UNIVERSIDADE DO EXTREMO SUL CATARINENSE – UNESC

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

DIRCEU PEREIRA TIEGS

REPLICAÇÃO DE BASES DE DADOS DO ZOPE OBJECT DATABASE

CRICIÚMA, JULHO DE 2008

DIRCEU PEREIRA TIEGS

REPLICAÇÃO DE BASES DE DADOS DO ZOPE OBJECT DATABASE

Trabalho de Conclusão de Curso apresentado para obtenção do grau de Bacharel em Ciência da Computação da Universidade do Extremo Sul Catarinense.

Orientador: Prof. MSc. Daniel Pezzi da Cunha

CRICIÚMA, JULHO DE 2008

Aos meus pais, Eunice Pereira e Dirceu Tiegs, e a minha namorada, Morgana Motta Fortuna.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, que sempre me apoiaram na minha carreira profissional, acadêmica e na minha vida. Agradeço também à minha namorada, meus grandes amigos e em geral a todas as pessoas que contribuíram de alguma maneira para que este trabalho pudesse ser realizado. A todos, meus mais sinceros agradecimentos.

“A mente que se abre a uma nova idéia jamais volta ao seu tamanho original.”

(Albert Einstein)

RESUMO

A quantidade de usuários de serviços *online* cresce em uma velocidade impressionante e as organizações que disponibilizam serviços de banco de dados precisam mantê-los disponíveis pelo maior tempo possível e com alto desempenho. Com o objetivo de manter serviços disponíveis distribuídos, o *Zope Object DataBase* (ZODB) - Sistema Gerenciador de Bancos de Dados Orientados a Objeto *open-source* - faz uso da replicação mestre-escravo. O problema fundamental dessa abordagem é que ela mantém um ponto único de falha, ou seja, se o nó mestre ficar indisponível os outros nós perdem sua utilidade. Como solução, o software *ZEORaid* disponibiliza um *gateway* experimental para replicação N para N. Este trabalho envolveu a pesquisa, a análise e o aprimoramento do *ZEORaid* para que os nós replicados mantenham a consistência. Como resultados, conseguiu-se aprimorar o emprego da alta disponibilidade mesmo em caso de falha de serviços e com um número muito grande acessos, aumentando também a escalabilidade e flexibilidade de configurações de sistemas distribuídos.

Palavras chave: ZODB, Alta Disponibilidade, Replicação, Bancos de Dados Distribuídos.

ABSTRACT

The online services user number is growing up every day at an impressive ratio while the organizations try to establish high available database services, keeping them running as long as possible. To achieve this goal the Zope Object DataBase (ZODB) - an open-source object oriented database management system - uses a master-slave replication approach. The major issue of this approach is a possible failure in the master node, what turns unavailable all the other nodes. Trying to solve this, the ZEORaid software provides an experimental gateway for N to N replication, removing the single point of failure. This research analysed and improved ZEORaid and were obtained valuable improvements, showing good results even during service failures, improving the general scalability and flexibility of the distributed systems configurations tested.

Keywords: ZODB, High Availability, Replication, Distributed Databases.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

2PC	2-Phase Commit
3PC	3-Phase Commit
OO	Orientado a Objetos
RAID	Redundant Arrays of Independent Disks
RPC	Remote Procedure Call
SGBD	Sistema Gerenciador de Banco de Dados
SGBD-OO	Sistema Gerenciador de Banco de Dados Orientado a Objeto
ZEO	Zope Enterprise Objects
ZODB	Zope Object Database
ZOPE	Z Object Publishing Environment
ZRS	Zope Replication Services

SUMÁRIO

1 INTRODUÇÃO

Enquanto muitas organizações podem se contentar em ter seus sistemas disponíveis em menos de 99% do tempo, outras, como bancos e sistemas governamentais precisam da maior disponibilidade possível. Alta Disponibilidade se refere às técnicas usadas para manter um sistema acessível aos usuários ou a outros sistemas. Um sistema é dito altamente disponível se, usando algumas técnicas pré-estabelecidas ele é confiável, auto-recuperável, tem detecção de erros e provê operações contínuas (ORACLE, 2007).

Uma das estratégias usadas para manter a alta disponibilidade é a replicação, onde vários serviços iguais ficam trabalhando ao mesmo tempo em locais diferentes, não deixando toda a carga e confiança no funcionamento do sistema em um ponto único (JALOTE, 1994). Isso é comumente utilizado no caso de servidores HTTP, *cache* e bancos de dados relacionais, mas isso foi pouco desenvolvido para bancos de dados orientados a objetos.

O Zope Object DataBase (ZODB) é um Sistema Gerenciador de Bancos de Dados Orientados a Objeto open-source (FERRI, 2002) utilizado por grandes corporações como o Bank Boston e vários governos, como o brasileiro e o inglês (PRODASEN, 2004).

Existe um problema fundamental no uso de ZODB em ambientes de alta disponibilidade: por ser um banco de dados orientado a objetos e, portanto, altamente dinâmico (ANTONIO, 2001), este sistema torna-se bastante lento, exigindo grandes

recursos de CPU e memória. O problema aumenta muito conforme a quantidade de acessos aos dados cresce, forçando o uso de múltiplos níveis de *cache* e melhorias no hardware.

Uma arquitetura estilo Mestre – Escravo chamada Zope Replication Services (ZRS) (ZOPE.COM, 2007) foi criada pela Zope Corporation (também responsável pelo Zope e ZODB), mas ela tem um problema grave: mantém um ponto único de falha, ou seja, se o nó mestre ficar indisponível os outros nós perdem sua utilidade.

Para a comunicação de múltiplos processos (N para N) costuma-se utilizar *gateways*. Um *gateway*, nesse caso, é um programa intermediário cujo propósito é fazer a comunicação, roteamento e *cache* de diferentes processos em diferentes máquinas. O software ZEORaid disponibiliza um *gateway* ainda experimental para replicação N para N de bancos de dados do ZODB.

Uma característica muito importante em sistemas de computação distribuída é a comunicação assíncrona. Na comunicação síncrona, ou bloqueante, quando um processo faz uma requisição a outro ele não pode executar outras operações enquanto a resposta não chega ou o tempo de espera se esgota; no caso da comunicação assíncrona, ou não-bloqueante, o processo pode efetuar todas as operações que não necessitem da resposta do outro processo enquanto espera pela mesma (JALOTE, 1994).

A fim de disponibilizar um serviço de replicação para o SGBD-OO ZODB este trabalho envolveu a pesquisa, a análise e o aprimoramento do *gateway* assíncrono ZEORaid para que bancos de dados do ZODB sejam replicados e mantenham consistência, podendo tornar transparente o uso de várias bases de dados e manter a alta

disponibilidade mesmo em caso de falha de serviços em um ou mais servidores ou em caso de um número muito grande acessos, aumentando também a escalabilidade e flexibilidade de configurações de sistemas distribuídos que utilizam o ZODB.

1.1 OBJETIVO GERAL

Analisar diferentes configurações de replicação para o ZODB utilizando o ZEORaid.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos dessa pesquisa consistem em:

a) realizar um estudo sobre técnicas de alta disponibilidade aplicadas em bancos de dados distribuídos;
b) compreender os mecanismos de controle de transação, concorrência e comunicação utilizados no ZODB;
c) identificar falhas no uso do *gateway* ZEORaid para replicação de dados;
d) propor soluções para os problemas encontrados;
e) aprimorar o sistema ZEORaid no uso de RAID nível 1.

1.3 JUSTIFICATIVA

A quantidade de usuários de serviços online cresce em uma velocidade impressionante e as organizações responsáveis por esses serviços precisam mantê-los disponíveis pelo maior tempo possível e com alto desempenho. Diferentes de técnicas de alta disponibilidade são utilizadas para atender as requisições de todos os usuários de uma maneira satisfatória (JALOTE, 1994).

Todos estes serviços precisam de uma maneira eficiente de armazenamento, capaz de manter os dados seguros e responder as requisições com performance adequada. Embora a maior parte do armazenamento de sistemas online seja feita usando Sistemas Gerenciadores de Bancos de Dados Relacionais, a crescente necessidade de características mais dinâmicas tem elevado o uso de Sistemas Gerenciadores de Bancos de Dados Orientados a Objeto.

O Zope Object Database (ZODB) é um Sistema Gerenciador de Bancos de Dados Orientados a Objeto open-source utilizado principalmente no Z Object Publishing (ZOPE), um framework de desenvolvimento e servidor de aplicativos web. Além do ZOPE, o ZODB pode ser utilizado em qualquer tipo de aplicação desenvolvida usando a linguagem Python ou que use essa linguagem como conector.

Grandes empresas como o governo brasileiro e o Bank Boston utilizam o ZODB em aplicações de grande porte, onde a necessidade de alta disponibilidade é maior. A população pode se beneficiar muito, mesmo que indiretamente, de recursos de alta disponibilidade em aplicações do governo, por exemplo: quanto mais performático, confiável e disponível for um sistema, melhor será o atendimento às necessidades dessas pessoas.

A Zope Corp, corporação responsável pelo ZODB e pelo o ZOPE, desenvolveu um sistema de replicação chamado Zope Replication Services (ZRS) que é capaz de utilizar até dois servidores de dados redundantes (ZOPE.COM, 2007), mas não é capaz de atender um número maior que esse, o que pode tornar o sistema lento ou até indisponível em alguns casos.

Com o intuito de pesquisar os problemas e vantagens da área de replicação em Sistemas Gerenciadores de Bancos de Dados Orientados a Objeto, este trabalho envolveu a análise e o aprimoramento do *gateway* ZEORaid para permitir a comunicação de um número maior de bases de dados distribuídas do ZODB, eliminando a restrição de apenas dois nós imposta pelo ZRS e permitindo maior flexibilidade, escalabilidade e disponibilidade dos sistemas que o utilizam.

1.4 ESTRUTURA DO TRABALHO

Este trabalho é formado por 8 seções. Na seção 1 há uma introdução aos objetivos do projeto, os temas envolvidos e a justificativa para realização do mesmo.

Nas seções 2, 3 e 4 são abordados os temas relacionados a Alta Disponibilidade, Bancos de dados Distribuídos e Bancos de Dados Orientados a Objeto.

A seção 5 apresenta trabalhos correlatos onde o objetivo dos mesmos é semelhante ao deste trabalho.

As ferramentas, métodos e dados utilizados para a realização dos testes de desempenho, bem como as suas especificações e problemas encontrados são abordados na seção 6.

A seção 7 apresenta o resultado dos testes no uso de replicação, de acordo com os resultados obtidos nas etapas anteriores.

Por fim, há uma conclusão e as possibilidades para trabalhos futuros.

2 ALTA DISPONIBILIDADE

Nos últimos 50 anos os computadores passaram de enormes e caras máquinas usadas por governos e grandes corporações para se tornar algo cotidiano. Os computadores são visíveis em todo lugar, na forma de desktops, notebooks, PDAs e de sistemas inteligentes embarcados, por exemplo, em carros, afirmam Koren e Krishna (2007).

A atuação dos computadores está cada vez mais atingindo pontos críticos, como contas bancárias, ações em bolsa de valores, computadores de bordo de aviões, carros e navios e aparelhos médicos, e conforme as facilidades se tornam parte do cotidiano, a população fica cada vez mais dependente de todas essas ações.

Computadores são máquinas complexas; a complexidade do hardware aumenta dia-a-dia com novos chips cada vez menores e ainda assim mais performáticos, e a complexidade do software é muito maior, provocando ainda mais chances de falha. É muito difícil encontrar um software ou hardware sem falhas, dizem Koren e Krishna (2007), mesmo na área espacial, onde tudo é testado meticulosamente.

Dye (1999) define Alta Disponibilidade como o emprego de técnicas de contenção e tolerância à falhas de hardware e software para permitir que um serviço fique disponível para seus usuários a maior parte do tempo possível. O uso de cada técnica depende da classificação das falhas encontradas.

2.1 CLASSIFICAÇÃO DE FALHAS

Segundo Koren e Krishna (2007) a definição de falha pode ser dividida em duas definições formais:

a) falha: um defeito de hardware ou um *bug* de software;

b) erro: a manifestação da falha.

Tanto falhas quanto erros podem se espalhar por um sistema; caso um microprocessador queime por causa de problemas de voltagem, esse erro não deve se manifestar em outras áreas do hardware. Para isso o projetista deve pensar em áreas de contenção para isolar o problema de forma que nenhum outro componente queime.

Falhas de hardware podem ser permanentes, transientes ou intermitentes. Falhas permanentes afetam de maneira permanente um dispositivo, possivelmente de forma irreversível; falhas transientes acontecem por um tempo em um dispositivo e depois, por algum motivo, o mesmo volta a funcionar corretamente; e falhas intermitentes oscilam, acontecendo de tempos em tempos (KOREN; KRISHNA, 2007).

2.1.1 Tipos de Redundância

Toda a tolerância à falhas é baseada na redundância. Por redundância entende-se a propriedade de existir mais de um recurso além do mínimo necessário para fazer o trabalho em questão. Quando falhas acontecem a redundância é utilizada para encobrir os erros e manter o sistema disponível para uso.

Koren e Krishna (2007) citam quatro tipos de redundância: hardware,

software, informações e tempo. Redundância via hardware normalmente é gerenciada por hardware, enquanto os outros tipos são gerenciados por software.

Tolerância via hardware é proporcionada pelo uso de hardware extra para detectar ou suprimir os erros causados por componentes com falhas. Um exemplo disso é o uso de três processadores desempenhando a mesma função; se um falhar os outros dois vão ser capazes de detectar o erro e continuar com suas operações. Isso é um exemplo de redundância estática, cujo objetivo é suprimir erros assim que eles acontecem. Outros tipos de redundância são a redundância dinâmica, onde um dispositivo é acionado assim que uma falha é detectada, e a redundância híbrida, que usa redundância estática e dinâmica para promover alta disponibilidade.

2.2 REDUNDÂNCIA DE INFORMAÇÕES

Erros nos dados podem acontecer enquanto os dados são transferidos de uma unidade para outra, de um sistema para outro ou até quando os dados estão armazenados em uma única unidade de memória. Para tolerar esse tipo de falha deve ser introduzida redundância nesses dados; isso é chamado redundância de informações. O tipo mais comum de redundância de informações é a codificação, que adiciona bits de checagem aos dados, que permite a detecção de erros e, em alguns casos, até a correção desses erros (KEMME, 2000).

Embora a codificação seja muito útil para arquivos individuais, é mais comum utilizar técnicas para grandes volumes de dados. Um exemplo disso é o uso de *RAID* (do inglês *Redundant Array of Independent Disks*), que conforme o nome indica,

utiliza conjuntos de discos independentes que armazenam os dados de maneira redundante (KOREN; KRISHNA, 2007).

Em sistemas distribuídos, onde os mesmos grupos de dados devem estar acessíveis para diferentes nós do sistema, a replicação de dados pode ajudar na acessibilidade dos dados. Manter os dados em um único nó pode fazer com que esse nó se torne um gargalo de performance e deixe os dados vulneráveis a falhas nesse nó. Uma forma de prevenir esse tipo de problema é manter cópias idênticas dos dados em múltiplos nós (DYE, 1999).

2.2.1 RAID

RAID é um bom exemplo de utilização de redundância em grandes volumes de dados, segundo Koren e Krishna (2007). As cinco estruturas (níveis) mais comuns de uso de RAID são RAID nível 1, 2, 3, 4 e 5.

2.2.1.1 RAID nível 1

RAID1 consiste em discos espelhados. No lugar de um disco existem dois discos, cada um sendo uma cópia do outro. Se um disco falha, o outro pode continuar a servir acesso aos dados. Se dos dois discos estiverem funcionando, RAID1 pode tornar o acesso a leitura mais rápido dividindo as requisições entre os dois discos; o acesso a escrita, contudo, se torna mais lento, pois os dois discos têm que terminar a atualização

antes que a operação de escrita se complete.

2.2.1.2 RAID nível 2

RAID2 consiste no uso conjunto de grupo de discos armazenando dados com um grupo de discos armazenando a codificação desses dados usando a teoria de Hamming. É o tipo menos utilizado de RAID por causa da lentidão e custo operacional.

2.2.1.3 RAID nível 3

RAID3 é uma modificação de RAID2 que usa detecção e correção de erros via codificação por setores do disco. São utilizados um grupo de discos de dados juntamente com um disco de paridade, e os dados são intercalados entre os discos de dados. Todas as operações de leitura e escrita são sincronizadas, o que diminui o tempo de transferência de dados. A Figura 1 mostra uma configuração de RAID3.

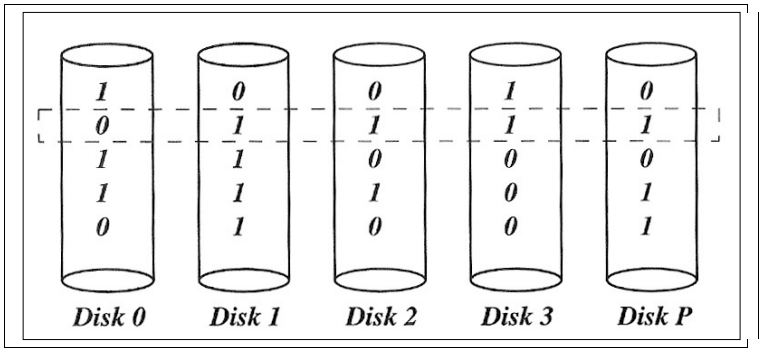


Figura 1. Configuração RAID3 com quatro discos de dados e um disco de paridade
Fonte: KOREN, I.; KRISHNA, C. M. (2007)

2.2.1.4 RAID nível 4

RAID4 é similar ao RAID3, exceto pelo fato da unidade de paridade não ser um único bit, mas um bloco de tamanho arbitrário chamado *stripe*. A vantagem do RAID4 sobre o RAID3 é que uma operação pequena de leitura pode ser contida em um único disco de dados, ao invés de serem intercalados todos os discos. Como resultado, pequenas operações de leitura são mais rápidas no RAID4. Um detalhe importante de se observar é que quando uma operação de escrita é realizada tanto o disco afetado quanto o disco de paridade precisam ser atualizados.

A Figura 2 mostra uma configuração de RAID4.

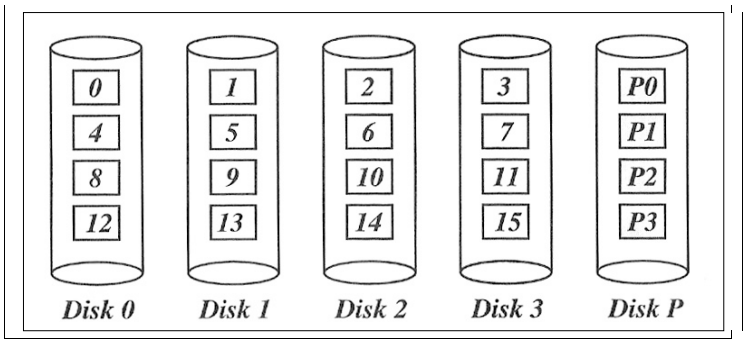


Figura 2. Configuração RAID4 com quatro discos de dados e um disco de paridade
Fonte: KOREN, I.; KRISHNA, C. M. (2007)

2.2.1.5 RAID nível 5

RAID5 é uma atualização do RAID4 que leva em conta o fato do disco de paridade poder tornar-se um gargalo de performance no sistema; no RAID4 cada operação de escrita em um disco de dados atualiza o disco de paridade também. Para resolver esse problema, RAID5 intercala os blocos de paridade entre todos os discos, não existindo assim um disco específico para paridade.

