usuário. O tempo de recuperação de documentos *Web* também depende do tempo de latência da rede. O protocolo HTTP 1.0 [BLFF96], atualmente usado na *Web*, é simples, mas está longe de ser considerado ótimo em relação à latência fornecida.

A Figura 2.1 mostra como ocorre a troca de pacotes para a recuperação de um documento HTML com pelo menos uma imagem. A troca de informações entre o cliente (*browser*) e o servidor *Web* acontece da seguinte forma:

1. O cliente abre uma conexão TCP, resultando em uma troca de pacotes *SYN*;
2. O cliente transmite uma requisição ao servidor, o servidor pode ter que ler dados do disco para responder à requisição do cliente, transmite a resposta para o cliente e fecha a conexão TCP [Pos81];
3. Depois de verificar no documento recebido se há algum objeto a ser recuperado, o cliente abre uma nova conexão para o servidor, resultando em mais trocas de pacotes *SYN*;
4. O cliente novamente faz uma requisição HTTP, desta vez para o primeiro objeto a ser recuperado. Uma nova requisição deve ser gerada pelo cliente para cada objeto a ser recuperado.

Devido ao fato do cliente abrir uma nova conexão para cada requisição HTTP, são inseridos custos à latência da rede:

1. O estabelecimento de uma nova conexão requer alocação de novos números de portas, alocação de recursos e criação de estruturas de dados apropriadas;
2. A *Web* claramente necessita de alguma forma de autenticação, possivelmente criptografia, para garantir a privacidade e integridade dos dados. Assim, fica extremamente caro realizar uma nova autenticação para cada requisição HTTP de um mesmo cliente.

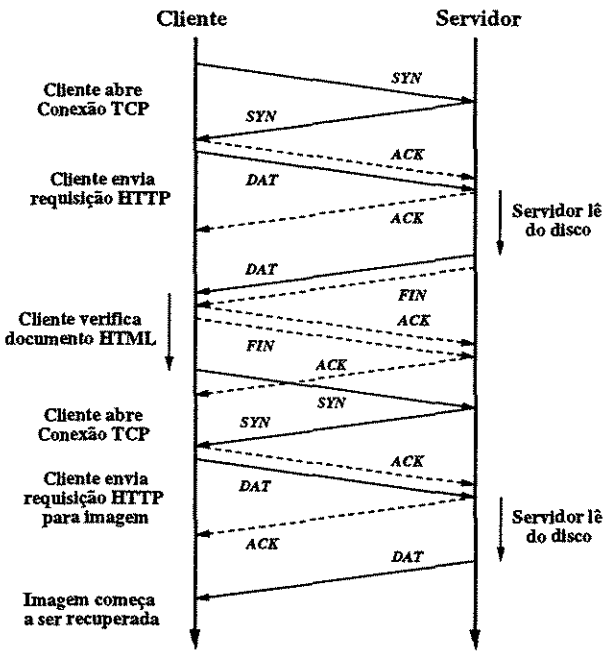


Figura 2.1: Troca de Pacotes no Protocolo HTTP 1.0.

Vários pesquisadores [PM94, Hei97, PM96, NGBS+97] têm analisado as ineficiências no uso da rede pelo protocolo HTTP 1.0 e têm proposto várias modificações para reduzir significativamente o tempo de latência.

Como o pequeno tempo de vida de uma conexão TCP causa um problema de desempenho, foi proposta em [Mog95, Pad95] uma mudança simples no protocolo HTTP 1.0: usar uma única conexão TCP com um tempo de vida maior durante várias requisições HTTP. A conexão permanece aberta para todos os objetos e documentos HTML que o cliente deseja recuperar do mesmo servidor. O protocolo HTTP 1.1 [FGM+97] utiliza essas idéias para definir um mecanismo de conexão persistente que resolve o mesmo problema. O mecanismo de conexão persistente (P-HTTP), usado no protocolo HTTP 1.1, evita a abertura de novas conexões TCP, e com isso, evita a troca desnecessária de pacotes entre o servidor e o cliente.

A Figura 2.2 mostra a recuperação de documentos da mesma forma que na Figura 2.1, exceto que o cliente já possui uma conexão TCP aberta para o servidor e a conexão não é fechada no fim da requisição HTTP. Para a conexão ter um longo tempo de vida, devem ser feitas algumas mudanças no comportamento do cliente e do servidor.

O cliente poderá manter um conjunto de conexões TCP abertas, uma para cada servidor com que tenha se comunicado recentemente. O cliente quando julgar necessário

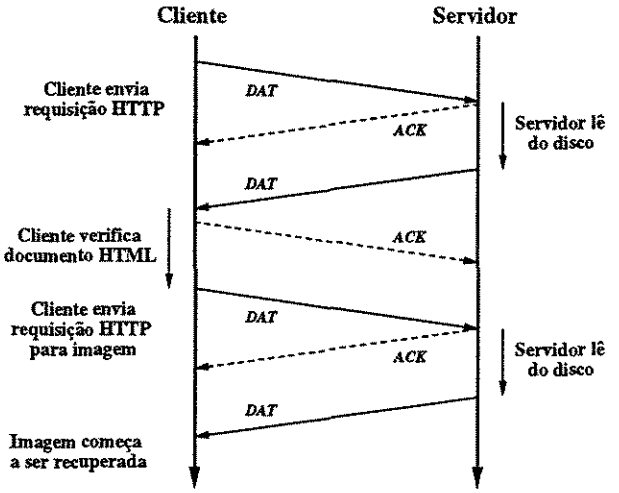


Figura 2.2: Troca de Pacotes no Protocolo HTTP 1.1.

poderá fechar conexões ociosas para limitar os recursos consumidos.

O servidor também poderá manter um conjunto de conexões TCP abertas. O servidor deverá ser capaz de limitar o número de conexões abertas para manter recursos suficientes para o processamento de novas requisições. O servidor poderá fechar uma conexão quando ocorrer um dos seguintes casos:

1. O número de conexões abertas exceder um limite máximo estabelecido;
2. Uma conexão ficar ociosa por mais tempo que um dado *timeout.*

No caso de uma requisição ter sido transmitida por um cliente mas ainda não ter sido recebida pelo servidor, o servidor poderá decidir fechar a conexão. Por isso, o cliente deverá ser capaz de restabelecer a conexão e refazer a requisição HTTP.

O método de conexão persistente (HTTP 1.1) tem inúmeras vantagens sobre o protocolo HTTP 1.0, tais como:

1. Devido ao fato de serem abertas e fechadas um número menor de conexões TCP, é utilizado menos tempo de CPU e menos memória;
2. Requisições e respostas HTTP podem ser "*pipelined*" em uma única conexão, isto é, um cliente pode fazer múltiplas requisições sem esperar por cada resposta, permitindo uma única conexão TCP ser usada mais eficientemente;
3. O congestionamento da rede é reduzido por diminuir o número de pacotes gerados pela abertura de uma nova conexão TCP.

2.3.2 Gerenciamento de Sessão

A natureza sem estado (*stateless*) do protocolo HTTP [BLFF96] continua a ser um problema no desenvolvimento de aplicações na *Web*. A comunicação entre um cliente *Web* e um servidor, através do protocolo HTTP, não mantém informações de estado. Isto significa que toda requisição de um cliente a um servidor é tratada independentemente das anteriores. Informações de conexões anteriores não são mantidas para uso em conexões futuras. Muitas vezes, é essencial manter informações de estado entre consecutivas interações cliente/servidor, como por exemplo, o conteúdo de um carrinho de compras enquanto o cliente está passeando por um supermercado virtual para fazer compras.

Mecanismos de Gerenciamento de Estado

Os métodos que veremos nesta subseção representam as informações de estado atribuindo valores a variáveis conhecidas como variáveis de estado. A preservação de estado na *Web* pode ser feita de várias formas e baseada em vários componentes de *software*. Algumas abordagens sugerem que somente o servidor mantenha traços de ações do usuário enquanto outras requerem o envolvimento ativo do *browser*.

Alguns dos métodos de preservação de estado propostos na literatura são:

- *Cookies*, uma extensão do protocolo HTTP proposto pela Netscape e padronizado no RFC 2109 [KM97];
- Variáveis escondidas em formulários HTML;
- Argumentos embutidos dinamicamente na URL [Iye97a].

Cookies

O RFC 2109 [KM97] define um mecanismo de gerenciamento de estado que é aplicado entre clientes (*browsers*) e servidor *Web*. O RFC 2109 resultou de propostas em [Kri98, Moo99, KM98] que especificam uma maneira de criar sessões de estado com requisições e respostas HTTP. São especificados dois novos cabeçalhos HTTP (*headers*), *Cookie* e *Set-Cookie*, que carregam informações de estado entre o servidor *Web* e o cliente.

Eles têm definições sintáticas comuns envolvendo o par atributo-valor. O servidor *Web* inicializa uma sessão se desejar manter o estado, retornando um cabeçalho de resposta *Set-Cookie*. O cliente se desejar continuar a sessão deverá retornar para o servidor um cabeçalho de resposta *Cookie*. O servidor pode ignorar esse cabeçalho ou usá-lo para determinar o estado corrente da sessão. O servidor pode fechar uma sessão enviando para o cliente um cabeçalho *Set-Cookie* com o atributo $Max\text{-}Age = 0$. O valor desse atributo

é *delta*-segundos, o tempo de vida do *cookie* em segundos. Quando o tempo de vida do *cookie* atingir *delta*-segundos o cliente descartará o *cookie*. Um valor zero significa que o *cookie* será descartado imediatamente.

O cliente mantém informações de estado que chegam via cabeçalho de resposta *Set-Cookie* de cada servidor *Web* (distinguido pelo nome ou endereço IP e número da porta). Para prevenir possíveis violações de segurança ou privacidade, o cliente pode rejeitar um *cookie*.

Na prática os clientes têm um limite para o número e tamanho dos *cookies* que eles podem armazenar. O cliente tentará armazenar todos os *cookies* que forem freqüentemente usados.

O cliente deverá satisfazer as seguintes condições:

1. Armazenar pelo menos 300 *cookies*;
2. Armazenar pelo menos 4096 *bytes* por *cookie*;
3. Armazenar pelo menos 20 *cookies* para um único *host* ou domínio.

O servidor não espera que o cliente seja capaz de exceder esse limite. Quando o limite dos 300 *cookies* ou os 20 *cookies* por servidor for excedido, o cliente removerá o *cookie* menos recentemente usado. Um exemplo da utilização de *cookies* [Iye97a] onde um *browser* recebe o seguinte *cookie* do servidor:

Set-Cookie: USERID=EDMAR; path=/dir1; expires=Wednesday, 28-Feb-2001 23:59:59 GMT

Esse *cookie* contém o valor para a variável *USERID*. Quando o cliente requisitar uma URL do servidor sob o *path* "*/dir*1", o seguinte *cookie* será enviado ao servidor:

Cookie: USERID=EDMAR

Nesse exemplo, o *cookie* expira às 23:59:59 horas do dia 28 de fevereiro de 2001. Se a data e hora de expiração não forem fornecidos, um *cookie* expira quando a sessão do *browser* termina.

Formulários HTML

Um método comum para manipular estado na *Web* envolve o uso de formulários HTML (*HyperText Markup Language*). Os servidores respondem a requisições de clientes com formulários HTML gerados dinamicamente contendo informações de estado embutidas em variáveis escondidas. Esses formulários HTML são tipicamente criados por programas

CGI. As variáveis escondidas são passadas de volta para o servidor quando o cliente submete o formulário.

Para ilustrar, suponha que um servidor está se comunicando com um cliente. O cliente se identifica junto ao servidor para que o acesso a sua conta seja permitido. Após essa identificação inicial, o servidor manterá alguma informação de estado para que o cliente não tenha que se identificar em todas as transações. Com formulários HTML, o servidor responde a toda requisição de cliente com um formulário gerado dinamicamente contendo o identificador do usuário como um argumento escondido. Quando o cliente submeter o formulário, o servidor identificará o cliente através do identificador de usuário do argumento escondido. A informação de estado é portanto passada entre o servidor e o cliente, enquanto a sessão de usuário for válida.

Essa abordagem limita os tipos de interações que são possíveis entre um cliente e um servidor enquanto o estado é preservado. O servidor deve sempre responder ao cliente com um formulário HTML gerado dinamicamente contendo variáveis escondidas, e não existe forma de preservar estado, por exemplo, quando o cliente está navegando entre documentos HTML estáticos.

Argumentos Embutidos Dinamicamente

Argumentos embutidos dinamicamente modificam *links* hipertexto para adicionar informações de estado [Iye97b, ID98]. *Links* hipertexto são alterados para invocar um programa especial conhecido como *argument embedder*. Esse programa passa as variáveis de estado para todos os *gateways* invocados pela aplicação.

Por exemplo, suponha que *embed* é o nome do *script* no qual é implementado o *argument embedder*. A aplicação deseja preservar as variáveis de estado x=*32* e y=*77*. Portanto, um *link* para um arquivo HTML:

<a href="http://davinci.unicamp.br/main.html">

será modificado para:

<a href="http://davinci.unicamp.br/cgi-bin/embed?url=http://davinci.unicamp.br /main.html&x=32&y=77">

Quando este *link* é selecionado, *embed* modificará os *links* no arquivo *main.html* para invocar o *script embed* com a URL original e as variáveis de estado como argumentos. Um *link* para um programa CGI:

<a href="http://davinci.unicamp.br/cgi-bin/prog?arg1=55">

será modificado para:

<a href="http://davinci.unicamp.br/cgi-bin/embed?url=http://davinci.unicamp.br /cgi-bin/prog&arg1=55&comma=1&x=32&y=77">

A *string comma=1* permite *embed* distinguir o argumento original *arg1* das variáveis de estado x e y. Se existe o perigo de *comma* entrar em conflito com uma outra variável de mesmo nome, um método mais sofisticado deverá ser usado para escolher um nome de variável único em vez de *comma*. Quando esse *link* é selecionado, *embed* invocará *prog* com o argumento original *arg1* juntamente com as variáveis de estado x e y. *Embed* então modificará os *links* na saída de *prog* para recursivamente invocar *embed* com a URL original e as variáveis de estado como argumentos.

Uma aplicação *Web* pode impedir que variáveis de estado sejam propagadas para um *link* colocando o comentário *<!-NO_STATE->* antes do *link* hipertexto.

Comparação dos Mecanismos de Gerenciamento de Estado

A seguir é realizada uma análise dos mecanismos de gerenciamento de estado estudados nesta seção; são apresentadas as vantagens e desvantagens de cada mecanismo com relação a: compatibilidade, tempo de vida das variáveis de estado, desempenho e segurança.

Compatibilidade

Primeiro, com relação à compatibilidade com o protocolo HTTP, formulários HTML e argumentos embutidos dinamicamente trabalham com qualquer *browser* e servidor *Web* que suportam HTTP. No entanto, *cookies* não são padronizados e requerem suporte especial e autorização por parte dos *browsers* e servidores *Web*. Segundo, formulários HTML podem manter estado somente quando todas as respostas do servidor são formulários gerados dinamicamente. Essa limitação compromete a utilidade desse método para uma grande classe de aplicações. Em contraste, *cookies* e argumentos embutidos dinamicamente podem preservar estado em aplicações que retornam páginas HTML estáticas ou dinâmicas.

A maioria dos *browsers* que suportam *cookies* permite aos usuários a configuração de rejeição dos *cookies*. A aplicação que depende de *cookies* para a preservação de estado não trabalhará corretamente se o *browser* do usuário não estiver configurado para aceitar *cookies*. Em contraste, não existe registro de qualquer *browser* que possa desabilitar o uso de formulários HTML ou argumentos embutidos dinamicamente.

Tempo de Vida das Variáveis de Estado

A aplicação *Web* que cria um *cookie* deve especificar uma data e hora de expiração do *cookie*. Isso pode ser um problema, pois é impossível saber quanto tempo o usuário ficará visualizando as páginas HTML. Se a data de expiração for muito próxima, a informação de estado pode ser invalidada no meio de uma aplicação; se a data de expiração estiver muito longe, *cookies* com informação de estado persistirão e confundirão outras aplicações (ou a mesma aplicação se os *cookies* não forem invalidados). Como regra, informações de estado confidenciais não terão um tempo de vida maior do que o tempo de vida da aplicação. Longos tempos de vida para tais dados introduzem riscos de segurança.

Variáveis de estado mantidas por formulários HTML e argumentos embutidos dinamicamente são uma parte integral da aplicação *Web*, e portanto datas de expiração não têm que ser especificadas. As variáveis de estado não expirarão no meio de uma aplicação ou permanecerão depois que a aplicação tiver sido finalizada. Formulários HTML e argumentos embutidos dinamicamente não dependem da corretude do sistema de *clock* do servidor e da máquina cliente, já os *cookies* dependem fortemente. Se o *clock* do servidor ou da máquina cliente estiverem incorretos, *cookies* não trabalharão corretamente.

Por outro lado, existem situações onde é desejável ter variáveis de estado expirando enquanto um *browser* ainda está visualizando páginas HTML que foram retornadas pela aplicação. Enquanto *cookies* lidam com essa situação naturalmente, formulários HTML e argumentos embutidos dinamicamente podem ser usados somente se a expiração das variáveis de estado for manuseada no nível de aplicação. Existem também situações onde variáveis de estado serão preservadas depois de uma aplicação ter sido finalizada. Para essas situações, somente *cookies* podem ser usados.

Desempenho

Cookies exigem pouca sobrecarga no servidor e, por isso, possuem um melhor desempenho entre as técnicas mencionadas. A principal causa de sobrecarga em formulários HTML e argumentos embutidos dinamicamente resulta da invocação de programas CGI para criar cada página, desde que ambas técnicas requerem que todas as páginas sejam geradas dinamicamente pelo servidor. CGI é tipicamente implementado pela criação de um processo separado que termina depois que o programa CGI é finalizado; esse mecanismo exige uma alta sobrecarga no servidor *Web*.

Existe um grande número de técnicas promissoras, como as que veremos na subseção 2.4, que permitem clientes executar programas no servidor sem criar novos processos para cada requisição. Uma abordagem é executar programas no servidor como parte do próprio servidor *Web*; uma outra abordagem é criar um *pool* de processos ou *threads*. O NSAPI (*Netscape Server API*) e o ISAPI (*Internet Server API*) da Microsoft usam a primeira

abordagem, enquanto *FastCGI* da *Open Market* usa a segunda abordagem. Com essas abordagens, a sobrecarga na utilização de formulários HTML e argumentos embutidos dinamicamente na preservação de estado diminuirá consideravelmente.

Segurança

Clientes e servidores *Web* freqüentemente precisam se comunicar seguramente sobre a *WWW* para evitar que usuários maliciosos obtenham informações confidenciais. SSL (*Secure Sockets Layer*) [Corb] da Netscape é um sistema de criptografia comumente usado fornecendo comunicação segura sobre a Internet.

Cookies, formulários HTML e argumentos embutidos dinamicamente todos suportam SSL; assim, variáveis de estado podem ser criptografadas antes delas serem enviadas através da Internet. SSL previne outros usuários de obter o identificador de sessão e realizar transações não autorizadas. A vantagem de se usar identificador de sessão em vez da *password* do cliente como uma variável de estado para autenticação é que as *passwords* são transmitidas através da rede uma única vez durante a autenticação inicial (identificação junto à aplicação) do usuário, assim é dado uma proteção extra à aplicação. S-HTTP (*Secure HTTP*) [EIT], da *Enterprise Integration Technologies*, é um outro sistema de criptografia bem conhecido que não é usado tão freqüentemente quanto SSL. Formulários HTML e argumentos embutidos dinamicamente também suportam S-HTTP.

Com formulários HTML, cada formulário dinâmico pode passar variáveis de estado para apenas uma URL. Com *cookies* e argumentos embutidos dinamicamente, os usuários podem continuar uma sessão pela seleção de uma grande variedade de *links* hipertexto [Iye97b]. *Cookies* e argumentos embutidos dinamicamente têm a capacidade de evitar a transmissão não criptografada de variáveis de estado através de *links* considerados inseguros.

É possível que *cookies* sejam passados para uma URL de um usuário malicioso que não faz parte da aplicação. Formulários HTML e argumentos embutidos dinamicamente passam variáveis de estado somente para *sites* que fazem parte da aplicação. Uma aplicação também pode receber um *cookie* criado por uma aplicação não conhecida, se a aplicação está esperando um *cookie* com o mesmo nome, resultará em um procedimento incorreto.

Devido aos *cookies* serem armazenados em disco, eles também podem causar problemas se o disco da máquina cliente é acessível a pessoas maliciosas. Formulários HTML e argumentos embutidos dinamicamente podem ser confiáveis nessa situação somente se o mecanismo de *cache* em disco estiver desabilitado e o cliente não armazenar no disco páginas HTML com informações confidenciais [Iye97b].

Para reduzir o risco dos identificadores de sessões serem roubados, o sistema pode periodicamente gerar novos identificadores para um cliente fazendo várias transações durante uma única sessão. No modo mais seguro, um novo identificador de sessão é gerado

depois de cada transação. Uma característica que também pode ser utilizada é registrar o endereço IP do cliente quando ele se autenticar pela primeira vez. Desta forma, em requisições futuras o sistema verificará se o endereço IP do cliente é igual ao endereço registrado pelo sistema. Se o IP não conferir o cliente terá que se identificar novamente, aumentando dessa forma a segurança no acesso à aplicação.

2.4 Integração de Bancos de Dados e Tecnologias *Web*

A *Web* é largamente baseada na tecnologia de sistemas de arquivos, que funciona bem com documentos estáticos e de natureza de leitura apenas (*read-only*). Contudo, com o grande aumento de fontes de informação, saber da existência e localização de informações, assim como uma forma de recuperá-las, têm tornado difícil para o usuário gerenciar e acessar as informações na *Web*. Sob tal circunstância, a tecnologia de sistemas de arquivos convencional está longe de ser a forma adequada para organizar, armazenar e acessar uma grande quantidade de informações na *Web*. Portanto, muitos dos grandes *sites* (provedores de informações) estão usando a tecnologia de bancos de dados para gerenciar essa grande quantidade de dados.

A tecnologia de bancos de dados tem sido a mais usada no campo de gerenciamento de informações. Essa tecnologia é muito eficiente na organização, armazenamento e recuperação de dados. Em [Per95a, Kra97, Mal98] são fornecidas análises das principais abordagens para a integração de bancos de dados com as tecnologias *Web*. Basicamente, elas podem ser divididas em três categorias: abordagem *server-side* (lado do servidor), abordagem *client-side* (lado do cliente) e abordagem *middle-layer* (camada intermediária).

A Figura 2.3 mostra algumas das formas de disponibilizar informações através da *Web* usando as abordagens citadas anteriormente: páginas HTML estáticas (arcos 1-4), Java *applets* (arcos 5-8) e geração dinâmica de páginas HTML usando *scripts* CGI (arcos 9-13).

2.4.1 Abordagem *Server-Side*

A abordagem *server-side* é focalizada em estender as funcionalidades do servidor *Web*. Os métodos usados incluem o CGI (*Common Gateway Interface*), *Gateways* como Servidores *Stand Alone*, SSI (*Server Side Includes*) e SAPI (*Server Application Programming Interface*).

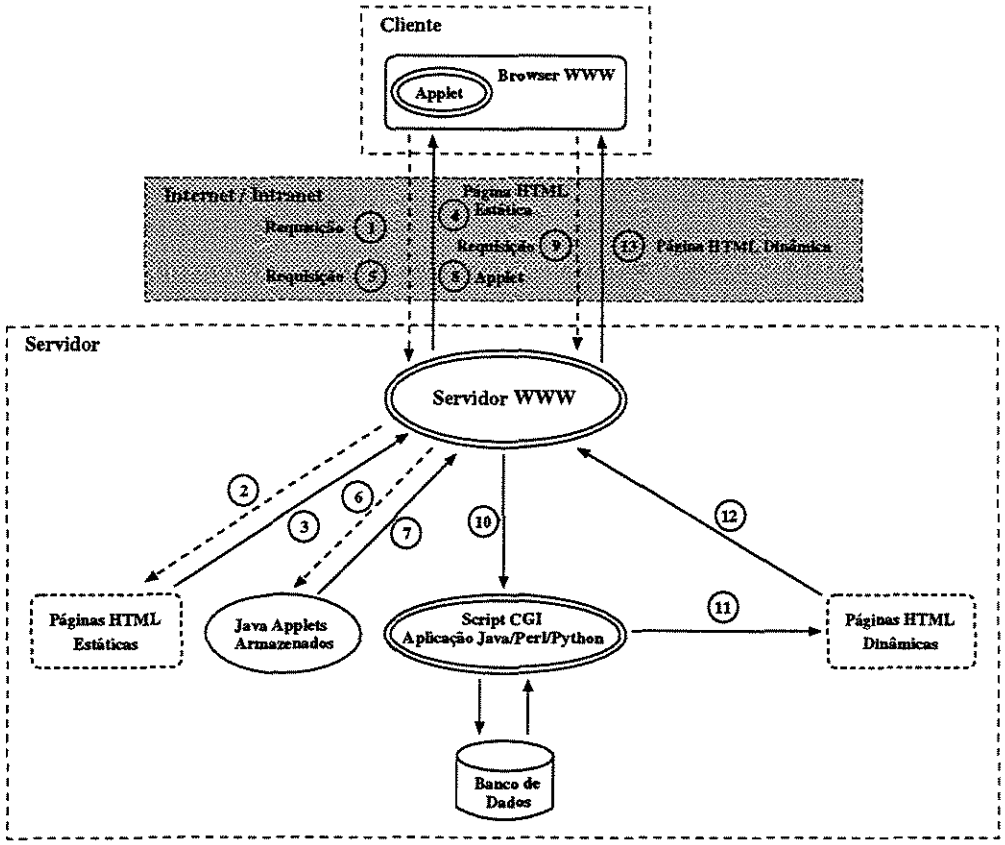


Figura 2.3: Usando Páginas HTML Estáticas, Java *Applets* e *Scripts* CGI.

