À

IM

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

Persistência e Alocação de Espaço em Armazenamento Distribuído

Dissertação de Mestrado

Aluno:

Danielle Fernandes Parga

Orientador:

Prof. Dr. Carlo Emmanoel Tolla de Oliveira

Rio de Janeiro

2007

Persistência e Alocação de Espaço em Armazenamento Distribuído

Danielle Fernandes Parga

Universidade Federal do Rio de Janeiro
Instituto de Matemática
Núcleo de Computação Eletrônica

Orientador:
Prof. Dr. Carlo Emmanoel Tolla de Oliveira
Ph. D., University College, 1993

Rio de Janeiro

2007

Persistência e Alocação de Espaço em Armazenamento Distribuído

Danielle Fernandes Parga

Dissertação submetida ao corpo docente do Instituto de Matemática – IM / Núcleo de Computação Eletrônica – NCE - Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre.

Data:

Aprovada por:

_____

Prof. Dr. Carlo Emmanoel Tolla de Oliveira - Orientador

Ph. D., University College, 1993

_____

Prof. Dr. Felipe Maia Galvão França.

Ph.D., Imperial College London, 1994.

_____

Profª. Drª. Vanessa de Paula Braganholo

DSc. , Universidade do Rio Grande do Sul, 2004.

Rio de Janeiro

2007

Parga, Danielle Fernandes.

Persistência e Alocação de Espaço em Armazenamento Distribuído / Danielle Fernandes Parga. – Rio de Janeiro, 2007.

xi, 84 p.; il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro - UFRJ, Instituto de Matemática / Núcleo de Computação Eletrônica, 2005.

Orientador: Carlo Emmanoel Tolla de Oliveira

1. Sistemas Distribuídos. 2. Sistemas Prevalentes. I. Oliveira, Carlo Emmanoel Tolla de (Orient.). II. Universidade Federal do Rio de Janeiro. Instituto de Matemática / Núcleo de Computação Eletrônica. III. Título.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me dado a sabedoria em optar por realizar este trabalho e por ter me dado a força que precisava para chegar até o fim.

Aos meus pais e minha irmã, que sempre contribuíram para o meu crescimento e incentivaram na busca de meus ideais, me tornando uma pessoa melhor, mais capaz e feliz.

Ao meu orientador Carlo Emmanoel, pela amizade e por todo apoio técnico e psicológico. Por não ter me deixado desistir nos momentos em que achei que não poderia continuar.

Aos meus amigos que sempre me empurraram para frente e que me proporcionaram horas de descontração nos momentos de *stress*.

Enfim, agradeço a todos que de alguma forma contribuíram para que este trabalho fosse finalizado. E espero um dia poder retribuir de alguma maneira toda força e apoio que me ofereceram.

RESUMO

PARGA, Danielle Fernandes. **Persistência e Alocação de Espaço em Armazenamento Distribuído**. Orientador: Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro: UFRJ/NCE, 2007. Dissertação (Mestrado em Sistemas de Informação).

À medida em que a tecnologia avança, os dispositivos computacionais estão cada vez mais difundidos em suas diversas formas. Com essa proliferação geográfica do acesso, a proposta centralizadora de recursos da *Internet* deixa de ser uma solução ótima. Uma melhor alternativa é distribuir a informação entre nós que estejam nas proximidades dos dispositivos requisitantes. Neste trabalho propomos a prevalência distribuída *peer-to-peer* (P2P), onde os nós compartilham sua memória com seus pares para manter a informação disponível.

O paradigma de memória compartilhada distribuída permite que usuários disseminem informações para todos os participantes. Por isso, é importante tornar as informações disponíveis a todo tempo e lugar. Para que isso seja possível, é necessário distribuir, alocar e sincronizar as informações persistidas em máquinas distribuídas. Além disso, é preciso qualificar os nós da rede segundo a oferta computacional e classificar esses nós segundo a disponibilidade, capacidade, desempenho e acesso.

Neste trabalho implementamos uma prova de conceito da prevalência distribuída e avaliamos diversas heurísticas de disponibilização de dados. Estas heurísticas foram simuladas e classificadas segundo as suas propriedades de otimização. Estas simulações mostram a capacidade da prevalência distribuída em compartilhar dados numa rede P2P.

ABSTRACT

PARGA, Danielle Fernandes. **Persistência e Alocação de Espaço em Armazenamento Distribuído**. Orientador: Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro: UFRJ/NCE, 2007. Dissertação (Mestrado em Sistemas de Informação).