A Figura 3 mostra uma configuração de RAID5.

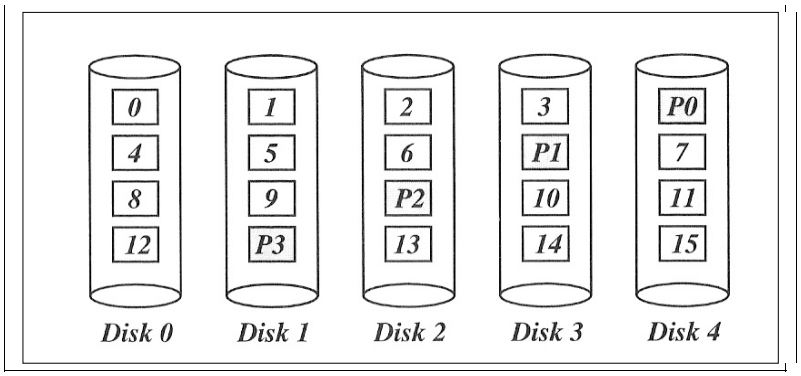


Figura 3. Blocos de paridade distribuídos em cinco discos em uma configuração RAID5
Fonte: KOREN, I.; KRISHNA, C. M. (2007)

2.2.2 Replicação de Dados

Replicação de dados em sistemas distribuídos é um outro exemplo de como a redundância de informações pode ser usada para melhorar a tolerância a falhas no nível de sistema. A replicação de dados consiste em manter cópias idênticas dos dados em dois ou mais nós de um sistema distribuído (KEMME, 2000). Assim como um

sistema RAID, a replicação pode prover tanto tolerância à falhas quanto melhorias na performance. Entretanto, é importante que as réplicas dos dados sejam mantidas de maneira consistente mesmo com possíveis falhas no sistema.

2.2.2.1 Pesos: Organização Hierárquica e Não-Hierárquica

Uma abordagem comum para o uso de replicação de dados é o uso de pesos para cópias individuais, seja em uma organização hierárquica ou não-hierárquica. Os pesos permitem preferir cópias que residem em um nó mais acessível ou melhor conectado (KOREN; KRISHNA, 2007).

2.2.2.2 Primary-Backup

Na abordagem *Primary-Backup* um nó é designado como primário e todos os acessos acontecem através desse nó. Os outros nós são designados backups, e todas as operações de escrita realizadas no nó primário são copiadas para os nós de backup; quando o nó primário falha um dos nós de backup é escolhido para tomar o seu lugar (URBANO, 2003).

Para manter a consistência existe o conceito de visão de grupo. Assim que o sistema é inicializado com um nó primário e seus nós de backup, todos os nós mantém uma lista disso - essa lista é a chamada visão de grupo. Quando acontecem problemas (ou reparos) em um determinado nó, esse problema é detectado e a visão de grupo dos

outros nós é atualizada.

Utilizando essa abordagem todas as requisições são recebidas pelo nó primário. Ele encaminha a requisição para os nós de backup e espera até receber uma confirmação de todos eles. Assim que a confirmação chega, ele responde a requisição do cliente. Isso é o que acontece no caso de não haver nenhuma falha.

No caso de acontecer uma falha que cause a desconexão de vários nós, apenas os que permanecerem ativos farão parte do algoritmo. Assim que a conexão for restabelecida, todos os nós que estavam desconectados devem ser reinicializados e sincronizados.

No caso de ocorrerem falhas em nós de backup, o nó primário pode ficar esperando indefinidamente a confirmação de requisição. Isso é facilmente remediável introduzindo um recurso de *timeout*; se o nó primário não recebe a confirmação em um determinado período de tempo ele assume que o nó de backup em questão está com problemas e o retira do grupo.

Se ocorrer uma falha no nó primário, existem 3 possibilidades:

a) se o nó primário ainda não encaminhou a requisição para nenhum dos nós de backup, não há inconsistência; a única coisa a fazer é designar um dos nós de backup para ficar no lugar do nó primário;
b) se o nó primário encaminhou a requisição para todos os nós de backup, também não há inconsistência; também é necessário escolher um nó de backup como nó primário;
c) se o nó primário encaminhou a requisição para alguns (mas não todos) os nós de backup; nessa situação é necessária alguma medida para corrigir a inconsistência entre os nós. Assim que um nó de backup

> detecta que o nó primário está inacessível, ele envia requisições para todos os outros nós atualizarem seu grupo de visão; os nós que já haviam recebido a requisição são sincronizados com os nós desatualizados, e assim que esse processo termina um novo grupo é estabelecido.

A detecção de falhas do nó primário pode acontecer de diferentes formas; por exemplo, cada nó pode executar diagnósticos sobre o estado dos outros nós. Outra forma é fazer o nó primário enviar *broadcasts* periodicamente para os nós de backup.

2.3 TOLERÂNCIA À FALHAS POR SOFTWARE

Pesquisadores reconhecem que existem dificuldades essenciais e acidentais na construção de software. Dificuldades essenciais vêm do desafio inerente de entender uma aplicação e um sistema operacional complexos, e de ter que construir uma estrutura compreendendo um número extremamente grande de estados com regras muito complexas de transição. Além disso, software é alvo de frequentes modificações, além de que o hardware e o sistema operacional mudam com o tempo. Dificuldades acidentais na construção de software vêm do fato de que pessoas cometem erros até em tarefas relativamente simples (KOREN; KRISHNA, 2007).

Muito trabalho vem sendo feito para reduzir a taxa de erros de softwares modernos. Essas técnicas se baseiam em procedimentos extensos de teste de software; esse tipo de teste, entretanto, não pode verificar conclusivamente se um programa tem erros ou não. Isso só pode ser alcançado através de uma prova matemática formal.

Construir essas provas formais é assunto de diversas pesquisas ativas; mesmo assim, o estado da arte atualmente ainda é muito primitivo e somente aplicável a pequenos softwares. Como resultado disso, é razoável assumir que todo grande software hoje contém defeitos (KEMME, 2000).

2.3.1 Testes de Aceitação

Assim como sistemas de hardware, um importante passo para promover a Alta Disponibilidade é detectar as falhas. Uma maneira comum de detectar defeitos em software é o uso de testes de aceitação, que são usados em *wrappers* e blocos de recuperação, dizem Koren e Krishna (2007).

A maioria dos testes de aceitação se encaixam em uma das seguintes categorias:

a) checagem de tempo, onde um processo temporizador verifica se a aplicação demorou mais que o tempo normalmente usado para terminar a execução. Nesse caso o sistema assume que uma falha ocorreu;
b) verificação da saída, onde é analizada a atuação da aplicação sobre uma determinada entrada, verificando se os dados de saída são compatíveis;
c) checagem de intervalos. Em alguns casos não existe uma saída exata que permita a verificação, então é assumido um limite de erro possível e é verificada a saída utilizando esse limite.

2.3.2 Wrappers

Um wrapper é um componente de software que encapsula um determinado programa (ou componente) que está sendo executado. É possível encapsular praticamente qualquer nível de software; exemplos incluem aplicações, *middleware* e até um *kernel* de sistema operacional (DYE, 1999). Alguns exemplos de uso de wrappers:

a) Tratamento de *buffer overflow*;

b) Checagem no agendamento de tarefas;

c) Suprimir erros conhecidos;

d) Checar saídas para testes de aceitação.

A Figura 4 mostra um exemplo de wrapper, onde o "Wrapper software" atua como um intermediário para a "wrapped entity", encapsulando-a:

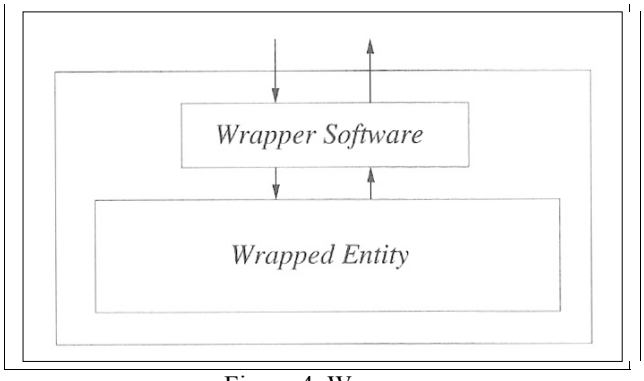


Figura 4. Wrapper
Fonte: KOREN, I.; KRISHNA, C. M. (2007)

2.3.3 Blocos de Recuperação

A abordagem de tolerância a falhas utilizando blocos de recuperação usa múltiplas versões de um software. Enquanto em outras abordagens várias versões são executadas em paralelo, usando blocos de recuperação apenas a última versão funcional do software é executada; se alguma falha é detectada, a execução é revertida para uma versão anterior (backup).

Nessa abordagem, a execução ocorre na versão primária, e então um teste de aceitação é feito sobre a saída - se o teste passa, a saída é aceita pelo sistema. Se o teste não passa, o estado dos dados e do sistema é revertido para o ponto anterior a execução da versão primária e a versão de backup é executada. Se nenhuma das versões de backup faz os testes passarem, o sistema deve tomar alguma medida corretiva específica.

Uma opção para agilizar o processo é utilizar blocos de recuperação distribuídos; nesse caso existem dois ou mais nós, cada um com versões idênticas do software e dos dados. Enquanto um nó executa uma versão (primária, possivelmente), o outro nó executa uma versão diferente. Cada nó pode servir como temporizador para os outros nós, detectando falhas passíveis de timeout. Se uma das execuções falha a outra pode efetuar alterações nos dados sem precisar reverter ao ponto inicial; se ambas as versões apresentam sucesso nos testes de aceitação a versão primária ou a versão que terminar a execução primeiro tem preferência (URBANO, 2003).

2.3.4 Chamada Remota de Procedimentos Tolerante à Falhas

Segundo Laurent, Johnston e Dumbill (2001) Chamada de Procedimentos

Remotos (do inglês *Remote Procedure Call*, ou RPC) é um mecanismo através do qual um processo pode fazer uma chamada a outro processo executando em algum outro processador. Conforme mostra a Figura 5 (onde aplicações construídas em Python, Perl e ASP trocam dados usando RPC), RPC pode ser usado independentemente de linguagem de programação ou plataforma de software / hardware.

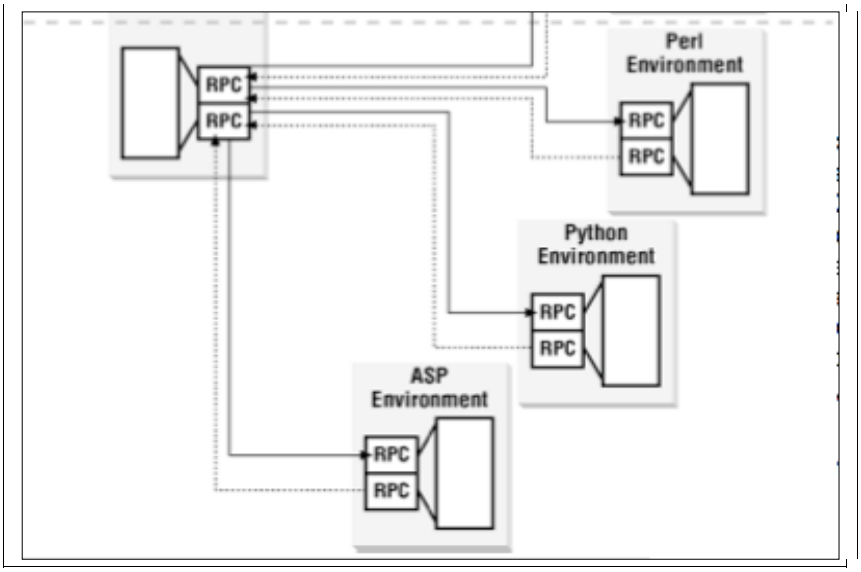


Figura 5. RPC em Ambientes Heterogêneos
Fonte: LAURENT, S. S.; JOHNSTON, J. & DUMBILL, E. (2001)

RPCs são largamente utilizadas na computação distribuída. Abaixo estão descritos dois meios de tornar RPCs tolerantes a falha, ambos utilizando replicação.

2.3.4.1 Abordagem Primary-Backup

Cada processo é implementado como primário e como backup, executando em nós separados. Chamadas são enviadas para ambas as cópias, mas normalmente

apenas o nó primário a executa. Se o primeiro falhar, o segundo é ativado e completa a execução.

A implementação dessa abordagem depende se as chamadas são repetíveis ou não-repetíveis. Uma chamada repetível é aquela que pode ser executada múltiplas vezes sem acarretar em erros; por exemplo, ler um banco de dados. Uma chamada não-repetível não pode ser executada mais de uma vez sem interferir na lógica ou acarretar em erros; por exemplo, incrementar um contador.

Caso existam chamadas não-repetíveis é importante assegurar que as operações serão executadas apenas uma vez, mesmo com múltiplos processos sendo executados paralelamente (KOREN; KRISHNA, 2007). Isso pode ser feito utilizando técnicas de *checkpointing*, não cobertas por esse trabalho.

2.3.4.2 Abordagem do Circo

A abordagem do Circo também envolve a replicação de processos. Processos cliente e servidor são replicados; seguindo a nomenclatura cada grupo de réplicas são chamados trupes.

Cada chamada tem um número de sequência associado que a identifica unicamente. São feitas chamadas idênticas para todos os processos servidores; um servidor espera até que tenha recebido chamadas idênticas de todos os clientes (ou timeout) antes de executar uma chamada. O resultado então é enviado de volta para todos os clientes; esse resultado também contém um número de sequência para identificá-lo.

Um cliente pode esperar até receber chamadas idênticas de todos os servidores (ou timeout) até aceitar o resultado, ou aceitar o primeiro resultado recebido e ignorar os restantes.

Existe uma complicação adicional: múltiplas trupes de clientes podem enviar chamadas concorrentes para uma mesma trupe de servidores; estas tem que responder com consistência e na mesma ordem. Existem duas maneiras de manter a consist6encia e a ordem, chamadas abordagem pessimista e abordagem otimista.

Na abordagem otimista nenhuma checagem inicial é feita; ao invés disso tudo é executado livremente e depois da execução o resultado é analisado para ver se a ordem foi mantida. Se a ordem não for mantida, o resultado é descartado e toda a operação é executada novamente. Isso pode se tornar um problema sério de performance caso a ordem seja perdida com frequência.

A abordagem pessimista, por outro lado, tem mecanismos que asseguram que a ordem será sempre preservada (KOREN; KRISHNA, 2007).

3 BANCOS DE DADOS DISTRIBUÍDOS

A tecnologia de Bancos de Dados Distribuídos é a união de duas abordagens de processamento de dados que a princípio parecem opostas: sistemas de bancos de dados e redes de computadores. Os bancos de dados mudaram a estrutura básica do tradicional processamento de arquivos; ao invés de cada aplicação definir e manter os seus próprios dados, os bancos de dados definiram um meio de fazer os dados serem organizados e mantidos de forma centralizada (OZNU; VALDURIEZ, 1999).

Uma das maiores motivações para o desenvolvimento e uso de bancos de dados é proporcionar um meio centralizado de controle de acesso, gerenciamento, armazenamento e integração dos dados.

Segundo Oznu e Valduriez (1999) um sistema distribuído de computação é um grupo de elementos processadores (não necessariamente homogêneos) que são inter-conectados por uma rede de computadores e que cooperam em uma determinada tarefa. Um banco de dados distribuído é uma coleção de múltiplos e logicamente inter-relacionados bancos de dados distribuídos sobre uma rede de computadores. Um Sistema Gerenciador de Bancos de Dados Distribuídos (SGBDD), então, é um software que permite o gerenciamento de bancos de dados distribuídos e torna a distribuição transparente para os usuários.

3.1 VANTAGENS

Muitas vantagens já foram citadas na literatura, indo de razões sociológicas até descentralização e economia. Essas vantagens podem ser resumidas em apenas quatro, segundo Urbano (2003):

a) gerenciamento transparente de dados distribuídos e replicados;
b) confiabilidade por meio de transações distribuídas
c) melhor performance
d) fácil expansão

3.1.1 Gerenciamento Transparente de Dados Distribuídos e Replicados

Transparência refere-se a separação da semântica de alto nível de um sistema das questões de implementação de baixo nível - em outras palavras, um sistema transparente esconde os detalhes de implementação de seus usuários. Existem vários tipos e níveis de transparência, descritos por Oznu e Valduriez (1999) e detalhados a seguir.

3.1.1.1 Independência de Dados

Independência de Dados refere-se a imunidade contra modificações na definição e organização dos dados feitas por aplicações de usuário.

Normalmente uma aplicação de usuário atua apenas sobre um pequeno grupo de dados, não sobre sua totalidade; por isso, não é necessário que a aplicação

possa alterar a organização e definições lógicas e físicas. Isso previne problemas quando múltiplas aplicações estão acessando o mesmo banco de dados e facilita para o desenvolvedor de aplicações, que não precisa lidar com detalhes de baixo nível.

3.1.1.2 Transparência de Rede

Além da independência de dados, outro fator é importante: transparência de rede. Os detalhes de estrutura e implementação da rede devem ficar escondidos dos usuários - é desejável que os usuários nem saibam da existência da rede.

Do ponto de vista do usuário, o uso de um banco de dados distribuído não deve ser diferente do uso de um banco de dados centralizado. Para isso são necessários dois tipos de transparência: transparência de local e transparência de nomes. Transparência de local refere-se ao fato de que o comando usado para realizar uma tarefa é independente tanto da localizações dos dados e da localização do sistema em que a operação deve ser executada. Transparência de nomes significa que um único nome é disponibilizado para cada objeto no banco de dados, para não ser necessária a identificação do local como parte do nome.

3.1.1.3 Transparência de Replicação

É desejável replicar dados por questões de performance, confiabilidade e disponibilidade de recursos, conforme visto no Capítulo 1. Assumindo que os dados são

replicados, o que deve ser tratado pela transparência de replicação é se os usuários devem ter conhecimento da presença de cópias ou se o sistema deve gerenciar isso de modo que os usuários vejam os dados como se fossem uma única cópia.

Do ponto de vista do usuário, a segunda opção é a melhor, por questões de usabilidade. Do ponto de vista do sistema, entretanto, obter essa transparência é altamente complexo, principalmente se tratando de áreas como concorrência e gerenciamento de transações.

3.1.1.4 Transparência de Fragmentação

É desejável, pelas mesmas razões da replicação, dividir um banco de dados em fragmentos menores e tratar cada fragmento como um banco de dados diferente. A fragmentação pode também reduzir os efeitos negativos da replicação: cada réplica não é a relação toda, apenas uma parte dela; então, menos espaço é necessário.