Common Gateway Interface

O método mais antigo e comum é usar o protocolo CGI [NCS] e programas *gateways* externos. CGI é um padrão para a interface de programas *gateways* externos com servidores *Web*. Um programa *gateway*, também conhecido como *script* CGI, é executado no lado do servidor.

Cada vez que é feita uma requisição a um documento dinâmico, o servidor *Web* cria um novo processo e carrega o interpretador da linguagem do *script*. Esse carrega o *script* apropriado, o executa, retorna a informação requerida para o usuário e termina [LDWN96]. Esse ciclo é aceitável em algumas situações, mas existem muitas nas quais isso não pode ser aceito. Por exemplo, quando as requisições são freqüentes e o tempo de processamento útil (execução do *script*) for pequeno comparado com o tempo de carga do interpretador. A carga do interpretador da linguagem e a criação de um novo processo para cada requisição do *script* são muito caros, pois consomem muitos recursos do servidor, tais como

memória RAM e tempo de processamento. Essa característica torna inviável em muitos casos a utilização do mecanismo CGI.

CGI fornece um método simples e genérico de executar programas no lado do servidor *Web* [NS96]. *Scripts* CGI podem ser implementados em várias linguagens de programação. Para garantir portabilidade da aplicação no lado do servidor, o *script* CGI pode ser implementado em C, Java, Perl ou Python, por exemplo. A interface CGI tem muitas vantagens, incluindo portabilidade entre servidores *Web* e grande disponibilidade de programas *gateways* de código aberto e de ferramentas de desenvolvimento.

Gateways como Servidores *Stand Alone*

Para superar os problemas do CGI, outro método usado é estabelecer *gateways* como servidores que funcionam independentemente do servidor *Web*. A maioria dos vendedores de bancos de dados usa essa abordagem para desenvolver servidores de bancos de dados baseados na *Web* com um alto desempenho. Desde que os servidores *Web* podem ser otimizados para interagir diretamente com os SGBDs, uma grande melhora de desempenho pode ser alcançada. O problema é que esses servidores normalmente são dedicados a um SGBD particular. Alguns vendedores requerem até mesmo seus próprios *browsers* para atuar como clientes. Para uma organização com sistemas legados de vários vendedores, essa é uma abordagem muito cara, pois requer diferentes servidores e *browsers* para sistemas diferentes. Além disso, ocorrerão problemas de integração no acesso a vários bancos de dados heterogêneos [Per95a, Per95b].

Server Side Includes

Um SSI consiste de uma seqüência especial de caracteres (*tokens*) embutidos em um documento HTML. Antes do documento ser enviado do servidor *Web* para o cliente que fez a requisição, o documento HTML é primeiro examinado pelo servidor à procura desses *tokens*. Quando um *token* é encontrado, o servidor interpreta a instrução no *token* e executa uma ação de acordo com a instrução, que pode ser um acesso a banco de dados. Os dados resultantes recuperados do banco de dados são então incluídos dentro do documento HTML e enviados ao cliente. Um dos problemas dessa abordagem é que o resultado obtido é limitado a somente um único documento com todos os dados nele. Além disso, o servidor percorrer o documento HTML à procura de *tokens* pode ser uma operação muito cara.

Server Application Programming Interface

Este método acrescenta ao servidor *Web* uma interface conhecida como SAPI. Essa interface consiste de uma biblioteca (*Dynamic Linkable Library*) que contém rotinas executáveis necessárias para a interface com diferentes sistemas de informação legados. A função das rotinas de uma DLL é a mesma que a de um programa *gateway*. Diferente dos programas externos, essas rotinas são carregadas no mesmo espaço de endereçamento do servidor *Web*. Com isso, existe uma sobrecarga mínima associado à execução dessas rotinas, pois não existe o custo de criação de um novo processo para cada requisição. Além disso, somente as funções ativas permanecem na memória. Como resultado, o desempenho do servidor é melhorado significativamente. Filtros SAPI podem ser usados para aplicações tais como autenticação personalizada, acesso ou *logging*. NSAPI (*Netscape Application Programming Interface*) e *Microsoft* ISAPI (*Internet Information Server Application Programming Interface*) são dois exemplos desse método.

2.4.2 Abordagem *Client-Side*

A abordagem *server-side* limita a interação do usuário com a aplicação, pois todo o processamento ocorre no lado do servidor. Para suportar aplicações mais complexas (por exemplo, atualizações em várias tabelas de bancos de dados que são requisitadas em diferentes formulários) uma abordagem conhecida como *client-side* foi desenvolvida. Essa abordagem aumenta a interatividade entre os clientes e os SGBDs. A idéia é distribuir a aplicação e enviá-la para o cliente. O cliente então executa o código localmente. Uma forma simples é enviar um programa compilado para ser executado na máquina do cliente. Essa abordagem permite um melhor desempenho e uma alta escalabilidade, pois a carga do servidor *Web* pode ser reduzida. A execução local também pode ser útil em casos onde é necessário interagir com um dispositivo local para o qual não existe um protocolo de rede disponível. De qualquer forma, esse tipo de abordagem requer mecanismos de controle de acesso adicionais, que normalmente incluem procedimentos de autenticação, controle de segurança e manutenção de integridade no computador do cliente.

Uma recente tendência na programação para a Internet é o uso da linguagem Java. O código Java é robusto, fácil de usar, fácil de entender e pode ser carregado automaticamente sobre a rede. É uma excelente linguagem para o desenvolvimento de aplicações baseadas na *Web*. Um dos motivos da proliferação de programas Java escritos para a *Web* é que Java permite uma melhor interação com sistemas de bancos de dados. Java supera as limitações da característica sem estado do protocolo HTTP. Com um *browser Web* que possui uma MVJ (Máquina Virtual Java) embutida, o cliente pode abrir uma conexão diretamente com o servidor de bancos de dados [Dua96]. Outras linguagens *script*, como

JavaScript, também podem ser interpretadas e executadas pelo cliente (*browser*), permitindo maior interatividade e ao mesmo tempo evitando comunicação através da Internet.

2.4.3 Abordagem *Middle-Layer*

A abordagem *middle-layer* permite a integração de fontes de dados heterogêneas e distribuídas sobre a *Web* [BBH+99]. Uma das técnicas de *middle-layer* é baseada em CORBA (*Common Object Request Broker Architecture*). Desenvolvido pelo OMG (*Object Management Group*), CORBA é um padrão para objetos distribuídos. CORBA fornece um mecanismo para objetos fazerem requisições e receberem respostas de forma transparente. CORBA ORB é um *framework* que suporta interoperabilidade entre objetos, construídos em diferentes linguagens de programação e executando sobre diferentes máquinas em ambientes distribuídos heterogêneos.

Dois métodos baseados em CORBA [Kra97] podem ser desenvolvidos para acessar bancos de dados via *Web*:

1. ***Scripts* CGI atuando como clientes CORBA**
   Esses clientes CORBA executando no servidor *Web* chamam um servidor CORBA que acessa os sistemas de bancos de dados;

2. **Clientes *Web* atuando como clientes CORBA**
   Uma comunicação direta entre um cliente *Web* e um servidor CORBA é estabelecida através do IIOP (*Internet Inter ORB Protocol*).

Dentre as abordagens para a integração de bancos de dados com as tecnologias *Web* apresentadas nesse capítulo, a mais utilizada em aplicações reais é a abordagem *server-side*. Por esse motivo, optou-se por se trabalhar com essa abordagem. A abordagem *server-side* apesar de limitar a interação do usuário com a aplicação, ela fornece mais segurança ao cliente do que a abordagem *client-side*. Isso ocorre, porque todo o processamento da aplicação é realizado no lado do servidor.

No desenvolvimento de aplicações comerciais, as três abordagens também podem ser combinadas para construir uma aplicação com as melhores características de cada uma das abordagens.

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta alguns trabalhos relacionados com o assunto da dissertação. Várias das técnicas apresentadas aqui são utilizadas no desenvolvimento da ferramenta proposta neste trabalho (GABD*Web*).

3.1 Gerenciador de Navegação

Características Abordadas

Controle de Diálogo

Em sistemas baseados na *Web*, para cada página acessada, existem dois tipos de eventos que podem ser disparados pelo usuário: eventos que levam a aplicação a outra página e eventos que têm aquela página como escopo [AD99a].

Não se deve impor nenhuma restrição aos eventos que não levem à transição de páginas, pois o *browser* os controla facilmente. Embora, às vezes, seja muito difícil implementar o diálogo desejado devido às limitações da linguagem HTML.

Contudo, eventos que causem transição de páginas devem ser controlados cuidadosamente por um mecanismo de controle de navegação. Infelizmente, não existe suporte para esse controle nem na linguagem HTML nem no protocolo HTTP. Por exemplo, não existe mecanismo para assegurar que um usuário chegue a uma página específica seguindo um caminho de navegação conhecido e permitido. Também, não existe mecanismo para evitar o *bookmarking* de páginas (acessar uma página diretamente através de uma URL armazenada anteriormente). Isso é um problema, pois o diálogo de algumas páginas requer ações passadas tomadas em outras páginas.

A falta de estado no protocolo HTTP tem, pelo menos, duas conseqüências diretas: dificulta o controle de acesso às informações e o controle do diálogo. A noção de estado

deve ser simulada pois os usuários estão acostumados a sistemas nos quais esta é uma característica intrínseca.

Perfis de Usuário

O controle de acesso às informações no protocolo HTTP foi desenhado para um protocolo sem estado e, como tal, protege apenas páginas individuais de acessos indevidos. Essa não é uma abordagem aceitável para sistemas interativos com muitos usuários, pois uma enorme quantidade de permissões devem ser atribuídas; isto porque para cada requisição de um usuário o sistema deverá validar o seu acesso às informações.

Sistemas reais têm o conceito de perfil de usuário. O conjunto de perfis de usuário em uma aplicação representa as categorias nas quais o usuário se enquadra como, por exemplo, gerentes, compradores, planejadores, etc. É possível que um usuário específico pertença a múltiplos perfis simultaneamente. Assim, a abordagem correta para o controle de acesso às informações é deixar o usuário herdar o acesso a uma dada página via seu conjunto de perfis [AD99a].

Idempotência do Protocolo HTTP

Como já foi dito, o protocolo HTTP foi inicialmente projetado para processamento de documentos estáticos. Nesse cenário, a idempotência realmente faz sentido; não existe risco se a mesma requisição for processada várias vezes, pois a cada vez, o resultado será o mesmo. Contudo, quando se está tratando com informação dinâmica não se pode assumir idempotência.

Os botões de *submit* e *back* do *browser* são as principais fontes de problemas de idempotência. Por exemplo, se o usuário pressionar várias vezes o botão *submit* em um formulário, a mesma ação será tomada a cada pressionamento, com efeitos indesejáveis.

O tratamento de idempotência, quando requerido, pode ser conseguido através do controle da navegação entre páginas. Por exemplo, pode-se proibir transições de uma página não idempotente para ela mesma [AD99b].

Arquitetura do Gerenciador de Navegação

Em [AD99a, AD99b] é proposta uma arquitetura em três níveis (*three-tier*) voltada para aplicações de bancos de dados tendo como *host* um único servidor de aplicação, podendo acessar simultaneamente diversos bancos de dados em servidores distintos e possivelmente heterogêneos. Essa arquitetura é fácil de implementar e tem um excelente ganho em desempenho, confiabilidade, segurança e escalabilidade.

Grafo de Navegação

Considerando as dificuldades citadas anteriormente, é proposto em [AD99a] um gerenciador de navegação baseado em um grafo conexo, orientado e colorido, no qual os nós representam páginas *Web*, as arestas representam transições entre páginas, e as cores representam os perfis dos usuários. É possível que um nó particular seja multi-colorido e que esse tipo de grafo seja bastante complexo. Tudo depende da aplicação a ser desenvolvida. A Figura 3.1 mostra um possível grafo de navegação para a arquitetura.

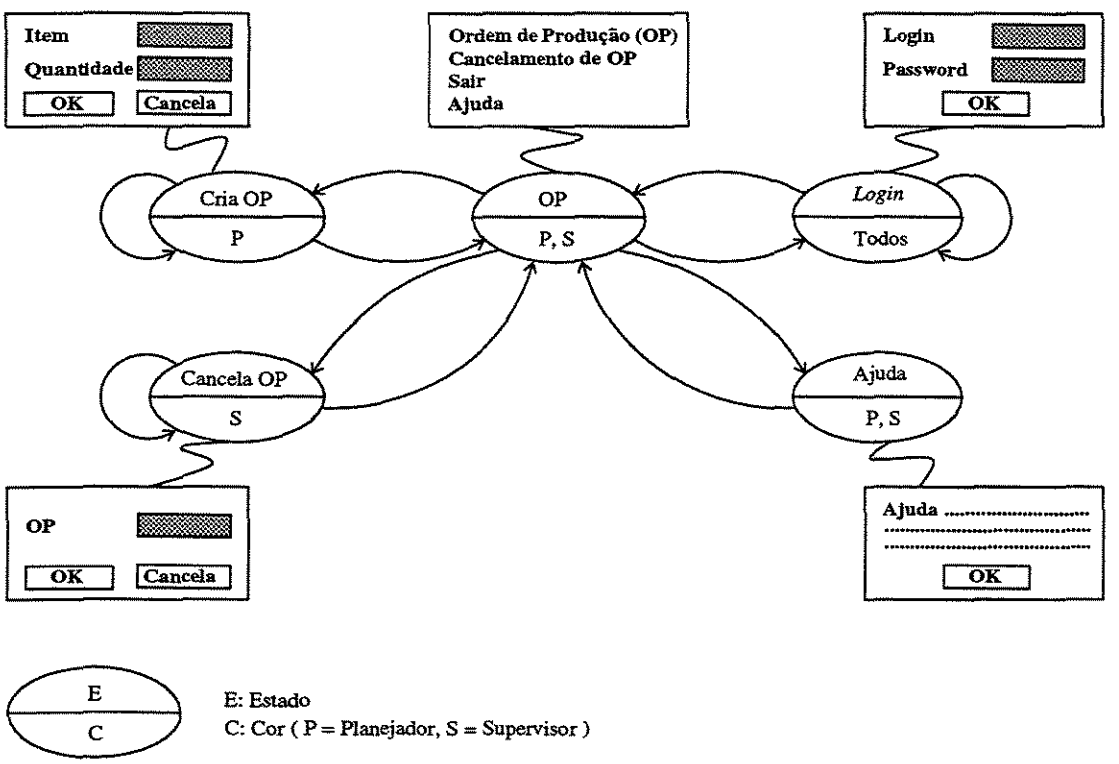


Figura 3.1: Exemplo de Grafo de Navegação.

Grafos orientados e conexos são ferramentas formais, compreensíveis e autocontidos para especificar e validar a navegação entre páginas e flexíveis o suficiente para serem modificados durante a etapa de projeto da aplicação. Além disso, tornam possíveis a simulação de estado e controle de segurança em tempo de execução. Deve ser notado que, embora o mecanismo de controle de navegação gerencie a transição entre páginas, ele não se importa com os eventos que causaram essas transições. Isso é tarefa da implementação do diálogo.

Cada sessão ativa de usuário possui um *token* (identificador de sessão) associado, que identifica as sessões para a aplicação. Como o protocolo HTTP não mantém estado, esse *token* deve ser sempre passado do *browser* para a aplicação e vice-versa. O mecanismo de controle de estado usado pela aplicação é o *cookie*. Esse *token* é gerado sempre que o usuário se identifica para a aplicação no nó inicial (tela de *login*). Todos os outros nós são descendentes do nó inicial, isto é, para cada nó existe um caminho conexo orientado ligando o nó inicial ao nó em questão.

Quando o usuário, em um nó particular, toma uma ação que o leva a um outro nó, entra em cena o mecanismo de validação da navegação. A transição será permitida se, e somente se, o nó destino for um descendente direto do nó origem e o conjunto de perfis (cores) do nó destino incluir pelo menos uma cor do conjunto de cores do usuário. A Figura 3.2 [AD99b] mostra a condição de validação de uma transição.

$$\exists\ EDGE(SOURCE, DESTINATION)\ \&\&$$
$$NODE\text{-}PROFILE\text{-}SET(DESTINATION) \cap$$
$$USER\text{-}PROFILE\text{-}SET(WEB\text{-}SESSION(SESSION\_ID).USER\_ID) \neq \emptyset$$
$$\Longrightarrow \text{Acesso Permitido}$$
$$\text{Ação: } WEB\text{-}SESSION(SESSION\_ID).NODE\text{-}ID = DESTINATION$$

Figura 3.2: Condição de Validação do Mecanismo de Navegação.

Assim, por indução, pode-se assegurar que este mecanismo efetivamente simula a conexão e leva em consideração questões de segurança via conceito de perfil de usuário.

Os usuários normalmente não se desconectam da aplicação *Web*. Eles fecham o *browser* ou vão para outra URL (*Uniform Resource Locator*). Por isso, a proposta em [AD99a] inclui um mecanismo de expiração de *tokens* a intervalos de tempo regulares.

O mecanismo de controle de navegação proposto resolve os problemas de idempotência e evita o *bookmarking*. Se o diálogo de uma página *Web* contiver alguma operação não idempotente, pode-se evitar re-execução não se criando uma aresta do nó que representa essa página para ele mesmo. Isso elimina alguns efeitos indesejáveis dos botões de *reload* e *submit*. *Bookmarking* é proibido naturalmente, pois não existe forma de se passar de um nó para outro se o nó alvo não for descendente direto do nó corrente. Além disso, o *token* da sessão já pode estar expirado. As transições disparadas pelo botão *back* do *browser* podem ser evitadas não se criando uma aresta conectando o nó corrente ao nó anterior.

3.2 Gerenciador de Sessão

O trabalho apresentado em [HPMP98, HMP98, HMP99] foi motivado principalmente pelo caráter sem estado (*stateless*) do protocolo HTTP. Nesse trabalho, é proposta uma arquitetura para o desenvolvimento de aplicações de banco de dados com estado para o ambiente *Web*. Para resolver esse problema foi introduzido um *software* especializado denominado gerenciador de sessão. O gerenciador de sessão mantém traços de estado de um cliente *WWW* e controla a sua interação com a aplicação de banco de dados.

Por razões de eficiência, a aplicação de banco de dados precisa ter uma estrutura multi-*threaded*; existe um mapeamento um para um entre os *threads* da aplicação e sessões de usuário. Para manter a aplicação de banco de dados coordenada com o *browser*, o gerenciador de sessão faz uso de *cookies*.

Arquitetura para Aplicações com Estado na *Web*

Como mostrado em [HM97], o re-estabelecimento de conexões com um sistema de banco de dados envolve um considerável retardo e *overhead* que degrada a velocidade que o serviço é oferecido à aplicação. Os *gateways* com um pequeno tempo de vida (tipicamente *scripts* CGI) são os responsáveis por esse problema. Processos com um pequeno tempo de vida precisam re-executar parte de um programa lógico para alcançar o ponto em que seu predecessor (isto é, o *script* que foi disparado pela requisição anterior do mesmo cliente) terminou a execução da requisição anterior.

Esse problema é resolvido com a introdução de um processo com um longo tempo de vida denominado gerenciador de sessão. A Figura 3.3 mostra a arquitetura para o desenvolvimento de aplicações de banco de dados com estado baseadas no gerenciador de sessão.

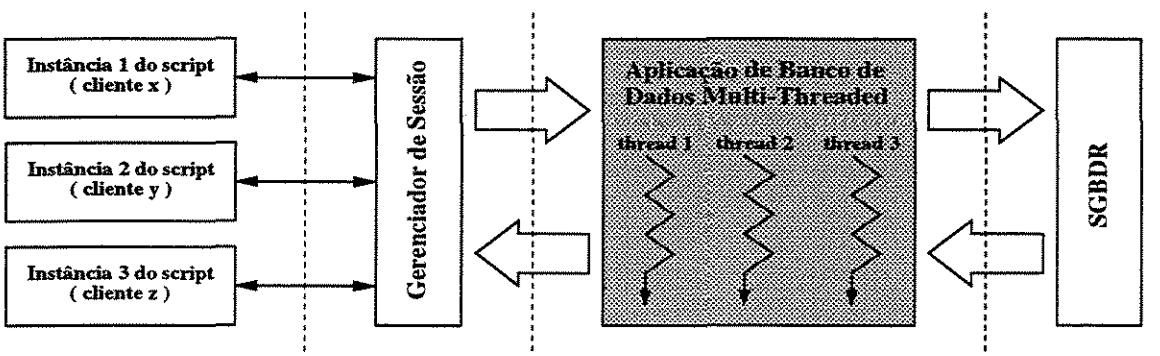


Figura 3.3: Arquitetura de Aplicações com Estado Baseadas no Gerenciador de Sessão.

As instâncias do *script* na Figura 3.3 são instâncias do mesmo *script* CGI para diferentes clientes. A comunicação de cada instância de um *script* com o *thread* de aplicação apropriado é garantida pelo gerenciador de sessão.

Arquitetura do Gerenciador de Sessão

O componente central desta arquitetura, o gerenciador de sessão, é um *software* multi-*threaded* que coordena a comunicação entre o *browser WWW* e a aplicação de banco de dados com estado. Como mostrado na Figura 3.4, o gerenciador de sessão tem duas interfaces; uma com a aplicação e outra com os *scripts* CGI.

Internamente o gerenciador de sessão tem uma estrutura com dois *threads*:

1. **I-thread**: é responsável por receber e processar as requisições que os *scripts* CGI submetem ao gerenciador de sessão;

2. **O-thread**: manuseia a saída produzida pela aplicação de banco de dados.

A estrutura com dois *threads* do gerenciador de sessão capacita-o a tratar requisições de entrada e saída concorrentemente. A estrutura interna do gerenciador de sessão é ilustrada na Figura 3.4.

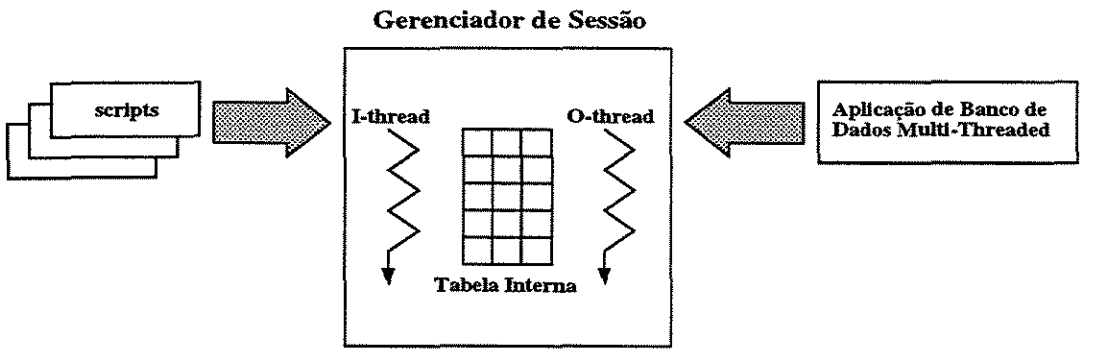


Figura 3.4: Arquitetura do Gerenciador de Sessão.

Os *scripts* CGI comunicam com o gerenciador de sessão através de conexões *socket*. Os *scripts* CGI transmitem ao gerenciador de sessão o conteúdo da variável de ambiente QUERY_STRING (ou da entrada padrão no caso do método POST) e o conteúdo da variável de ambiente HTTP_COOKIE. Para separar os dois elementos de informação (de tamanhos desconhecidos), é introduzido um separador de campo especial.

A operação do gerenciador de sessão é baseada em uma tabela interna que mantém informações vitais sobre a sincronização do *browser* com o *thread* de aplicação. O gerenciador de sessão mantém uma entrada na tabela para cada cliente que está utilizando a aplicação (o número de entradas na tabela é igual ao número de *threads* da aplicação de banco de dados). Os campos mais importantes da tabela são:

- O *timestamp* da sessão;
- O valor do *cookie* que está atualmente armazenado no *browser* do cliente;
- O identificador do *thread* que serve a sessão considerada;
- O descritor do *socket* através do qual a requisição alcança o gerenciador de sessão.

Se uma nova requisição é recebida pelo *I-thread*, o gerenciador de sessão pode agir de quatro formas diferentes. Caso a requisição não tenha um valor HTTP_COOKIE associado, é atribuído um novo *thread* de aplicação (do *pool* de *threads* disponíveis) para o cliente (isto é, reserva uma entrada em sua tabela interna). Caso a requisição possua um valor HTTP_COOKIE mas não no formato usado pelo gerenciador de sessão, o gerenciador age da mesma forma que no primeiro caso. Caso a requisição tenha um valor HTTP_COOKIE no formato usado pelo gerenciador, mas a requisição não possui uma entrada na tabela interna, o gerenciador envia uma indicação de erro ao *browser* informando que por alguma razão o *browser* não está coordenado com a aplicação de banco de dados, ou que a sessão expirou. Por último, a requisição possui um valor HTTP_COOKIE no formato correto e possui uma entrada na tabela interna; nesse caso, o gerenciador de sessão valida a requisição.