As technology advances, different forms of computational devices are spreading. With the proliferation of geographic access, the centralizing approach of the Internet is no longer an optimal solution. A better alternative is to distribute the information among the nodes that are around the requesting devices. In this work, we propose the distributed peer-to-peer (P2P) information prevalence, where memory is shared among peers to keep information available.

The distributed shared memory paradigm allows users to spread information to all participants. For this reason, it is important to make information available everywhere and at all times. In order to do so, it is necessary to distribute, allocate and synchronize information saved in distributed machines. Furthermore, there is the need to qualify the network nodes according to the computational availability and classify these nodes with respect to availability, capacity, performance and access.

In this work, we implemented a proof of concept for the distributed prevalence and assessed different heuristics for data availability. These heuristics were simulated and classified according to their optimization properties. These simulations show the ability of the distributed prevalence in sharing data in a P2P network.

LISTA DE ABREVIATURAS

API	*Application Programming Interface*
BDD	Banco de Dados Distribuídos
HTTP	*HiperText Transfer Protocol*
IP	*Internet Protocol*
JXTA	*Juxtapose*
MIB	*Management Information Base*
OLPC	*One Laptop Per Child*
OO	Orientação a Objetos
OOD	Orientadas a Objetos Distribuídos
ORB	*Object Request Broker*
P2P	*Peer-to-Peer*
PDA	*Personal Digital Assistant*
PYRO	*Python Remote Objects*
RAM	*Random Access Memory*
RMI	*Remote Method Invocation*
RPC	*Remote Process Call*
SGBD	Sistema Gerenciador de Bando de Dados
SNMP	*Simple Network Management Protocol*
TCP	*Transfer Control Protocol*
UDP	*User Datagram Protocol*
XML	*Extensible Markup Language*

LISTA DE FIGURAS

LISTAGENS

GRÁFICOS

TABELAS

SUMÁRIO

1. Introdução

A Internet trouxe mudanças em todas as áreas de negócios. A utilização de um meio tão poderoso e que está sempre disponível aos seus usuários, causou uma revolução na maneira de ter acesso à informação e de fazer negócios.

Com a crescente necessidade de persistência de dados, principalmente devido à popularização da Internet e ao grande volume de informações, uma grande variedade de soluções vem sendo desenvolvida (Hibernate, 2003; KELLER, 1997; LERMEN, 1998; ZIMBRÃO, 2003). É consenso entre os desenvolvedores a elevada importância da persistência nos sistemas atuais. Porém, com a disseminação do paradigma de Orientação a Objetos (OO), com grandes vantagens sobre o paradigma procedural, os desenvolvedores se viram obrigados a ter de conviver com dois mundos distintos, o mundo formado por conceitos relacionais e o mundo de OO.

Além disso, o desenvolvimento em camadas (BATTISTI, 2003) trouxe um salto qualitativo de produção de software, garantindo total flexibilidade de desenvolvimento. Independente do número de camadas envolvidas (uma, duas, três camadas até o modelo n-camadas), uma das maiores dificuldades encontradas pelos desenvolvedores é o desempenho e a flexibilidade da camada de manipulação de dados, também conhecida como camada de persistência. A necessidade de manipular dados de forma universal, independente do paradigma de armazenamento a ser utilizado, traz flexibilidade à aplicação.

Como dito anteriormente, o uso da tecnologia OO no desenvolvimento de aplicações vem se tornando cada vez mais difundido. Porém, a maior parte dos Sistemas Gerenciadores de Banco de Dados (SGBDs) utilizados nos últimos anos são baseados no modelo relacional. Neste modelo, todos os dados são armazenados em tabelas e o sistema de armazenamento é fundamentado nos relacionamentos entre elementos de dados, buscando uma normalização desses dados para evitar redundâncias e possibilitar um maior desempenho nas consultas.

Entretanto, esse modelo não atende às demandas cada vez mais complexas das aplicações. Um dos grandes problemas que existe nos sistemas distribuídos que utilizam o modelo do tipo centralizado, isto é, vários clientes acessando a mesma base de dados, é a capacidade limitada de escalonamento. Com o aumento do número de usuários, isso resulta num desempenho degradado pelo gargalo existente entre os clientes e o servidor.