3.1.2 Confiabilidade Por Meio de Transações Distribuídas

Bancos de dados distribuídos são usados para aumentar confiabilidade e disponibilidade, já que eles replicam componentes e, por causa disso, eliminam pontos únicos de falha. A falha de um único ponto, ou a falha de um link de comunicação que faz com que um ou mais componentes fiquem inacessíveis não é suficiente para derrubar o sistema todo. No caso dos dados, isso significa que algumas partes dos dados

podem ficar inacessíveis, mas que os usuário devem ter a possibilidade de acessar outras partes dos dados. Isso é proporcionado por transações distribuídas e protocolos de aplicação.

Uma transação é uma unidade básica de computação consistente e confiável que consiste em uma sequência de operações de bancos de dados executadas como uma operação atômica. Ela transforma um estado consistente de banco de dados em outro estado consistente mesmo quando um grande número de transações é executado concorrentemente, mesmo quando falhas ocorram.

Proporcionar suporte à transações requer a implementação protocolos de controle de concorrência - em particular *two-phase commit* (2PC) e protocolos de recuperação distribuída.

3.1.3 Melhor Performance

A melhoria de performance proporcionada sobre um banco de dados distribuído geralmente vem de dois pontos: fragmentação / replicação e paralelismo.

A fragmentação e replicação podem tornar o acesso aos dados mais rápido por distribuir carga em diferentes máquinas e em diferentes locais, diminuindo latência de rede e uso de memória / CPU. O paralelismo melhora a performance por causa do uso de *inter-query* e *intra-query*. Inter-query é a habilidade de executar múltiplas *queries* ao mesmo tempo, enquanto intra-query é alcançada dividindo uma *query* única em várias *sub-queries*, cada uma executando em um nó diferente.

3.1.4 Fácil Expansão

Em um banco de dados distribuído é muito simples lidar com o aumento de tamanho de uma base de dados - a única coisa necessária é adicionar mais capacidade de processamento e armazenamento. Enquanto isso também é possível em um banco de dados centralizado, isso se torna muito mais barato em um banco de dados distribuído. Ao invés de atualizar os servidores que já existem, existe a possibilidade de adicionar mais servidores na rede.

Isso pode ser muito mais barato enquanto for tomado cuidado com a performance de comunicação na rede.

3.2 ARQUITETURA DE BANCOS DE DADOS DISTRIBUÍDOS

A arquitetura de um sistema define a sua estrutura. Isso significa que os componentes de um sistema são identificados, a função de cada componente é especificada e os relacionamentos e interações entre componentes são definidas (TANENBAUM; STEEN, 1999).

Existem várias arquiteturas possíveis para bancos de dados distribuídos, a maioria idealizadas e somente teóricas. A arquitetura mais usada nos vários tipos de bancos de dados é a arquitetura Cliente / Servidor, detalhada a seguir.

3.2.1 Cliente / Servidor

A arquitetura cliente / servidor para bancos de dados apareceu no começo da década de 90 e causou um grande impacto tanto na tecnologia de bancos de dados quanto na computação em geral. A idéia é dividir toda a funcionalidade que um sistema deve proporcionar em duas classes: funções de servidor e funções de cliente. Isso proporciona uma arquitetura de duas camadas que torna mais fácil gerenciar a complexidade dos bancos de dados modernos (TANENBAUM; STEEN, 1999).

A distinção entre tarefas de cliente e servidor é bem clara: o servidor fica com a maior parte do gerenciamento de dados, ou seja, com tarefas como otimização de queries, gerenciamento de transações e concorrência e gerenciamento de armazenamento. O cliente fica responsável pela interface com o usuário, conexão com o servidor e tarefas secundárias mas importantes como caching (OZNU; VALDURIEZ, 1999).

Em determinadas arquiteturas os nós podem assumir múltiplos papéis dependendo do contexto; por exemplo, na arquitetura mostrada na Figura 6, o nó do meio da imagem é ao mesmo tempo servidor (de requisições vindas de clientes HTTP, FTP e Web-DAV) e cliente (obtendo dados do servidor de dados).

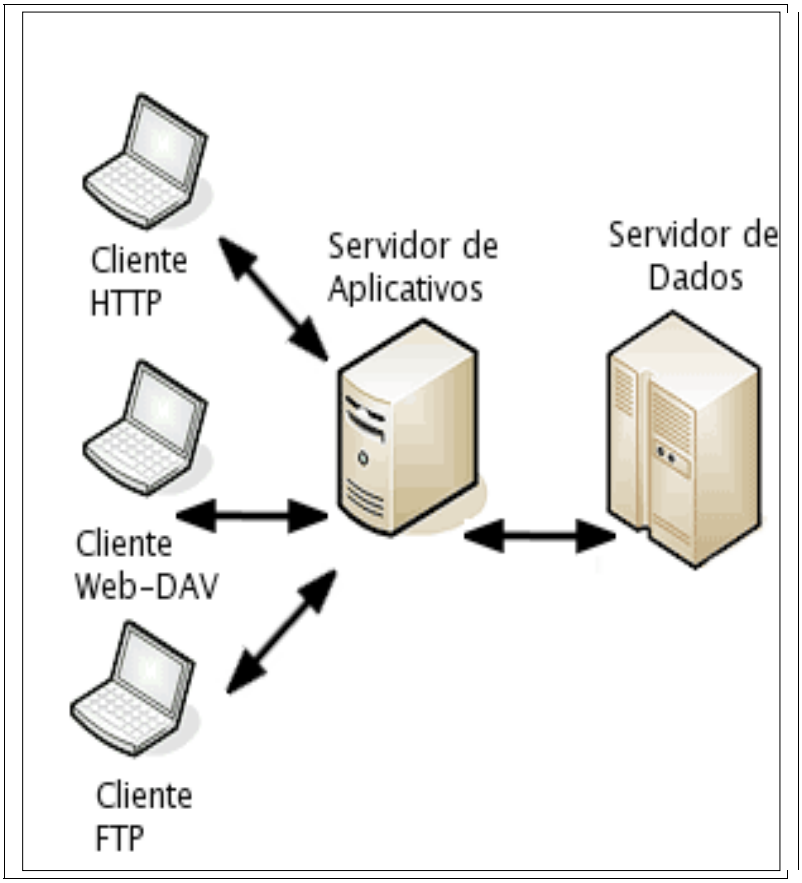


Figura 6. Cliente / Servidor em dois contextos

Uma variante comum da arquitetura cliente / servidor é a arquitetura Peer-to-Peer (ou P2P), que trata cada nó da rede como cliente e como servidor ao mesmo tempo, utilizando assim recursos como largura de banda, conectividade e processamento dos participantes da rede.

3.2.2 Gateways

Um *gateway* é um nó de rede (seja um sistema de hardware ou um aplicativo de software) que pode prover recursos como comunicação entre diferentes protocolos, isolamento de falhas, tradução de endereços de rede e / ou tradução de

sinais, conforme for necessário para disponibilizar interoperabilidade entre sistemas (LUCAS, 2002).

Também conhecido como proxy em alguns contextos, um *gateway* atua como intermediário entre diferentes nós de uma rede; no caso de sistemas de bancos de dados, um *gateway* pode ser responsável por tarefas como balanceamento de carga, *cache*, gerenciamento de tráfego global e redundância de dados.

3.2.2.1 Balanceamento de Carga

Balanceamento de Carga é uma técnica utilizada para distribuir trabalho (geralmente processamento) entre diferentes computadores, processos, discos ou outros recursos para obter o melhor uso destes recursos (BOURKE, 2001).

Segundo Lucas (2002) esta técnica é comumente utilizada em servidores web, para distribuir uniformemente ou não (dependendo das características das máquinas e do balanceador de carga) as requisições feitas pelos clientes para os vários servidores utilizados para disponibilizar um recurso via web, conforme mostra a Figura 7.

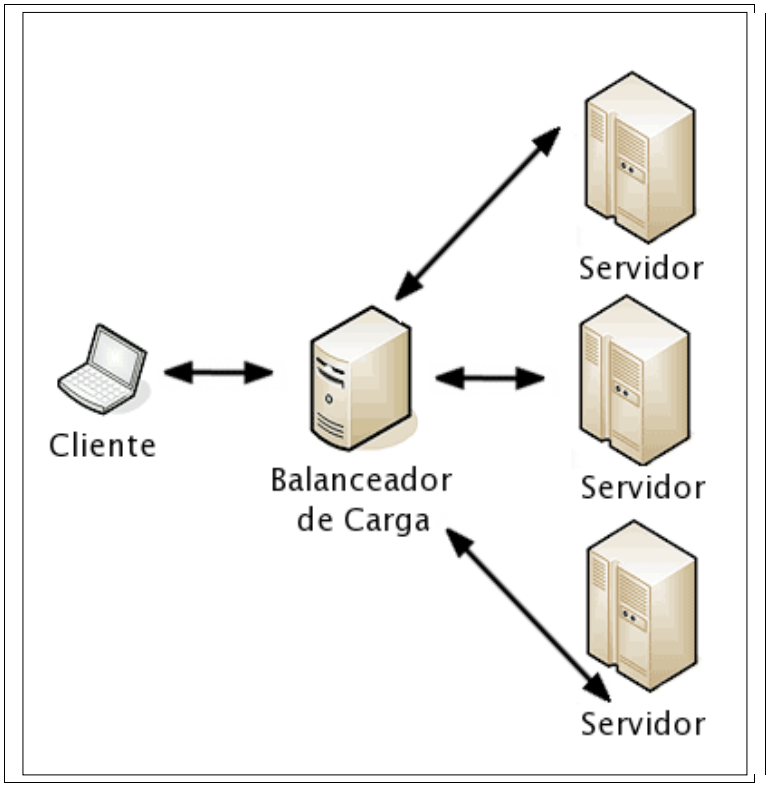


Figura 7. Balanceamento de Carga

3.2.2.2 Cache

Cache é uma coleção temporária de dados duplicados que são armazenados para acesso rápido. Os dados são originados de um local para o qual o acesso é mais lento, comparando com o acesso ao *cache*, seja por causa de impedância de rede ou tempo de processamento.

O *cache* é muito útil no caso de servidores de dados e servidores web, diz Bourke (2001); os clientes podem manter cópias dos seus dados mais acessados localmente, evitando o custo de processamento e largura de banda que seriam usados para cada requisição. Desse modo, os clientes acessam os dados localmente e, de tempos em tempos verificam a validade do *cache*; quando o *cache* se tornar inválido (ou

seja, quando os dados foram alterados no servidor) os clientes fazem uma nova requisição para alimentar novamente o *cache*.

3.2.2.3 Gerenciamento de Tráfego Global

Alguns recursos, como servidores web e servidores de bancos de dados, precisam ser acessados por clientes de todo o mundo, o que torna o acesso mais lento para clientes que estiverem geograficamente mais distantes dos servidores; no pior caso, algumas regiões podem ficar totalmente sem acesso por falha de rotas, backbones, etc (LUCAS, 2002).

O Gerenciamento de Tráfego Global é um tipo de balanceamento que distribui a carga entre servidores geograficamente distribuídos, assim eliminando os problemas de indisponibilidade e tornando os serviços mais rápidos para resposta aos clientes.

3.3 PROTOCOLOS DISTRIBUÍDOS DE DISPONIBILIDADE

Protocolos distribuídos de disponibilidade são utilizados para manter a atomicidade e durabilidade de transações distribuídas que executam sobre um determinado número de bancos de dados.

Para facilitar a explicação, Oznu e Valduriez (1999) assumem que o nó que originou uma transação é o processo que executa suas operações. Esse processo é chamado coordenador. O coordenador comunica-se com processos chamados

participantes que o auxiliam na execução das operações da transação.

3.3.1 Componentes de Protocolos Distribuídos de Disponibilidade

As técnicas de alta disponibilidade usadas por bancos de dados distribuídos consistem em protocolos de commit, terminação e recuperação. Os protocolos de commit e recuperação funcionam da mesma forma que em bancos de dados centralizados, mas o protocolo de terminação só existe em bancos de dados distribuídos. O protocolo de terminação atua quando um nó apresenta falhas no meio da execução de uma transação; o protocolo de terminação gerencia os demais nós para terminarem essa execução (OZNU; VALDURIEZ, 1999).

O requisito primário de um protocolo de commit é que ele mantenha a atomicidade das transações distribuídas. Isso significa que mesmo quando a execução de uma transação envolva múltiplos nós e alguns deles apresentam falhas, a transação deve aplicar os efeitos de todas as operações ou voltar ao estado anterior (BESTAVROS; WANG, 1994).

3.3.2 Protocolo Two-Phase Commit

Two-Phase Commit (2PC) é um protocolo simples que assegura o commit atômico de transações distribuídas. Ele faz com que todos os nós envolvidos em uma transação concordem em fazer commit na transação antes que seus efeitos sejam

aplicados permanentemente. Existem várias razões pelas quais essa sincronização de informações entre os nós seja necessária, como por exemplo *deadlocks* que façam com que um nó aborte uma transação e conflitos de leitura (BESTAVROS; WANG, 1994).

As regras básicas para definir se será feito commit de uma transação são:

a) se apenas um participante votar para abortar uma transação, o coordenador deve chegar a uma decisão global de abort.
b) se todos os participantes votarem para fazer commit de uma transação, o coordenador deve chegar a uma decisão global de commit.

A Figura 8 mostra as possíveis transições de estado do protocolo 2PC.

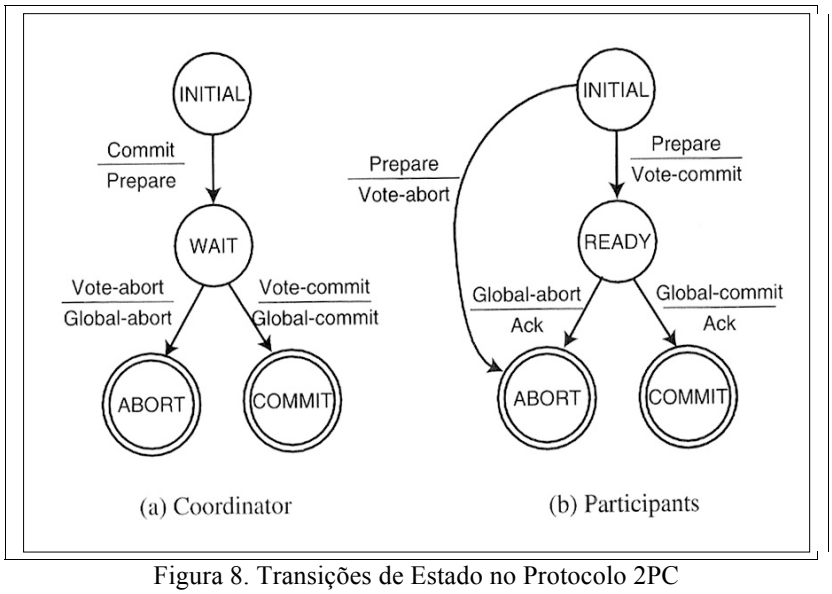


Figura 8. Transições de Estado no Protocolo 2PC
Fonte: OZNU, M. T.; VALDURIEZ, P. (1999)

Existem ainda duas variações do protocolo 2PC, chamadas Abort Presumido e Commit Presumido. O protocolo de Abort Presumido é otimizado para gerenciar transações somente leitura e transações que façam pouca escrita; o protocolo de Commit Presumido é otimizado para gerenciar transações que lidam mais com escrita (OZNU;

VALDURIEZ, 1999).

3.3.3 Protocolo Three-Phase Commit

Three-Phase Commit (3PC) é um protocolo derivado do 2PC que possui uma característica muito importante: é não-bloqueante (assíncrono). A observação básica feita pelos criadores do 3PC sobre o 2PC é que, enquanto um estado está em "preparado para commit", outros estados podem estar tanto como "commit" ou "abort".

Essa diferença pode ser observada na Figura 9, onde é mostrado um novo estado entre WAIT e COMMIT, que funciona como um estado temporário onde o processo está preparado para o commit mas ainda não o fez. Este novo estado elimina determinadas violações ao assincronismo que podem acontecer no protocolo 2PC.

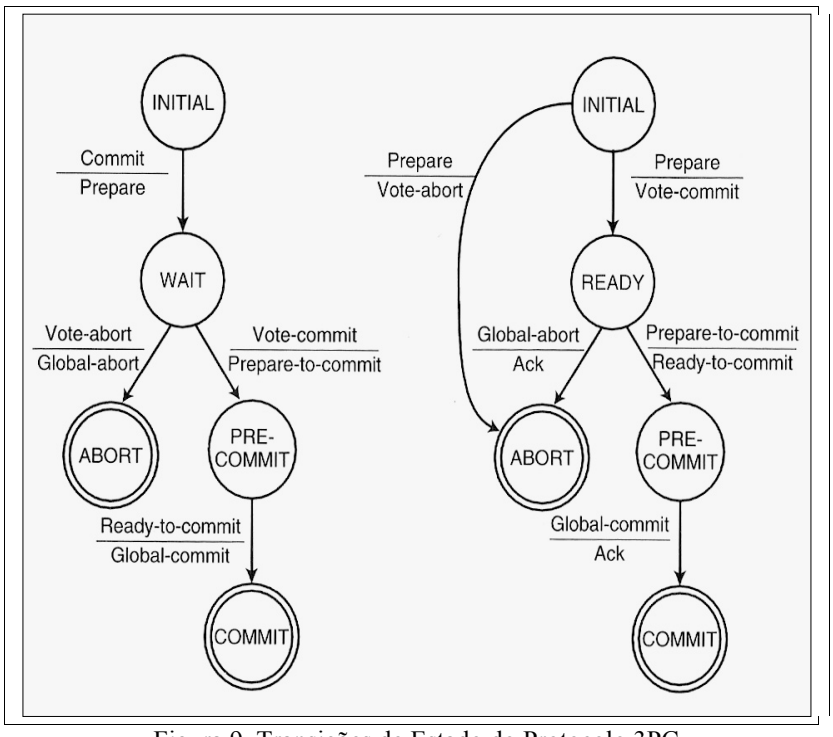


Figura 9. Transições de Estado do Protocolo 3PC
Fonte: OZNU, M. T.; VALDURIEZ, P. (1999)

4 BANCOS DE DADOS ORIENTADOS A OBJETO E O ZODB

A computação trouxe para o mundo meios mais rápidos, fáceis e seguros de armazenar dados e informações. O equivalente a uma grande biblioteca pode ser gravado em uma mídia somente para leitura (como um DVD) e replicado de uma maneira muito simples e rápida.

O armazenamento, entretanto, não é o principal problema; as áreas de pesquisa com maior concentração são as relacionadas à organização, relacionamento e busca.

Apesar da teoria de bancos de dados ser uma das áreas com mudança mais lenta, ela está mudando. Nos últimos tempos apareceram duas grandes vertentes: o uso de um novo modelo de dados lógico (o modelo orientado a objeto) e a integração de elementos orientados a objeto em bancos de dados relacionais (HARRINGTON, 2000).

Mesmo sendo muito menos aceito pelo público geral do que bancos de dados relacionais, os bancos de dados orientados a objeto já são maduros o suficiente para fazer parte de projetos de missão crítica de governos e grandes corporações (PRODASEN, 2007).