Quando uma nova requisição é recebida pelo *I-thread* e a variável HTTP_COOKIE é validada, a cadeia de consulta é decodificada e encapsulada em uma mensagem destinado ao *thread* da aplicação responsável por manipular essa sessão. As mensagens são colocadas em uma fila de mensagens que é acessível por todos os *threads* da aplicação. Cada *thread* recupera da fila aquelas mensagens que possuem seu identificador de *thread*. O *O-thread* recebe o resultado da operação realizada pelo *thread* da aplicação através de uma conexão *socket*. O *thread* da aplicação também envia o seu identificador ao *O-thread*, isso permite que o *O-thread* identifique rapidamente na tabela interna do gerenciador de sessão o descritor de *socket* através do qual a resposta do *thread* de aplicação será transmitida ao *script* CGI.

O *O-thread* transmite um novo valor para o cabeçalho *Set-Cookie*; o novo valor do *cookie* e o *timestamp* da sessão atualizado também são armazenados na tabela interna do gerenciador de sessão. Quando uma sessão fica ociosa por mais que um período de tempo pré-estabelecido, o gerenciador de sessão invalida a entrada na tabela interna para

essa sessão e o *I-thread* envia uma mensagem especial ao *thread* de aplicação. Quando o *thread* recebe essa mensagem ele retorna ao *pool* de *threads* disponíveis para servir uma nova sessão de usuário.

Aplicação de Banco de Dados Multi-*Threaded*

A aplicação de banco de dados é um *software* estruturado de forma *multi-threaded.* Essa aplicação pode ter várias conexões ativas para o mesmo ou diferentes SGBDs (distribuídos em vários servidores). Desenvolver aplicações de banco de dados como um *software multi-thread* traz um ganho de desempenho considerável pois várias sessões de usuário podem ser atendidas simultaneamente pela aplicação.

No momento de inicialização do sistema um número pré-definido de *threads* de aplicação são criados. Cada *thread* implementa um programa lógico idêntico. No momento que chega a primeira requisição o *thread* de aplicação se conecta ao banco de dados, ele é capaz de executar qualquer classe de comandos SQL (SELECT, INSERT, UPDATE, etc.). Após o processamento da operação o *thread* retorna o resultado para o *O-thread* através da conexão *socket* e permanece em um estado bloqueado, aguardando por uma nova requisição. O *thread* da aplicação produz a saída em formato HTML encapsulando as informações recuperadas do banco de dados.

3.3 Servidor de Aplicação *Oracle*

Oracle9*i*AS (*Oracle9i Application Server* [Net]) é o novo servidor de aplicação da Oracle. Ele fornece uma plataforma simples, completa e integrada para o desenvolvimento e execução de aplicações *Web* e todos os tipos de aplicações Internet.

Visão Geral da Arquitetura

A fim de servir conteúdo *Web* aos seus usuários de forma confiável e robusta, o Oracle9*i*AS fornece um servidor *Web* com capacidade para servir um número grande de usuários de maneira escalável.

Oracle9*i*AS possui o servidor HTTP Oracle como componente central da arquitetura. Esse componente é constituído pelo produto mais popular da ASF (*Apache Software Foundation*), o servidor *Web* Apache. O servidor HTTP Oracle fornece uma versão do servidor Apache completamente configurada e testada (o servidor Apache representa cerca de 60% de todos os servidores *Web* no mundo).

A Figura 3.5 mostra a estrutura interna do Oracle9*i*AS. Nas próximas seções, citaremos algumas das principais características do Oracle9*i*AS; uma definição completa das características consideradas menos importantes para esse trabalho são encontradas em [Net01a].

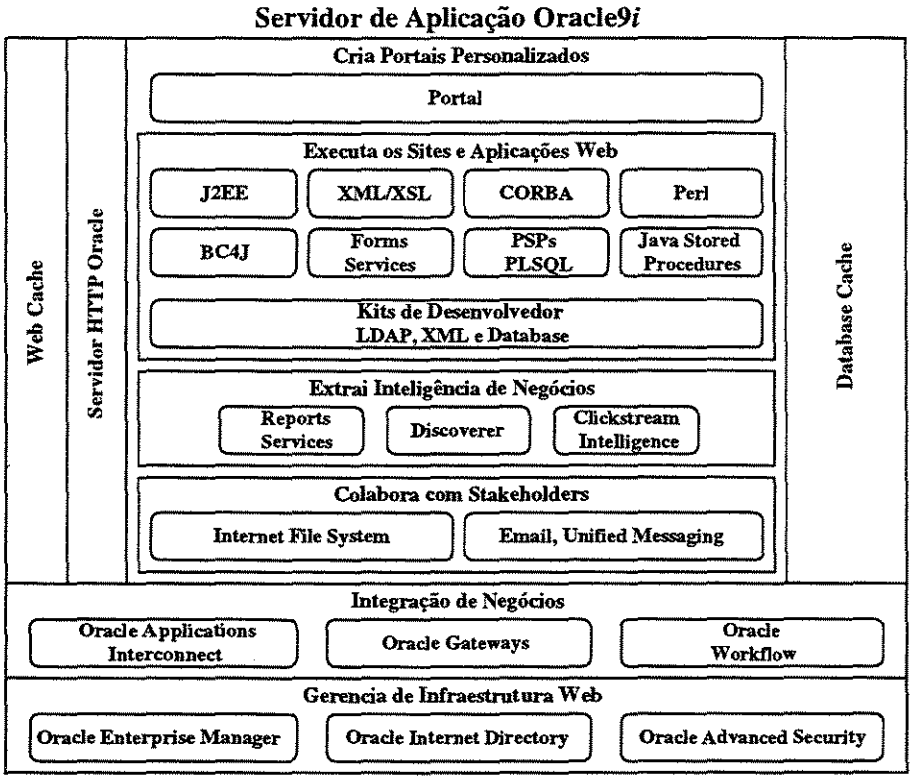


Figura 3.5: Arquitetura Interna do Oracle9*i*AS.

Oracle9*i*AS fornece uma plataforma de desenvolvimento que suporta as mais recentes linguagens de programação e tecnologias padrão, tais como: Java, Perl, C++, CORBA, XML, JSP (*Java Server Pages*), etc.

Extensibilidade do Servidor HTTP *Oracle*

A arquitetura do servidor HTTP Oracle, derivada do servidor Apache, é bastante modular: o componente central do servidor *Web* é muito pequeno; todas as suas funcionalidades são implementadas como módulos que são adicionados ao servidor e invocados dinamicamente durante o ciclo de vida de uma requisição HTTP.

Algumas das principais funcionalidades adicionadas pelos módulos suportados pelo servidor HTTP Oracle são:

1. Uma máquina Java Servlet (*Java Servlet Engine*) baseada no Apache JServ. O módulo mod_jserv transfere requisições HTTP feitas a aplicações Servlet para serem executadas na máquina Servlet (Apache JServ). Esse tópico será coberto em mais detalhes na subseção 4.1.1;

2. Uma máquina JSP (*Java Server Page Engine*) que executa sobre a máquina Servlet;

3. Uma máquina Perl (*Perl Engine*) que permite executar *scripts* Perl no mesmo processo que o servidor *Web*. O módulo mod_perl transfere requisições feitas a aplicações Perl para o interpretador Perl que está embutido no Servidor HTTP Oracle;

4. Uma máquina PLSQL (*PLSQL Engine*) que permite a execução de *stored procedures* em um banco de dados Oracle, diretamente do servidor *Web*, sem qualquer programação adicional. O módulo mod_plsql transfere requisições HTTP feitas a *stored procedures* para serem executados em um servidor Oracle;

5. *PLSQL Server Pages* permite a inserção de código PLSQL em páginas HTML. Essas páginas utilizam a máquina PLSQL mencionada anteriormente na execução dos *stored procedures*;

6. Componentes de negócios para Java (*Business Components for Java* - BC4J) que fornecem uma infra-estrutura para a construção de páginas JSP e Servlets;

7. *XML Development Toolkit* fornece as bibliotecas necessárias para a utilização de código XML nas aplicações;

8. Suporte SSL para garantir a segurança no tráfego de informações entre o usuário (*browser*) e as aplicações *Web*. O módulo mod_ssl fornece suporte padrão para o protocolo HTTPS. Esse módulo permite conexões seguras entre um usuário e o servidor HTTP utilizando um mecanismo de criptografia fornecido pela Oracle sobre um SSL (*Secure Sockets Layer*);

9. Suporte à execução de *scripts* CGI e FastCGI. O módulo mod_fastcgi permite a execução desses programas, que podem ser escritos em C, C++ ou Java;

10. Suporte à manutenção de sessão entre requisições HTTP consecutivas. O módulo mod_rewrite fornece essa funcionalidade através da codificação de variáveis de estado nas URLs (*URL Rewriting*);

11. Oracle9*i*AS possui suporte ao protocolo HTTP 1.1, permitindo a reutilização de uma conexão entre sucessivas requisições HTTP de um mesmo usuário.

A maioria das funcionalidades acima é herdada do servidor Apache, com exceção dos módulos *PLSQL Engine* (mod_plsql), *PLSQL Server Pages*, *Business Components for Java* e *XML Development Toolkit*.

A utilização do servidor Apache pela Oracle na construção de seu servidor HTTP prova que os produtos de código aberto, principalmente do Projeto Apache, são de qualidade e muito bem aceitos no mercado.

Desempenho

Oracle9*i*AS *Web* Cache

O Oracle9*i*AS *Web* Cache melhora significativamente o desempenho e escalabilidade de aplicações *Web* armazenando conteúdos estáticos e dinâmicos gerados pelas aplicações, reduzindo, assim, a carga nos servidores *Web*. O Oracle9*i*AS *Web* Cache opera como um servidor *proxy* que está situado em frente ao Oracle HTTP Server.

O Oracle9*i*AS melhora o desempenho de servidores *Web* armazenando na memória páginas acessadas freqüentemente, eliminando a necessidade de se processar várias vezes as mesmas requisições a aplicações ou consulta a bancos de dados. Ao contrário da maioria dos servidores *proxy* existentes, que são capazes de manusear somente conteúdo estático, o Oracle9*i*AS *Web* Cache acelera a entrega de conteúdo *Web* estático e dinâmico.

Oracle9*i*AS *Database* Cache

O Oracle9*i*AS *Database* Cache melhora o desempenho e escalabilidade de aplicações que acessam o banco de dados Oracle, armazenando dados e *stored procedures* que são usados freqüentemente. O Oracle9*i*AS *Database* Cache fornece os seguintes benefícios:

1. **Desempenho**: processar consultas de banco de dados no *middle-tier* reduz o tempo gasto enviando e recebendo dados sobre a rede;

2. **Escalabilidade**: reduz a carga no servidor de banco de dados, permitindo o suporte de mais usuários;

3. **Transparência**: aplicações existentes construídas usando o OCI (*Oracle Call Interface*), a interface cliente para aplicações conectando ao banco de dados Oracle, podem usar o Oracle9*i*AS *Database* Cache sem precisar fazer qualquer modificação no seu código.

Escalabilidade, Disponibilidade e Balanceamento de Carga

Em [Net01b], são descritas três áreas que determinam o desempenho de uma aplicação *Web*: escalabilidade, disponibilidade e a habilidade de balancear as requisições em vários servidores.

Escalabilidade

Qualidade que indica com que eficiência uma aplicação *Web* pode responder requisições à medida que as requisições de usuários aumentam. Essa melhora de eficiência pode ser obtida através da escalabilidade de *hardware*, dados, requisições e aplicação. A escalabilidade de dados e requisição são fornecidas com a utilização do Oracle9*i*AS *Database* Cache e Oracle9*i*AS *Web* Cache respectivamente.

Disponibilidade

Qualidade que indica com que eficiência uma aplicação *Web* responde no caso de uma falha de *software* ou *hardware*. Oracle9*i*AS utiliza isolação de sessão para isolar execuções de sessões de usuário; dessa forma, uma falha na aplicação tem impacto mínimo nos outros usuários. Oracle9*i*AS detecta automaticamente falhas de componentes do sistema, redireciona conexões, reinicia processos que falharam e pode mover código de aplicações que estão executando para um nó diferente.

Balanceamento de Carga

Característica que permite a distribuição de requisições entre diferentes servidores que estão operando juntos como um único servidor virtual. O balanceamento de carga ajuda a maximizar a escalabilidade de um sistema devido a uma melhor utilização de seus recursos de processamento. O Oracle9*i*AS faz o balanceamento de carga entre *threads* e processos em um único servidor e entre servidores em um ambiente com múltiplos nós.

3.4 Servidor de Aplicação *Enhydra*

O Projeto *Enhydra* [Enhb] é similar ao Projeto Apache, mas com foco em *softwares* E-*Business* girando em torno do servidor de aplicação. O Projeto *Enhydra* consiste de milhares de desenvolvedores em mais de 50 países.

Enhydra é um servidor de aplicações de código aberto (*Open Source* [Ray]) que facilita o desenvolvimento rápido de aplicações baseadas em Java e XML. *Enhydra* é um servidor escalável e de alto desempenho que é baseado nos principais padrões atuais, tais

como: XML, DODS e Java Servlet. O servidor *Enhydra* pode ser integrado com qualquer servidor *Web*, incluindo suporte para WAI, ISAPI, CGI, Apache JServ e RMI. *Enhydra* também pode funcionar como um servidor *Web standalone* aceitando diretamente requisições HTTP e oferecendo as funcionalidades fundamentais de um servidor *Web* (GET, POST, MIME, etc.).

Esse produto permite desenvolvedores construirem e executarem aplicações *multi-tiered*. Por razões de segurança, *Enhydra* incorpora suporte a SSL. Nessa arquitetura, a maioria das aplicações baseadas na *Web* consiste de três tipos fundamentais de componentes:

1. **Objeto de Apresentação**: contém a lógica que apresenta informação a uma fonte externa e recupera informações dessa fonte. A lógica de apresentação normalmente fornece menus de opções para permitir o usuário navegar através de diferentes partes da aplicação, e manipular *input* e *output*. Freqüentemente o componente de apresentação também executa uma quantidade limitada de validação de dados de entrada;

2. **Objeto de Negócio**: contém a lógica de aplicação que administra as funções e processos de negócios. Objetos de negócios são invocados por um componente de apresentação quando um usuário solicita uma opção, ou por uma outra função de negócio. As funções de negócio geralmente executam algum tipo de manipulação de dados;

3. **Objeto de Dado**: contém a lógica que faz interface com o sistema de armazenamento de dados, tais como: sistemas de bancos de dados, sistemas de arquivos hierárquicos ou com algum outro tipo de fonte de dados externo à aplicação. Objetos de negócios invocam objetos de dados para salvar dados persistentes.

Enhydra usa JDBC para se conectar a bancos de dados relacionais. JDBC fornece um nível de abstração que torna o SGBD usado transparente ao desenvolvedor da aplicação. *Enhydra* atualmente suporta *drivers* JDBC tipos 3 e 4 para Oracle, Microsoft SQL Server, Informix, Sybase e outras implementações padrões de JDBC.

A Figura 3.6 mostra em alto nível a arquitetura *Enhydra* [Enha]. Os principais componentes da arquitetura *Enhydra* são:

- Multi-Servidor *Enhydra*;
- Gerenciador de Administração;
- Gerenciador de Apresentação;

- Gerenciador de Sessão;
- Gerenciador de Banco de Dados.

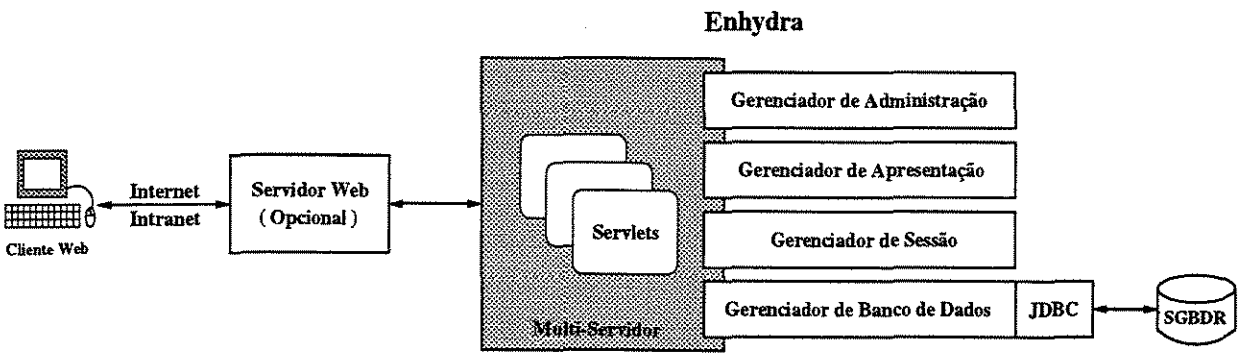


Figura 3.6: Visão geral da Arquitetura *Enhydra*.

Multi-Servidor *Enhydra*

Este componente gerencia o ciclo de vida e a conectividade dos servlets, incluindo aplicações *Enhydra*. O Multi-Servidor também é responsável por gerenciar o arquivo de configuração do servidor *Enhydra*, que define o estado inicial do servidor *Enhydra*.

Enhydra também fornece um carregador de classes junto com o Multi-Servidor. Existe uma instância do carregador de classes que inicializa cada servlet no Multi-Servidor. Esta correspondência um-para-um entre servlets e carregadores de classes permite novos servlets e aplicações serem carregados e executados sem ter que reinicializar o servidor inteiro. Para atualizar uma aplicação existente, basta reinicializar o seu carregador de classes.

Enhydra pode conectar a servidores *Web* através de um grande número de protocolos ou diretamente a um *browser* via protocolo HTTP. Devido a muitos servlets serem projetados para serem componentes reutilizáveis, o Multi-Servidor permite que servlets sejam acessados através de várias opções de conectividade. O Multi-Servidor suporta um grande número de opções de conectividade (WAI, RMI, CGI-Relay e HTTP). Isto significa que um único servlet pode ser conectado a vários servidores *Web*, usando diferentes protocolos. Essa funcionalidade aumenta muito a reusabilidade de servlets sendo executados no ambiente *Enhydra*. Além disso, através do uso de WAI e RMI é possível executar aplicações distribuídas no ambiente *Enhydra*.

Gerenciador de Administração

O gerenciador de administração é uma ferramenta gráfica que permite um gerenciador de sistema configurar e monitorar uma instância do *Enhydra* e as aplicações associadas a ela. Todas as informações de configuração para *Enhydra* e aplicações *Enhydra* são armazenadas em arquivos de configuração. Quando o servidor *Enhydra* inicializa, ele lê esses arquivos de configuração e inicializa o processo servidor e todas as aplicações especificadas no arquivo. Uma vez que uma instância *Enhydra* está executando, o gerenciador de administração é capaz de executar operações de gerenciamento. Todas as operações de gerenciamento funcionam da mesma forma; o estado ativo de *Enhydra*, uma aplicação, ou servlet é mudado e a alteração pode ser registrada nos arquivos de configuração.

Algumas das operações de gerenciamento que podem ser executadas pelo gerenciador de administração são:

- **Start/Stop**: inicializa ou finaliza uma aplicação ou servlet que está atualmente sendo executado na instalação *Enhydra*;
- **Add/Remove**: adiciona ou remove uma aplicação ou servlet de uma instalação *Enhydra*;
- **Modify**: modifica os atributos operacionais para *Enhydra*, uma aplicação ou servlet;
- **Check Status**: verifica o *status* do estado ativo de *Enhydra*, uma aplicação ou servlet. Incluindo o número de aplicações ativas, o *time-out* de uma sessão e o tamanho de um *pool* de conexões de banco de dados.

Gerenciador de Apresentação

O gerenciador de apresentação é responsável pela carga e execução de objetos de apresentação no contexto de uma aplicação *Enhydra*. O objeto de apresentação é responsável por apresentar informações para uma fonte externa e obter informações (*input*) dessa fonte. O gerenciador de apresentação transforma o nome do objeto de apresentação em uma URL, usa o carregador (*loader*) da classe especificada para carregar o objeto de apresentação e então executa o objeto de apresentação através da chamada do método *run()* do objeto. Existe uma instância do gerenciador de apresentação para cada aplicação *Enhydra*. O gerenciador de apresentação também pode armazenar objetos de apresentação em memória e quaisquer arquivos associados que fazem parte da aplicação.

Gerenciador de Sessão

O gerenciador de sessão e objetos de sessão fornecem um mecanismo simples e flexível que permite a manutenção de estado na *Web*. *Enhydra* fornece uma implementação geral de gerenciamento de sessão que serve como uma base para modelos de estado mais sofisticados. O Gerenciador de sessão fornece a servlets e outras aplicações que executam do lado do servidor a capacidade de manter estado sobre um usuário à medida que ele se move através da aplicação.

Enhydra mantém estado ao criar um objeto de sessão para cada usuário. Esses objetos de sessão são armazenados e mantidos no servidor. Quando um usuário faz a primeira requisição a uma aplicação, o gerenciador de sessão atribui ao usuário um novo objeto de sessão com um ID de sessão único. O objeto de sessão é então passado como parte da requisição para o servlet que manuseia a requisição. Servlets podem adicionar informações ao objeto de sessão ou ler informações dele. Se o usuário ficar ocioso por mais que um certo período de tempo, a sessão de usuário torna-se inválida e o gerenciador de sessão destrói o objeto de sessão correspondente à sessão do usuário.

Gerenciador de Banco de Dados

O gerenciador de banco de dados é o componente que gerencia os *pools* de conexões através de qualquer número de bancos de dados lógicos. Um banco de dados lógico é uma abstração que esconde as diferenças entre diferentes tipos de bancos de dados (bancos de dados heterogêneos).

Um banco de dados lógico usa JDBC para se conectar com servidores de bancos de dados tais como: Oracle, Microsoft SQL Server, InterBase, Postgres, Informix, Sybase, etc. O gerenciador de banco de dados é responsável pelo estado da conexão com o banco de dados. Especificamente, o gerenciador de banco de dados aloca e libera conexões para os bancos de dados lógicos.

A Figura 3.7 mostra o gerenciador de banco de dados e seu relacionamento com os bancos de dados lógicos. As aplicações não interagem diretamente com os SGBDs, o gerenciador de banco de dados faz a interface entre a aplicação e os SGBDs. Sempre que a aplicação precisar acessar o banco de dados, essa fará uma requisição ao gerenciador que retornará à aplicação uma conexão já aberta.

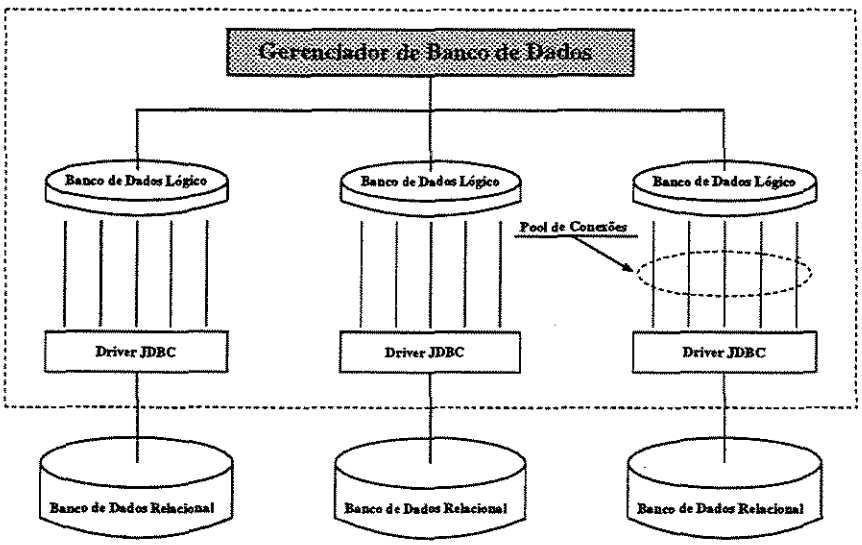


Figura 3.7: Gerenciador de Banco de Dados e Bancos de Dados Lógicos.

3.5 Gerenciador de Banco de Dados *InterBase*

InterBase [Cora] é um banco de dados relacional recentemente colocado como código aberto (*open source*) que combina facilidade de uso, baixo custo de manutenção e poder. Desde 1985 InterBase tem fornecido um SGBD de alto desempenho com tecnologias sofisticadas que aplicações de banco de dados necessitam. Em janeiro de 2000, a *Borland* anunciou que a nova versão do SGBD InterBase - InterBase 6.0 - seria de código aberto.

InterBase oferece alto desempenho para aplicações complexas e mantém a facilidade de desenvolvimento de pequenas aplicações. Oferece as melhores características de um servidor de banco de dados [Cor01] tais como: *triggers*, *stored procedures*, *two-phase commit*, restrições de integridade, etc. Essas características reduzem o tempo de desenvolvimento e melhoram a confiabilidade do servidor.