Para suprir esses tipos de limitações surgiram várias soluções, como por exemplo, os sistemas gerenciadores de bancos de dados objeto-relacional (SILVA, 1999) (STONEBRAKER, 1996). Nestes sistemas, a persistência dos objetos pode ser obtida pelo mapeamento de classes para banco de dados relacionais. Além disso, implementam de forma natural os modelos orientados a objetos, isto é, os conceitos de classe, objeto, tipo, identidade, igualdade, encapsulamento, herança, agregação, método e polimorfismo (*overloading* e *overriding*). Ainda possuem mecanismos especiais para controlar transações entre objetos, técnicas de armazenamento que facilitam a recuperação rápida de objetos complexos (*clustering*) e funções adicionais para coordenar atividades cooperativas, tais como mecanismos para avisar os usuários sobre mudanças de estado nos objetos e para notificar a disponibilidade dos objetos. De toda forma, o paradigma relacional e orientado a objetos são intrinsecamente divergentes. O resultado é o não-casamento de impedâncias, isto é, o atrito em converter os conceitos de um nos conceitos do outro.

A estratégia objeto-relacional amplia o problema do não-casamento de impedâncias, proporcionado pela incompatibilidade de tipos entre a linguagem de programação utilizada e a forma de armazenamento das informações. Uma forma comum de se obter a persistência de objetos consiste em estender uma linguagem de programação orientada a objetos por um conjunto de classes que implementam a persistência, gravando os objetos em um meio não-volátil. Porém, o desempenho na execução de transações utilizando um meio não-volátil como forma de persistência de objetos é muito menor que em um meio volátil. Isto é explicado pelo fato de todos os objetos estarem na memória RAM (*Random Access Memory*), o que torna o acesso aos objetos muito mais rápido.

Outro problema está no fato de o desenvolvimento da tecnologia e das ciências proporcionar o crescimento qualitativo e quantitativo de informações. As aplicações apresentam uma demanda crescente de processamento e a massa de dados também cresce com a maior necessidade de informação. Surge, então, a necessidade de criar um modelo prevalente distribuído, onde as informações e aplicações precisam estar largamente disponíveis. A justificativa dessa abordagem está no fato de um sistema prevalente utilizar um modelo transparente de persistência de objetos baseado no armazenamento desses objetos na memória principal do computador, característica que a diferencia dos sistemas convencionais. E sendo um sistema distribuído, os objetos ficarão dispersos na memória de várias máquinas.

A alta disponibilidade das informações é uma das propriedades mais desejáveis em sistemas computacionais, principalmente em aplicações comerciais que, tipicamente,

envolvem acesso a banco de dados e usam transações. Alta disponibilidade pode ser alcançada através de distribuição de dados. Essa distribuição é realizada utilizando técnicas de replicação e fragmentação (que neste trabalho de dissertação chamaremos de migração). Ambas as técnicas melhoram o desempenho do sistema e das aplicações, pois o compartilhamento dos dados através da distribuição permite que cada nó possa reter um grau de controle sobre os dados armazenados localmente.

Com a replicação há uma disseminação de cópias de dados onde for necessário. Através da replicação dos dados, os usuários têm acesso local aos dados correntes, reduzindo o tráfico da rede. Se uma das réplicas não está operacional, outra réplica garante que um determinado serviço seja oferecido. No entanto, réplicas requerem protocolos que assegurem consistência de estado.

É nesse sentido que se utilizada a computação ubíqua, área de pesquisa idealizada por Mark Weiser (1991), que sugere novas formas de interação entre usuário e máquina, desenvolvendo ambientes onde a inserção de tecnologia no cotidiano ocorre de modo transparente. Nesses ambientes, ditos ubíquos, é crescente a presença de mobilidade e informações como identificação, atividade, preferências e histórico de usuários que são monitorados, processados e armazenados por diferentes dispositivos e aplicações.

Problema

Apesar do contínuo avanço na tecnologia de computadores, o progresso nos sistemas de computação não tem acompanhado o mesmo ritmo. Isto advém do fato de os sistemas atuais estarem pesadamente ancorados a um legado tecnológico, que limita as possibilidades de inovação. Este legado impõe um descasamento de impedância entre a representação da informação e o seu efetivo processamento.

Para que se possa usufruir dos avanços tecnológicos, é necessário que a representação da informação seja reconciliada com o paradigma de processamento. Sendo o paradigma predominante a orientação a objetos, é necessário que toda a representação da informação seja compatível com sua forma estrutural e comportamental. Neste trabalho, utilizou-se a prevalência computacional como forma de representação da informação, já que ela preserva toda a flexibilidade e desempenho do paradigma orientado a objetos.