4.1 MODELOS DE DADOS

Antes do desenvolvimento do primeiro banco de dados, o acesso aos dados era gerenciado por aplicações que acessavam arquivos no sistema de arquivos do

sistema operacional. Os problemas relacionados a integridade dos dados e a inabilidade de representar relacionamentos lógicos entre os dados logo fez com que o primeiro modelo de dados fosse criado - o modelo hierárquico (HARRINGTON, 2000).

4.1.1 Modelo Hierárquico

O modelo hierárquico, que foi implementado primeiramente pelo *Information Management System* (IMS) da IBM, permite apenas relacionamentos um-para-um e um-para-muitos; não há uma maneira nativa de usar relacionamentos muitos-para-muitos, explica Harrington (2000).

Bancos de dados hierárquicos são navegáveis; isso significa que o acesso aos dados somente acontece por meio de relacionamentos predefinidos. Além disso, o acesso a hierarquia é tipicamente por meio da entidade no topo da hierarquia (a raiz) e deve obedecer a ordem hierárquica.

O benefício de um banco de dados hierárquico é que sua natureza navegável faz o acesso muito rápido quando é seguido um relacionamento predefinido. Entretanto, a rigidez do modelo de dados - em particular, a inabilidade de dar a uma entidade múltiplos pais e a falta de acesso direto aos dados - torna-o inviável em ambientes onde buscas muito específicas são necessárias.

A Figura 10 mostra uma das possíveis saídas para evitar o problema da necessidade de relacionamentos muitos-para-muitos: a duplicação de entidades (no caso, “Entrada do Catálogo”).

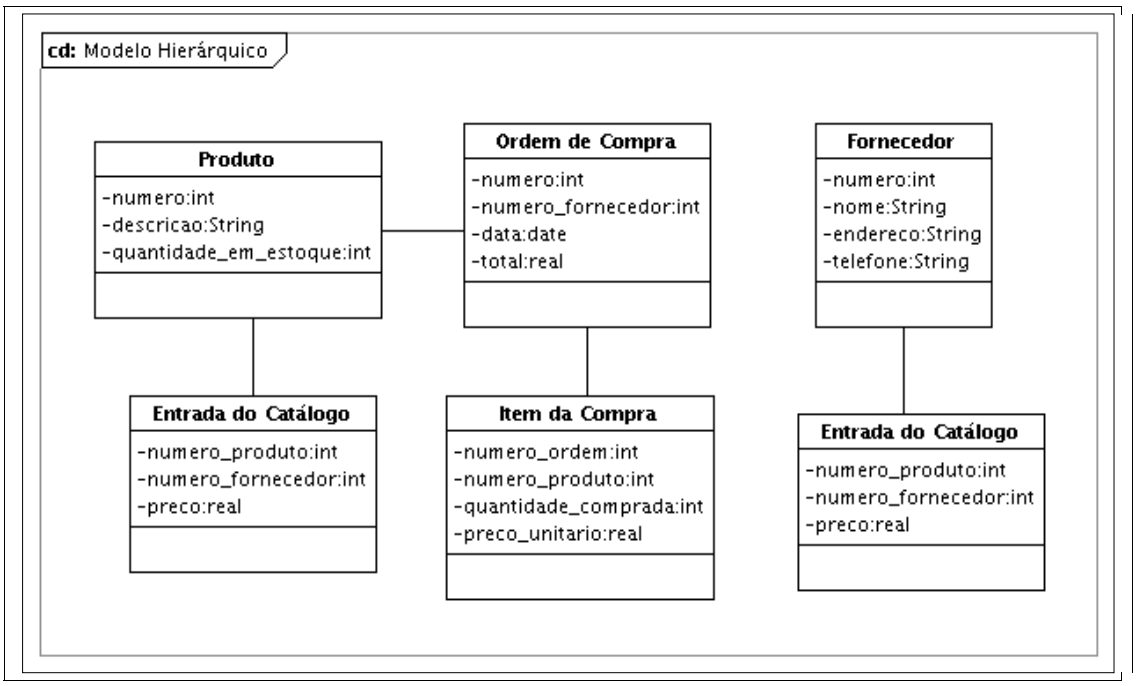


Figura 10. Modelo Hierárquico

4.1.2 Modelo de Rede

Muito antes da primeira rede de computadores, alguns projetistas de bancos de dados criaram dois modelos baseados em redes - o modelo de rede simples e o modelo de rede complexo. O objetivo desses modelos é eliminar as restrições impostas pelo modelo hierárquico de dados (HARRINGTON, 2000).

4.1.2.1 Modelo de Rede Simples

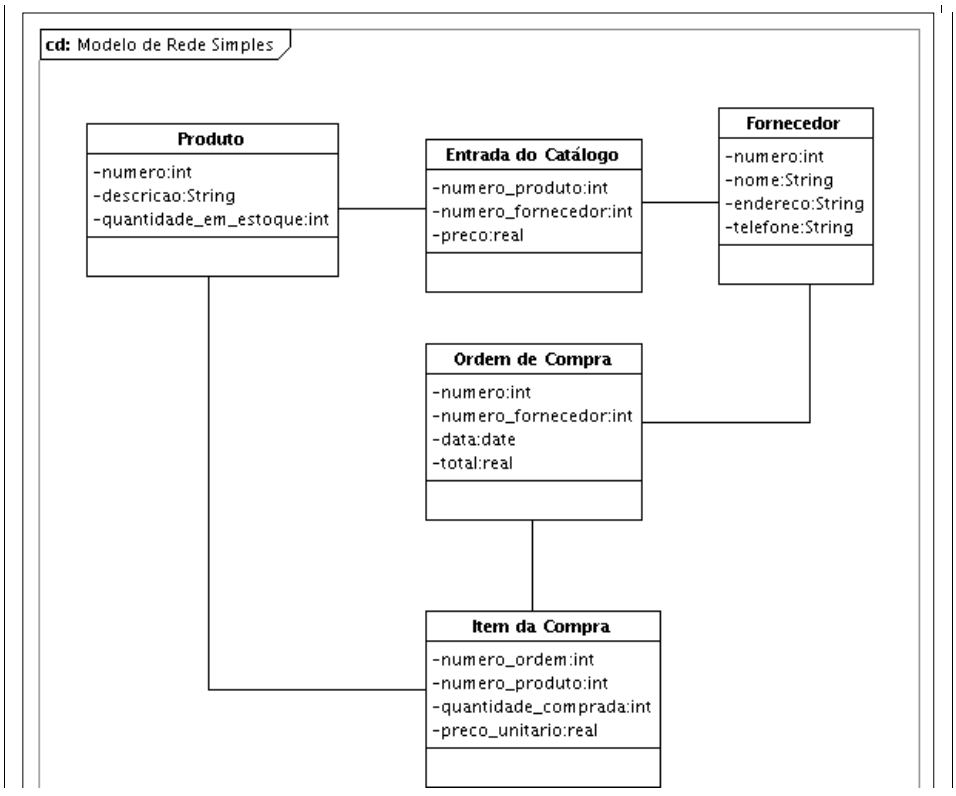
No modelo de rede simples, todos os relacionamentos são um-para-um ou um-para-muitos; relacionamentos de muitos-para-muitos não são permitidos, mas é

possível que uma entidade tenha múltiplos pais. Isso elimina os problemas de entidades duplicadas desnecessariamente e de restrições de acesso.

Bancos de dados de rede simples são navegáveis: a maior parte dos acessos acontece por meio de relacionamentos predefinidos. Um tipo de acesso direto as instâncias de entidades é suportado por meio de *hashing*, mas pelo fato de hashing afetar a organização física dos dados, na prática apenas uma entidade da hierarquia de entidades pode ser habilitada para acesso rápido via hash. Todas as outras instâncias de entidades só poderão ser acessadas usando a hierarquia.

Por causa de suas características, bancos de dados de rede simples tem boa performance quando o acesso segue relacionamentos mas tem má performance se buscas manuais são feitas fora da tabela hash. Dependendo do projeto do banco de dados, buscas específicas podem ser difíceis de serem satisfeitas, diz Harrington (2000).

A Figura 11 mostra que o uso de múltiplos pais na entidade "Entrada do Catálogo", evitando a necessidade de duplicar essa entidade.



4.1.2.2 Modelo de Rede Complexo

O modelo de dados de rede complexo é parecido com o modelo de rede simples, mas permite implementação direta de relacionamentos muitos-para-muitos, o que elimina a necessidade de entidades compostas apenas para representar relacionamentos. Apesar disso entidades compostas se tornam necessárias quando existe a necessidade de serem armazenados dados sobre os relacionamentos.

Segundo Harrington (2000) manter relacionamentos diretos de muitos-para-muitos é muito complicado, o que fez com que nenhum banco de dados de rede complexo tivesse sucesso comercial.

A Figura 12 mostra o uso de um relacionamento direto de muitos-para-muitos.

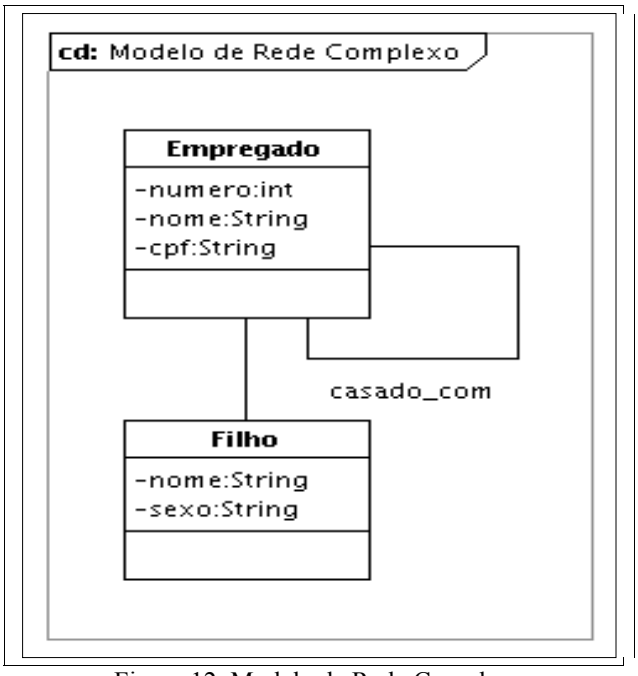


Figura 12. Modelo de Rede Complexo

4.1.3 Modelo Relacional

O modelo relacional foi uma revolução no projeto de bancos de dados; enquanto os modelos anteriores eram similares, com alterações apenas em tipos de relacionamento e restrições de acesso, o modelo relacional tem uma implementação significativamente diferente.

Bancos de dados relacionais não são navegáveis, e seus relacionamentos não são armazenados diretamente no banco de dados - são utilizados conjuntos de chave primária / chave estrangeira para obter os relacionamentos conforme necessário. Por conta de suas características o uso de buscas específicas é muito facilitado; além disso o banco de dados é mais fácil de ser alterado, porque na maioria dos casos as modificações podem ocorrer com o banco de dados em uso (DYE, 1999).

Uma desvantagem do modelo relacional para os modelos anteriores é a performance. Os modelos hierárquico e de rede são muito atrelados ao armazenamento físico dos dados, e a estrutura de dados para relacionamentos realmente fazem parte do banco de dados, o que torna o acesso a dados relacionados muito mais rápido.

Apesar da desvantagem de performance (que se torna quase irrelevante com o poder computacional que temos hoje), o modelo relacional se tornou o padrão de-facto. As necessidades de negócio de apoiam fortemente das buscas específicas, o que dá uma larga vantagem aos bancos de dados relacionais (URBANO, 2003).

A Figura 13 mostra o uso de uma entidade intermediária usando um conjunto de chaves para definir um relacionamento de muitos-para-muitos.

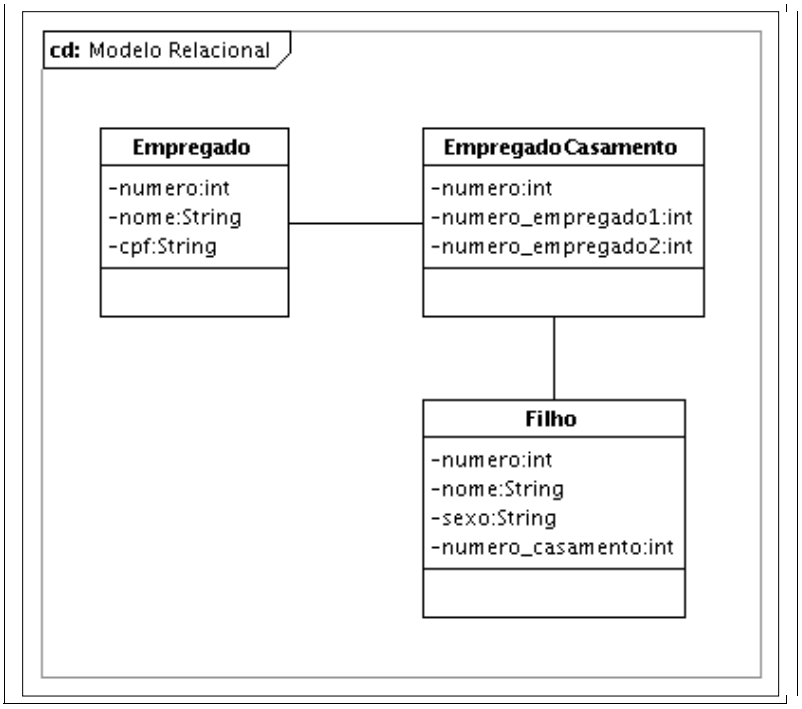


Figura 13. Modelo Relacional

4.1.4 Modelo Orientado a Objeto

O desenvolvimento do paradigma orientado a objeto alterou fundamentalmente a maneira de olhar para os dados e os procedimentos que são aplicados sobre esses dados. Tradicionalmente, dados e procedimentos são armazenados separadamente, estando os dados em um banco de dados e os procedimentos na aplicação. A orientação a objeto, por outro lado, combina os procedimentos de uma entidade com os seus dados (BEERI; MILO, 1991).

Essa combinação geralmente é considerada um avanço no gerenciamento de dados. Entidades tornam-se unidades auto-contidas que podem ser reutilizadas e

movidas com relativa facilidade. Ao invés do comportamento de uma entidade ser atrelado a uma aplicação específica, o comportamento é parte da entidade em si, então não importa onde a entidade for usada, o comportamento será previsível.

Bancos de dados orientados a objeto são navegáveis: o acesso aos dados acontece através de relacionamentos armazenados com os dados em si. Isso é um passo para trás - bancos de dados orientados a objeto não são tão adaptados ao uso de buscas específicas como bancos de dados relacionais (HARRINGTON, 2000).

A Figura 14 demonstra o uso de uma hierarquia orientada a objetos, onde a entidade “Empresa” é composta por várias entidades “Empregado”.

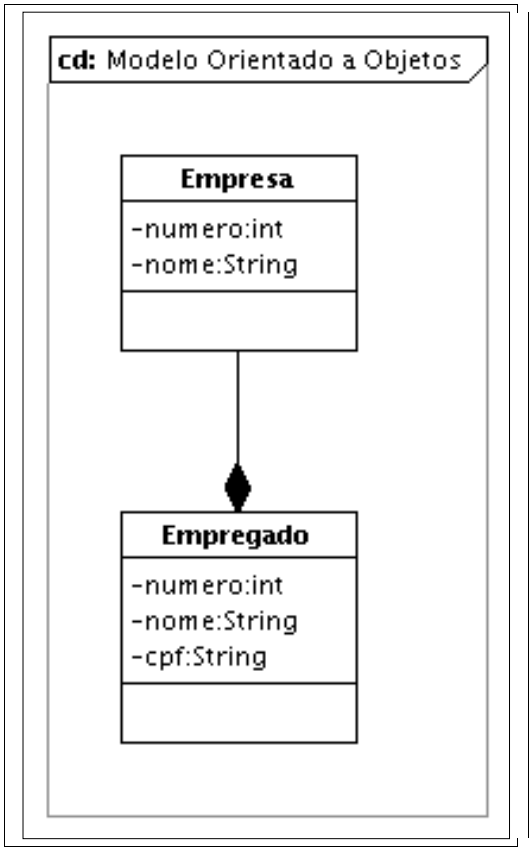


Figura 14. Modelo OO

4.2 ZODB

O ZODB é uma implementação open-source do modelo de dados orientado a objetos. ZODB significa *Zope Object Database* e foi criado inicialmente para uso conjunto com o Zope, um servidor de aplicações orientado a objetos construído utilizando a linguagem de programação Python.

Assim como a maior parte dos bancos de dado orientados a objeto, o ZODB só pode ser utilizado com a linguagem de programação Python. Python é uma linguagem de altíssimo nível (do inglês *Very High Level Language*, ou VHLL) orientada a objetos, de código aberto, multiplataforma e com tipagem dinâmica e forte (FERRI, 2002).

O ZODB é um software robusto e maduro, com mais de 10 anos de existência e é utilizado em grandes corporações, como CIA, NASA, Bank Boston e vários governos, inclusive o brasileiro. O Plone, vencedor de vários prêmios na categoria de gerenciamento de conteúdo e um dos projetos de software livre mais usados no mundo, utiliza o ZODB (PRODASEN, 2007).

Além de implementar o modelo de dados básico - classes, objetos, relacionamentos - o ZODB provê uma arquitetura extensível de diferentes tipos de armazenamentos (*Storages*, na terminologia do ZODB) e uma implementação de um catálogo de objetos (ZCatalog) que permite buscas específicas sem a necessidade de caminhar sobre a hierarquia de objetos (ZODB, 2007).

4.2.1 Arquitetura

Segundo Kuchling (2006) os principais componentes da arquitetura do ZODB são:

a) *Connection* (Conexões): Faz o *cache* de objetos e move-os para dentro e fora de um *Storage*. Um programa com múltiplas *threads* normalmente usa uma instância de *Connection* separada para cada *thread*. Assim, diferentes *threads* podem alterar os objetos e finalizar suas transações independentemente;
b) *DB* (Bases de Dados): Atua entre uma ou mais *Connections* e um *Storage*. Uma instância de *DB* é usada por processo;
c) *Storage* (Armazenamentos): É o nível mais baixo da arquitetura que gerencia escrita e leitura de objetos a partir de um armazenamento confiável. Existem alguns tipos padrão de *Storage*, como *FileStorage*, que usa o sistema de arquivos do sistema operacional, *BDBFullStorage*, que usa o banco de dados *BerkeleyDB* e *ClientStorage*, que permite o uso do ZODB em rede.

4.2.2 ZEO

O Zope Enterprise Objects (ZEO) é um módulo que disponibiliza uma arquitetura distribuída para instalações do ZODB por meio de um *Storage* chamado *ClientStorage*. O *ClientStorage* não faz a escrita em uma mídia física, apenas encaminha todas as requisições através da rede para um servidor. Este, por sua vez, executa uma instância da classe *StorageServer*, simplesmente agindo como um *front-*

end para algum *Storage* físico (como *FileStorage*, por exemplo) (ANTONIO, 2001). A Figura 15 demonstra um exemplo de uso de ZEO, onde três clientes ZEO acessam a mesma base de dados em um servidor ZEO (também chamados de ZSS / ZEO *Storage Server* por alguns autores).