InterBase é portável, podendo ser executado sobre Windows 95/98, Linux, HP/UX e outros sistemas Unix. Um banco de dados *InterBase* possui características técnicas comparáveis às de servidores proprietários tais como Oracle, Microsoft SQL Server, etc.

Ferramenta de Administração

InterBase fornece uma interface gráfica de usuário denominada IBConsole. O IBConsole permite configurar e manter um servidor InterBase, criar e administrar bancos de dados no servidor, executar comandos SQL de forma interativa, gerenciar usuários e administrar

a segurança no servidor.

O IBConsole é um *front-end* GUI para ferramentas do InterBase que funcionam através da linha de comando. O IBConsole executa sobre *Windows*, mas pode gerenciar um banco de dados em qualquer servidor InterBase na rede local.

Banco de Dados Ativo

O InterBase suporta um grande número de características de banco de dados avançados, incluindo *triggers*, gerenciador de eventos baseado no servidor e funções definidas pelo usuário para habilitar regras e funcionalidades de negócio diretamente no servidor de banco de dados.

O benefício de se ter essas características é que muitas das regras de negócios normalmente usadas podem ser carregadas da aplicação cliente para o servidor. Isso resulta em uma aplicação cliente mais eficiente, manutenção de código centralizado e alto desempenho.

Restrições de Integridade

As restrições (*constraints*) de integridade mantêm de forma eficiente e segura a integridade dos dados no servidor. O InterBase possui quatro categorias de restrições de integridade:

1. **Chave primária e chave única**: garante que apenas uma linha na tabela possui o mesmo valor para o conjunto de colunas;
2. **Referencial**: garante que relacionamentos pai-filho entre tabelas são válidos; isto é, um conjunto de colunas em uma chave estrangeira tem os mesmos valores que as colunas da chave primária da tabela referenciada;
3. **Controle de restrições de integridade**: determina que a condição de pesquisa associada será válida para toda linha na tabela;
4. **Domínio**: pode ser usado para especificar um alcance de valores aceitáveis para uma coluna ou enumerar uma lista de valores válidos, bem como, valores *default*.

Triggers

Combinado com *stored procedures*, o InterBase possui uma poderosa implementação de *triggers*. O InterBase também suporta a noção de uma classe de *triggers*, que é usada para determinar a ordem em que *triggers* são executados.

Gerador de Eventos

Os *triggers* no InterBase também podem ser usados para enviar eventos ao gerenciador de eventos do InterBase. O Gerador de eventos habilita o banco de dados a funcionar como um banco de dados ativo que automaticamente notifica processos interessados quando uma mudança de estado acorre. O gerador de eventos economiza tempo de desenvolvimento e fornece um projeto de aplicação mais eficiente.

Arrays Multi-dimensionais

O InterBase fornece suporte a *arrays* multi-dimensionais usados em aplicações científicas e financeiras. Um único campo em um banco de dados pode armazenar um *array* de dezesseis dimensões. Para dados que devem necessariamente ser organizados como *arrays*, o InterBase simplifica o projeto do banco de dados e aumenta o desempenho. Infelizmente, esse recurso não faz parte do padrão ANSI/ISO SQL-92.

Banco de Dados Distribuído

O InterBase oferece a capacidade de executar o protocolo *two-phase commit* (2PC) de forma distribuída. Isso permite ao usuário alcançar automaticamente um alto grau de confiabilidade em um ambiente de banco de dados distribuído.

Sistema de *Backup*

O InterBase permite manter uma cópia exata do banco de dados original, denominado banco de dados *shadow*. Essa cópia é mantida em tempo real pelo servidor InterBase, fornecendo imediatamente um *backup* disponível em caso de falha de *hardware*. O banco de dados *shadow* executa automaticamente e adiciona uma penalidade mínima de desempenho.

3.6 Gerenciador de Banco de Dados *PostgreSQL*

PostgreSQL é um SGBD de código aberto [Teaa, Pos00] com características semelhantes às de SGBDs proprietários. O antecessor do PostgreSQL, desenvolvido pela Universidade da Califórnia em Berkeley (1977-1985), foi o Ingres. Em 1986 começou a implementação do Postgres ("*Post Ingres*") conduzido pelo Dr. Michael Stonebraker. Em 1995 foi adicionado suporte SQL ao Postgres, e seu nome foi mudado para PostgreSQL para refletir o seu relacionamento com o SQL padrão.

PostgreSQL é comparável a bancos de dados comerciais em termos de funcionalidades, desempenho e segurança. PostgreSQL oferece poder adicional ao SGBD por incorporar algumas características tais como: classes, herança (permite que uma tabela herde atributos de uma outra tabela), tipos de dados definidos pelo usuário e funções. Essas características colocam o PostgreSQL na categoria de bancos de dados conhecida como Objeto-Relacional. PostgreSQL é um sofisticado servidor de bancos de dados objeto-relacional, que suporta quase todos os construtores SQL, transações, visões, *stored procedures*, integridade referencial, *commit/rollback*, *checkpoints* e *triggers*.

O SGBDOR PostgreSQL (Sistema de Gerenciamento de Bancos de Dados Objeto-Relacional) é uma combinação das características de Sistemas de Gerenciamento de Bancos de Dados Orientado a Objetos (SGBDOO) e Sistemas de Gerenciamento de Bancos de Dados Relacionais (SGBDR). Ele permite novas características de bancos orientados a objetos, sem deixar de dar suporte a linguagens de banco de dados relacional.

Arquitetura

A arquitetura PostgreSQL utiliza o modelo cliente/servidor para servir as requisições feitas pelos usuários. Uma sessão PostgreSQL consiste da cooperação dos seguintes processos:

1. Processo *daemon* supervisor (POSTMASTER);
2. Aplicação do usuário;
3. Um ou mais processos servidores de bancos de dados.

Um único POSTMASTER gerencia vários bancos de dados em uma máquina servidora. As aplicações que desejam acessar o PostgreSQL devem fazer chamadas à biblioteca LIBPQ. Essa biblioteca envia requisições de usuário sobre a rede para o POSTMASTER, que inicia um novo processo servidor e conecta a aplicação a esse novo processo servidor. A partir desse ponto, a aplicação de banco de dados e o processo servidor PostgreSQL se comunicam sem a intervenção do POSTMASTER. Portanto, o POSTMASTER está

sempre executando, aguardando por novas requisições de usuário. A biblioteca LIBPQ permite uma única aplicação de banco de dados fazer várias conexões aos processos servidores PostgreSQL.

A Figura 3.8 [Teab] mostra como uma conexão é estabelecida nessa arquitetura.

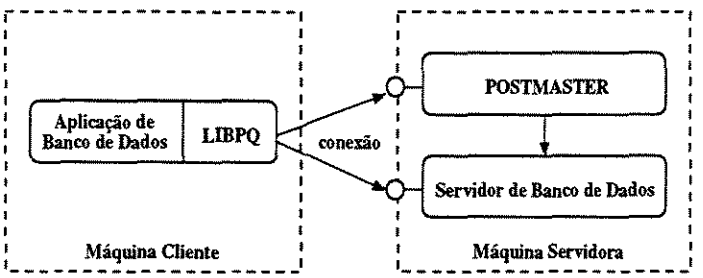


Figura 3.8: Processo de Conexão na Arquitetura PostgreSQL.

Plataformas

Sistema Operacional

PostgreSQL suporta vários sistemas operacionais de domínio público e proprietários (tais como Unix e MS Windows NT).

Linguagens de Programação

PostgreSQL suporta um grande número de interfaces de programação, incluindo:

- Java (JDBC);
- Perl 4/5 (DBI);
- Python (*Database* API);
- C e C++;
- PHP;
- TCL/TK.

Outras linguagens e ambientes de desenvolvimento podem acessar um banco de dados PostgreSQL através da interface ODBC (*Open DataBase Connectivity*).

Capítulo 4

Ferramentas de Código Aberto para Acesso a Bancos de Dados

Atualmente existem diversas ferramentas de código aberto que podem ser usadas no desenvolvimento de aplicações de bancos de dados. Todas as ferramentas abertas para acesso a bancos de dados apresentadas neste capítulo são capazes de serem utilizadas em uma aplicação que faz acesso concorrente a bancos de dados heterogêneos. A seguir, são apresentadas três tecnologias que estão sendo largamente usadas em aplicações de banco de dados baseadas na *Web*.

4.1 Ambiente Java

4.1.1 API Java Servlet

Java Servlet [Ber] é uma ferramenta muita poderosa que resolve muitas das deficiências geradas pelos programas CGIs. Servlets são módulos de código em Java que executam como uma aplicação servidora do lado do servidor *Web* (por isso o nome "servlet", similar a "applet" que é executado no lado do cliente). Servlets são portáveis; ao contrário de CGIs e APIs proprietárias (NSAPI da *Netscape* ou ISAPI da *Microsoft*), uma aplicação servlet é escrita uma única vez e pode ser executada em qualquer plataforma disponível para a aplicação. A aplicação servlet pode conectar-se com outros computadores utilizando *sockets* ou RMI (*Remote Method Invocation*). Também pode facilmente conectar-se a bancos de dados relacionais usando o JDBC (*Java DataBase Connectivity*) [Dara]. Pelo fato de serem totalmente escritos em Java, servlets podem ser facilmente utilizados no desenvolvimento de *middleware*.

Servlets têm várias vantagens sobre CGI, tais como:

- Um servlet não é executado em um processo separado. Desta forma, é removido o *overhead* de criação de um novo processo para cada requisição HTTP;
- Um servlet permanece em memória entre requisições. Um programa CGI (e provavelmente o seu interpretador) precisará ser carregado e inicializado para cada requisição HTTP;
- Existe uma única instância que responde a todas as requisições concorrentemente, permitindo assim servlets gerenciarem facilmente dados persistentes. Ao contrário de programas CGIs, uma conexão que for criada ainda existirá na próxima vez em que o servlet for acessado;
- Permitem gerenciamento de informações de estado no topo do protocolo HTTP [BLFF96].

Essas vantagens mostram claramente o grande ganho em desempenho que uma aplicação desenvolvida utilizando a tecnologia servlet tem sobre uma aplicação utilizando CGI [Darb, Gil].

Uma Máquina Servlet executa uma aplicação servlet em três diferentes estágios que definem o seu ciclo de vida. Esses estágios são:

1. **Inicialização**: neste primeiro estágio a Máquina Servlet carrega o arquivo *bytecode* do servlet no espaço de memória da Máquina Virtual Java (MVJ), cria uma instância (converte o arquivo carregado em um objeto válido) e inicializa o servlet. Neste estágio o método *init()* do servlet é chamado uma única vez. O estágio de inicialização pode ser usado pelo servlet que acabou de ser carregado, para alocar os recursos que serão usados mais tarde durante a execução. Recursos de conexões (URL, JDBC e arquivos) e alocação de memória são as operações mais comuns executadas neste estágio;
2. **Execução**: toda vez que uma requisição servlet é feita à Máquina Servlet, a instância do servlet requisitado já está presente no espaço de memória da máquina servlet, ou, se ainda não está presente, o estágio de inicialização é executado. Com a requisição servlet, a Máquina Servlet recebe todos os parâmetros da requisição que são usados para construir um objeto *ServletRequest*. Esse objeto será usado pelo servlet para acessar todas as informações necessárias sobre a requisição feita pelo cliente. Um outro objeto, denominado *ServletResponse*, contém todas as facilidades que o servlet precisa para retornar a saída para o cliente (*browser*). Esse estágio será executado quantas vezes forem feitas requisições à Máquina Servlet. Múltiplos

threads (um para cada requisição) podem executar esse estágio (método *service()*) concorrentemente;

3. **Destruição**: neste último estágio um servlet libera qualquer recurso que foi alocado durante o estágio de inicialização. O método *destroy()* é chamado uma única vez antes do servlet ser descarregado da memória.

Esses três estágios são controlados pela Máquina Servlet que os gerencia chamando o método apropriado na classe servlet (o método construtor e *init()* durante a inicialização, *service(ServletRequest, ServletResponse)* na execução e *destroy()* na destruição).

A Figura 4.1 mostra o ciclo de vida típico de um servlet:

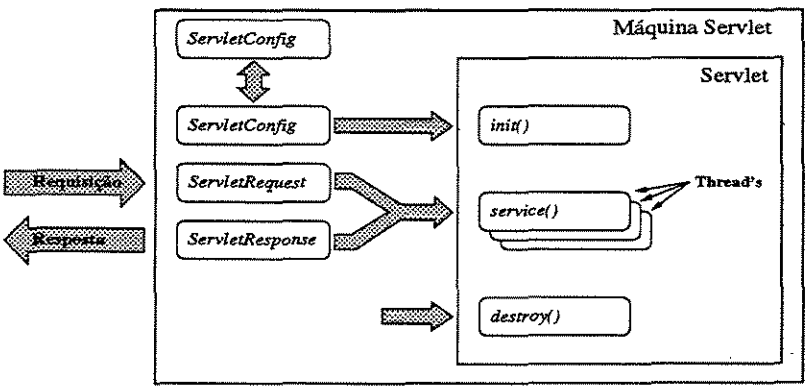


Figura 4.1: Ciclo de Vida de um Servlet.

Apache JServ

Analisando o problema de adicionar suporte a servlets em servidores *Web*, várias soluções têm sido desenvolvidas e alguns pacotes de *software* já estão disponíveis no mercado para alcançar esse objetivo. O Apache JServ [MFb] é uma dessas soluções.

Executar um servlet é uma tarefa similar à execução de um *script* CGI. Como os *scripts* que precisam de um interpretador (um *shell* no caso de um CGI genérico ou um interpretador de linguagem) para serem executados, o servlet, que é compilado em *bytecode* Java, precisa de uma Máquina Virtual Java para ser interpretado.

A diferença significativa entre um *shell* ou interpretador de linguagem e uma Máquina Virtual Java é o tempo de criação (*spawning time*) do processo. Os dois primeiros podem ser criados em um tempo limitado e utilizam pouquíssima memória (GNU *bash* 2.0 inicia em menos de 1/2 segundo em um PC e gasta em torno de 500Kb de RAM). A Máquina

Virtual Java é um processo muito mais pesado, requerendo dez vezes mais recursos (sobre o mesmo PC, a Máquina Virtual Java, SUN JVM, leva 5 segundos para iniciar e 4Mb de RAM). Deste ponto de vista, é fácil entender porque a Máquina Virtual Java não pode ser executada toda vez que uma requisição é feita.

A solução para esse problema implica em ter uma máquina virtual pronta e executando quando uma requisição a um servlet for feita. Em um servidor como *JavaSoft Java Web Server*, escrito em Java, esse problema não ocorre desde que servlets são executados pela mesma máquina virtual executando o próprio programa servidor. Esse modelo é chamado modelo *two-tier*, isto significa que quando uma requisição é feita por um cliente o próprio servidor *Web* executa o servlet.

Incluir uma máquina virtual no Servidor Apache [Apa] criaria vários problemas, tais como:

- Uma Máquina Virtual Java seria incluída em cada processo Apache, portanto multiplicando o seu "tamanho" e reduzindo o seu desempenho;
- Comprometeria a estabilidade do próprio Apache, desde que construir uma Máquina Virtual Java é uma tarefa extremamente complicada e difícil.

Para evitar tais problemas, o Apache JServ foi projetado seguindo o modelo *three-tier* [MFa]. A máquina servlet não é parte do servidor *Web*, mas uma aplicação servidora separada (*standalone*). Quando executando uma requisição servlet, o servidor *Web* atua como um cliente, subrequisitando servlets através da rede (usando o *Apache JServ Protocol*, comumente conhecido como protocolo AJP), convertendo os dados retornados e enviando de volta para o cliente que fez a requisição.

O Apache JServ é dividido em dois módulos principais: o **mod_jserv**, incluído no servidor *Web* e atuando como um cliente AJP, e o **Apache JServ**, atuando como um servidor AJP.

1. **mod_jserv**: é um módulo para o servidor Apache escrito em C. Sua principal tarefa é reenviar uma requisição HTTP disparada pelo servidor Apache para a Máquina Servlet do Apache JServ usando o protocolo AJP. Quando a Máquina Servlet termina de processar uma requisição AJP, os resultados são convertidos de volta para HTTP e enviados para o cliente;
2. **Apache JServ**: é uma Máquina Servlet escrita em Java. O Apache JServ é um servidor *standalone* e por essa razão não requer um servidor *Web* particular para operar (qualquer servidor *Web* modular terá seu próprio mod_jserv).

O Apache JServ opera da seguinte forma:

- Um cliente chama o servidor Apache requisitando um servlet;
- O servidor Apache redireciona a requisição HTTP para o módulo mod_jserv;
- O mod_jserv traduz a requisição HTTP do cliente em uma requisição AJP e contacta a Máquina Servlet do Apache JServ através do protocolo TCP/IP;
- O Apache JServ já executou o seu processo de inicialização e está pronto para receber requisições AJP;
- O Apache JServ traduz a requisição AJP em um objeto *ServletRequest*, cria um novo objeto *ServletResponse* usado pelo servlet para retornar dados para o cliente;
- Durante o estágio de execução, todos os dados passados para o objeto *ServletResponse* são convertidos em uma resposta AJP e enviados de volta ao mod_jserv;
- O mod_jserv traduz a resposta AJP em uma resposta HTTP e a envia de volta ao cliente.

A Figura 4.2 mostra o modo de operação do Apache JServ:

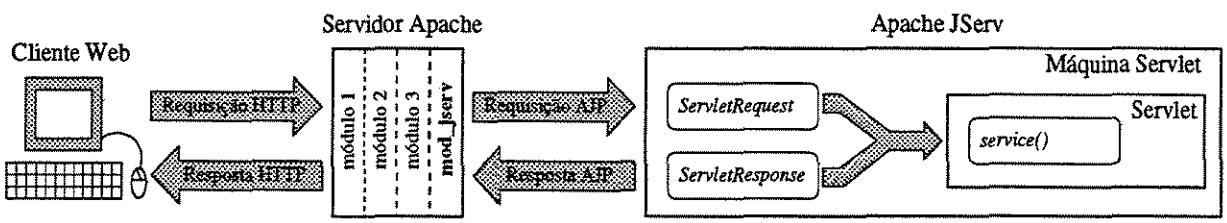


Figura 4.2: Modo de Operação do Apache JServ.

4.1.2 JDBC

O JDBC (*Java DataBase Connectivity*) é um conjunto de classes para integração da linguagem Java com bancos de dados relacionais. O JDBC é uma API (*Application Programming Interface*) que possibilita a adição de comandos SQL (*Structured Query Language*) em uma aplicação Java. A especificação do JDBC foi baseada no padrão X/Open SQL *Call Level Interface* (CLI), deixando-o muito parecido com um *driver* ODBC (*Open DataBase Connectivity*), que também foi baseado nesse padrão. A principal diferença é que

o ODBC é uma implementação em linguagem C nativa enquanto que o JDBC pode ser implementado independentemente de plataforma, seguindo os ideais da linguagem Java.

Um dos maiores problemas no desenvolvimento de banco de dados pode ser a curva de aprendizagem associada com cada pacote de banco de dados que existe no mercado. Todos os fornecedores de bancos de dados oferecem uma API diferente, e proprietária. Se desejar realizar qualquer programação com um banco de dados proprietário, necessitará aprender algo a respeito da API.

As APIs abertas são projetadas para dar independência de banco de dados, definem meios universais de fazer consultas, recuperar conjuntos de dados e fazer descobertas sobre meta-dados do banco de dados.

O JDBC possibilita desenvolver aplicações Java independentes de banco de dados. Muitos revendedores oferecem suporte ao JDBC, inclusive aqueles que lançaram pacotes de conectividade de banco de dados anteriores à época do JDBC. Utilizando o JDBC, é fácil construir aplicações portáveis para uma grande variedade de servidores de banco de dados.

Arquitetura do JDBC

A estrutura do JDBC inclui dois conceitos chaves:

- O objetivo do JDBC é ser uma interface independente de qualquer Sistema de Gerenciamento de Banco de dados (SGBDs), permitir um acesso genérico a banco de dados através de SQL e ter uma interface uniforme para diferentes fontes de dados;
- O programador deve escrever apenas uma interface para interagir com um banco de dados relacional e, utilizando o JDBC, o programa poderá acessar qualquer fonte de dados sem ter uma programação adicional para realizar essa operação.

A Figura 4.3 mostra a arquitetura do JDBC. A classe *DriverManager* é utilizada para abrir uma conexão com um banco de dados através de um *driver* JDBC, que precisa estar registrado antes de realizar uma conexão. Quando uma conexão é iniciada, o *DriverManager* escolhe um *driver* de uma lista de *drivers* disponíveis para efetuar a conexão, dependendo da URL especificada. Uma vez estabelecida a conexão com sucesso, as chamadas para consultas (*queries*) e busca de resultados (*fetch*) serão feitas diretamente ao *driver* JDBC.

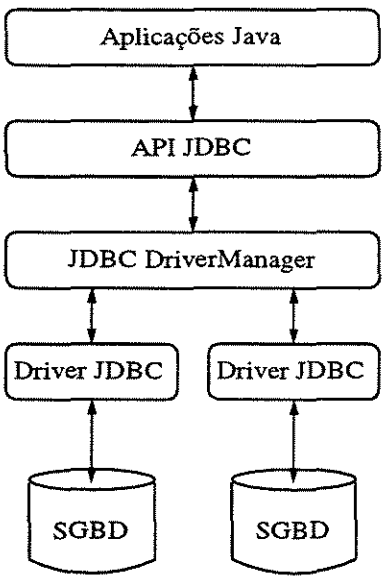


Figura 4.3: Arquitetura do JDBC.

As aplicações compartilham uma interface comum com o gerenciador de *drivers*. Especificamente, existem quatro tipos de *drivers* JDBC:

- **Tipo 1**: o *driver* traduz uma chamada JDBC em chamada ODBC, que é então processada por um *driver* ODBC. Contudo, o código binário ODBC, e em muitos casos o código do banco de dados cliente, deve ser carregado em todas as máquinas que usam esse *driver*. Neste método ocorre uma degradação no desempenho causado pela adição de uma camada extra (ODBC) entre os clientes e o SGBD;

- **Tipo 2**: o *driver* traduz chamadas JDBC em chamadas da API nativa do SGBD a ser conectado. A maioria dos SGBDs fornece chamadas de API padrão para que as aplicações possam se comunicar com seus SGBDs. Contudo, esse método também requer que algum código binário seja carregado em cada máquina cliente. Esse método proporciona uma melhora no desempenho porque chamadas de API de SGBDs normalmente são mais eficientes que chamadas ODBC;

- **Tipo 3**: o *driver* traduz chamadas JDBC dentro de um protocolo de rede independente de SGBD, que é então traduzido para o protocolo do SGBD pelo servidor. Este servidor de rede é capaz de conectar clientes escritos em Java a diferentes bancos de dados;

- **Tipo 4**: o *driver* traduz diretamente chamadas JDBC em protocolos de rede usados pelo SGBD a ser conectado. Essa abordagem é similar à abordagem do *driver* tipo 2. Ela permite chamadas diretas de máquinas cliente a SGBDs; essa é uma excelente solução para acessos a bancos de dados em Intranets. Contudo, desde que muitos desses protocolos são proprietários, os vendedores de bancos de dados são os principais desenvolvedores desse tipo de *driver*.

4.2 Ambiente Perl

4.2.1 Mod_Perl

Como mencionado anteriormente, o problema mais crítico na execução de *scripts* CGI é que são criados processos adicionais em cada vez que um *script* é executado. Quando um usuário faz uma requisição a um *script* CGI, o processo CGI ocorre externamente ao servidor *Web*, isso significa que o servidor *Web* deve criar um novo processo para executar o *script* CGI. No caso de um *script* CGI implementado em Perl, o interpretador Perl deve ser carregado na memória e executado. O interpretador Perl também é executado como um processo separado, esse compila e executa o código em Perl, e retorna o resultado ao servidor *Web*. Esse método funciona bem, exceto por dois problemas: o processo é lento, desde que o *script* Perl tem que ser re-interpretado toda vez que ele é executado, e isto consome muita memória e tempo, devido ao fato de que o interpretador Perl deve ser executado para cada requisição ao *script* Perl.

Acima é descrito o processo padrão de execução de um *script* Perl. Para aplicações *Web* com baixo volume de requisições e/ou baixa demanda de processamento, CGI é fácil de implementar e os custos são ainda razoáveis. Mas, para aplicações *Web* com um grande volume de requisições simultâneas, esse processo de execução fará com que o sistema fique lento e frustará o usuário que deverá esperar muito tempo pela resposta de sua requisição.

Uma solução para esse problema é a utilização de um módulo Apache, denominado mod_perl. Este módulo, escrito por *Doug MacEachern*, embute o interpretador Perl diretamente dentro do servidor *Web*. Apache::Registry é um módulo Perl que trabalha em conjunto com mod_perl, esse módulo é usado para emular o ambiente CGI, com isso, *scripts* CGI escritos em Perl podem ser usados com mod_perl sem ter que reescrevê-los. Com Apache::Registry, cada programa CGI individual é compilado e armazenado em memória na primeira vez que for requisitado, e então permanece disponível para todas as próximas requisições desse *script* CGI. Esse processo evita o custo de inicialização (*startup*) do *script*. O efeito é que os *scripts* CGI são pré-compilados pelo servidor e executados sem a necessidade de criação de novos processos, portanto executando muito mais rapidamente e eficientemente os *scripts* Perl. Os benefícios da integração do servidor *Web* com

o interpretador Perl podem ser divididos em duas partes:

1. Devido ao interpretador Perl ser construído dentro do processo filho do Apache, os *scripts* Perl podem ser executados muito mais rapidamente: o interpretador Perl não precisa ser carregado na memória para cada invocação do *script*. Dependendo da configuração, módulos e/ou *scripts* Perl podem ser completamente ou parcialmente pré-compilados e armazenados na memória;

2. Mod_perl possibilita a escrita de módulos Apache em Perl. Com isso, um código em Perl pode intervir em qualquer estágio de processamento do servidor Apache.