A prevalência permite que a informação seja preservada sem perdas e com um mínimo de sobrecarga sobre o sistema original. Com isso, toda a capacidade latente já

existente nos parques computacionais pode ser resgatada. Por isso, é necessário explorar todas as dimensões em que se possa aplicar a prevalência e criar ferramentas que facilitem a transição do modelo legado.

A aplicação da prevalência deve explorar a transparência, capacidade de busca, interatividade, escalabilidade e autonomia dos sistemas computacionais. A transferência permite implantar a prevalência com um mínimo de interferência com a codificação do negócio. Uma vez implantada a prevalência, sistemas de informação podem ser migrados com o desenvolvimento de mecanismos de busca baseado nessa prevalência. Seguindo além, toda a potencialidade inovadora da prevalência pode ser aplicada em uma evolução do paradigma atual para o paradigma prevalente.

O presente trabalho de pesquisa propõe a criação de uma solução chamada Pantheon, que visa implantar a persistência utilizando um engenho de armazenamento de comandos em arquivos, a serem retomados toda vez que a aplicação for reinicializada. O armazenamento em arquivo utiliza a capacidade de serialização que as linguagens orientadas a objetos possuem (*Python* é uma delas). Para isto, a utilização do *Prevayler* (WUESTEFELD, 2001) é fundamental, já que este possui todo o mecanismo de serialização de comandos.

O mecanismo de comandos para a persistência pode ser extensivo à replicação e migração das informações e serviços pela rede distribuída da aplicação. Para garantir uma maior disponibilidade das informações e dos serviços oferecidos, o Pantheon utiliza as redes ponto-a-ponto, que replicam e migram essas informações e serviços pelos diversos pontos. Desta forma, o conteúdo que está sendo servido por um nó que tem a necessidade de se desconectar da rede, pode ser servido por outro nó que permanecerá conectado.

Esta redundância provê uma solução de alta disponibilidade a um custo baixo. Para que isto seja possível, é preciso um mecanismo de replicação e migração e, conseqüentemente, um de sincronização dessas informações. Sincronização porque, como o conteúdo dos nós são dinâmicos, existe a possibilidade de edição do conteúdo por outros nós. Logo, quando o conteúdo que estava desconectado, conectar-se, ou quando um conteúdo que está hospedado em mais de um lugar for modificado, terá que ocorrer uma sincronização das informações que garanta a consistência das mesmas.

Este mecanismo é o mesmo utilizado pelo *Prevayler*, ou seja, cada modificação realizada em um nó é encapsulada na forma de um comando. Este comando é enviado pela

rede aos pontos que possuem cópia do nó. O gerenciador de modificações do nó recebe o comando e se encarrega de reproduzi-lo no seu nó local, para que o mesmo se torne sincronizado com as demais cópias. Exatamente da mesma forma como o *Prevayler* realiza a carga dos dados de um sistema ao inicializá-lo. Entretanto, a abordagem atual do *Prevayler* não adota uma arquitetura de prevalência distribuída utilizando o modelo P2P, mas sim uma arquitetura distribuída utilizando o modelo “mestre-escravo”.

Nas arquiteturas Orientadas a Objetos Distribuídos (OOD) existentes, a distribuição de objetos é estática e centralizada, ou seja, não se reconfiguram de acordo com o comportamento do processamento. Essa abordagem implica num pequeno conjunto de máquinas servidoras configuradas previamente para realizar processamento. Esta distribuição estática impede tirar proveito maior da capacidade de processamento nas máquinas ditas “clientes”. Além disso, provoca um grande tráfego na rede e uma sobrecarga de processamento nos “servidores”. Isso representa um grande desperdício de recursos computacionais, devido ao aumento do número de computadores pessoais e PDA’s (*Personal Digital Assistant*) utilizados no mundo, além do aumento da capacidade individual de recurso dos equipamentos já utilizados.

A junção do mundo de computadores pessoais com a computação móvel é a origem da computação ubíqua. Uma vez que o usuário destas máquinas estará trocando continuamente de plataforma, esta mobilidade resultará em um requisito de que seu ambiente de trabalho o acompanhe. Este ambiente de trabalho juntamente com os dados que o integram, não poderão mais estar restritos a um nó físico, nem depender de um servidor central, que pode não estar disponível. Um exemplo importante é o modelo de OLPC (*One Laptop Per Child*) (HELMERSEN, 2006). Nele as máquinas confiam umas nas outras, porém nem sempre podem contar com uma ligação à *Internet*. Em um modelo de computação ubíqua, os dados têm que ser movimentados sob demanda. A demanda sobre os dados é feita pelas aplicações que estão requisitando-os e que precisam que eles estejam disponíveis e em nós próximos.