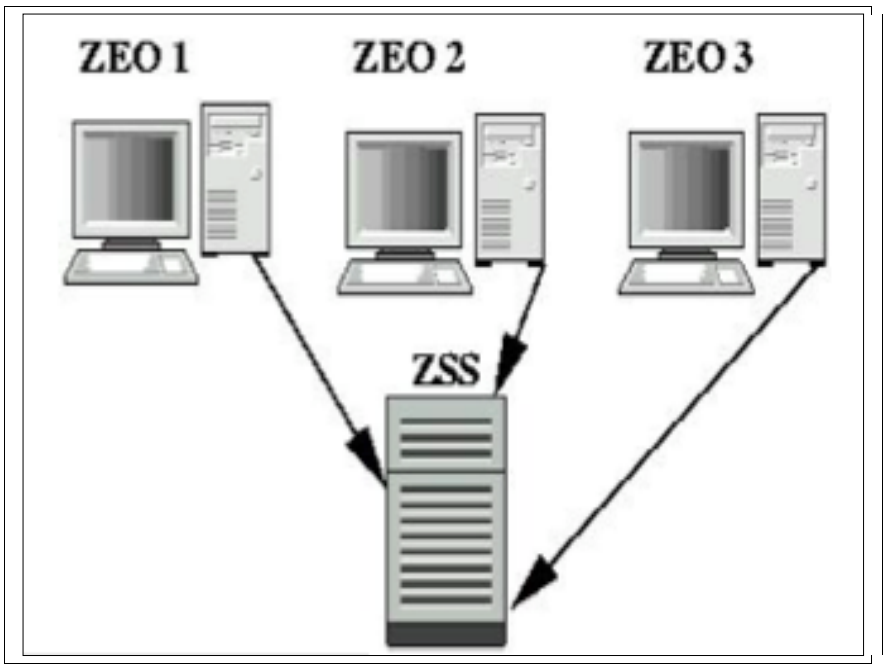


Figura 15. Arquitetura ZEO
Fonte: ANTONIO, M. S. (2001)

O ZEO também disponibiliza recursos para *cache* dos dados nos clientes e resolução de conflitos.

4.2.2.1 ZRPC

Toda a comunicação entre os clientes ZEO e o servidor ocorrem utilizando o protocolo RPC usando uma implementação própria chamada ZRPC. O ZRPC faz parte do ZEO (KUCHLING, 2006), embora seja auto-contido.

O ZRPC é composto dos seguintes módulos:

a) *client*: Gerencia criação de conexões com o servidor remoto;

b) *connection*: *Dispatcher* de objetos;

c) *log*: Funções para registro de *log*;

d) *error*: Definição de exceções que podem ser lançadas pelo ZRPC;

e) *marshal*: Gerencia diferenças entre versões do protocolo interno do ZEO;

f) *server*: Gerencia conexões vindas de clientes;

g) *smac*: Gerencia comunicação assíncrona;

h) *trigger*: Controle de recursos em comunicação assíncrona.

A autenticação no ZRPC é opcional e é feita usando uma versão modificada do protocolo HMAC (*Keyed-Hashing for Message Authentication*), definido na RFC 2104 como um protocolo criptográfico utilizado para autenticação em troca de mensagens baseado em um algoritmo específico e uma chave secreta (KRAWCZYK; BELLARE; CANETTI, 1997).

O ZRPC utiliza-se de chamadas síncronas (bloqueantes) e assíncronas (não-bloqueantes). As mensagens enviadas pelo ZRPC são compostas pelos seguintes campos: identificador, *flags*, nome do método e argumentos.

Quando um dos lados da conexão (o cliente) faz uma chamada de procedimento, ele envia uma mensagem um novo identificador. O outro lado (o servidor) responde com uma mensagem utilizando o mesmo identificador, a *string* ".reply" como o nome do método e o retorno do método no campo dos argumentos. Ambos os lados podem iniciar chamadas de procedimentos.

4.2.2.2 Alta disponibilidade no ZODB e o ZRS

Do ponto de vista da alta disponibilidade o ZEO tem um problema: não existe a possibilidade de utilizar múltiplos servidores, apenas múltiplos clientes. Esse problema é solucionado pelo ZRS (*Zope Replication Services*), que permite o uso de até dois servidores de dados usando a abordagem *Primary-Backup*; se o servidor primário falha, o servidor de backup assume o seu lugar (ZOPE.COM, 2007).

O problema do ZRS é que a arquitetura é limitada para até dois servidores, além do software ser proprietário e cobrado por CPU.

4.2.3 Controle de Transações

O ZODB utiliza *Multi-version concurrency control* (MVCC) como controle de transações. MVCC é uma técnica padrão utilizada para evitar conflitos entre leituras e escritos do mesmo objeto. MVCC garante que cada transação veja uma versão consistente do banco de dados através da leitura de dados não-concorrentes de objetos alterados por transações concorrentes. É uma técnica bem comum na implementação de banco de dados - PostgreSQL, por exemplo, usa MVCC (POSTGRES, 2007)

A técnica usada pelo ZODB é uma variante otimista do MVCC que provê consistência em tempo de execução. Ao invés de disparar um erro para sinalizar um problema de consistência, o ZODB automaticamente lê dados não-concorrentes.

Conflitos de leitura ocorrem quando o banco de dados não pode disponibilizar uma visão consistente dos dados para a aplicação. Especificamente, o

ZODB sempre lê a revisão mais recente de um objeto. Se a revisão mais recente do objeto foi escrita depois que a transação iniciou, não é possível garantir consistência.

Esses conflitos de leitura fazem as transações abortarem e recomeçarem todo o trabalho. Transações cujo tempo de execução é muito longo são mais propensas a apresentar conflitos; o MVCC então é um grande benefício, pois evita que esse tipo de transação seja reiniciada a cada conflito encontrado (ZODB, 2007).

5 TRABALHOS CORRELATOS

Antonio publicou em 2001 um artigo chamado "*Distributed Zope Clustering*". Neste trabalho ele investigou o problema da limitação de bases de dados em ambientes ZEO.

A solução proposta consiste na execução de um *daemon* para cada servidor ZEO (chamados de ZSS no artigo), que comunicam-se entre si verificando alterações usando diferentes algoritmos para a sincronização de dados. A arquitetura da solução é apresentada na Figura 16, onde podemos ver três servidores ZEO / ZSS fazendo sincronização entre si e três clientes ZEO acessando todos os servidores.

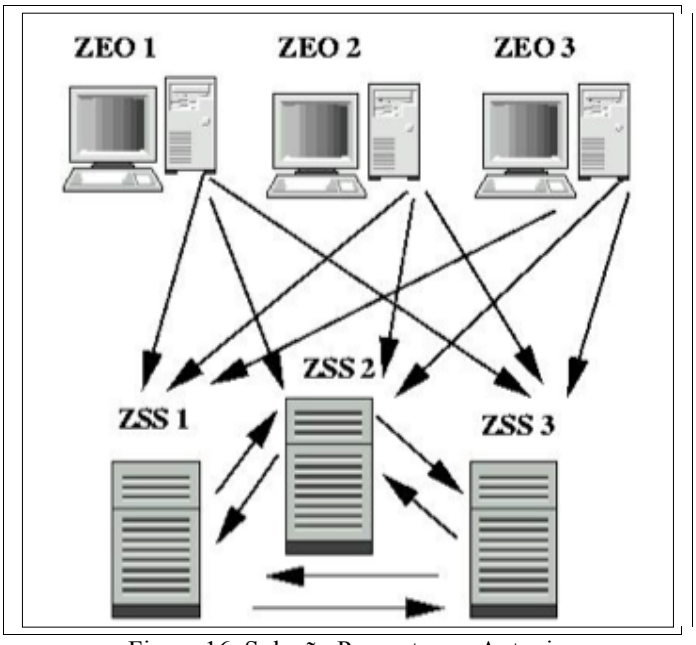


Figura 16. Solução Proposta por Antonio
Fonte: ANTONIO, M. S. (2008)

Segundo Antonio (2001), a solução proposta mostra um grande avanço no uso distribuído de ZODB, pois assim clientes e servidores ZEO distribuídos geograficamente podem se comunicar de maneira mais eficiente, escolhendo o nó mais

próximo.

6 AMBIENTE DE TESTES

Para tornar possível o principal objetivo deste projeto, o projeto ZEORaid foi aprimorado com a implementação de funcionalidades e correção de erros, e um aplicativo foi desenvolvido e configurado para demonstrar o uso de uma arquitetura distribuída com o ZODB. Após a realização destas etapas, foi possível executar os testes e analisar os resultados obtidos pela aplicação desenvolvida. As seções seguintes descrevem as ferramentas utilizadas no desenvolvimento e nos testes.

6.1 PYTHON

Python é uma linguagem de alto nível, interpretada, orientada à objetos e dinâmica. Suas estruturas de alto nível, combinadas com sua tipagem dinâmica a faz muito atrativa para desenvolvimento de largos aplicativos assim como para uso como linguagem de script. Ela foi desenvolvida pelo holandês Guido Van Rossum, no final de 1990. A linguagem surgiu enquanto Van Rossum passava o tempo entre o Natal de 1990 e o ano novo de 1991, trabalhando na linguagem ABC, a qual ele havia participado do grupo que a desenvolvera. Seu nome surgiu a partir do famoso show "*Monty Python's Flying Circus*" (LUTZ, 2006).

O interpretador do Python é facilmente extensível, incorporando novas funções e tipos de dados implementados em C ou C++ (ou qualquer outra linguagem acessível a partir de C). Python também se adequa como linguagem de extensão para customizar aplicações (PYTHON, 2008).

A biblioteca padrão inclui módulos para processamento de texto e expressões regulares, protocolos de rede, acesso aos serviços do sistema operacional, criptografia, interface gráfica, web, etc. Além da biblioteca padrão, existe uma grande variedade de extensões adicionais para todo tipo de aplicação.

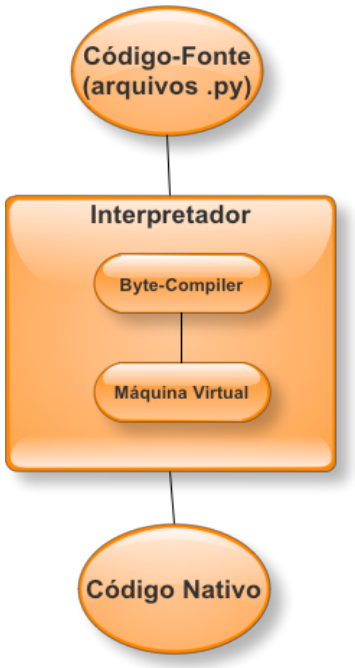


Figura 17. Execução de um Programa Python

A Figura 17 mostra o ciclo de interpretação de um programa Python: um arquivo de texto contendo código-fonte é passado para o interpretador, que por sua vez ativa o *byte-compiler* para traduzir o código-fonte para um objeto intermediário chamado *bytecode*. O *bytecode* é executado pela máquina virtual que por sua vez o converte para código nativo do sistema operacional (LUTZ, 2006).

6.1.1 ftplib

ftplib é uma biblioteca criada para a linguagem Python que atua como um cliente FTP, tornando possível o uso de todas as operações cobertas pelo protocolo FTP de uma forma programática (PYTHON 2008). A ftplib foi utilizada para simular acessos de escrita na aplicação de testes, fazendo *upload* de arquivos para o servidor Zope.

```
from ftplib import FTP
ftp = FTP('ftp.cwi.nl')   # conecta ao host na porta padrão
ftp.login()               # efetua login como usuário anônimo
ftp.retrlines('LIST')     # lista o conteúdo do diretório
ftp.retrbinary('RETR README', open('README', 'wb').write)
ftp.quit()
```

Figura 18. Código usando ftplib

A Figura 18 mostra um exemplo simples de código usando ftplib para executar operações como um cliente FTP.

6.2 ZOPE

Zope é um ambiente para construção e gerenciamento de aplicações web com foco em gerência de conteúdo. Ele foi desenvolvido pela Digital Creations Inc., hoje Zope Corporation (WEITERSHAUSEN, 2007).

A Digital Creations incorporou à distribuição três pacotes de software *open source* para suportar publicação na Web: Bobo, Document Template, e BoboPOS. Estes três componentes desenvolveram-se no coração do Zope, fornecendo o Object Request Broker, a linguagem de scripting DTML e a base de dados orientada à objetos ZODB.

Posteriormente a Digital Creations desenvolveu um servidor de aplicações comercial baseado nestes três componentes Open Source. Este produto chamou-se Principia. Estimulada por um investidor externo, a Digital Creations abriu o código fonte do Principia em Novembro de 1998 e renomeou-o como Zope.

Como possui seu próprio servidor Web, o ZServer (Zope Server), o Zope dispensa a presença de qualquer outro servidor. Esse servidor mapeia e acessa os objetos passados na URL do navegador através do ORB. Isso implica em que todo objeto, com permissão para tal, pode ser acessado via URL, o que lhe confere uma poderosa estrutura para trabalho na WWW (WEITERSHOUSEN, 2007).

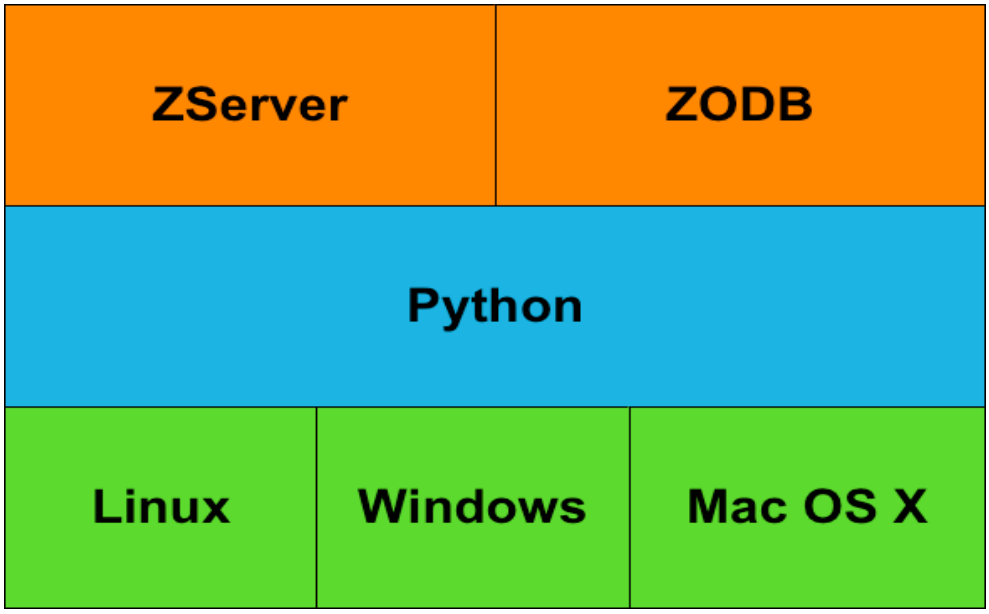


Figura 19. Arquitetura do Zope

A Figura 19 mostra que os principais componentes do Zope, o ZServer e o ZODB, são executados sobre a máquina virtual Python, que por sua vez é executada sobre um sistema operacional. Isso demonstra que, assim como a linguagem Python, o Zope é multiplataforma.

6.3 PLONE

Plone é um gerenciador de conteúdo criado por Alan Runyan e Alexander Limi em 1999, quando ambos decidiram que poderiam melhorar o CMF em vários aspectos, como usabilidade, acessibilidade e funcionalidade (ASPELI, 2007). O Content Management Framework (CMF) por sua vez, é um framework criado por funcionários da Zope Corporation para o desenvolvimento de aplicações cujo foco é o conteúdo (ASPELI, 2007).

Segundo Aspeli (2007) além de funcionar como um gerenciador de conteúdos o Plone conta com o framework CMF para o desenvolvimento de aplicações que utilizam toda a sua infra-estrutura. Esse framework foi utilizado no desenvolvimento da aplicação de teste, que é executada sobre o Plone.

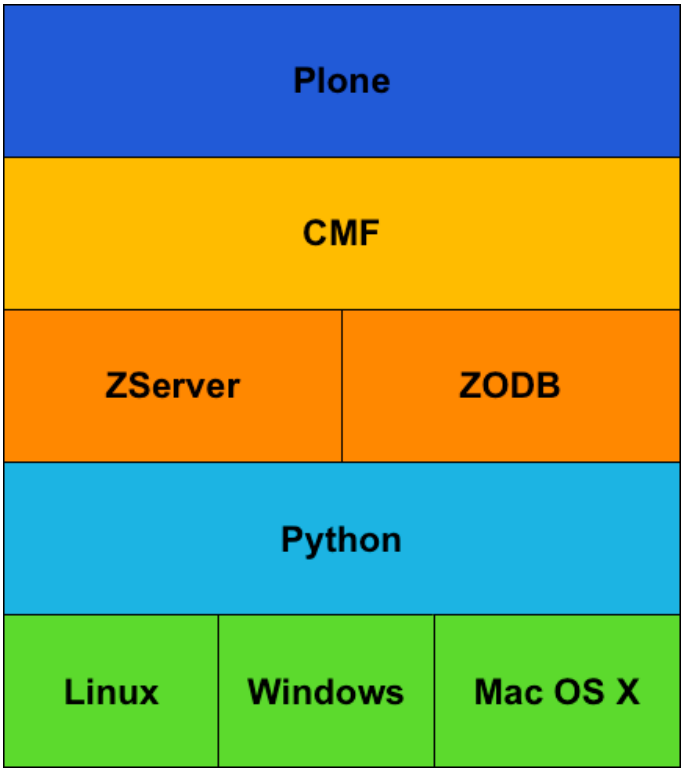


Figura 20. Arquitetura do Plone

A Figura 20 mostra a "pilha tecnológica" do Plone, que é dependente do CMF e que por sua vez é executado sobre o Zope.

6.4 ZEORaid

ZEORaid é um software de código aberto criado na empresa Gocept, Inc. para disponibilizar um novo storage (armazenamento) para o ZODB que utiliza conceitos da arquitetura RAID para fazer replicação de servidores ZEO. Para tornar isso possível é disponibilizado um servidor chamado ZEORaid Server, que mantém dados sobre os servidores ZEO e estende o ZRPC para permitir chamadas de verificação de estado.

A implementação tem como o objetivo usar o máximo possível da infra-

estrutura já existente e disponibilizar recursos para configurar replicação em servidores ZEO da maneira mais simples possível (THEUNE, 2008).

Para aproveitar a infra-estrutura atual, um servidor ZEORaid é criado da mesma forma que um servidor ZEO, mas configurado usando opções específicas. Essas opções fazem com que um servidor ZEORaid atue:

a) como um servidor ZEO quando acessado por clientes ZEO;

b) como um cliente ZEO quando acessando um servidor ZEO.