4.2.2 DBI

O DBI (*DataBase Interface* [Des]) é uma API para a linguagem Perl, que fornece acesso a bancos de dados. O DBI define um conjunto de funções, variáveis e convenções que fornecem uma interface consistente para banco de dados, independente do banco de dados que está sendo usado. O DBI abstrai a necessidade de entender o nível mais baixo da tecnologia do banco do dados. O DBI é uma camada transparente que fica entre uma aplicação e um ou mais DBDs (*DataBase Drivers*), isto é, são os *drivers* que realmente interagem com os bancos de dados.

Arquitetura do DBI

Com a explosão da popularidade da linguagem Perl como uma das principais linguagens de programação CGI, se fez necessário a definição de uma interface de conexão a banco de dados simples e padronizada.

A Figura 4.4 mostra os componentes da arquitetura DBI. O *Switch* é o código responsável por chamar o método para o *driver* apropriado para a execução atual. O *switch* também é responsável, entre outras coisas, pela carga dinâmica dos *drivers*, checagem e manipulação de erros. Os DBDs implementam suporte para um dado tipo de sistema de banco de dados. Existe um grande número de DBDs para SGBDs comerciais e de código aberto, tais como:

- **DBD::Oracle** - *driver* para Oracle 7 e Oracle 8;
- **DBD::FreeTDS** - *driver* para SQL Server e Sybase;
- **DBD::Sybase** - *driver* para Sybase;
- **DBD::Pg** - *driver* para PostgreSQL;

- **DBD::mSQL(MySQL)** - *driver* para mSQL;
- **DBD::Informix** - *driver* para Informix.

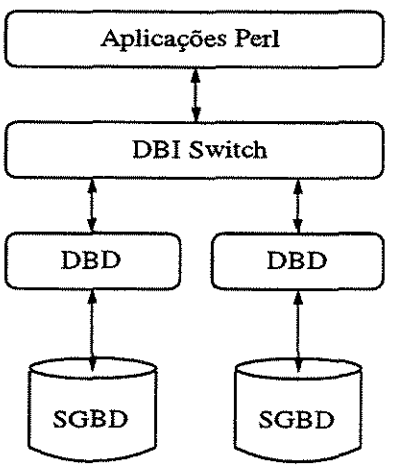


Figura 4.4: Arquitetura do DBI.

4.2.3 Apache::DBI

A principal limitação de um *script* CGI em relação aos bancos de dados é que as conexões com os bancos de dados não são persistentes. Toda requisição de um *script* CGI tem que reconectar ao banco de dados, e quando a requisição terminar a conexão é fechada.

Apache::DBI foi escrito para superar essa limitação. Esse módulo inicializa um *pool* de conexões persistentes ao banco de dados. Todo pedido de conexão ao banco de dados que é feita através do DBI, será transferido ao módulo Apache::DBI. Esse módulo verifica se o *handle* da requisição de conexão anterior já está armazenado e verifica usando o método *ping* se esse *handle* ainda é válido. Se estas duas condições forem verdadeiras o Apache::DBI apenas retorna o *handle* do banco de dados. Se não existir um *handle* apropriado ou se o método *ping* falha, uma nova conexão é estabelecida e o *handle* é armazenado para futuramente ser reutilizado. Quando o *script* Perl é executado novamente por um processo filho do Apache que ainda está conectado, o Apache::DBI apenas verifica o *cache* de conexões abertas comparando os parâmetros *host*, *username* e *password*. Uma conexão com os mesmos parâmetros é retornada se estiver disponível ou uma nova conexão é criada e depois retornada. Não existe necessidade de apagar o comando "*disconnect*" do código de um *script* Perl, pois, o módulo Apache::DBI faz uma sobrecarga desse método com um método vazio. Assim, quando um *script* Perl deseja conectar-se a um banco de

dados, o Apache::DBI retorna imediatamente uma conexão válida e o *script* Perl pode conectar-se ao banco de dados sem ter que abrir uma nova conexão com a base de dados. Isso somente é possível com *scripts* Perl sendo executados em um servidor *Web* com o módulo mod_perl.

Se é desejado que uma conexão já esteja aberta quando um *script* Perl é executado pela primeira vez depois da inicialização do servidor *Web*, o módulo Apache::DBI permite que durante o processo de inicialização do servidor *Web* sejam abertas todas as conexões que serão inicialmente usadas pelos *scripts* Perl. Essa característica deverá ser usada somente se todos os usuários tiverem permissão de conectar-se ao banco de dados através dessas conexões.

O módulo Apache::DBI ainda possui algumas limitações: esse módulo mantém persistentes as conexões com os bancos de dados através dos processos filhos do servidor Apache. Se um usuário acessa várias vezes um banco de dados, as requisições HTTP serão manuseadas muito provavelmente por diferentes processos filhos do Apache. Todo processo filho do Apache precisa fazer sua própria conexão. Não haverá problema, se todos os processos filhos do Apache puderem compartilhar os *handles* das conexões. Uma possível solução pode ser a utilização de uma versão *multi-thread* do servidor Apache.

Com essa limitação em mente, existem cenários, onde o uso do Apache::DBI não é aconselhável. Por exemplo, uma aplicação *Web* com um grande volume de requisições onde todo usuário conecta-se ao banco de dados com um identificador que é diferente para cada usuário. Todo processo filho do Apache abrirá muitas conexões ao banco de dados. Em conseqüência disso, em pouco tempo o servidor Apache entrará em colapso.

Um outro problema são os *timeouts*: alguns bancos de dados disconectam-se do cliente depois de um certo período de tempo de inatividade. O Apache::DBI tenta validar o *handle* do banco de dados usando o método *ping* do módulo DBI. Por *default* esse método retorna que o *handle* é válido. Se o *handle* não é válido e o *driver* não tem implementado o método *ping*, será retornado ao *script* Perl um *handle* não válido que provocará um erro ao se tentar acessar o banco de dados com esse *handle*.

4.3 Ambiente Python

Python é uma linguagem de programação interpretada, interativa, orientada a objetos e escrita em C. Seu projeto combina características de engenharia de *software* de linguagens tradicionais com a grande aplicabilidade de linguagens *script*. A linguagem Python é freqüentemente comparada às linguagens: Java, Perl e Tcl.

Python combina um grande poder de expressão com uma sintaxe muito clara e enxuta, podendo ser estendido pela adição de novos módulos implementados em linguagens compiladas, tais como: C ou C++. A implementação de Python é portável: podendo

ser executada sobre Unix (Solaris, Linux e outros), Windows, OS/2 e outros; ou seja, a implementação de Python pode ser suportada por qualquer sistema operacional que possua um compilador C.

4.3.1 Python Database API

Esta API foi definida para padronizar a interface entre módulos Python e diferentes SGBDs. Com isso, é esperado alcançar uma alta consistência conduzindo a módulos mais facilmente compreensíveis, códigos que são geralmente mais portáveis através de bancos de dados e um amplo alcance de conectividade de bancos de dados.

Capítulo 5

Aspectos de Implementação

Algumas arquiteturas comerciais são intrinsecamente muito complexas, pois assumem requisitos de implementação muito complexos. Isso tem um impacto direto no custo do sistema, o que não é aceitável em muitos projetos. E, além disso, esses sistemas comerciais incluem funcionalidades não necessárias à maioria das aplicações como, por exemplo, transações distribuídas controladas pelo *browser*. Assim, existem muitos sistemas nos quais as soluções comerciais não fornecem uma solução com o custo/benefício apropriado.

Qualquer arquitetura de desenvolvimento de aplicações com interface *Web* deve resolver a maioria dos problemas citados na subseção 3.1, os quais podem ser divididos em duas categorias distintas: projeto (*design*) e implementação. Os problemas de projeto incluem diálogo da aplicação e especificação da navegação entre páginas. Os problemas de implementação incluem simulação de estado (com conexão persistente), implementação do diálogo e mecanismo seguro de controle de navegação entre páginas utilizando o conceito de perfil de usuário.

Para alcançar todos esses objetivos, foi introduzido nas abordagens tradicionais de desenvolvimento de aplicações *Web*, um agente especializado denominado GABD*Web* (Gerenciador de Aplicações de Bancos de Dados na *Web*). O GABD*Web* faz uso de um módulo interno, denominado Gerenciador de Sessões, que mantém traços de estado de um cliente *WWW* (*browser*) e coordena a sua interação com a aplicação de banco de dados.

5.1 Arquitetura do GABD*Web*

5.1.1 Ambiente Java

GABD*Web* foi implementado como um pacote Java. Esse pacote fornece capacidade à aplicação de realizar a preservação de estado, através de uma tabela de sessões, entre consecutivos acessos de um mesmo usuário. GABD*Web* possui um *pool* de conexões com-

partilhadas. Quando uma aplicação servlet termina de usar uma conexão, essa conexão é reciclada e não destruída. Com isso, as conexões tornam-se persistentes e podem ser utilizadas entre várias requisições HTTP.

A estrutura com vários *threads* (*multi-threaded*) das aplicações servlets fornece uma vantagem de desempenho sobre as implementações convencionais com um único *thread* (*single-threaded*). Dessa forma, várias requisições HTTP podem ser manipuladas simultaneamente diminuindo o tempo de resposta do usuário. A máquina servlet utilizada na execução dos servlets foi o Apache JServ [MFb]. Os *drivers* utilizados no acesso aos bancos de dados foram: JDBC OCI (*driver* nativo do Oracle para Java) para acessar o SGBD Oracle e JDBC FreeTDS para acessar o SGBD MS SQL Server.

GABD*Web* possui um *pool* de conexões para cada banco de dados utilizado pelo sistema. A Figura 5.1 apresenta a arquitetura geral do GABD*Web* utilizando servlets:

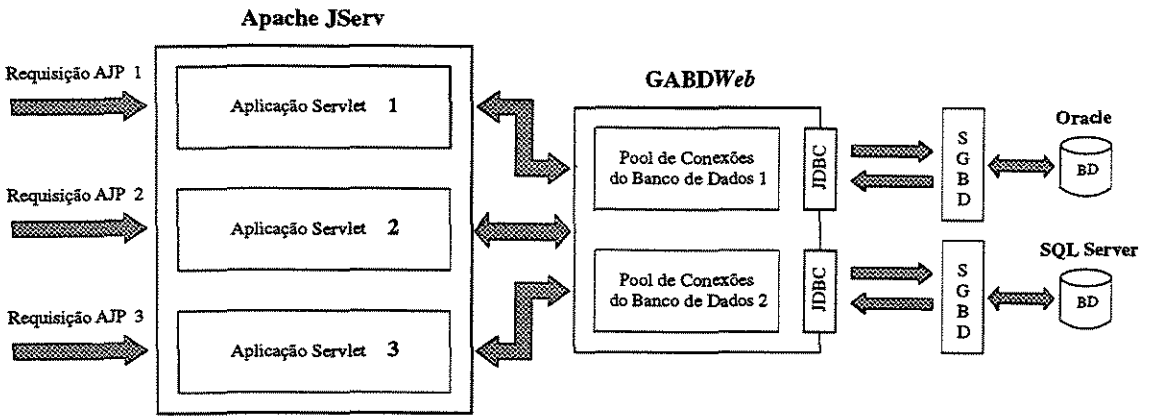


Figura 5.1: Arquitetura Geral do GABD*Web* Utilizando Servlets.

GABD*Web* fornece controle sobre os *drivers* de vários bancos de dados, segurança sobre o acesso de usuários aos dados da aplicação e controle de concorrência das requisições HTTP dos clientes.

A classe GABD*Web* possui métodos para:

- Pegar uma conexão do *pool* de conexões, passando como parâmetro o nome do *pool*;
- Retornar uma conexão para o *pool*;
- Criar uma nova sessão de usuário;
- Fechar uma sessão de usuário;
- Verificar a autorização de um usuário;

- Validar uma transição de um usuário;
- Liberar (*release*) todos os recursos e fechar todas as conexões durante o processo de *shutdown*;
- Manter um arquivo de *log* (*ApplicationManager.log*) do sistema. Esse arquivo é usado para verificar o funcionamento correto do sistema.

A classe GABD*Web* foi desenvolvida de acordo com o padrão *Singleton*. Um *Singleton* é uma classe com apenas uma instância. Após a instância de GABD*Web* ter sido criada por uma aplicação servlet, outras aplicações poderão somente pegar uma referência para essa instância através de um método estático da classe.

A Figura 5.2 apresenta a estrutura do GABD*Web* em Java:

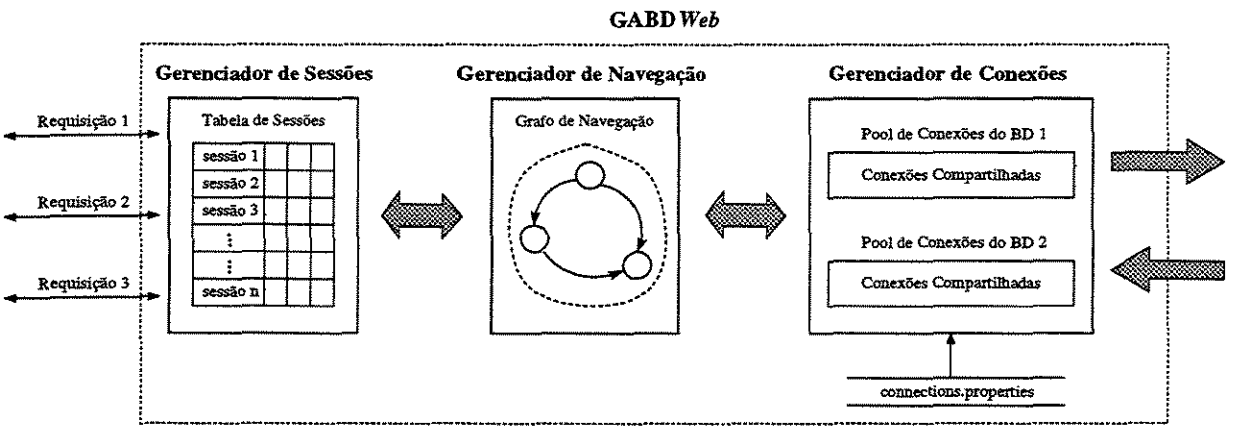


Figura 5.2: Estrutura do GABD*Web* em Java.

A arquitetura de GABD*Web* é dividida em três módulos principais:

1. **Gerenciador de Sessões**: responsável por gerenciar as sessões de usuários entre várias requisições HTTP;
2. **Gerenciador de Navegação**: responsável por validar uma transição de uma página para outra;
3. **Gerenciador de Conexões**: mantém e coordena os *pools* de conexões compartilhadas.

Gerenciador de Sessões

Este módulo é o componente central do GABD*Web*. O Gerenciador de Sessões coordena a comunicação entre o cliente (*browser WWW*) e a aplicação de banco de dados com estado. Uma sessão é criada pelo Gerenciador de Sessões após o usuário ser autenticado em uma página inicial da aplicação, onde o usuário fornece o seu nome (*username*) e sua senha (*password*). Dessa forma, uma sessão (com identificador único) será criada para esse usuário. Para manter a aplicação de banco de dados coordenada com o *browser*, o Gerenciador de Sessões faz uso do mecanismo de gerenciamento de estado, apresentado na subseção 2.3.2, denominado *cookie*.

A Figura 5.3 mostra a estrutura interna do Gerenciador de Sessões:

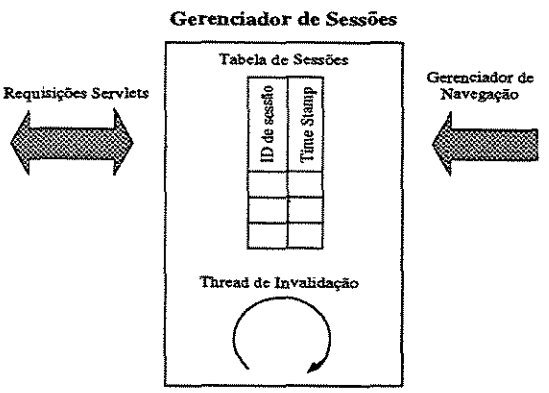


Figura 5.3: Estrutura do Gerenciador de Sessões.

O Gerenciador de Sessões fornece alguns métodos para o controle das sessões de usuário. Esses métodos são usados para:

- Criar uma nova sessão de usuário;
- Validar uma sessão de usuário;
- Fechar uma sessão de usuário.

A operação do Gerenciador de Sessões é fortemente baseada nos dados armazenados na tabela de sessões, que mantém informações vitais sobre a sincronização dos *browsers* com as aplicações servlets. O Gerenciador de Sessões mantém um registro na tabela de sessões para cada usuário servido pelas aplicações servlets que possuem uma sessão de usuário válida. Esse número é dinâmico, podem ser criadas quantas sessões forem necessárias para servir as requisições feitas pelos usuários às aplicações servlets.

Internamente, o Gerenciador de Sessões possui uma estrutura com dois componentes:

1. **Tabela de Sessões**: armazena os *timestamps* das sessões de usuário válidas;
2. *Thread* **de Invalidação**: responsável por invalidar uma sessão após o seu tempo de validade (*timeout*) ter expirado.

Tabela de Sessões

Este componente é responsável por armazenar todos os *timestamps* das sessões de usuário que são correntemente válidas no sistema. Quando uma sessão expirar ou for fechada pelo usuário ela deverá ser excluída da Tabela de Sessões.

Os campos da Tabela de Sessões são:

- **Identificador de Sessão**: identificador criado pelo sistema para identificar univocamente uma sessão de usuário;
- **Timestamp**: instante (ms) da última utilização da sessão.

Quando uma nova requisição HTTP é recebida pela aplicação servlet, o Gerenciador de Sessões tentará validar a sessão de usuário. Desta forma, três alternativas podem ocorrer:

1. A requisição HTTP não possui um *cookie* com um identificador de sessão;
2. A requisição HTTP possui um identificador de sessão, mas esse identificador não possui uma entrada na Tabela de Sessões;
3. A requisição HTTP possui um identificador de sessão, e esse identificador possui uma entrada na Tabela de Sessões.

Na alternativa 1, como não existe um identificador de usuário na requisição, o sistema retornará uma página HTML de identificação (*login*) para o usuário.

Na alternativa 2, GABD*Web* verificará junto ao Gerenciador de Sessões se existe uma sessão válida para esse usuário. Como não existe, o sistema retornará novamente uma página de identificação para o usuário. Dessa forma, após a identificação do usuário e a validação do sistema uma nova sessão será criada para esse usuário. Para isso, o Gerenciador de Sessões enviará um *cookie* ao *browser* do usuário contendo o identificador dessa sessão. O tempo de vida do *cookie* é igual ao *timeout* da sessão.

Na alternativa 3, existe uma sessão válida para o usuário. Logo após ser completada a requisição do usuário, o *timestamp* da sessão é atualizado e um novo *cookie* é enviado ao *browser* do usuário. Quando uma sessão for invalidada, um novo *cookie* com um tempo de

vida igual a zero será enviado ao usuário, isto significa que o *cookie* na máquina do usuário deverá ser descartado imediatamente. Dessa forma, quando um usuário que possui uma sessão válida fizer uma requisição de serviço à aplicação servlet, não haverá necessidade de se identificar novamente.

Thread de Invalidação

A função deste componente é invalidar todas as sessões de usuário que permanecerem sem atividades por mais que um período de tempo (*timeout*) pré-definido. Para essa finalidade é armazenado e atualizado na Tabela de Sessões o *timestamp* de cada sessão válida. Esse método do Gerenciador de Sessões é executado em um *thread*, sendo executado portanto de forma concorrente com todo o sistema. O *Thread* de Invalidação permanece inativo durante um período de tempo, e novamente verifica se existem sessões expiradas. Esse processo se repete durante todo o tempo em que o sistema estiver em uso.

Depois que o *timeout* da sessão tiver expirado, o Gerenciador de Sessões identificará que a sessão está inativa (isto é, que a sessão foi terminada). Com isso, o *Thread* de Invalidação invalidará a sessão, retirando o seu identificador da Tabela de Sessões e liberando todos os recursos alocados para essa sessão de usuário.

Gerenciador de Navegação

A função do Gerenciador de Navegação é validar todas as transições entre páginas que um usuário da aplicação servlet tentar realizar. Para isso, toda vez que for feita uma requisição à aplicação, o Gerenciador de Navegação, logo após a validação da sessão, verificará se o usuário possui ou não permissão para acessar a página solicitada. Se não possuir, será enviado uma mensagem de erro para o *browser* do cliente.

O Gerenciador de Navegação é o componente da arquitetura responsável por fazer o controle de acesso dos usuários da aplicação servlet. Com isso, pode-se ter uma aplicação com os usuários distribuídos geograficamente, mas com um conjunto específico de permissões de acesso à aplicação.

Gerenciador de Conexões

A Figura 5.4 apresenta a estrutura do Gerenciador de Conexões. O Gerenciador de Conexões funciona como um servidor de conexões persistentes, servindo como um componente central entre as aplicações servlets e os SGBDs. Apesar de ser transparente ao sistema, quando estiver sendo usado o protocolo HTTP 1.1 [FGM+97], com conexão persistente, o desempenho do mesmo melhorará consideravelmente em relação a quando estiver sendo usado o protocolo HTTP 1.0, que não possui conexão persistente [Pad95, PM94, BLFF96].

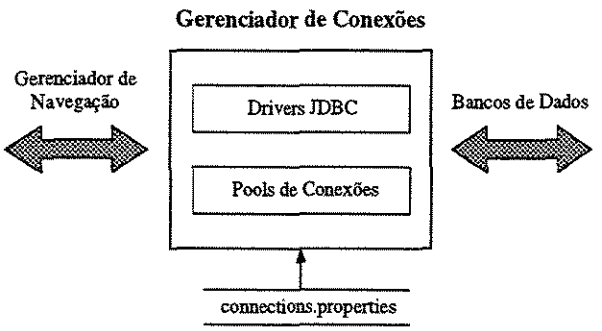


Figura 5.4: Estrutura do Gerenciador de Conexões.

A classe Gerenciador de Conexões é um invólucro (*wrapper*) dos *pools* de conexões que são criados pelo sistema. O Gerenciador de Conexões é responsável por:

- Carregar e registrar todos os *drivers* JDBC que serão utilizados pelo sistema. Durante o processo de *shutdown* todos os *drivers* são descarregados do sistema;
- Criar os *pools* de conexões baseado nos parâmetros definidos no arquivo de propriedades (*connections.properties*);
- Manter traços das aplicações servlets que estão utilizando o sistema para finalizar todos os *pools* quando o último cliente (aplicação servlet) for finalizado.

Configuração do Gerenciador de Conexões

Durante o processo de inicialização do GABD*Web*, o arquivo *connections.properties* fornece os parâmetros de configuração do sistema. Esse arquivo é fornecido no formato *properties* contendo pares chave/valor (*key/value*) que definem os *pools* de conexões. As propriedades comuns entre os *pools* são as seguintes:

- **drivers**: lista de *drivers* JDBC que serão carregados pelo Gerenciador de Conexões. O Gerenciador de Conexões conecta-se aos bancos de dados usando um *driver* JDBC. Por isso, é importante especificar o *path* do *driver* JDBC no *CLASSPATH* do sistema;
- **logfile**: arquivo de *log* do sistema.

Outras propriedades são fornecidas para cada *pool* de conexões. O nome de cada propriedade inicia com o identificador do banco de dados:

- **<database_id>.url**: URL JDBC que define a base de dados;
- **<database_id>.user**: nome do usuário para a base de dados;
- **<database_id>.password**: senha do usuário;
- **<database_id>.numconn**: número de conexões que são abertas automaticamente pelo GABD*Web* durante o processo de inicialização;
- **<database_id>.maxconn**: número máximo de conexões que podem estar abertas simultaneamente em cada *pool* de conexões.

O identificador *database_id* é usado no arquivo de configuração para identificar um *pool* de conexões. O nome do usuário e sua correspondente senha devem ser válidos para o banco de dados definido pela propriedade "url". Durante a inicialização do sistema todos os *drivers* fornecidos pela propriedade "drivers" são carregados dentro da máquina virtual Java e uma instância para a classe é criada. Com isso, todos os *drivers* são registrados e armazenados em um vetor. Esse vetor é utilizado para descarregar todos os *drivers* durante o processo de *shutdown*.

A classe Gerenciador de Conexões possui métodos para:

- Pegar uma conexão do *pool* de conexões, passando como parâmetro o nome do *pool* (*database_id*);
- Retornar uma conexão para o *pool*;
- Liberar (*release*) todos os recursos e fechar todas as conexões durante o processo de *shutdown*.