Esta dissertação apresenta e descreve uma nova abordagem, onde sistemas OOD tornam-se ponto-a-ponto, ao invés de usarem o modelo *3*-camadas. A distribuição de objetos é feita dinamicamente usando estratégias que levam a um processamento uniforme.

Este trabalho é organizado em oito capítulos. O primeiro deles corresponde a esta introdução, que exibe a motivação, e o objetivo deste trabalho, o problema que este trabalho pretende resolver, além de sua organização.

O segundo capítulo descreve algumas das principais arquiteturas e tecnologias que têm sido utilizadas no desenvolvimento da prevalência distribuída, servindo de base para o desenvolvimento deste trabalho. Além disso, esse capítulo aponta a relação entre a utilização de tais tecnologias e o problema descrito no primeiro capítulo.

O terceiro capítulo apresenta o estado da arte, onde para cada aspecto teórico relatado no segundo capítulo, descreve diversas abordagens propostas na literatura.

O quarto capítulo faz uma proposição de uma Prevalência Distribuída, apresentando uma proposta de arquitetura do Pantheon e a implementação desta arquitetura.

O quinto capítulo contém a análise de todo o processo de desenvolvimento deste trabalho. Além disso, esse capítulo também apresenta a avaliação deste trabalho, apresentando seus pontos positivos e negativos.

O sexto capítulo corresponde à conclusão. Esse capítulo apresenta uma visão geral deste trabalho, assim como suas contribuições e perspectivas futuras.

2. Revisão Teórica

O desenvolvimento da prevalência distribuída envolve a pesquisa em várias áreas. A prevalência distribuída pressupõe os seguintes aspectos: replicação da informação; sincronização e consistência; otimização do uso de memória e da rede e seus recursos; acesso e busca distribuída; mobilidade, segurança e robustez dos dados. Os aspectos teóricos abordados nesta seção procuram cobrir as necessidades relatadas anteriormente.

2.1. Aplicações Distribuídas

O desenvolvimento de aplicações baseadas na utilização de objetos distribuídos constitui um campo de pesquisa em crescente desenvolvimento. Apoiado por conceitos de orientação a objetos e utilizando-se da capacidade de comunicação provida pelo uso de redes de computadores, as aplicações de objetos distribuídos ganharam um impulso significativo com a expansão da Internet.

A característica básica desse tipo de aplicação é o fato de que os objetos que a compõem encontram-se dispersos em máquinas diferentes, podendo estar em diferentes locais, rodando sobre diferentes plataformas e comunicando-se mediante a utilização de um componente responsável por gerenciar as chamadas remotas, tais como ORB (*Object Request Broker*), RPC (*Remote Process Call*) e outros. A Figura 1 mostra a diferença entre aplicações centralizadas e aplicações distribuídas.

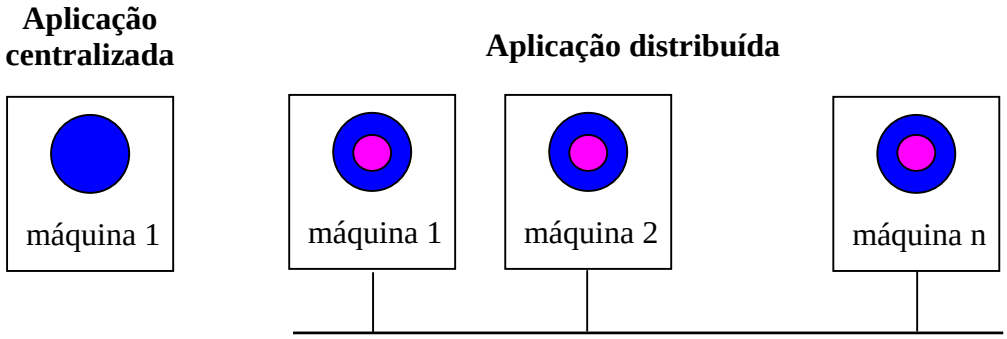

Figura 1 – Aplicação Centralizada x Aplicação Distribuída.