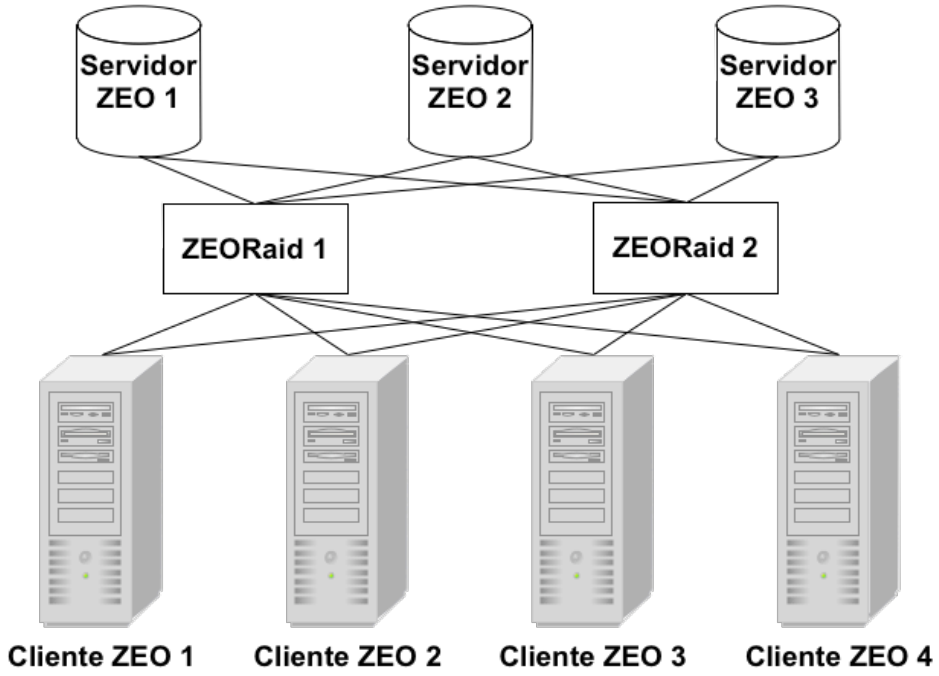


Figura 21. Ambiente Utilizando o ZEORaid

Isso tudo é transparente: nem os clientes ZEO e nem os servidores ZEO precisam de softwares adicionais ou configurações específicas do ZEORaid. A Figura 21 mostra um exemplo disso: os clientes ZEO 1, 2, 3 e 4 estão configurados para acessar os servidores ZEORaid 1 e 2 como se fossem servidores ZEO comuns; os servidores ZEORaid, por sua vez, estão configurados como clientes ZEO e estão

acessando os servidores ZEO 1, 2 e 3 como tais.

O uso de dois servidores ZEORaid ao mesmo tempo ainda é experimental; entretanto, usar apenas um servidor acrescentaria à arquitetura um ponto único de falha, ou seja: se o servidor ZEORaid falhasse a aplicação falharia, e possivelmente dados de transações que estavam ocorrendo ou sendo gravadas seriam perdidos.

O próprio ZEORaid, de certa forma, ainda é experimental. A empresa Google, Inc., através do projeto Google Summer of Code financiou um projeto cujo intuito é aprimorar o ZEORaid. O projeto aconteceu entre 26 de maio e 17 de agosto de 2008, e as tarefas incluíram a otimização do processo de invalidação do *cache*, paralelização das requisições para múltiplos servidores ZEO e criação e remoção de servidores ZEO de maneira simplificada sem parar os serviços (TIEGS, 2008).

Os servidores ZEO e os servidores ZEORaid precisam usar uma versão específica do ZODB: a versão 3.9, que inclui recursos e correções necessárias para a execução do maquinário do ZEORaid. Essa versão do ZODB foi desenvolvida primariamente pela equipe da empresa Gocept, Inc. e ainda não está disponível como versão estável (empacotada).

6.5 APLICAÇÃO DE TESTES

A aplicação desenvolvida foi um site plone para uma empresa (hipotética) de consultoria e desenvolvimento de software chamada XyZ Consultoria. O site conta com uma área institucional, uma área de desenvolvimento (onde existe uma lista de clientes e projetos) e uma área de produtos (onde são apresentados os livros e vídeo-

aulas criadas pelos profissionais da XyZ Consultoria).

A Figura 22 mostra o mapa do site utilizado nos testes.

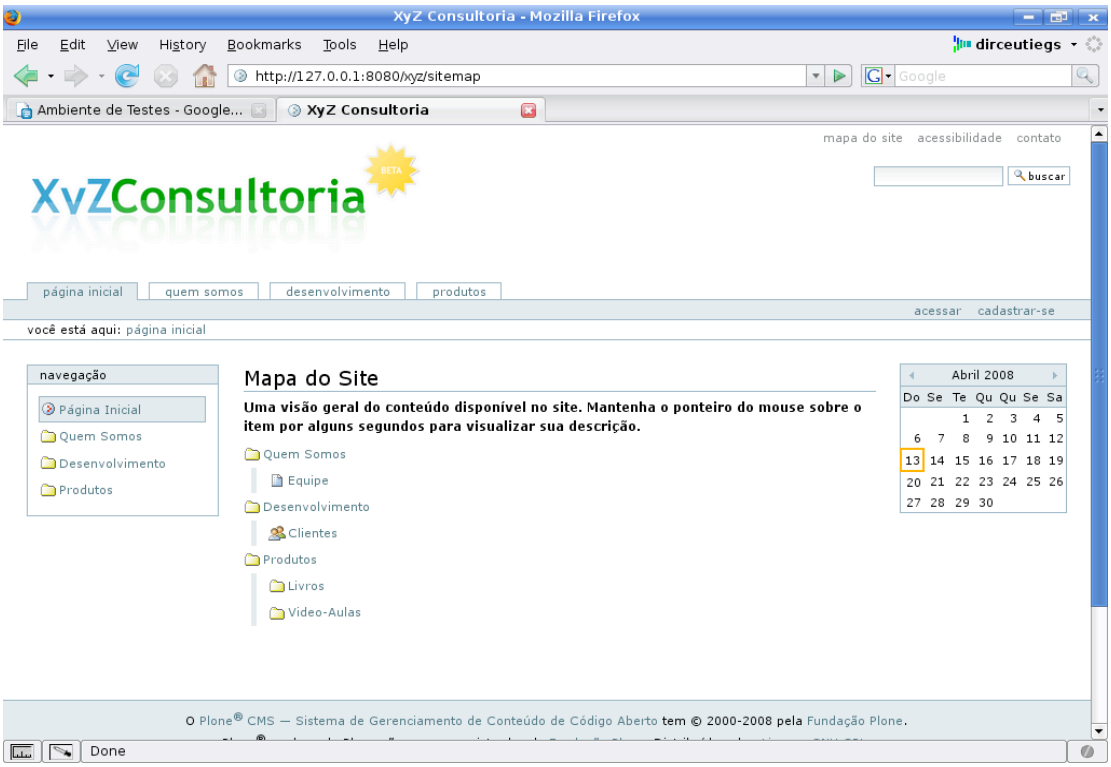


Figura 22. Mapa do Site da Aplicação de Teste

6.5.1 Máquinas de Teste

O desenvolvimento foi feito em um notebook Apple iBook G4 com as seguintes características:

a) processador: PowerPC G4 de 1.2 GHz

b) memória principal: 512 MB;

c) disco rídigo: ATA-6 de 30 GB;

d) rede: 802.11g;

e) sistema operacional: Mac OS X 10.5;

f) hostname: notebook.

Um segundo computador (um desktop) também foi utilizado nos testes, e conta com as seguntes características:

a) processador: Athlon XP 2600+ de 1.9 GHz;

b) memória principal: 512 MB;

c) disco rídigo: SATA de 80 GB;

d) rede: 802.11g;

e) sistema operacional: Kubutu Linux 7.10 com kernel 2.6.22;

f) hostname: desktop.

Os softwares utilizados foram:

a) linguagem de programação: Python 2.4.4;

b) servidor de aplicações: Zope 2.9.8 com ZODB 3.7;

c) framework / CMS: Plone 2.5.4;

d) software de replicação: ZEORaid 1.0b1

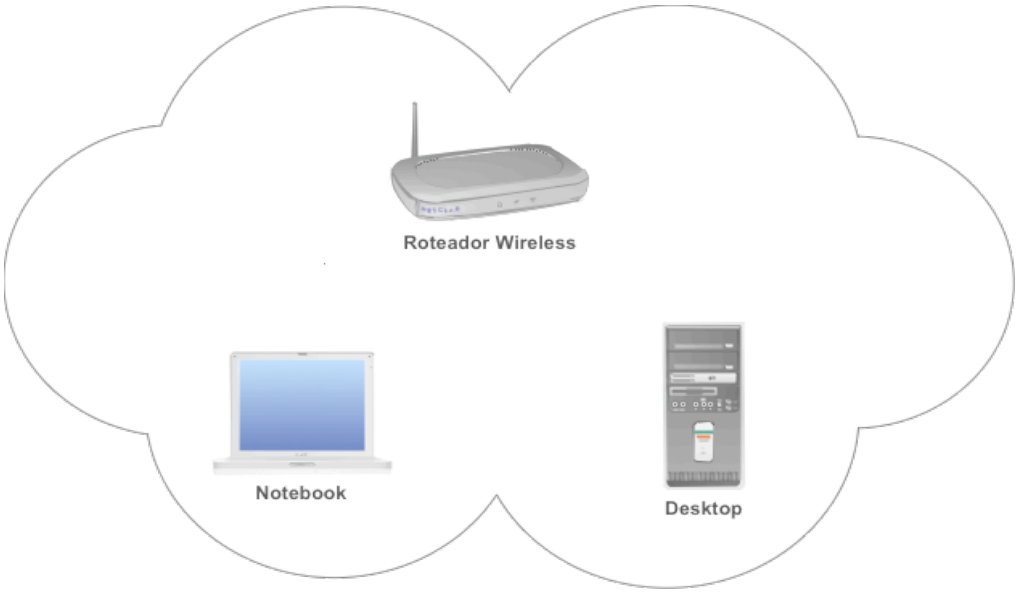


Figura 23. Arquitetura da Rede de Testes

A Figura 23 mostra a arquitetura da rede utilizada nos testes: o notebook e o desktop conectados por um roteador wireless.

O notebook inicialmente foi usado apenas para o desenvolvimento, enquanto o desktop foi usado para servir a aplicação usando um servidor ZEO e um cliente ZEO.

6.5.2 Configuração do Ambiente

Foram montadas quatro configurações para os testes:

a) dois clientes ZEO e um servidor ZEO no desktop;
b) dois clientes ZEO, dois servidores ZEO e um servidor ZEORaid no desktop;
c) um cliente ZEO e um servidor ZEO no notebook, um cliente ZEO e um servidor ZEO no desktop e um servidor ZEORaid no desktop;
d) um cliente ZEO, um servidor ZEO e um servidor ZEORaid no notebook e no desktop.

Essas diversas configurações foram montadas para verificar diversos fatores na utilização do servidor de replicação, como uso de processamento, reposta de chamadas, uso da rede, performance para escrita e performance para leitura.

6.6 TESTES EFETUADOS

A primeira etapa dos testes constituiu a instalação das quatro configurações de rede de processos descritas anteriormente. Os testes utilizando a aplicação desenvolvida foram feitos baseados na seguinte metodologia:

a) 10000 acessos de leitura com 200 requisições simultâneas a cada vez aos clientes ZEO;
b) 200 acessos de escrita com 5 requisições simultâneas a cada vez aos clientes ZEO;
c) 200 acessos de escrita com 5 requisições simultâneas a cada vez aos clientes ZEO durante a recuperação de um servidor ZEO;
d) cada experimento foi repetido 10 vezes para obter-se um resultado médio adequado.

Todos os resultados foram armazenados em arquivos de texto e de imagem para análise comparativa final.

Os testes foram efetuados para detectar possíveis anomalias em uso de memória, processador e rede e também outros problemas quando utiliza-se replicação para ZODB em um ambiente de alta disponibilidade.

Cinco ferramentas foram utilizadas para coletar os dados dos testes:

a) top, ferramenta padrão para monitoramento de uso de recursos como tempo de processador e uso de memória em sistemas Unix;
b) vmstat, presente em sistemas Linux para análise do uso de memória;
c) ab (Apache Benchmark Tool), um software usado para testes de

requisições HTTP;

d) ntop, ferramenta que funciona de maneira semelhante ao “top” mas que monitora uso de rede;

e) teste_escrita_ftp.py, criado pelo acadêmico para gerar acessos de escrita nos clientes ZEO.

6.7 PROBLEMAS ENCONTRADOS

Durante os testes foi encontrado um problema no ZEORaid versão 1.0b1, revisão 86540 que causava erros na resolução de conflitos. O problema só ocorria na última versão do ZODB e era causado pelo uso experimental do recurso de desfazer transações (*Undo*) para *blobs* (grandes arquivos binários armazenados no sistema de arquivos). Este problema foi corrigido.

Outro problema encontrado foi o fato de que, quando um *storage* falha por causa de algum erro ele não é reiniciado automaticamente e o processo de recuperação de transações também não. É necessária intervenção humana ou o uso de softwares de monitoramento para reiniciar o servidor do *storage* problemático e depois iniciar o processo de recuperação de transações.

Os processos de *packing* e de recuperação de transações não podem ser executados concorrentemente, ou seja: se dois ou mais *storages* caírem por causa de alguma falha, apenas um por vez deverá ser reiniciado e passar pelo processo de recuperação de transações, fazendo com que os demais aguardem fora do sistema. Uma solução seria usar um *lock* distribuído para evitar que esses processos ocorram paralelamente.

Foi observado que existe um problema de sincronização de transações caso o relógio interno (*clock*) dos nós da rede não estiverem sincronizados. Isso pode ser resolvido utilizando um serviço de NTP (*Network Service Protocol*).

Durante a recuperação de transações podem haver conflitos irreparáveis caso o volume de novas transações no sistema seja muito grande. Além disso, o tempo de resposta de leitura do sistema aumenta muito, o que pode ser um problema em sistemas onde há carga muito alta.

7 ANÁLISE DOS TESTES

Durante a análise foi dada atenção especial aos resultados significativamente diferentes do sistema de referência definido na rede “A”, que não utiliza replicação. A análise foi dividida basicamente em três partes: inicialmente foi analisada a rede “A” para fins comparativos, logo após foram analisadas as redes “B”, “C” e “D” e por último foi realizada um comparativo final.

7.1 REDE “A”

A rede “A” é composta por dois clientes ZEO e um servidor ZEO sendo executados localmente no desktop, conforme demonstrado na Figura 24. Esta rede apresentou resultados já esperados: quando não estão recebendo requisições o consumo de memória e CPU é quase nulo; quando fazemos os testes de requisições de leitura o uso de CPU aumenta para cerca de 70%, enquanto o uso de memória aumenta para aproximadamente 12%. Os testes de escrita demonstram apenas um aumento de 10% no uso de CPU com relação ao testes de escrita.

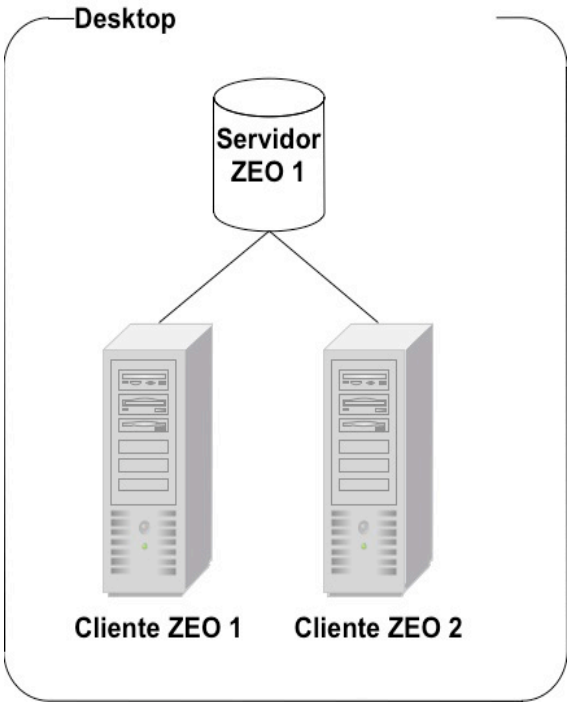


Figura 24: Arquitetura da Rede “A”

7.2 REDE “B”

A rede “B” é composta por dois servidores ZEO, dois clientes ZEO e um servidor ZEORaid. Essa arquitetura de rede utiliza replicação para manter os dois servidores ZEO idênticos, utilizando o servidor ZEORaid para gerenciar as transações e objetos que são transmitidos entre os clientes e servidores ZEO, conforme demonstrado pela Figura 25.

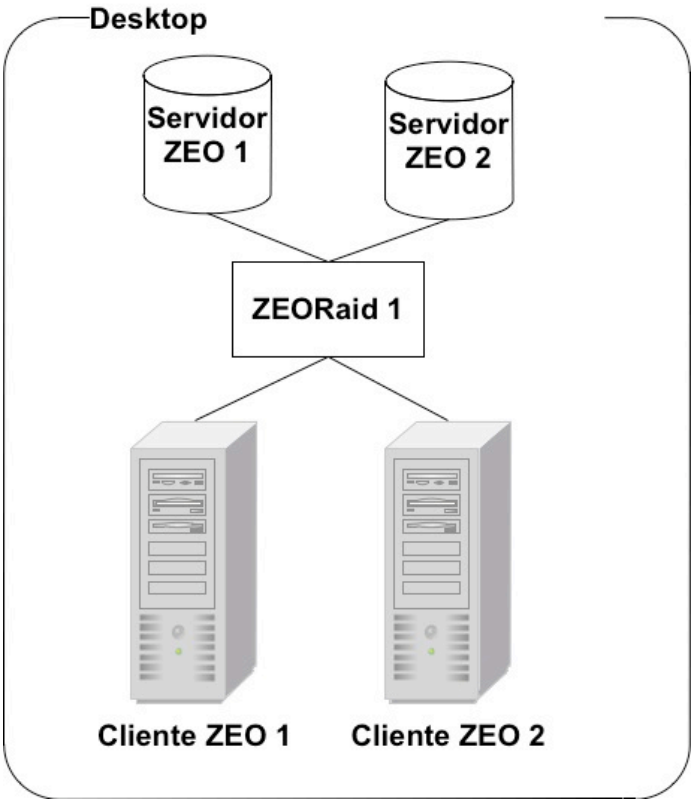


Figura 25: Arquitetura da Rede “B”

Nos testes foi detectado um aumento considerável no uso de memória mesmo quando o sistema não está recebendo requisições: 16% do total de memória do sistema operacional. Durante os testes de escrita o uso de CPU aumentou para 95% e o uso de memória foi para 47%; os testes de leitura apresentaram um consumo de CPU de 91% e uso de memória consideravelmente menor, 16.9%, o que demonstra o uso de *cache* de objetos nos clientes ZEO.

A replicação foi verificada utilizando os seguintes passos:

a) O servidor ZEO 2 foi parado;

b) O teste de escrita foi efetuado, usando apenas o servidor ZEO 1 como *backend*;

c) O servidor ZEO 2 foi reiniciado;

d) A opção de recuperação de transações foi ativada no servidor ZEORaid para que os dois bancos de dados fossem sincronizados.