Pool de Conexões

Esta classe armazena e gerencia todas as conexões compartilhadas que estão abertas para cada base de dados. Os bancos de dados são identificados por uma URL JDBC. Uma URL JDBC consiste de três partes: o identificador do protocolo (sempre jdbc), o identificador do *driver* (odbc, oracle, etc.) e o identificador do banco de dados.

Cada banco de dados do sistema possui um *pool* de conexões. Esses *pools* de conexões são criados durante o processo de inicialização do GABD*Web*, baseados no arquivo de propriedades. Devido ao alto custo de uma licença de usuário em SGBDs comerciais (por exemplo, Oracle, Microsoft SQL Server, etc.), as conexões de um *pool* serão todas abertas

para um mesmo usuário. Com isso, será minimizado o custo do sistema com relação ao número de licenças de usuários de um SGBD.

O sistema inicialmente cria, para cada *pool*, um número pré-definido (*numconn*) de conexões para os usuários, as conexões são definidas pelas variáveis *user* e *password*. Durante a execução do sistema somente um número máximo (*maxconn*) de conexões podem estar abertas simultaneamente para cada *pool*. Se esse limite de conexões for alcançado, durante uma requisição o usuário ficará bloqueado até uma conexão ser liberada, evitando assim que usuários mal intencionados derrubem o sistema fazendo um número muito grande de requisições acima do limite que o sistema consegue gerenciar.

5.1.2 Ambiente Perl

Com o propósito de resolver os principais problemas encontrados na arquitetura CGI padrão, utilizamos os módulos mod_perl e Apache::DBI no desenvolvimento da arquitetura GABD*Web*. Nessa arquitetura, o GABD*Web* foi implementado com a utilização dos módulos citados acima. Os *scripts* CGI são escritos em Perl e o servidor *Web* é o Apache com um interpretador Perl ligado a ele via módulo *mod_perl*. Não há necessidade de carregar o interpretador Perl para toda execução de um *script*, pois esse já está ligado ao código do servidor *Web* através do módulo *mod_perl*. Mais ainda, consegue-se aumento adicional de desempenho devido a um *cache* de *scripts* pré-compilados mantidos por *mod_perl*. Essas escolhas têm um grande impacto no desempenho global e conectividade do sistema. O desempenho pode ser aumentado consideravelmente pois pode-se minimizar ao máximo o tempo de processamento dos *scripts* CGI devido à combinação de características individuais.

Com relação à conectividade, pode-se usar como banco de dados qualquer banco suportado pela combinação de módulos DBI (*Data Base Interface*) e DBD (*Data Base Driver*) da biblioteca Perl, estando incluídos os bancos de dados Oracle, SQL Server, Informix e Sybase, dentre outros. Os *drivers* utilizados no acesso aos bancos de dados foram: DBD::Oracle para acessar o SGBD Oracle e DBD::Sybase para acessar o SGBD MS SQL Server, como o próprio nome diz, esse *driver* também pode ser usado para acessar o SGBD Sybase. O mecanismo de acesso ao banco de dados não impõe perda de desempenho ao sistema pois cada instância filha do servidor *Web* possui um *pool* de conexões persistentes com o banco de dados através do módulo Apache::DBI. A instância pai do servidor Apache passa a requisição a uma instância filha inativa. Se todas estiverem ocupadas servindo outras requisições então uma nova instância filha do servidor *Web* é disparada, desde que o número máximo de instâncias filhas não tenha sido excedido. A instância filha carrega e executa o *script* Perl, esse *script* acessa o GABD*Web* que coordena todas as atividades necessárias para responder à requisição do usuário.

A Figura 5.5 apresenta a arquitetura geral do GABD*Web* utilizando Perl:

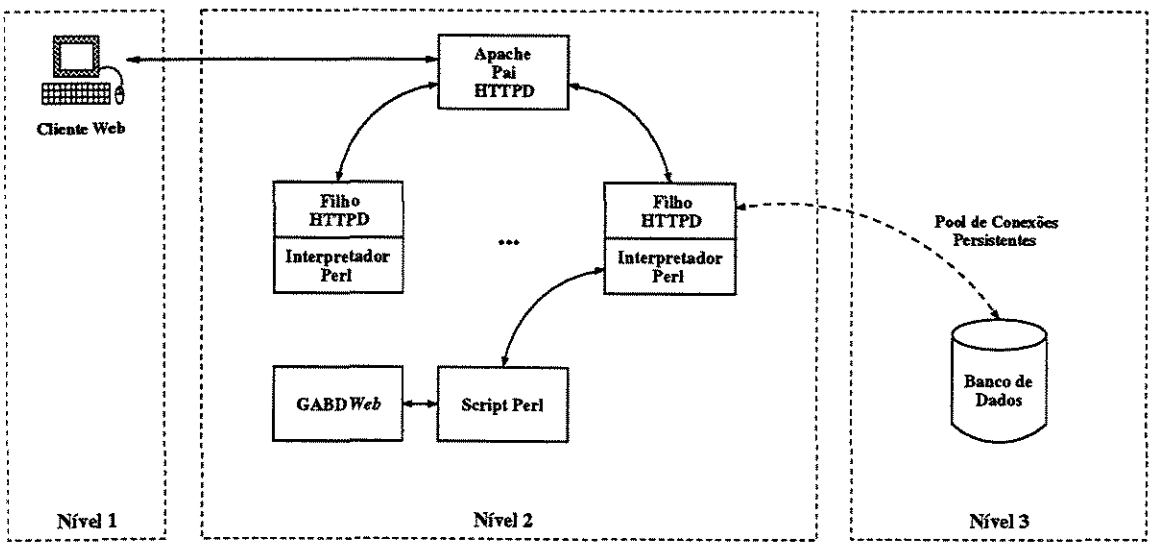


Figura 5.5: Arquitetura Geral do GABD*Web* Utilizando Perl.

5.1.3 Ambiente Python

A implementação do agente de banco de dados (GABD*Web*), em [Gui01], é proposta como a junção do "*Session Manager*" e do "*Database Application*" [HMP98], descritos na subseção 3.2, em um único processo de longa duração denominado DBCM (*Database Connection Manager*). Essa solução é composta de duas partes: um programa Python (*daemon*) que implementa o DBCM e um pequeno programa CGI escrito em C, que é executado em menos de 0.5 ms em uma instalação Linux moderna.

Nessa implementação é utilizada uma nova e eficiente técnica de comunicação de dados entre o servidor *Web* e o DBCM, onde o *script* CGI conecta-se ao DBCM, através de um *socket* Unix, e envia seus descritores (*input* e *output*). Essa implementação requer somente uma conexão *socket* para comunicação de dados entre o DBCM e o SGBD. Uma outra conexão *socket* entre o DBCM e o *script* CGI é usada somente para receber os descritores.

O principal benefício dessa técnica, em relação a [HMP98], é que além da recepção dos dois descritores de E/S do *script* CGI que são passados ao DBCM, nenhuma comunicação adicional é necessária entre eles: a entrada do usuário é lida diretamente do servidor *Web* pelo DBCM e a saída gerada é escrita diretamente para o servidor *Web* usando o descritor de saída recebido do *script* CGI. Dessa forma, é reduzido o número de conexões *socket* utilizadas na transmissão dos dados entre o servidor *Web* e o DBCM. Essa técnica tem o

potencial de reduzir o tempo de reposta de aplicações onde um grande volume de dados é retornado do SGBD para o cliente (*browser*).

O DBCM foi implementado como um servidor concorrente convencional, que dispara um processo filho para cada requisição de um *script* CGI. Esse mantém informações de estado para cada usuário ativo em uma tabela residente em memória denominada Tabela de Conexões, contendo: *username* e *password* criptografada, nome da instância do SGBD, *status* da conexão, *handle* da conexão, *timeout* da conexão e o número de vezes que a conexão foi usada. O *username* e *password* são aquelas necessárias para a autenticação no SGBD. Por razões de segurança e privacidade, a *password* não criptografada (no formato ASCII) é usada na abertura da conexão com o SGBD e depois descartada. O *status* de uma conexão está ativo se existe uma interação em andamento com o SGBD e inativo caso contrário. Se um usuário submeter uma consulta e existir uma conexão inativa atribuída a ele, essa conexão é reutilizada e, portanto, torna-se ativa. O *timeout* da conexão é usado como *timeout* da sessão de usuário: a conexão com o SGBD será fechada se o *timeout* expirar, então sua entrada será descartada da Tabela de Conexões. A conexão do DBCM com o SGBD Oracle é feita através do *driver* DCOracle da *Digital Creations Oracle Interface*, que foi instalado como um módulo Python.

A Figura 5.6 mostra os caminhos (*paths*) de comunicação entre o servidor *Web*, *scripts* CGI e DBCM:

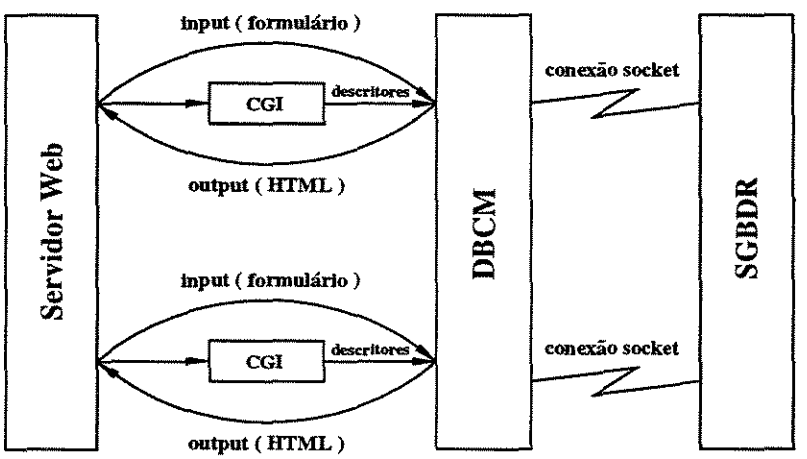


Figura 5.6: Caminhos de Comunicação no DBCM.

Capítulo 6

Análise de Desempenho

Neste capítulo, é apresentada uma série de experimentos que permitem a quantificação do *overhead* de tempo imposto pelas arquiteturas de *gateways* convencionais e os benefícios que podem ser obtidos pela arquitetura proposta nesse trabalho. Para analisar o desempenho da arquitetura, é preciso definir o significado de desempenho no contexto específico de plataforma cliente/servidor. Existem dois tipos de desempenho, um mais geral para qualquer paradigma de execução; e um mais específico, especialmente definido para modelos cliente/servidor:

- **Desempenho Absoluto**: mede o tempo que uma aplicação leva para executar. Essa definição é bem adaptada para programas sem retardo assíncrono (interação do usuário, congestionamento de rede, carga de máquina) e portanto é usada como um índice de velocidade nesse contexto. É fácil entender porque esse tipo de desempenho não pode ser usado para testar a velocidade do servidor;

- **Desempenho Percebido**: mede o tempo que um servidor leva para executar e terminar uma ação requisitada pelo cliente. Essa definição é bem adaptada ao paradigma cliente/servidor. O Desempenho e a velocidade do servidor são usados para definir esse tipo de medida.

Como em [HM97, HVM00, Zha99], foram realizados vários testes para demonstrar que a arquitetura proposta é significativamente mais eficiente que os *gateways* tradicionais. Das três técnicas de avaliação de desempenho (medição, simulação e modelagem analítica), medição é considerada a mais precisa [CHS93]. Esse tipo de teste somente é possível ser realizado depois da implementação do sistema considerado. Portanto, é possível realizar essas medidas desde que o GABD *Web* foi implementado.

Nos experimentos apresentados neste capítulo, utilizamos os SGBDs Oracle e MS SQL Server e não sistemas gerenciadores abertos como InterBase e PostgreSQL porque esses

ainda são pouco usados em aplicações corporativas, ao contrário do Oracle e MS SQL Server. Além disso, o SGBD InterBase não era um sistema aberto no momento do início dessa dissertação.

O tempo total (t_{total}) gasto pelos *gateways* tradicionais na recuperação e transmissão da informação solicitada de volta ao *browser* é dado por:

$$t_{total} = t_{processo} + t_{interpretador} + t_{script} + t_{conexao} + t_{aplicacao} + t_{SQL} \quad (6.1)$$

Na fórmula 6.1, os fatores que implicam diretamente no tempo total (t_{total}) de execução de um *script* CGI padrão são representados por:

1. $t_{processo}$: tempo gasto na criação de um novo processo CGI;
2. $t_{interpretador}$: tempo que o sistema gasta para carregar o interpretador do *script* na memória;
3. t_{script}: tempo gasto pelo sistema para carregar o código do *script* na memória;
4. $t_{conexao}$: tempo gasto na reserva de recursos e no estabelecimento de uma conexão com o banco de dados;
5. $t_{aplicacao}$: tempo de execução do código do *script* (esse tempo é diretamente proporcional à complexidade do algoritmo do *script*);
6. t_{SQL}: tempo de processamento gasto pelo SGBD na execução do comando SQL submetido pelo cliente.

Os quatro primeiros fatores são todos controláveis, enquanto que $t_{aplicacao}$ e t_{SQL} variam dependendo respectivamente do tamanho do código útil do *script* [Pre00] e do tipo de comando SQL submetido pelo cliente. O t_{SQL} depende também se o SGBD utilizado implementa ou não um sistema de *cache* para consultas SQL: na próxima vez em que um cliente executar um mesmo *script* CGI, cujo resultado de consulta já esteja na memória, esse tempo será muito menor que o tempo gasto na primeira execução do *script* CGI.

Ao contrário dos *gateways* tradicionais, onde todos os seis fatores são determinantes do tempo total de execução de um *script* CGI, nas arquiteturas Java, Perl e Python, vários desses fatores podem ser reduzidos ou até mesmo eliminados com a utilização do GABD*Web* (Java), módulos mod_perl e Apache::DBI (Perl) e DBCM (Python).

6.1 Configuração dos Experimentos

Foram utilizadas quatro máquinas (*hosts*) durante o processo de desenvolvimento e avaliação dos experimentos. As configurações de *hardware* e *software* utilizadas foram:

- **Host 1**: Pentium I 200 MHz com 64 MB de RAM. Essa máquina hospeda os seguintes *softwares*:
  1. Sistema operacional: Linux Red Hat 6.2 (kernel 2.2.14-5.0);
  2. Apache JMeter 1.3 (gerador de carga HTTP).
- **Host 2**: AMD-K6 350 MHz com 128 MB de RAM. Essa máquina hospeda os seguintes *softwares*:
  1. Sistema operacional: Linux Red Hat 6.2 (kernel 2.2.14-5.0);
  2. Servidor Apache 1.3.14.
- **Host 3**: AMD-K6 300 MHz com 64 MB de RAM. Essa máquina hospeda os seguintes *softwares*:
  1. Sistema operacional: Linux Red Hat 6.2 (kernel 2.2.14-5.0);
  2. Oracle 8.0.5.
- **Host 4**: Pentium I 166 MHz com 32 MB de RAM. Essa máquina hospeda os seguintes *softwares*:
  1. Sistema operacional: Microsoft Windows NT Server 4.0;
  2. Microsoft SQL Server 6.5.

A Figura 6.1 mostra a configuração dos cenários utilizados no desenvolvimento dos experimentos:

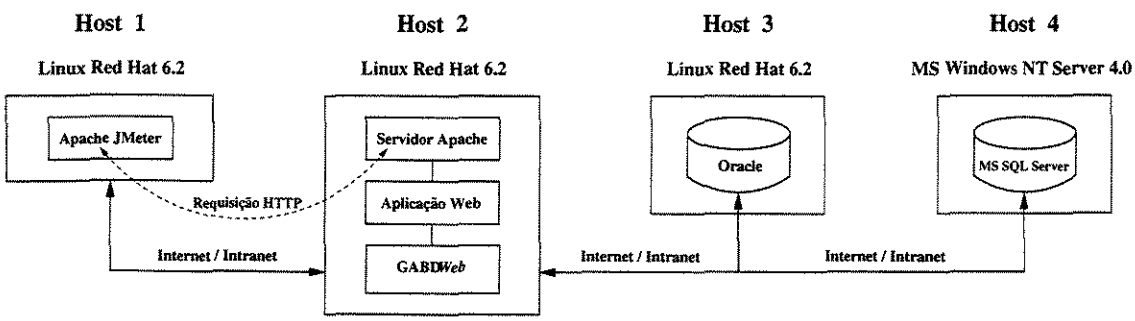


Figura 6.1: Configuração dos Cenários dos Experimentos.

Os experimentos consistem de uma série de testes em que um programa gerador de carga (Apache JMeter) direciona uma grande quantidade de requisições HTTP ao servidor Apache. Na subseção 6.1.1, o Apache JMeter é apresentado em mais detalhes.

6.1.1 Apache JMeter

O Apache JMeter é uma ferramenta de código aberto que faz parte do Projeto Java Apache [Pro]. O Apache JMeter é uma aplicação, escrita em Java, projetada para testar o comportamento e medir o desempenho de servidores *WWW* na recuperação de documentos HTML estáticos ou gerados dinamicamente (CGI, Servlets, Perl, Python, etc.). O Apache JMeter também pode ser usado para simular um período de alta carga sobre o servidor *WWW* ou sobre a rede para testar a sua solidez ou para analisar o seu desempenho sob diferentes tipos de carga.

O Apache JMeter pode ser usado para fazer uma análise gráfica do servidor *WWW* ou para testar o comportamento de um *script*/servidor sob uma grande quantidade de requisições concorrentes.

As características do Apache JMeter incluem:

- É completamente portável, pois é 100% escrito em Java;
- Permite o uso de ambos métodos GET e POST;
- Devido à característica *multi-threaded* da linguagem Java, o Apache JMeter possui um *framework multi-threaded* que permite requisições HTTP concorrentes realizadas por vários *threads*, onde cada *thread* simula o tráfego causado por um cliente HTTP;
- Vários tipos de visualização das estatísticas de carga podem ser escolhidas para uma melhor análise, incluindo a possibilidade de salvar essas estatísticas em um arquivo.

A Figura 6.2 mostra a ferramenta Apache JMeter utilizada na geração de carga HTTP para a realização dos experimentos:

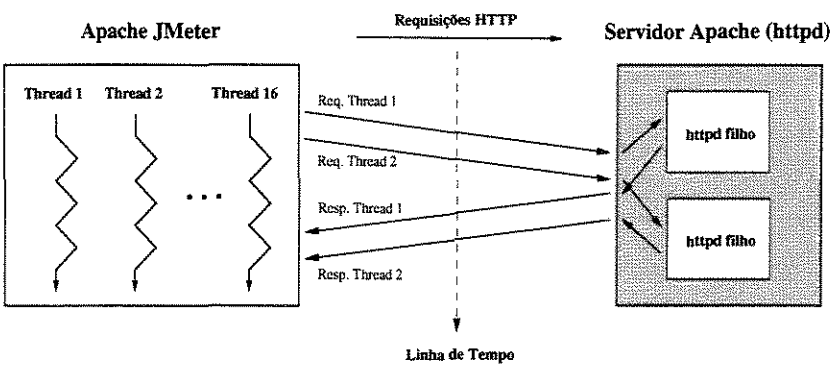


Figura 6.2: Utilização do Apache JMeter na Geração de Carga HTTP.

Como pode ser visto na Figura 6.2, a linha de tempo mostra que o número máximo de requisições concorrentes que podem ser feitas ao servidor Apache pelo Apache JMeter depende exclusivamente do número de *threads* utilizados no experimento.

Da forma que o Apache JMeter foi implementado não era possível realizar as medidas para os testes propostos. O *software* simplesmente envia requisições HTTP a partir de cada *thread*, sem controlar quantas e quais requisições foram realizadas por cada *thread*. Por isso, foi necessário reimplementar algumas das funcionalidades do *software* para que ele fizesse o controle de todas as variáveis necessárias para a sincronização de cada *thread* com o *script* CGI. Para resolver esse problema é necessário que cada *thread* envie um *cookie*, contendo o número do *thread*, para o *script* CGI. O valor do *cookie* é utilizado somente na amostragem do tempo de conexão, já que para o tempo de resposta não é necessário essa sincronização.

O fato do Apache JMeter ser uma ferramenta de código aberto facilitou muito o processo de amostragem, pois não houve a necessidade de implementar totalmente um ferramenta que se adequasse ao experimento, e sim apenas alterar uma parte do código de um *software* que já vem sendo usado e aprovado por muitos usuários.

6.2 Cenários dos Experimentos

O Apache JMeter foi configurado para simular o tráfego causado por até 16 clientes HTTP (*threads*) concorrentes, começando por um *thread* e incrementando de 1 até 16 *threads* concorrentes. Cada *thread* faz 10 repetições de uma mesma requisição, permitindo portanto que o experimento alcance um estado estável. Durante os testes, arquivos de *log* de atividades foram atualizados com as seguintes informações:

- **Tempo de Conexão**: tempo gasto na alocação de recursos e no estabelecimento de uma conexão com o banco de dados;
- **Tempo de Resposta**: tempo total gasto para completar a transferência dos dados requisitados pelo cliente (*browser*), incluindo o tempo de conexão com o servidor *Web*.

A Figura 6.3 mostra os cenários de acesso a banco de dados sujeitos à avaliação através dos experimentos. Os cenários avaliados foram:

1. *Script* CGI/Java padrão: *script* e interpretador Java são carregados a cada requisição;
2. *Script* Java Servlet: *script* e interpretador Java são pré-carregados;

3. *Script* Java Servlet com *pool* de conexões (Gerenciador de Conexões - GABD*Web*);
4. *Script* CGI/Perl padrão: *script* e interpretador Perl são carregados a cada requisição;
5. *Script* CGI/Perl com módulo mod_perl: *script* e interpretador Perl são pré-carregados;
6. *Script* CGI/Perl com módulos mod_perl e *pool* de conexões (Apache::DBI);
7. *Script* CGI/Python padrão: *script* e interpretador Python são carregados a cada requisição;
8. *Script* CGI/C com DBCM/Python: *script* e interpretador Python são pré-carregados;
9. *Script* CGI/C com DBCM/Python e *pool* de conexões (Gerenciador de Conexões).

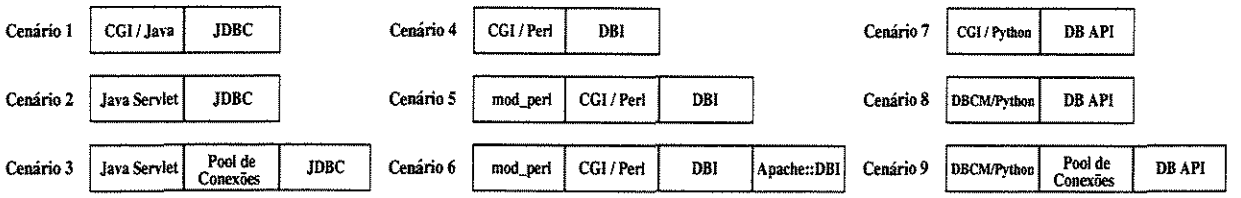


Figura 6.3: Cenários dos Experimentos Avaliados.

Todos os testes foram realizados duas vezes, uma vez para cada um dos SGBDs que estão sendo avaliados (Oracle e MS SQL Server), com exceção dos cenários 7, 8 e 9 onde foram realizados testes somente com o SGBD Oracle. Em todos os casos, o acesso aos bancos de dados envolve a execução de uma consulta SQL simples sobre uma tabela com poucos registros: o tamanho do documento HTML gerado é de 247 *bytes*. O desempenho de todos os cenários com respeito às métricas consideradas é ilustrado na seção 6.3.

6.3 Resultados Experimentais

6.3.1 Avaliação do Tempo de Conexão

Ambiente Java

A Figura 6.4(a) mostra que o tempo de abertura de uma conexão ($t_{conexao}$), no cenário 3, foi totalmente eliminado devido à utilização do *pool* de conexões, mostrando claramente uma grande melhora de desempenho em relação à arquitetura do cenário 2. Também pode ser verificado que com um *thread* o tempo médio de abertura de uma conexão no Oracle é aproximadamente 80% maior que o tempo médio de abertura de uma conexão no MS SQL Server; para os testes com cinco ou mais *threads* os tempos se tornam praticamente iguais. O tempo de conexão no cenário 3 é sempre zero porque existem tantas conexões abertas quanto requisições HTTP que chegam ao Gerenciador de Conexões (GABD*Web*). Caso existam mais requisições HTTP concorrentes do que conexões abertas, haverá um *overhead* inicial associado à abertura da conexão, mas nas próximas requisições esse custo não ocorrerá mais.

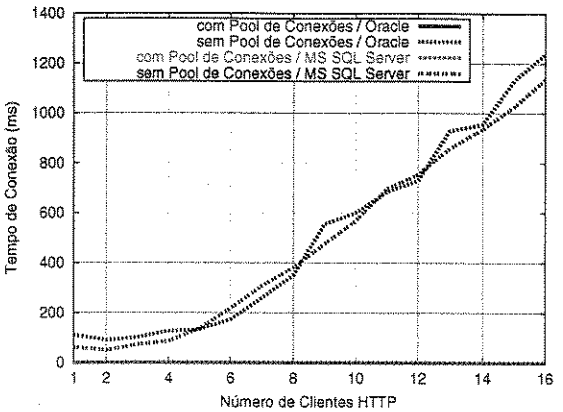
Os tempos de conexões para o cenário 1 não estão presentes no gráfico da Figura 6.4(a) porque eles são iguais aos tempos de conexão do cenário 2. Isso ocorre porque o fato de um pedido de conexão ser feito por um *script* Java padrão ou por um Servlet não influencia no tempo de abertura dessa conexão já que todos os recursos devem ser alocados da mesma forma (o mesmo ocorre nos ambientes Perl e Python).

Ambiente Perl

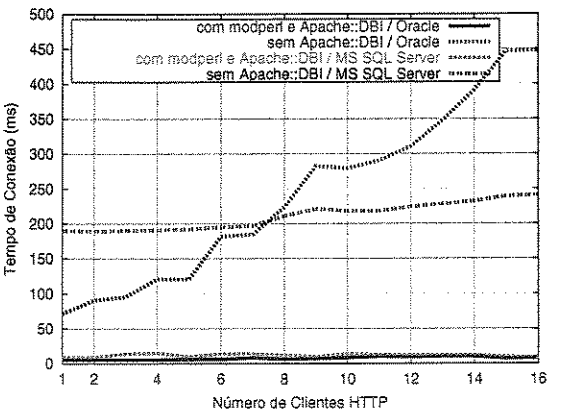
Como mostrado na Figura 6.4(b), o tempo de conexão foi quase totalmente eliminado devido à utilização do módulo Apache::DBI, reduzindo os tempos de conexão em torno de 10 ms. Os tempos de conexão no Oracle para testes realizados até 7 *threads* são muito menores que com MS SQL Server, a partir desse ponto os tempos para Oracle crescem quase linearmente e os tempos para MS SQL Server permanecem quase constantes. Isto é, com baixa carga os tempos de conexão no Oracle são melhores, já em períodos de alta carga o MS SQL Server se comporta de forma mais eficiente. O tempo de abertura de uma única conexão no MS SQL Server é aproximadamente 150% maior que o tempo de conexão no Oracle.

Ambiente Python

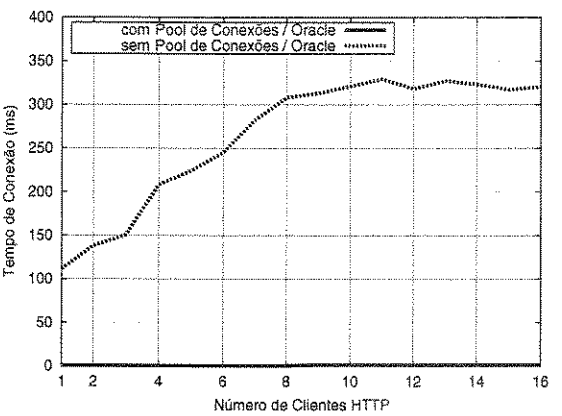
A Figura 6.4(c) mostra a eliminação do tempo de conexão no cenário 9 com a utilização do *pool* de conexões do DBCM.



(a) Cenários 2 e 3 - Java



(b) Cenários 5 e 6 - Perl



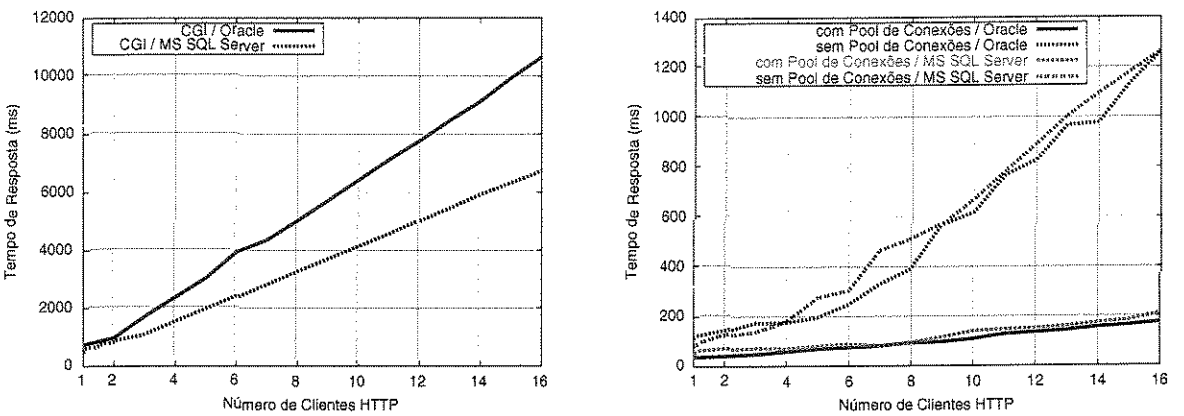
(c) Cenários 8 e 9 - Python

Figura 6.4: Tempo de Conexão x Número de Clientes.

6.3.2 Avaliação do Tempo de Resposta

Ambiente Java

Nos cenários 2 e 3, o tempo de resposta (t_{total}) é reduzido drasticamente em relação ao cenário 1 devido a três motivos. Primeiro, o $t_{processo}$ é significativamente menor que no cenário 1 (*script* CGI padrão) devido a não ser necessário criar novos processos a cada requisição HTTP que é feita à Máquina Servlet. Segundo, sempre que chegar uma requisição, a Máquina Servlet (Apache JServ) já possui uma Máquina Virtual Java na memória, eliminando com isso o fator $t_{interpretador}$. Por último, após a primeira requisição feita ao servlet o seu código executável sempre estará na memória, eliminando dessa forma o t_{script}.



(a) Cenário 1

(b) Cenários 2 e 3

Figura 6.5: Tempo de Resposta x Número de Clientes (Java).

Analisando os gráficos das Figuras 6.4(a) e 6.5(b), podemos verificar que a grande redução no tempo de resposta no cenário 3 ocorre exclusivamente devido à eliminação do fator $t_{conexao}$ em relação ao cenário 2. O fator $t_{conexao}$ é muito próximo do valor do tempo de resposta total:

$$t_{cenario\_3} = t_{cenario\_2} - t_{conexao} \tag{6.2}$$

Uma vez que o $t_{conexao}$ é o principal contribuinte para o tempo total de execução do *script*, pela Fórmula 6.2, podemos verificar que o tempo de resposta do cenário 3 será pequeno em relação ao cenário 2, o que está de acordo com o gráfico obtido na Figura 6.5(b).

Devido à eliminação dos fatores: $t_{processo}$, $t_{interpretador}$ e t_{script}; podemos verificar através da Figura 6.5 que o tempo de resposta de uma requisição HTTP é muito menor nos cenários 2 e 3 do que no cenário 1. Também fica claro que a arquitetura do cenário 3 executa sistematicamente mais rápido que as arquiteturas dos cenários 1 e 2, aproximadamente 6 vezes mais rápido que a arquitetura Servlet padrão (cenário 2) e 55 vezes mais rápido que a arquitetura CGI/Java padrão (cenário 1) com Oracle e 35 vezes mais rápido que a arquitetura CGI/Java padrão com MS SQL Server.

De acordo com a Figura 6.5(a), o tempo de resposta de um *script* CGI/Java padrão acessando o Oracle é bem maior que o tempo de resposta para o mesmo *script* acessando o MS SQL Server. Isso ocorre porque o tempo de abertura de uma nova conexão no Oracle gasta muito mais recursos (memória) no *host* 2 do que o MS SQL Server e consequentemente mais tempo. Mesmo os SGBDs estando instalados em máquinas diferentes de onde o *script* é executado, o consumo de recursos nessa máquina é muito maior devido as chamadas de rotinas que o *driver* faz à biblioteca OCI (*Oracle Call Interface*), essas chamadas são então enviadas sobre o protocolo Net8 para o SGBD Oracle (o mesmo ocorre no ambiente Perl).

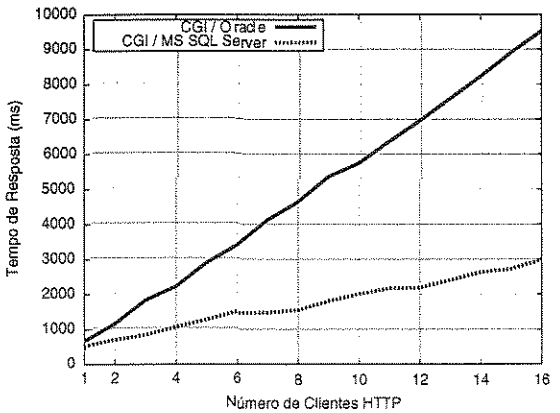
Ambiente Perl

A Figura 6.6 mostra a grande redução no tempo de resposta do cenário 6 em relação aos cenários 4 e 5; a melhora de desempenho é aproximadamente 7,5 vezes em relação ao cenário 5 com Oracle e 4 vezes com MS SQL Server, em relação ao cenário 4 o tempo de resposta do cenário 6 é aproximadamente 120 vezes mais rápido com Oracle e 37,5 vezes mais rápido com MS SQL Server.

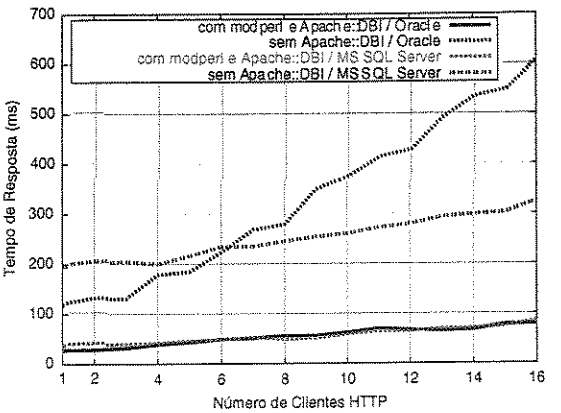
Podemos verificar, na Figura 6.6(b), que o tempo de resposta no cenário 6 com Oracle e MS SQL Server são praticamente iguais, isso ocorre devido aos *pools* de conexões utilizados nos dois casos. No cenário 5 o tempo de resposta com Oracle é menor em relação ao MS SQL Server até os testes realizados com 6 *threads*, a partir desse ponto o tempo de resposta com MS SQL Server se torna bem menor em relação ao Oracle.

Ambiente Python

A Figura 6.7 mostra o efeito da utilização do DBCM em aplicações de banco de dados na *Web*, o tempo de resposta do cenário 9 é bastante reduzido. O cenário 9 executa a mesma requisição aproximadamente 3,5 vezes mais rápido que o cenário 8 e aproximadamente 100 vezes mais rápido que o cenário 7 para o SGBD Oracle.

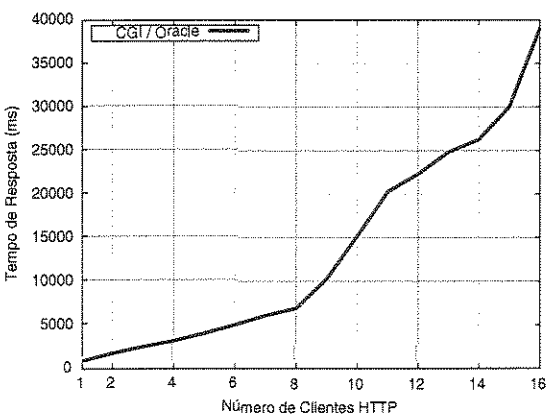


(a) Cenário 4

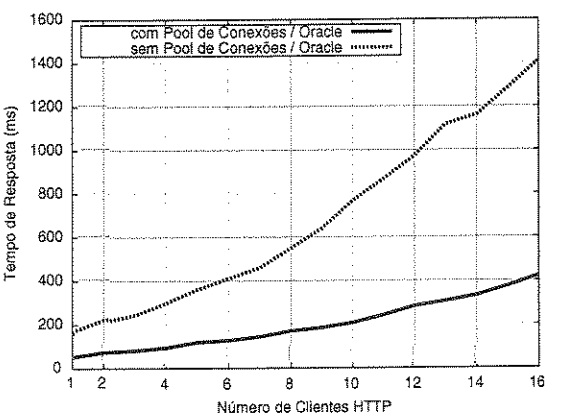


(b) Cenários 5 e 6

Figura 6.6: Tempo de Resposta x Número de Clientes (Perl).



(a) Cenário 7



(b) Cenários 8 e 9

Figura 6.7: Tempo de Resposta x Número de Clientes (Python).

6.3.3 Avaliação do *Throughput*

O *Throughput*, em nosso contexto, representa o tempo total gasto na execução de 16 requisições concorrentes de um mesmo *script*.

Consulta Pequena - 2 Registros

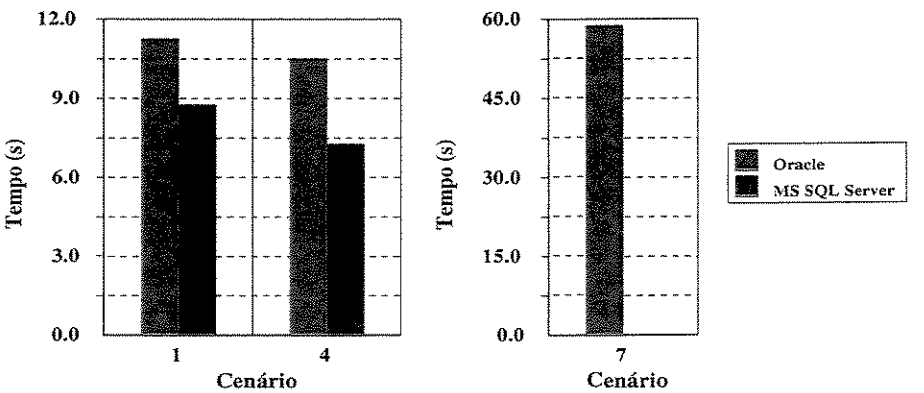
A Figura 6.8 mostra os tempos de *throughput* para todos os cenários do experimento. Podemos ver através da Figura 6.8(a) que o *throughput* do cenário 4 é melhor que nos cenários 1 e 7, isso ocorre devido às arquiteturas Java e Python consumirem mais recursos na execução de um CGI padrão que a arquitetura Perl. O tamanho de um interpretador Java que deve ser carregado na memória toda vez que chega uma requisição é muito maior que o interpretador Perl, com isso é gasto mais tempo e o *throughput* aumenta. Nos cenários 1 e 4 o *throughput* para o MS SQL Server é melhor pois o processo de abertura de uma nova conexão no Oracle consome mais recursos que no MS SQL Server, como nesses cenários já existe um grande consumo de recursos, esse fator torna o processo mais lento e em conseqüência o *throughput* com o MS SQL Server é melhor.

Na Figura 6.8(b), podemos ver que os cenários 3, 6 e 9 possuem *throughputs* melhores que os outros cenários, isso ocorre pois não existem custos associados à abertura de novas conexões. O melhor desempenho é do cenário 6 (Perl), em seguida o cenário 3 (Java) e por último o cenário 9 (Python). Em todos os cenários da Figura 6.8(b) o *throughput* com Oracle é melhor, isso ocorre porque apesar do Oracle consumir mais recursos para cada conexão ele consegue trabalhar com requisições concorrentes de forma mais eficiente que o MS SQL Server.

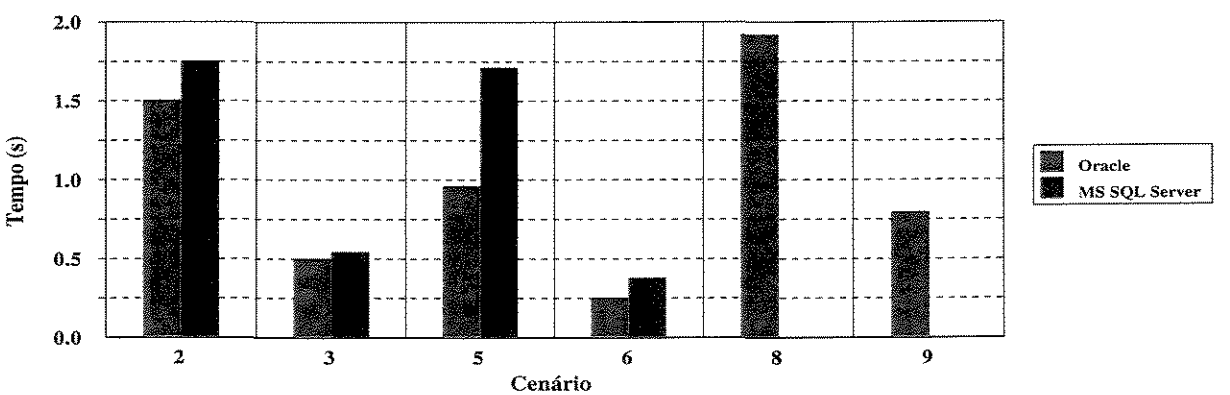
Consulta Grande - 3618 Registros

A Figura 6.9 mostra o *throughtput* para os cenários 3, 6 e 9. Nesses casos, o acesso aos bancos de dados envolve a execução de uma consulta SQL sobre uma tabela com muitos registros: o tamanho do documento HTML gerado é de 188k *bytes*.

Na Figura 6.9(a) podemos ver que o melhor desempenho é do cenário 6 (Perl), em seguida o cenário 3 (Java) e por último o cenário 9 (Python). No cenários 6 o *throughput* com o MS SQL Server é melhor, isso ocorre porque apesar do Oracle trabalhar de forma mais eficiente com conexões concorrentes, existe um *overhead* adicional associado ao aumento de registros na tabela de consulta. Dessa forma, como o Oracle consome mais recursos por conexão, o *throughput* para o MS SQL Server é melhor que para o SGBD Oracle.

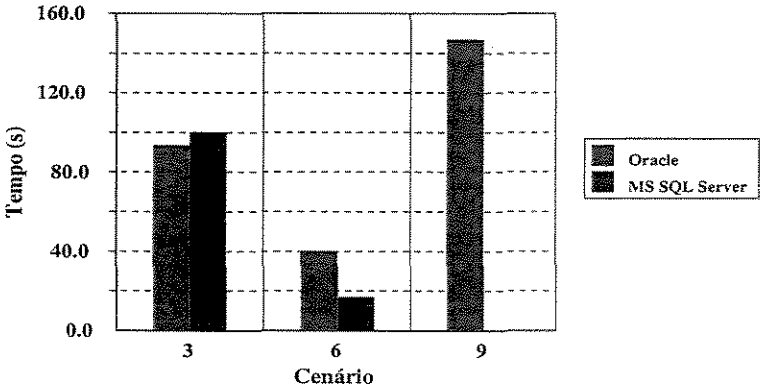


(a) Cenários 1 (Java), 4 (Perl) e 7 (Python)



(b) Cenários 2 e 3 (Java), 5 e 6 (Perl), 8 e 9 (Python)

Figura 6.8: Comparação do *Throughput* para Pequenas Consultas.



(a) Cenários 3 (Java), 6 (Perl) e 9 (Python)

Figura 6.9: Comparação do *Throughput* para Grandes Consultas.

6.4 Tempo Total dos Cenários

A tabela 6.1 mostra os fatores envolvidos no t_{total} de cada cenário:

Cenário	Ambiente Java			Ambiente Perl			Ambiente Python		
Tempo	Cenário 1	Cenário 2	Cenário 3	Cenário 4	Cenário 5	Cenário 6	Cenário 7	Cenário 8	Cenário 9
$t_{processo}$	●			●			●		
$t_{interpretador}$	●			●			●		
t_{script}	●			●			●		
$t_{conexao}$	●	●		●	●		●	●	
$t_{aplicacao}$	●	●	●	●	●	●	●	●	●
t_{SQL}	●	●	●	●	●	●	●	●	●

Tabela 6.1: Fatores Envolvidos no Tempo Total de Cada Cenário.

Os tempos envolvidos nos cenários 3, 6 e 9 são: $t_{aplicacao}$ e t_{SQL}. Esses fatores não podem ser eliminados do (t_{total}) desses cenários. O $t_{aplicacao}$ depende da complexidade do código da aplicação e da capacidade de processamento do sistema que executa a aplicação. O t_{SQL} depende da complexidade do comando SQL, da capacidade de processamento das consultas pelo SGBD, do *overhead* de tempo associado ao *driver* (esse tempo pode ser significativo em consultas pequenas) e do sistema de *cache* do SGBD. Em todos os nossos experimentos foram utilizados os sistemas de *cache* do Oracle e do MS SQL Server.

Capítulo 7

Conclusão

7.1 Contribuições

Nessa dissertação foi proposta uma arquitetura "*three-tier*", denominada GABD*Web*, para utilização no desenvolvimento de aplicações de bancos de dados com estado no contexto da *Web*; essa arquitetura é totalmente baseada em ferramentas de código aberto. Com isso, pode-se criar aplicações para a *Web* com um baixo custo de desenvolvimento. A motivação para esse projeto foi realizar a preservação de estado, a redução da taxa de estabelecimento de novas conexões com os SGBDs e o controle sobre o acesso de usuários às aplicações de bancos de dados na *Web*.

GABD*Web* fornece capacidade ao sistema de realizar a preservação de estado, através do Gerenciador de Sessões, entre consecutivos acessos de um mesmo usuário a uma aplicação de banco de dados. Nessa arquitetura, GABD*Web*, através do Gerenciador de Conexões, permanece com as conexões abertas, permitindo suas reutilizações entre múltiplas requisições HTTP, resolvendo portanto o problema de abertura de uma nova conexão para toda requisição HTTP de um mesmo usuário.

GABD*Web* pode ser utilizado para facilitar a programação da interface de sistemas *Web* e resolver alguns problemas de projeto associados a essa idéia. Dentre as vantagens de utilização do GABD*Web* podemos citar:

- Tratamento do controle de diálogo através do gerenciador de navegação, evitando que o usuário siga um caminho de navegação entre páginas proibido na especificação do projeto;
- Permite que exista uma especialização dos diálogos segundo o tipo de usuário;
- Trata a característica sem estado (*stateless*) do protocolo HTTP através da definição de sessão para a aplicação;

- Melhora o desempenho do sistema reduzindo a taxa de estabelecimento de novas conexões com o banco de dados.

Também foram descritas algumas APIs de código aberto (Java Servlet, Perl e Python), seus benefícios no desenvolvimento de aplicações de bancos de dados no contexto da *Web* e realizada uma avaliação comparativa do desempenho dessas APIs através de experimentos. Vários testes foram realizados, a fim de quantificar os benefícios da utilização dessa técnica.

Verificamos que nos testes realizados a implementação no ambiente Perl teve um melhor tempo de resposta em relação a Java e Python, e Java teve um tempo de resposta melhor que Python. Verificamos também que o Oracle, no ambiente Java, gasta mais tempo que o MS SQL Server no processo de abertura de uma conexão. Isso ocorre devido à grande quantidade de recursos que devem ser alocados pelo SGBD. No ambiente Perl, em períodos de baixa taxa de requisições concorrentes, o Oracle se comporta melhor, pois consegue servir os pedidos de conexões de forma mais eficiente. É possível, no entanto, que essas medidas sejam influenciadas pelos respectivos *drivers*, e não reflitam a eficiência intrínseca dos SGBDs ou dos ambientes de programação utilizados.

7.2 Trabalhos Futuros

Diversas direções podem ser seguidas para dar continuidade a essa dissertação. Dentre os trabalhos futuros os que mais se destacam são:

- Realização de testes mais exaustivos com um grande número de requisições HTTP concorrentes;
- Utilização de *drivers* mais confiáveis e possivelmente mais eficientes nos testes com Perl e Python. Dessa forma, seria minimizada a interferência dos *drivers* nos resultados dos experimentos;
- Realização de testes com Python acessando o SGBD MS SQL Server. Essas medidas não foram realizadas devido à não disponibilidade de um *driver* para acessar o SGBD;
- Realização de testes com os SGBDs abertos PostgreSQL e InterBase;
- GABD*Web* pode ser estendido para fazer o gerenciamento de transações no contexto da *Web*. O conceito de transação distribuída no ambiente *Web* tem se tornado cada vez mais comum com o grande crescimento da área de comércio eletrônico.

Apêndice A

Agência de Viagens *Travel*

Este apêndice mostra como aplicar os conceitos descritos no capítulo 5 no desenvolvimento de aplicações de banco de dados com estado baseadas na *Web*. A validação das técnicas foi feita através da implementação de uma aplicação real: um sistema simplificado de gerenciamento de reservas de passagens aéreas em uma agência de viagens, denominado agência de viagens *Travel*. Uma implementação desse sistema interativo via *Web* foi dividida em dois módulos:

- **Módulo GABD** *Web*: é o gerenciador da aplicação *Travel*. Esse módulo controla as requisições do usuário, validando as permissões de acesso e navegações entre páginas da aplicação;
- **Módulo** *Travel*: é definido levando-se em consideração as tarefas que fazem parte da solução da aplicação.

A.1 Módulo GABD *Web*

Entre as atividades realizadas pelo gerenciador de aplicação, GABD *Web*, podemos citar:

- Controle do diálogo entre páginas, baseado no grafo de navegação, também chamado de grafo de diálogo entre páginas;
- Recebe todas as mensagens enviadas pelo *browser*. Essas mensagens são requisições de transição entre páginas HTML;
- Simula a conexão (estado) entre a aplicação e o *browser*. Cada sessão de usuário ativa tem um *token* associado, mantido pelo gerenciador de sessão, identificando essa sessão para a aplicação. O gerenciador de aplicação, através do gerenciador de sessão, é também responsável pelo mecanismo de expiração de *tokens*;

- Coordena todos os *pools* de conexões compartilhadas. Faz a interface entre a aplicação e os SGBDs, abrindo e fechando conexões quando necessário.