Esse testes demontrou sucesso na replicação, embora o tempo de resposta para leitura durante a recuperação do servidor ZEO 2 tenha aumentado muito: o sistema ficou cerca de 70% mais lento. Isso pode inviabilizar o uso de ZEORaid em ambientes onde além da alta disponibilidade, a performance bruta é importante.

7.3 REDE “C”

Os testes na rede “C”, que conforme demonstra a Figura 26, utiliza clientes e servidores ZEO distribuídos entre dois nós da rede, demonstrou pouca diferença na utilização de recursos com relação a rede “B”. Um fato muito importante a ser observado é que o uso de rede se manteve estável durante todos os testes, exceto o teste de replicação.

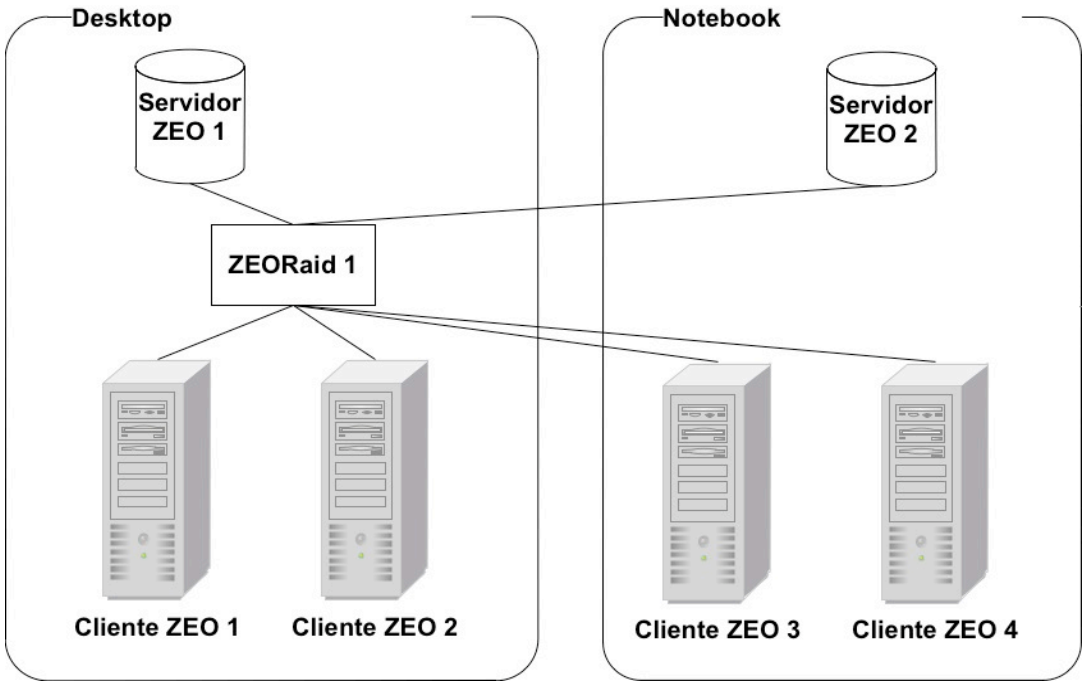


Figura 26: Arquitetura da Rede “C”

Durante o teste de replicação foi encontrado um problema na recuperação de dados entre os dois servidores ZEO, que além de causar indisponibilidade de serviços causa um uso excessivo de rede – o servidor ZEORaid se desconecta do servidor ZEO problemático e fica conectando e desconectando até que o mesmo seja desativado. Este problema é descrito em detalhes na Seção 6.7.

7.4 REDE “D”

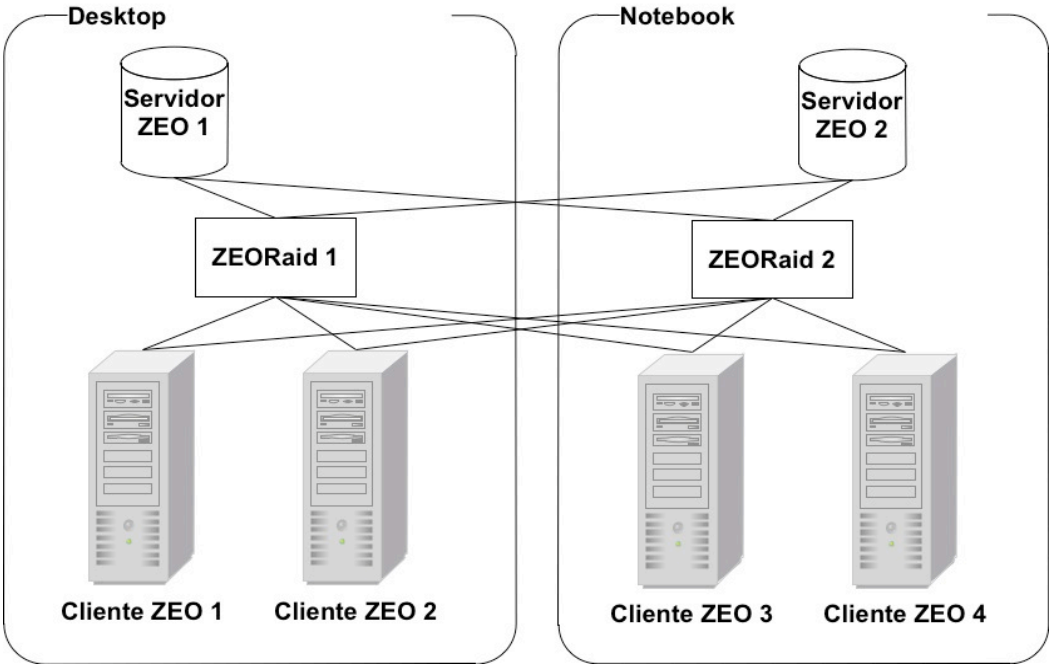


Figura 27: Arquitetura da Rede “D”

A rede “D” utiliza um servidor ZEO, um cliente ZEO e um servidor ZEORaid em cada um dos dois nós da rede, conforme visto na Figura 27. Os testes demonstraram um uso muito alto de recursos: o uso de CPU ficou em 98% durante os testes de escrita, e o uso de memória ficou em 57% no mesmo teste. O uso de rede se manteve estável, aumentando junto com o aumento de requisições.

Os mesmos problemas encontrados nos testes da rede “C” foram demonstrados novamente nesta rede. Além disso foi detectado um problema novo: os clientes ZEO conectam por padrão no servidor ZEORaid considerado “ótimo”, ou seja, o que estiver no mesmo endereço IP. Isso gera diversos conflitos na sincronização de transações e objetos, pois cada servidor ZEORaid grava alterações nos servidores ZEO separadamente. Esta falha é descrita em detalhes na Seção 6.7.

7.5 RESULTADO FINAL

Tendo em vista os resultados apresentados nas seções anteriores, constatou-se que o uso de replicação com ZEORaid é viável e eficiente no que tange alta disponibilidade, mas é inviável em ambientes onde alta performance é necessária. Os fatores que contribuíram para esta conclusão foram respectivamente:

a) A replicação funcional e a recuperação rápida e eficiente de servidores ZEO;

b) O uso excessivo de memória e CPU durante testes de carga quando utiliza-se múltiplos servidores ZEORaid;

c) Tempo de resposta lento durante recuperações.

O uso de múltiplos servidores ZEORaid ainda se demonstra instável. De acordo com os resultados obtidos, o ZEORaid pode ser utilizado em ambientes com carga moderada, mas é necessário pensar em outras formas de backup e recuperação para o caso de um servidor ZEORaid não ser suficiente.

CONCLUSÃO

Este trabalho apresentou um estudo sobre replicação de bases de dados orientadas a objeto, sendo que o objetivo primordial foi utilizar e aprimorar o ZEORaid para disponibilizar recursos de alta disponibilidade para o banco de dados ZODB. Uma extensa pesquisa foi realizada sobre alta disponibilidade, bancos de dados distribuídos e bancos de dados orientados a objeto, especialmente sobre o ZODB.

Diferentes configurações foram testadas com carga elevada e em circunstâncias passíveis de problemas para verificar as limitações desse tipo de recurso. Um portal simulando o uso real de acessos de leitura e escrita foi desenvolvido usando Plone para que fosse possível armazenar os resultados para análise posterior.

A análise apontou problemas no uso de ZEORaid em ambientes com elevada carga de escrita, além de erros de recuperação e sincronização em ambientes onde são usados muitos servidores de replicação, mas também demonstrou que o ZEORaid funciona muito bem para replicar ambientes quando são tomados certos cuidados.

O uso de replicação com apenas um servidor ZEORaid causa um problema na arquitetura de alta disponibilidade: o servidor de replicação se torna um ponto único de falha. Para resolver isso algumas soluções podem ser pesquisadas, como o uso de sistemas de arquivos compartilhados sobre a rede e sistemas de monitoramento. Entretanto, isso não resolve o problema da perda de dados de transações no caso do servidor ZEORaid estar enviando dados, recuperando transações ou fazendo *packing*.

Infelizmente este trabalho não é totalmente conclusivo, pois ainda há

metodologias diferentes para fazer replicação em bases de dados orientadas a objeto e diversas correções e aprimoramentos podem ser aplicados ao ZEORaid. Como trabalhos futuros sugerem-se:

a) Análise no uso de replicação mestre-escravo para bancos de dados orientados a objeto;

b) Uso de uma camada de abstração compartilhada para que um servidor ZEORaid não se torne um ponto único de falha;

c) Otimização no uso de recursos computacionais em ambientes que utilizem mais de um servidor ZEORaid;

d) Uso de diferentes níveis de RAID.

REFERÊNCIAS

ANTONIO, M. S. **Distributed zope clustering**. In: XXVI ASR ’2001 Seminar, Instruments and Control, 26., 2001, Brno. Anais Eletrônicos. Disponível em: <http://www.fs.vsb.cz/akce/2001/asr2001/Proceedings/papers/3.pdf> Acesso em: 20 ago. 2007.

ASPELI, M. **Professional Plone Development**. [S.l.]: Pack Publishing, 2007.

BEERI, C.; MILO, T. **A model for active oriented database**. In: Proceedings of the 17th International Conference of Very Large Data Bases, 17., 1991, Jerusalem. Anais Eletrônicos. Disponível em: <http://www.vldb.org/conf/1991/P337.PDF> Acesso em: 15 set. 2007.

BESTAVROS, A.; WANG, B. **Multi-version speculative concurrency control with delayed commit.** Proceedings of the 1994 International Conference on Computers and their Applications, 1994, Boston. Anais Eletrônicos. Disponível em: <http://www.cs.bu.edu/techreports/pdf/1993-014-scc-delayed-commit.pdf> Acesso em: 20 ago. 2007.

BOURKE, T. **Server Load Balancing**. [S.l.]: O’Reilly, 2001.

DYE, C. **Oracle Distributed Systems.** [S.l.]: O’Reilly, 1999.

FERRI, J. R. **Ambiente para Contrução de Textos (ACT)**: Um Ambiente para a Construção Colaborativa e Publicação de Textos na Web. 2002. Monografia (Bacharelado em Informática) — Universidade Regional do Noroeste do Estado do Rio Grande do Sul, Ijuí.

HARRINGTON, J. L. **Object-Oriented Database Design** - Clearly Explained. [S.l.]: Academic Press, 2000.

JALOTE, P. **Fault Tolerance in Distributed Systems**. New Jersey: Prentice Hall, 1994.

KEMME, B. **Database Replication for Clusters of Workstations**. 2000. Dissertação (Doutorado em Ciências Técnicas) — Swiss Federal Institute of Technology Zurich, Zurich.

KOREN, I.; KRISHNA, C. M. **Fault-Tolerant Systems**. [S.l.]: Elsevier, 2007.

KUCHLING, A. **ZODB/ZEO Programming Guide**. 2003. Disponível em: <http://www.zope.org/Wikis/ZODB/FrontPage/guide/index.html>. Acesso em: 20 jun. 2007.

KRAWCZYK, H.; BELLARE, M., CANETTI, R. **HMAC: Keyed-Hashing for Message Authentication**. 1997. Disponível em: <http://www.ietf.org/rfc/rfc2104.txt>. Acessado em 14 de Março de 2008.

LAURENT, S. S.; JOHNSTON, J.; DUMBILL, E. P**rogramming Web Applications With XML-RPC**. [S.l.]: O'Reilly, 2001.

LUCAS, M. **Absolute BSD**. The Ultimate Guide to FreeBSD. [S.l.]: O'Reilly, 2002.

LUTZ, M. **Programming Python**. [S.l.]: O'Reilly, 2006.

ORACLE. **Oracle Database High Availability Architecture and Best Practices**. California: Oracle Corporation, 2007.

OZNU, M. T.; VALDURIEZ, P. **Principles of Distributed Database Systems**. [S.l.]: Prentice Hall, 1999.

POSTGRESQL. **PostgreSQL 8.2.5 Documentation**. 2006. Disponível em: <http://www.postgresql.org/docs/8.2/static/mvcc.html>. Acessado em 18 de Outubro de 2007.

PRODASEN. **Boletim do Prodasen - N 41 - de 16 a 31 de Outubro de 2004**. 2004. Disponível em: <http://www5.senado.gov.br/boletimprodasen/boletins/boletim14/jean>. Acesso em: 20 ago. 2007.

PYTHON. **Python Programming Language**. Official Website. 2008. Disponível em: <http://www.python.org>. Acessado em 06 de Abril de 2008.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems Principles and Paradigms**. [S.l.]: Prentice Hall, 1999.

THEUNE, C. **gocept.zeoraid README**. 2007. Disponível em: <http://svn.zope.org/gocept.zeoraid/trunk/README.txt>. Acesso em: 10 abr. 2008.

TIEGS, D. P. **Improved Replication for ZODB through ZEORaid**. 2008. Disponível em: <http://code.google.com/soc/2008/zope/appinfo.html?csaid=480505ACAC256B7D>. Acesso em: 26 mai. 2008.

URBANO, R. **Oracle Database Advanced Replication**. [S.l.]: Oracle Corporation, 2003.

WEITERSHAUSEN, P. **Web Component Development With Zope 3**. [S.l.]: Springer, 2007.

ZODB. **Zope 2 wiki ZODB**. 2007. Disponível em: <http://wiki.zope.org/zope2/ZODB>. Acesso em: 20 jun. 2007.

ZOPE.COM. **Zope Replication Services**. 2007. Disponível em: <http://www.zope.com/products/zopereplicationservices.html>. Acesso em: 20 jun. 2007.

APÊNDICE A

Replicação de Bases de Dados do Zope Object Database

Dirceu Pereira Tiegs[1], Daniel Pezzi da Cunha[2]

[1]Weimar Consultoria – Criciúma, SC – Brazil

[2]Departamento de Ciência da Computação – Universidade do Extremo Sul Catarinense (UNESC) – Criciúma, SC - Brazil

dirceutiegs@gmail.com, dpc@unesc.net

Abstract. *The online services user number is growing up every day at an impressive ratio, as well as the need to keeping these services running as long as possible and with high performance. To achieve this goal the Zope Object DataBase (ZODB) uses a master-slave replication approach, whose major issue is a possible failure. This research analysed and improved ZEORaid, a software provides an experimental gateway for N to N replication. As for obtained results, the use of high availability was improved even during service failures and with high number of accesses, and ZEORaid problems were pointed and fixed.*

Resumo. *A quantidade de usuários de serviços online cresce diariamente, assim como a necessidade de manter esses serviços disponíveis pelo maior tempo possível e com alto desempenho. Para atingir esse objetivo o Zope Object Database (ZODB) faz uso da replicação mestre-escravo, cujo problema fundamental é manter um ponto único de falha. Este trabalho envolveu a pesquisa, a análise e o aprimoramento do software ZEORaid, que disponibiliza um gateway experimental para replicação N para N. Como resultados, conseguiu-se aprimorar o emprego da alta disponibilidade mesmo em caso de falha de serviços e com um número grande acessos, além de apontar e corrigir problemas no ZEORaid.*

1. Introdução

Enquanto muitas organizações podem se contentar em ter seus sistemas disponíveis em menos de 99% do tempo, outras, como bancos e sistemas governamentais precisam da maior disponibilidade possível. Alta Disponibilidade se refere às técnicas usadas para manter um sistema acessível aos usuários ou a outros sistemas. Um sistema é dito altamente disponível se, usando algumas técnicas pré-estabelecidas ele é confiável, auto-recuperável, tem detecção de erros e provê operações contínuas (ORACLE, 2007).

Todos estes serviços precisam de uma maneira eficiente de armazenamento, capaz de manter os dados seguros e responder as requisições com performance adequada. Embora a maior parte do armazenamento de sistemas online seja feita usando Sistemas Gerenciadores de Bancos de Dados Relacionais, a crescente necessidade de características mais dinâmicas tem elevado o uso de Sistemas Gerenciadores de Bancos de Dados Orientados a Objeto.

Uma das estratégias usadas para manter a alta disponibilidade é a replicação,

onde vários serviços iguais ficam trabalhando ao mesmo tempo em locais diferentes, não deixando toda a carga e confiança no funcionamento do sistema em um ponto único (JALOTE, 1994). Isso é comumente utilizado no caso de servidores HTTP, *cache* e bancos de dados relacionais, mas isso foi pouco desenvolvido para bancos de dados orientados a objetos.

O Zope Object Database (ZODB) é um Sistema Gerenciador de Bancos de Dados Orientados a Objeto open-source utilizado principalmente no Z Object Publishing (ZOPE), um framework de desenvolvimento e servidor de aplicativos web. Além do ZOPE, o ZODB pode ser utilizado em qualquer tipo de aplicação desenvolvida usando a linguagem Python ou que use essa linguagem como conector.

Grandes empresas como o governo brasileiro e o Bank Boston utilizam o ZODB em aplicações de grande porte, onde a necessidade de alta disponibilidade é maior. A população pode se beneficiar muito, mesmo que indiretamente, de recursos de alta disponibilidade em aplicações do governo, por exemplo: quanto mais performático, confiável e disponível for um sistema, melhor será o atendimento às necessidades dessas pessoas.

A Zope Corp, corporação responsável pelo ZODB e pelo o ZOPE, desenvolveu um sistema de replicação chamado Zope Replication Services (ZRS) que é capaz de utilizar até dois servidores de dados redundantes (ZOPE.COM, 2007), mas não é capaz de atender um número maior que esse, o que pode tornar o sistema lento ou até indisponível em alguns casos.

Com o intuito de pesquisar os problemas e vantagens da área de replicação em Sistemas Gerenciadores de Bancos de Dados Orientados a Objeto, este trabalho envolveu a análise e o aprimoramento do *gateway* ZEORaid para permitir a comunicação de um número maior de bases de dados distribuídas do ZODB, eliminando a restrição de apenas dois nós imposta pelo ZRS e permitindo maior flexibilidade, escalabilidade e disponibilidade dos sistemas que o utilizam.

2. O ZODB e o Problema de Replicação

Dye (1999) define Alta Disponibilidade como o emprego de técnicas de contenção e tolerância à falhas de hardware e software para permitir que um serviço fique disponível para seus usuários a maior parte do tempo possível. O uso de cada técnica depende da classificação das falhas encontradas.

Toda a tolerância à falhas é baseada na redundância. Por redundância entende-se a propriedade de existir mais de um recurso além do mínimo necessário para fazer o trabalho em questão. Quando falhas acontecem a redundância é utilizada para encobrir os erros e manter o sistema disponível para uso.

RAID1 consiste em discos espelhados. No lugar de um disco existem dois discos, cada um sendo uma cópia do outro. Se um disco falha, o outro pode continuar a servir acesso aos dados. Se dos dois discos estiverem funcionando, RAID1 pode tornar o acesso a leitura mais rápido dividindo as requisições entre os dois discos; o acesso a escrita, contudo, se torna mais lento, pois os dois discos têm que terminar a atualização antes que a operação de escrita se complete.

O ZODB é uma implementação open-source do modelo de dados orientado a objetos. ZODB significa *Zope Object Database* e foi criado inicialmente para uso

conjunto com o Zope, um servidor de aplicações orientado a objetos construído utilizando a linguagem de programação Python.

O Zope Enterprise Objects (ZEO) é um módulo que disponibiliza uma arquitetura distribuída para instalações do ZODB por meio de um *Storage* chamado *ClientStorage*. O *ClientStorage* não faz a escrita em uma mídia física, apenas encaminha todas as requisições através da rede para um servidor. Este, por sua vez, executa uma instância da classe *StorageServer*, simplesmente agindo como um *front-end* para algum *Storage* físico (como *FileStorage*, por exemplo) (ANTONIO, 2001). A Figura 1 demonstra um exemplo de uso de ZEO, onde três clientes ZEO acessam a mesma base de dados em um servidor ZEO (também chamados de ZSS / ZEO *Storage Server* por alguns autores).

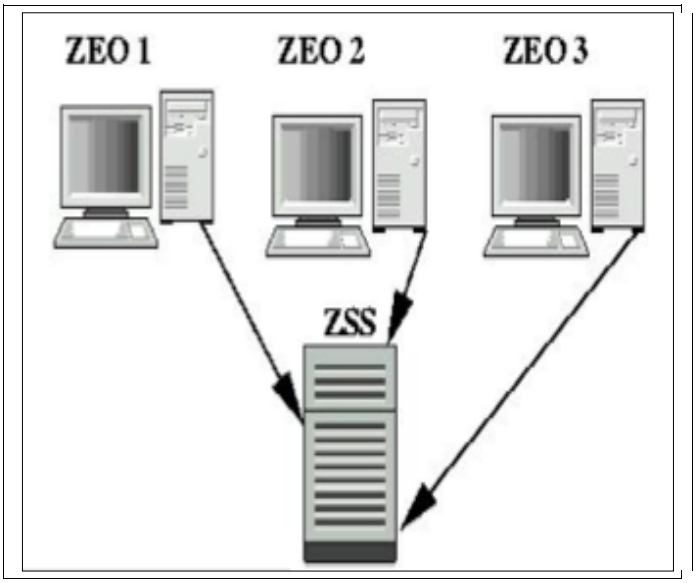


Figura 1. Arquitetura ZEO
Fonte: ANTONIO, M. S. (2001)

Do ponto de vista da alta disponibilidade o ZEO tem um problema: não existe a possibilidade de utilizar múltiplos servidores, apenas múltiplos clientes. Esse problema é solucionado pelo ZRS (*Zope Replication Services*), que permite o uso de até dois servidores de dados usando a abordagem *Primary-Backup*; se o servidor primário falha, o servidor de backup assume o seu lugar (ZOPE.COM, 2007). A arquitetura do ZRS é demonstrada na Figura 2.

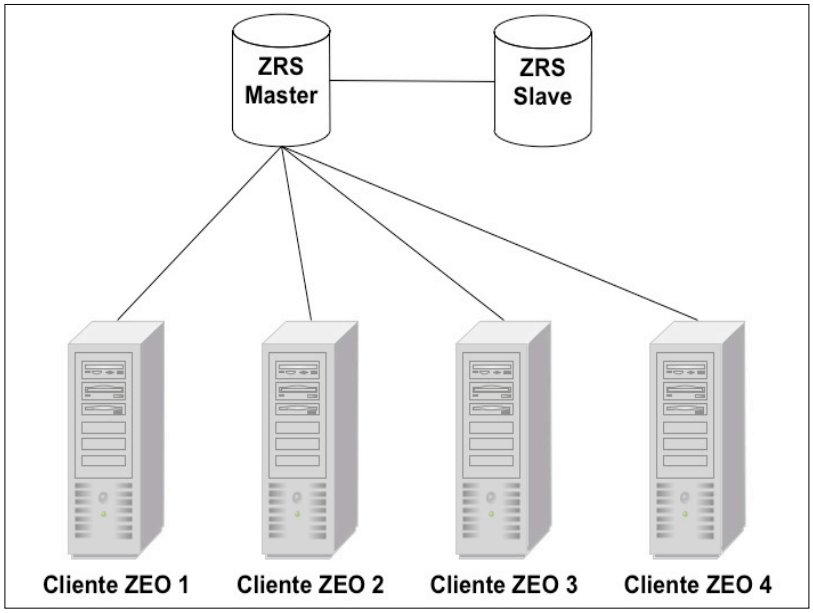


Figura 2. Ambiente Utilizando o ZRS

O problema do ZRS é que a arquitetura é limitada para até dois servidores, além do software ser proprietário e cobrado por CPU.

3. Arquitetura de Replicação Múltipla Utilizando ZEORaid

Para tornar possível o principal objetivo deste projeto, o projeto ZEORaid foi aprimorado com a implementação de funcionalidades e correção de erros, e um aplicativo foi desenvolvido e configurado para demonstrar o uso de uma arquitetura distribuída com o ZODB. Após a realização destas etapas, foi possível executar os testes e analisar os resultados obtidos pela aplicação desenvolvida.

O ZEORaid é um software de código aberto criado na empresa Gocept, Inc. para disponibilizar um novo storage (armazenamento) para o ZODB que utiliza conceitos da arquitetura RAID para fazer replicação de servidores ZEO. Para tornar isso possível é disponibilizado um servidor chamado ZEORaid Server, que mantém dados sobre os servidores ZEO e estende o ZRPC para permitir chamadas de verificação de estado.

Para aproveitar a infra-estrutura atual, um servidor ZEORaid é criado da mesma forma que um servidor ZEO, mas configurado usando opções específicas. Essas opções fazem com que um servidor ZEORaid atue como um servidor ZEO quando acessado por clientes ZEO e como um cliente ZEO quando acessando um servidor ZEO. A Figura 3 mostra um exemplo disso: os clientes ZEO 1, 2, 3 e 4 estão configurados para acessar os servidores ZEORaid 1 e 2 como se fossem servidores ZEO comuns; os servidores ZEORaid, por sua vez, estão configurados como clientes ZEO e estão acessando os servidores ZEO 1, 2 e 3 como tais.

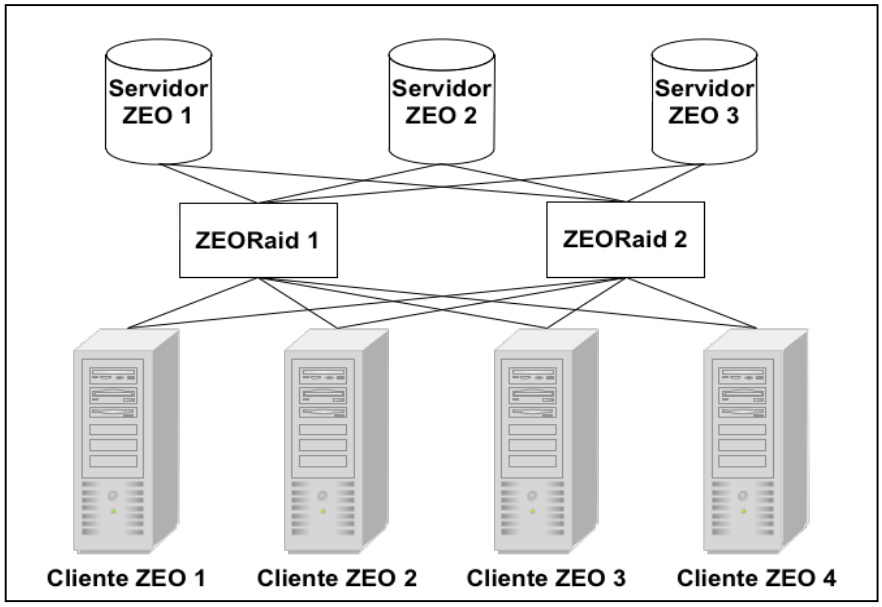


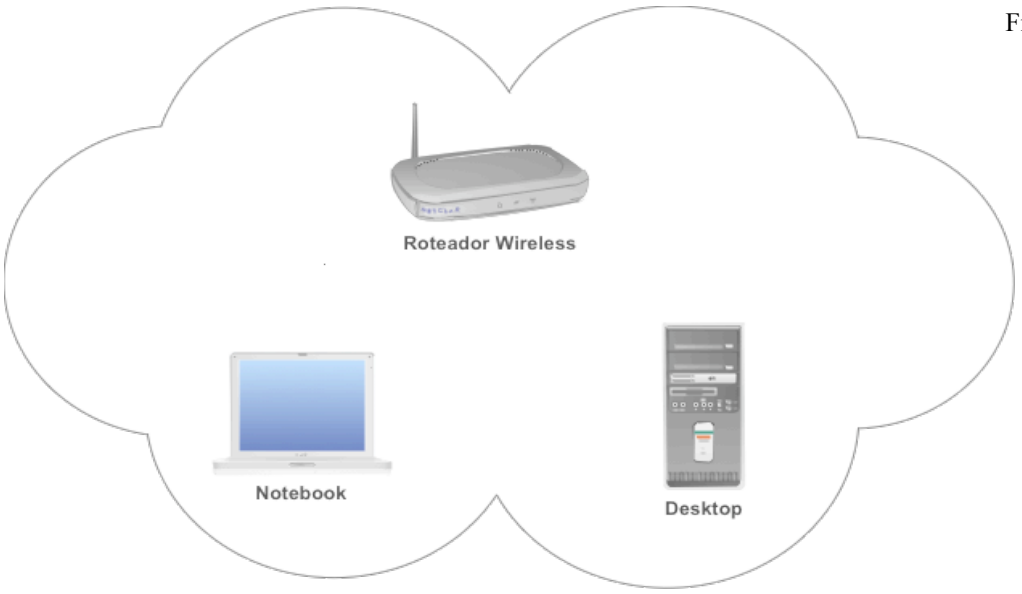
Figura 3. Ambiente Utilizando o ZEORaid

O próprio ZEORaid, de certa forma, ainda é experimental. A empresa Google, Inc., através do projeto Google Summer of Code financiou um projeto cujo intuito é aprimorar o ZEORaid. O projeto aconteceu entre 26 de maio e 17 de agosto de 2008, e as tarefas incluíram a otimização do processo de invalidação do *cache*, paralelização das requisições para múltiplos servidores ZEO e criação e remoção de servidores ZEO de maneira simplificada sem parar os serviços (TIEGS, 2008).

A aplicação desenvolvida para testes foi um site plone para uma empresa (hipotética) de consultoria e desenvolvimento de software chamada XyZ Consultoria. O site conta com uma área institucional, uma área de desenvolvimento (onde existe uma lista de clientes e projetos) e uma área de produtos (onde são apresentados os livros e vídeo-aulas criadas pelos profissionais da XyZ Consultoria).

O desenvolvimento foi feito em um notebook Apple iBook G4 com processador PowerPC G4 de 1.2 GHz, 512 MB de memória principal, disco rídigo ATA-6 de 30 GB e sistema operacional Mac OS X 10.5. Um segundo computador (um desktop) também foi utilizado nos testes, e conta com processador Athlon XP 2600+ de 1.9 GHz, 512 MB de memória principal, disco rídigo SATA de 80 GB e sistema operacional Kubutu Linux 7.10 com kernel 2.6.22. Foi utilizado um roteador *wireless* com padrão 802.11g para a comunicação em rede.

Figura 4.



Arquitetura da Rede de Testes

A Figura 4 mostra a arquitetura da rede utilizada nos testes: o notebook e o desktop conectados por um roteador wireless.

A primeira etapa dos testes constituiu a instalação de quatro configurações de rede de processos diferentes. Os testes utilizando a aplicação desenvolvida foram feitos baseados na seguinte metodologia:

10000 acessos de leitura com 200 requisições simultâneas a cada vez aos clientes ZEO;

200 acessos de escrita com 5 requisições simultâneas a cada vez aos clientes ZEO;

200 acessos de escrita com 5 requisições simultâneas a cada vez aos clientes ZEO durante a recuperação de um servidor ZEO;

cada experimento foi repetido 10 vezes para obter-se um resultado médio adequado.

Os testes foram efetuados para detectar possíveis anomalias em uso de memória, processador e rede e também outros problemas quando utiliza-se replicação para ZODB em um ambiente de alta disponibilidade.

Durante a análise foi dada atenção especial aos resultados significativamente diferentes do sistema de referência definido na rede “A”, que não utiliza replicação. A análise foi dividida basicamente em três partes: inicialmente foi analisada a rede “A” para fins comparativos, logo após foram analisadas as redes “B”, “C” e “D” e por último foi realizada um comparativo final.

A rede “A” é composta por dois clientes ZEO e um servidor ZEO sendo executados localmente no desktop. Esta rede apresentou resultados já esperados: quando não estão recebendo requisições o consumo de memória e CPU é quase nulo; quando fazemos os testes de requisições de leitura o uso de CPU aumenta para cerca de 70%, enquanto o uso de memória aumenta para aproximadamente 12%. Os testes de escrita demonstram apenas um aumento de 10% no uso de CPU com relação ao testes de escrita.

A rede “B” é composta por dois servidores ZEO, dois clientes ZEO e um servidor ZEORaid. Essa arquitetura de rede utiliza replicação para manter os dois servidores ZEO idênticos, utilizando o servidor ZEORaid para gerenciar as transações e objetos que são transmitidos entre os clientes e servidores ZEO. Nos testes foi detectado um aumento considerável no uso de memória mesmo quando o sistema não está recebendo requisições: 16% do total de memória do sistema operacional. Durante os testes de escrita o uso de CPU aumentou para 95% e o uso de memória foi para 47%; os testes de leitura apresentaram um consumo de CPU de 91% e uso de memória consideravelmente menor, 16.9%, o que demonstra o uso de *cache* de objetos nos clientes ZEO.

A replicação foi verificada utilizando os seguintes passos:

a) O servidor ZEO 2 foi parado;

b) O teste de escrita foi efetuado, usando apenas o servidor ZEO 1 como *backend*;

c) O servidor ZEO 2 foi reiniciado;

d) A opção de recuperação de transações foi ativada no servidor ZEORaid para que os dois bancos de dados fossem sincronizados.

Esse teste demontrou sucesso na replicação, embora o tempo de resposta para leitura durante a recuperação do servidor ZEO 2 tenha aumentado muito: o sistema ficou cerca de 70% mais lento. Isso pode inviabilizar o uso de ZEORaid em ambientes onde além da alta disponibilidade, a performance bruta é importante.

Os testes na rede “C”, que utilizam clientes e servidores ZEO distribuídos entre dois nós da rede, demonstraram pouca diferença na utilização de recursos com relação a rede “B”. Um fato muito importante a ser observado é que o uso de rede se manteve estável durante todos os testes, exceto o teste de replicação.

Durante o teste de replicação foi encontrado um problema na recuperação de dados entre os dois servidores ZEO, que além de causar indisponibilidade de serviços causa um uso excessivo de rede – o servidor ZEORaid se desconecta do servidor ZEO problemático e fica conectando e desconectando até que o mesmo seja desativado.

A rede “D” utiliza um servidor ZEO, um cliente ZEO e um servidor ZEORaid em cada um dos dois nós da rede, conforme visto na Figura 27. Os testes demonstraram um uso muito alto de recursos: o uso de CPU ficou em 98% durante os testes de escrita, e o uso de memória ficou em 57% no mesmo teste. O uso de rede se manteve estável, aumentando junto com o aumento de requisições.

Os mesmos problemas encontrados nos testes da rede “C” foram demonstrados novamente nesta rede. Além disso foi detectado um problema novo: os clientes ZEO conectam por padrão no servidor ZEORaid considerado “ótimo”, ou seja, o que estiver no mesmo endereço IP. Isso gera diversos conflitos na sincronização de transações e objetos, pois cada servidor ZEORaid grava alterações nos servidores ZEO separadamente.

4. Conclusão

Tendo em vista os resultados obtidos, constatou-se que o uso de replicação com ZEORaid é viável e eficiente no que tange alta disponibilidade, mas é inviável em ambientes onde alta performance é necessária. Os fatores que contribuíram para esta conclusão foram respectivamente:

a) A replicação funcional e a recuperação rápida e eficiente de servidores ZEO;
b) O uso excessivo de memória e CPU durante testes de carga quando utiliza-se múltiplos servidores ZEORaid;
c) Tempo de resposta lento durante recuperações.

O uso de múltiplos servidores ZEORaid ainda se demonstra instável. De acordo com os resultados obtidos, o ZEORaid pode ser utilizado em ambientes com carga moderada, mas é necessário pensar em outras formas de backup e recuperação para o caso de um servidor ZEORaid não ser suficiente.

A análise apontou problemas no uso de ZEORaid em ambientes com elevada carga de escrita, além de erros de recuperação e sincronização em ambientes onde são usados muitos servidores de replicação, mas também demonstrou que o ZEORaid funciona muito bem para replicar ambientes quando são tomados certos cuidados.

O uso de replicação com apenas um servidor ZEORaid causa um problema na arquitetura de alta disponibilidade: o servidor de replicação se torna um ponto único de falha. Para resolver isso algumas soluções podem ser pesquisadas, como o uso de sistemas de arquivos compartilhados sobre a rede e sistemas de monitoramento. Entretanto, isso não resolve o problema da perda de dados de transações no caso do servidor ZEORaid estar enviando dados, recuperando transações ou fazendo *packing*.

Referências

Antonio, S. **Distributed zope clustering**. In: XXVI ASR ’2001 Seminar, Instruments and Control, 26., 2001, Brno. Anais Eletrônicos. Disponível em: <http://www.fs.vsb.cz/akce/2001/asr2001/Proceedings/papers/3.pdf> Acesso em: 20 ago. 2007.

Dye, C. **Oracle Distributed Systems.** [S.l.]: O’Reilly, 1999.

Jalote, P. **Fault Tolerance in Distributed Systems**. New Jersey: Prentice Hall, 1994.

Oracle. **Oracle Database High Availability Architecture and Best Practices**. California: Oracle Corporation, 2007.

Tiegs, D. **Improved Replication for ZODB through ZEORaid**. 2008. Disponível em: <http://code.google.com/soc/2008/zope/appinfo.html?csaid=480505ACAC256B7D> . Acesso em: 26 mai. 2008.

ZODB. **Zope 2 wiki ZODB**. 2007. Disponível em: <http://wiki.zope.org/zope2/ZODB>. Acesso em: 20 jun. 2007.

Zope.com. **Zope Replication Services**. 2007. Disponível em: <http://www.zope.com/products/zopereplicationservices.html>. Acesso em: 20 jun. 2007.