A fim de realizar todas as atividades descritas acima, o GABD*Web* precisa manter informações persistentes no lado do servidor. O GABD*Web* utiliza um pequeno banco de dados, composto de tabelas relacionais, onde são armazenadas informações vitais para o gerenciamento da aplicação *Travel*. A Figura A.1 mostra o diagrama entidade relacionamento mantido pelo GABD*Web*:

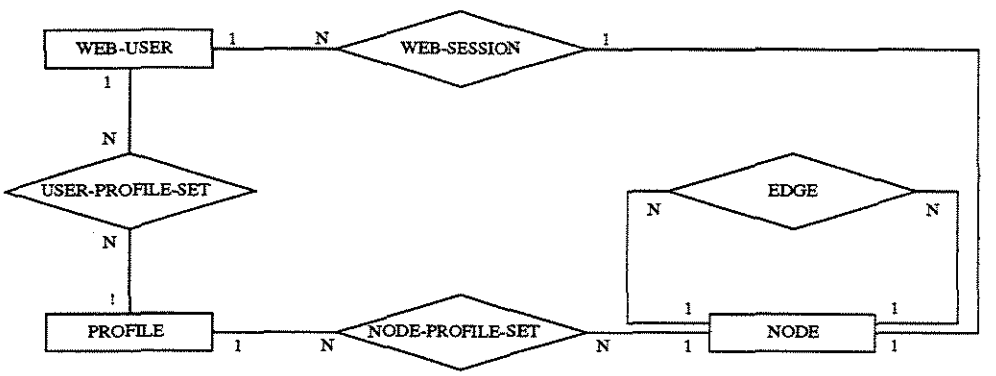


Figura A.1: Modelo Entidade Relacionamento do GABD*Web*.

O diagrama entidade relacionamento do GABD*Web* é composto por:

- PROFILE: define os perfis reconhecidos pela aplicação;
- WEB-USER: identifica os usuários e suas respectivas senhas;
- USER-PROFILE-SET: conjunto de perfis de cada usuário;
- NODE: conjunto dos nós do grafo de navegação;
- NODE-PROFILE-SET: conjunto dos perfis associados a cada nó do grafo de navegação;
- EDGE: conjunto das arestas do grafo de navegação;
- WEB-SESSION: conjunto de sessões válidas em um dado instante.

O esquema do banco de dados mantido pelo GABD*Web* é definido a seguir:

- PROFILE (PROFILE_ID, DESCRIPTION)
- WEB_USER (USER_ID, USERNAME, PASSWORD, PEOPLE_ID)
- USER_PROFILE_SET (USER_ID, PROFILE_ID)
- NODE (NODE_ID, DESCRIPTION)
- NODE_PROFILE_SET (NODE_ID, PROFILE_ID)
- EDGE (SOURCE, DESTINATION)
- WEB_SESSION (SESSION_ID, USER_ID, NODE_ID)

A Figura A.2 mostra o grafo de navegação da aplicação *Travel*. O grafo de navegação é totalmente baseado nas funcionalidades da aplicação.

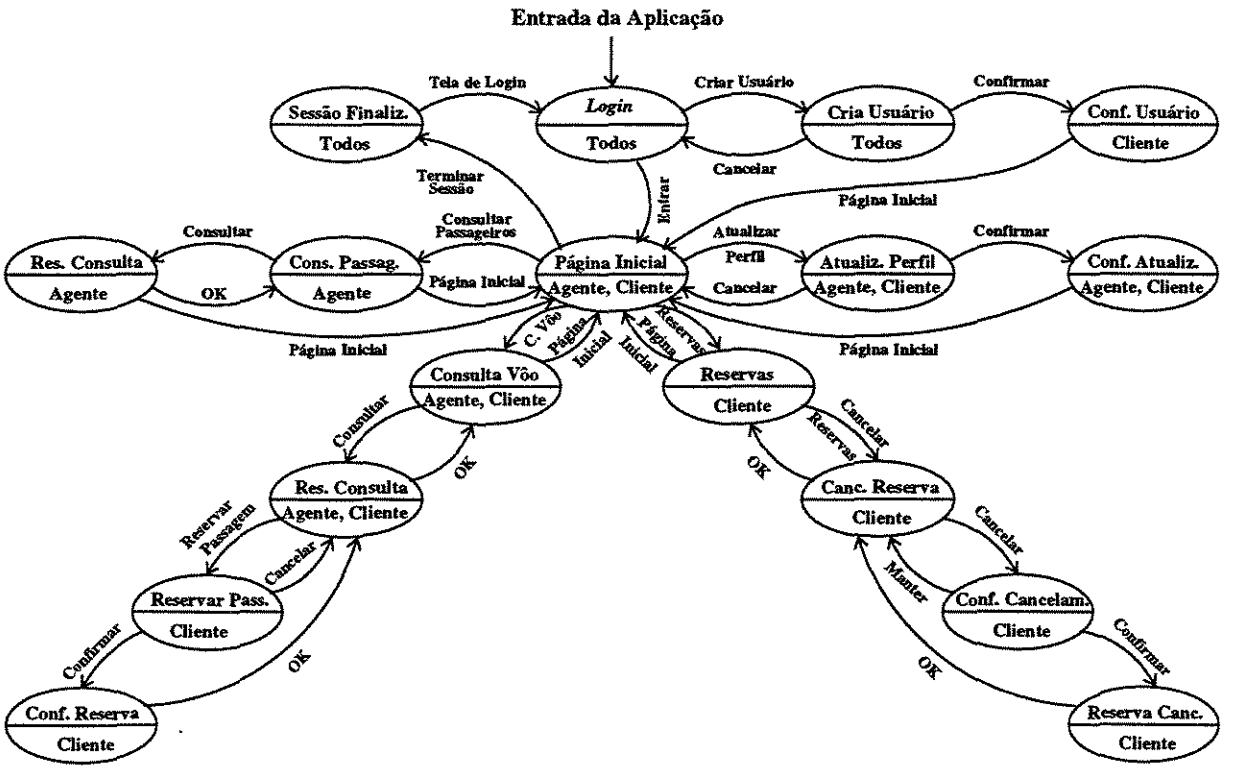


Figura A.2: Grafo de Navegação da Aplicação *Travel*.

A.2 Módulo *Travel*

A agência de viagens *Travel* deseja disponibilizar para seus clientes um sistema com interface *Web* que permita a consulta de horários de vôos, reserva de passagens aéreas, o cancelamento e consulta de reservas.

Embora a aplicação *Travel* tenha muitos usuários definidos na tabela *WEB-USER*, o GABD *Web* se conecta ao banco de dados através de um mesmo usuário do banco de dados. Dessa forma, os usuários da aplicação compartilharão as mesmas conexões com o banco de dados, aumentando a escalabilidade da aplicação e diminuindo a quantidade de licenças do SGBD utilizado.

A aplicação *Travel* necessita de um banco de dados para armazenar as informações referente às reservas de passagens aéreas. A Figura A.3 mostra o diagrama entidade relacionamento mantido pela aplicação *Travel*:

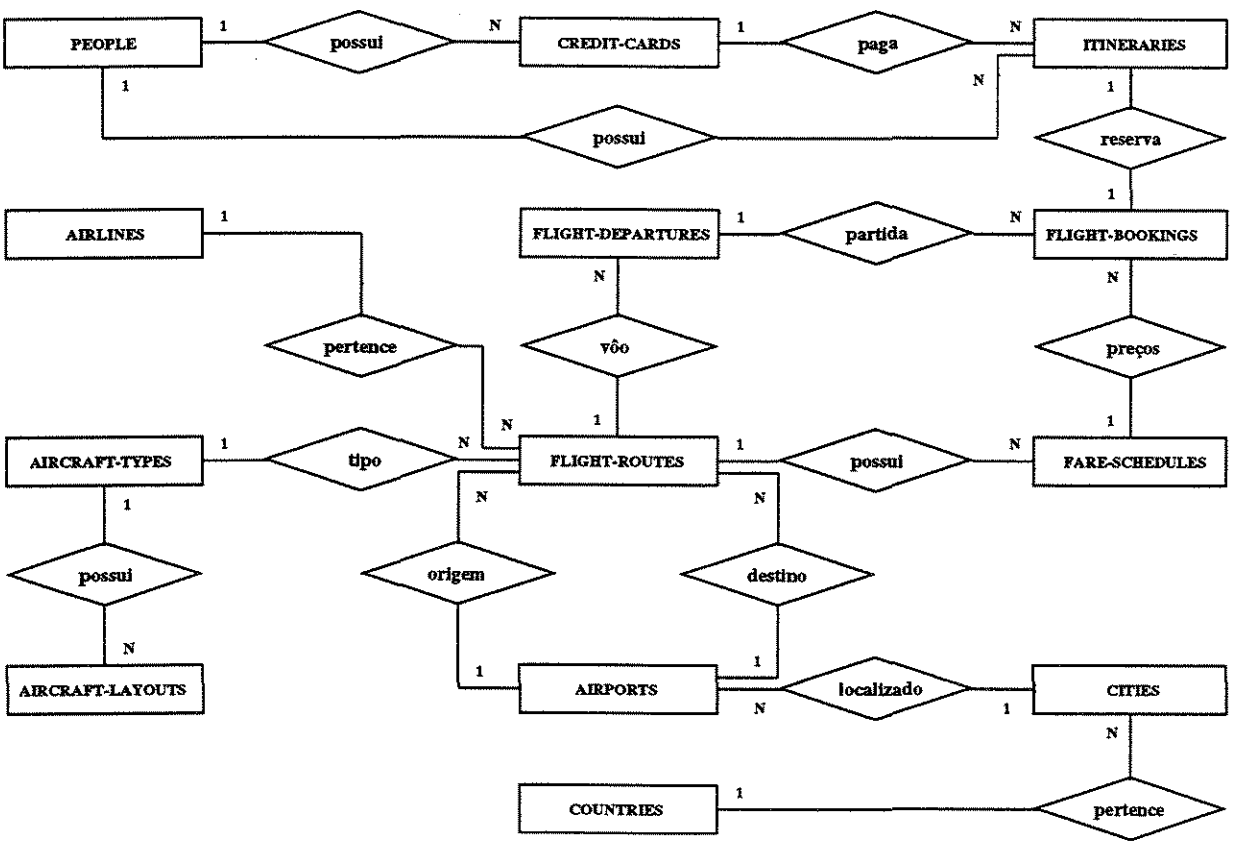


Figura A.3: Modelo Entidade Relacionamento da Aplicação *Travel*.

O diagrama entidade relacionamento da aplicação *Travel* é composto por:

- COUNTRIES: conjunto de países que são atendidos por uma rota aérea;
- CITIES: conjunto de cidades que fazem parte de uma rota aérea;
- AIRPORTS: conjunto de aeroportos que estão em uma rota aérea;
- AIRLINES: conjunto de linhas aéreas;
- FLIGHT_ROUTES: conjunto de rotas aéreas;
- FLIGHT_DEPARTURES: conjunto de vôos pertencentes a uma rota aérea;
- FLIGHT_BOOKINGS: reservas de passagens aéreas;
- AIRCRAFT_TYPES: tipos de aeronaves;
- AIRCRAFT_LAYOUTS: disposição dos assentos nas aeronaves;
- PEOPLE: usuários cadastrados no sistema;
- ITINERARIES: itinerários dos passageiros;
- FARE_SCHEDULES: preços das passagens aéreas;
- CREDIT_CARDS: cartões de crédito dos usuários cadastrados no sistema.

O esquema do banco de dados mantido pela aplicação *Travel* é definido a seguir:

- COUNTRIES (ID, NAME, GEOGRAPHY, CURRENCY, FLAG_PHOTO_FILENAME, CURRENCY_DESCRIPTION)
- CITIES (ID, NAME, STATE_PROVINCE, TIME_ZONE_CITY, TIME_ZONE_TO_GMT, CITY_MAP_LEVEL, WORLD_MAP_X, WORLD_MAP_Y, CON_ID, CITY_CODE)
- AIRPORTS (AIRPORT_CODE, CTY_ID, DESCRIPTION)
- AIRLINES (CODE, NAME, PARTNER)
- FLIGHT_ROUTES (ROUTE_ID, AIR_CODE, FLIGHT_NUMBER, ORIGIN_ARP_CODE, DEST_ARP_CODE, FLIGHT_CATEGORY, AVAILABLE_DAYS, MEAL, MOVIE, NUMBER_OF_STOPS, FLIGHT_MILES, ACT_CLASS)
- FLIGHT_DEPARTURES (ID, FLR_ID, DEPARTURE_DATE, DEPARTURE_TIME, ARRIVAL_TIME, FLIGHT_STATUS, FLIGHT_NOTES)

- FLIGHT_BOOKINGS (ID, PRICE_PAID, ASSIGNED_SEAT, ITN_ID, FLD_ID, FAS_ID)
- AIRCRAFT_TYPES (AIRCRAFT_CLASSIFICATION, MANUFACTURER, DESCRIPTION, PHOTO_FILENAME)
- AIRCRAFT_LAYOUTS (ACT_CLASS, SEAT_ROW, SEAT_LETTER, Y_POSITION, X_POSITION, SEAT_CLASS, SMOKING, EXTRA_LEG, ROOM, SEAT_WIDTH, BABY_FRIENDLY, EXIT_ROW, AISLE_SEAT)
- PEOPLE (ID, DESIGNATION, FIRST_NAME, LAST_NAME, JOB_TITLE, LOB, EMAIL, CONTACT_PHONE, HOME_PHONE, MOBILE_PHONE, ADDRESS, START_DATE, TERMINATION_DATE, NATIONALITY_ID, RESIDENT_ID, COST_CENTRE, PHOTO_FILENAME, AUTHORISATION_ID, EMPNO)
- ITINERARIES (ID, TICKET_NUMBER, PEO_ID, DEPART_DATE, TRIP_DAYS, INVOICE_DATE, RESERVATION_DATE, DOM_INTL, RECORD_TYPE, CREDIT_CARD_CODE, VALIDATING_CARRIER, SEGMENT_COUNT, HOTEL_COUNT, CAR_COUNT, TICKET_DAYS_IN_ADV, FLIGHT_FULL_FARE, FLIGHT_TAX_AMOUNT, FLIGHT_BASE_AMOUNT, CAR_COSTS, HOTEL_COSTS, TRIP_DESC, AUTHORISATION_ID, CCD_ID)
- FARE_SCHEDULES (ID, FLR_ID, FARE_CATEGORY, STANDARD_PRICE, CABIN_CLASS, START_DATE, END_DATE, ENDORSEMENTS, TAX_RATE)
- CREDIT_CARDS (CARD_ID, CARD_NUMBER, TYPE, EXPIRATION_DATE, PEO_ID)

A aplicação *Travel*, como pode ser visto na Figura A.2, possui dois perfis de usuários que definem as permissões de acesso à aplicação:

1. **Agente**: perfil que define privilégios especiais ao usuário. Quem tem esse perfil pode consultar, alterar e cancelar as reservas de qualquer cliente da aplicação. Esse perfil é utilizado por agentes de viagem da empresa;
2. **Cliente**: perfil que define privilégios de consulta, alteração e cancelamento de reservas feitas somente pelo próprio cliente. Esse perfil é utilizado por qualquer usuário da aplicação.

Com a definição dos módulos GABD*Web* e *Travel* foi possível realizar a implementação de algumas funcionalidades do grafo de navegação utilizando Java Servlets. Não foram implementadas todas as funcionalidades da aplicação *Travel*, pois, o objetivo de validação das técnicas apresentadas nessa dissertação não exigia a implementação de toda a aplicação.

Bibliografia

[AD99a] Cecília S. Arias and Beatriz M. Daltrini. Base Architecture for Network Application Development. *International Conference on Enterprise Information System, ICEIS'99, Setúbal, Portugal*, 1999.

[AD99b] Cecília S. Arias and Beatriz M. Daltrini. WWW User Interface Specification and Implementation Using AIAs. *21st International Conference on Information Technology Interfaces, ITI'99, Pula, Croácia*, 1999.

[Apa] Apache. Apache HTTP Server Project. *http://www.apache.org/*.

[BBH+99] Athman Bouguettaya, Boualem Benatallah, Lily Hendra, James Beard, Kevin Smith, and Mourad Ouzzani. World Wide Database - Integrating the Web, CORBA and Databases. *ACM SIGMOD International Conference on Management of Data, Philadelphia, USA*, pages 594–596, June 1999.

[Ber] Hans Bergsten. An Introduction to Java Servlets. *http://WebDevelopersJournal.com/articles/*.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol - HTTP/1.0. *RFC 1945*, May 1996.

[CHS93] Ibe O. C., Choi H., and Trivedi K. S. Performance Evaluation of Client-Server Systems. *IEEE Transactions on Parallel and Distributed Systems*, 04(11), 1993.

[Cora] Borland Software Corporation. InterBase. *http://www.borland.com/interbase/*.

[Corb] Netscape Communications Corporation. The SSL Protocol. *http://home.netscape.com/eng/ssl3/ssl-toc.html*.

[Cor01] Borland Software Corporation. InterBase 4.0 - White Paper. *http://www.borland.com/interbase/papers/*, April 2001.

[Dara] Chád Darby. Developing 3-Tier Database Applications with Java Servlets. *http://www.sys-con.com/java/feature/3-2/3-tier/*.

[Darb] Chád Darby. Migrating CGI Scripts to Java Servlets. *http://www.sys-con.com/java/feature/3-1/cgi-scripts/*.

[Des] Alligator Descartes. DBI - A Database Interface for Perl 5. *The Perl Journal - http://www.symbolstone.org/technology/perl/DBI/doc/tpj5/index.html*.

[Dua96] Nick N. Duan. Distributed Database Access in a Corporate Environment using Java. *Computer Networks and ISDN Systems*, 28(7-11):1149–1156, May 1996.

[EIT] EIT. Enterprise Integration Technologies. *http://www.eit.com/*.

[Enha] Enhydra. Enhydra Java Application Server Architecture. *http://www.enhydra.org/*.

[Enhb] Enhydra. Open Source Java/XML Application Server. *http://enhydra.enhydra.org/*.

[FGM+97] R. Fielding, J. Gettys, Jeffrey C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. *RFC 2068*, January 1997.

[Gil] Dan Gildor. Replacing your CGIs with Java Servlets. *http://www.developer.com/news/techworkshop/*.

[Gui01] Célio C. Guimarães. A WWW SQL Programming Tool with Persistent Database Connections. *9th International Python Conference, Long Beach, CA*, March 2001.

[Hei97] John Heidemann. Performance Interactions between P-HTTP and TCP Implementations. *ACM SIGCOMM*, 27(02):65–73, April 1997.

[HM97] S. Hadjiefthymiades and D. Martakos. Improving the Performance of CGI Compliant Database Gateways. *Sixth International WWW Conference, Santa Clara, CA, USA*, 29:08–13, April 1997.

[HMP98] S. Hadjiefthymiades, D. Martakos, and C. Petrou. State Management in WWW Database Applications. *22nd Annual International Computer Software and Applications Conference (IEEE COMPSAC'98), Vienna, Austria*, 1998.

[HMP99] S. Hadjiefthymiades, D. Martakos, and C. Petrou. Stateful Relational Database Gateways for the World Wide Web. *Journal of Systems and Software (JSS), Elsevier Science*, 48(03), 1999.

[HPMP98] S. Hadjiefthymiades, S. Papayiannis, D. Martakos, and C. Petrou. Integrating Databases on the WWW. *Panhellenic Conference on New Information Technology (NIT'98), Athens, Greece*, 1998.

[HVM00] S. Hadjiefthymiades, I. Varouxis, and D. Martakos. Performance of RDBMS-WWW Interfaces Under Heavy Workload. *Journal of Universal Computer Science, Springer Verlag*, 06(06), 2000.

[ID98] Aruan Iyengar and Daniel Dias. Distributed Virtual Malls on the World Wide Web. *18th IEEE International Conference on Distributed Computing Systems (ICDCS'98), Amsterdam, Netherlands*, May 1998.

[Iye97a] Aruan Iyengar. Dynamic Argument Embedding: Preserving State on the World Wide Web. *IEEE Internet Computing*, 01(02):50–56, March/April 1997.

[Iye97b] Aruan Iyengar. Preserving State on the World Wide Web Using Dynamic Argument Embedding. *IBM Research Report RC 20924(92672)*, July 1997.

[KM97] David M. Kristol and Lou Montulli. HTTP State Management Mechanism. *RFC 2109*, February 1997.

[KM98] David M. Kristol and Lou Montulli. HTTP State Management Mechanism. *Internet Draft draft-ietf-http-state-man-mec-09.ps, IETF*, July 1998.

[Kra97] Ralf Kramer. Databases on the Web: Technologies for Federation Architectures and Case Studies. *ACM International Conference on Management Data, Tucson, Arizona*, 1997.

[Kri98] David M. Kristol. HTTP Proxy State Management Mechanism. *Internet Draft draft-kristol-http-proxy-state-00.ps, IETF*, May 1998.

[LDWN96] Yew Huey Liu, Paul Dantzig, C. Eric Wu, and Lionel M. Ni. A Distributed Connection Manager Interface for Web Services on IBM SP Systems. *International Conference for Parallel and Distributed Systems (ICPADS'96), Tokyo, Japan*, June 1996.

[LF98] Hongjun Lu and Ling Feng. Integrating Database and Web Technologies. *International Journal of World Wide Web*, 01(02):73–86, 1998.

[LF99] Hongjun Lu and Ling Feng. Towards a Web-Based Database Application Development System. *Asia Pacific Web Conference (APWeb'99), Kowloon, Hong Kong, China*, pages 27–29, September 1999.

[Mal98] Susan Malaika. Resistance is Futile: The Web Will Assimilate Your Database. *IEEE Bulletin of Data Engineering*, 21(02):04–13, June 1998.

[MFa] Stefano Mazzocchi and Pierpaolo Fumagalli. Advanced Apache JServ Techniques. *http://www.apache.org/~stefano/papers/*.

[MFb] Stefano Mazzocchi and Pierpaolo Fumagalli. Servlet Performance and Apache JServ. *http://www.apache.org/~stefano/papers/*.

[Mog95] Jeffrey C. Mogul. The Case for Persistent-Connection HTTP. *ACM SIGCOMM*, 25(04):299–313, October 1995.

[Moo99] Keith Moore. Applicability Statement for HTTP State Management. *Internet Draft draft-iesg-http-cookies-00.txt, IETF*, May 1999.

[NCS] NCSA. CGI - The Common Gateway Interface. *http://hoohoo.ncsa.uiuc.edu/cgi/*.

[Net] OTN Oracle Technology Network. Oracle9*i*AS - Oracle9*i* Application Server. *http://technet.oracle.com*.

[Net01a] OTN Oracle Technology Network. Oracle9*i* Application Server - Product White Paper. May 2001.

[Net01b] OTN Oracle Technology Network. Oracle9*i* Application Server - Scalability, Availability, and Load Balancing Options. February 2001.

[NGBS+97] H. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1 and PNG. *ACM SIGCOMM*, September 1997.

[NS96] Tam Nguyen and V. Srinivasan. Accessing Relational Databases from the World Wide Web. *ACM SIGMOD International Conference on Management of Data, Montreal, Canada*, pages 529–540, June 1996.

[Pad95] Venkata N. Padmanabhan. Improving World Wide Web Latency. *Report No. UCB/CSD-95-875, Computer Science Division, University of California at Berkeley, Chicago, USA*, May 1995.

[Per95a] Louis Perrochon. On the Integration of Legacy Information Servers into the World Wide Web. *Internet-Tutorial of the First Joint Annual Meeting 1995 of the Gesellschaft für Informatik (GI) and the Schweizer Informatiker Gesellschaft (SI) (GISI'95), Zürich, Switzerland*, September 1995.

[Per95b] Louis Perrochon. W3 'Middleware': Notions and Concepts. *Fourth International World Wide Web Conference, Boston, MA*, December 1995.

[PM94] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP Latency. *Second International World Wide Web Conference, Chicago*, pages 995–1005, October 1994.

[PM96] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *ACM SIGCOMM*, 26(03):22–36, July 1996.

[Pos81] J. Postel. Transmission Control Protocol. *RFC 793*, September 1981.

[Pos00] Great Bridge PostgreSQL. Open Source Object-Relational Database: Introduction and Overview. *White Paper - GreatBridge*, December 2000.

[Pre00] Lutz Prechelt. An Empirical Comparison of Seven Programming Languages. *IEEE Computer*, 33(10):23–29, October 2000.

[Pro] Java Apache Project. Apache JMeter. *http://java.apache.org/jmeter/index.html*.

[Ray] Eric Steven Raymond. Open Source. *http://www.opensource.org/docs/definition.html*.

[Teaa] PostgreSQL Development Team. PostgreSQL. *http://www.postgresql.org*.

[Teab] PostgreSQL Development Team. PostgreSQL Tutorial.

[XLF99] Quan Xia, Hongjun Lu, and Ling Feng. Supporting Web-Based Database Application Development. *6th International Conference on Database Systems for Advanced Applications (DASFAA'99), Taiwan*, pages 17–26, April 1999.

[Zha99] Weigang Zhao. A Study of Web-Based Application Architecture and Performance Measurements. *Fifth Australian World Wide Web Conference (AusWeb'99), Lismore, Australia*, July 1999.