A utilização de sistemas de objetos distribuídos apresenta um conjunto de vantagens como (ORFALI, 1996):

- Transparência – esconde a natureza distribuída dos recursos da aplicação, permitindo que o desenvolvimento e manipulação dos objetos seja planejado e

utilizado sem a preocupação com localização, protocolos ou plataformas;

- Balanceamento de carga – um serviço pode estar replicado em várias máquinas, de tal modo que aumenta a disponibilidade do sistema e permite um balanceamento de uso de recursos, permitindo um melhor desempenho. No caso de objetos, permite que os objetos possam ser duplicados, fazendo com que na ocorrência de falha de um objeto, uma cópia do objeto possa assumir as funções do objeto inicial;
- Processamento distribuído – um serviço da aplicação também pode ser realizado por um conjunto de servidores que trabalham em cooperação. Essa distribuição das tarefas é mais uma maneira de otimizar o desempenho do sistema;
- Interoperabilidade e independência de plataforma – os clientes e servidores de uma aplicação podem estar distribuídos em plataformas de software e hardware diferentes. Dessa forma, os sistemas de objetos distribuídos buscam ser independentes de plataforma sendo capazes de trocar informações mesmo que sejam escritos em linguagens diferentes ou estejam sendo executados em sistemas operacionais e arquiteturas diversas;
- Escalabilidade – a replicação de dados e serviços dos sistemas, possível através da utilização de objetos distribuídos, permitem uma maior escalabilidade, ou seja, as aplicações ganham maior poder de atender a mudanças de demanda;
- Tolerância a falhas **–** a utilização de objetos distribuídos permite a replicação e a redundância de determinados serviços em servidores diferentes. Esta característica pode ser explorada para assegurar que tarefas essenciais sejam alocadas para outro servidor quando ocorrer falha no sistema.

A computação distribuída surge como forma de melhorar e mesmo resolver necessidades de troca e partilha de dados, assegurando a interligação de coleções heterogêneas de equipamentos dispersos geograficamente. A necessidade de sistemas distribuídos resulta de três motivos principais: os dados para as aplicações não estão acessíveis localmente; o equipamento necessário não existe localmente; e da necessidade de maior capacidade de processamento.

Igualmente, os sistemas distribuídos introduzem melhorias de custo e desempenho, através da otimização de partilha de recursos, de melhoria da gestão dos sistemas e pelo aumento de facilidades e recursos do sistema. Um sistema distribuído pode ser descrito como

tendo por base uma rede de componentes, locais e remotos. As aplicações distribuídas permitem novos modos de utilização dos sistemas existentes e a criação de novos sistemas com novas formas de interagir com as aplicações já existentes.

O grande campo de aplicações distribuídas se encontra nas modalidades comerciais e científicas, que amealham recursos e reconhecimentos suficientes para justificá-las. A intenção deste trabalho é reduzir o patamar de entrada no domínio distribuído para que possa ser usado em aplicações do dia-a-dia. A principal característica de sistemas distribuídos explorada é a distribuição geográfica que possa permitir a sobrevivência de dados independentemente da conectividade de nós na rede.

2.2. Computação Ubíqua (Computação Pervasiva)

Hoje em dia, é possível assumir que um novo paradigma de computação está emergindo. Este novo paradigma de computação que está sendo proposto sugere muito mais do que apenas trazer a computação para atividades onde seu uso não era trivial, mas uma revolução na forma de interação homem máquina. O novo paradigma, apresentado por Mark Weiser (1991), define ambientes de computação ubíqua (*ubiquitous computing* - computação em todo lugar) como espaços onde computadores estão disponíveis no ambiente físico, mas efetivamente invisíveis para o usuário. Outros autores também se referem a este tipo de computação como computação pervasiva. Ao longo da história o homem tem criado diversos meios e ferramentas que melhorem sua convivência e a comunicação social. Quem não gostaria de levar o seu computador a qualquer lugar e utilizar os seus arquivos e programas a qualquer hora? Ou interagir com os dados que estão armazenados em dispositivos remotos sem ter que estar sentado a frente de um computador em sua casa ou empresa, mas enquanto está no avião ou no meio da mata realizando uma pesquisa sobre plantas ou animais, ou sentado debaixo de uma árvore na sua casa de campo? Isto não era possível até há pouco tempo atrás.

A idéia básica da computação pervasiva (*pervasive computing*) é disponibilizar acesso computacional de modo invisível em todo o lugar todo o tempo. Invisível no sentido de que o usuário não precisa conhecer a tecnologia, mas apenas desfrutar dos benefícios que ela possa oferecer, sendo na utilização dos computadores pessoais, de PDA’s (*Personal Digital Assistant*), de celulares, de acessórios como relógios ou óculos e até mesmo na própria roupa e corpo humano. A tecnologia vai migrar do computador pessoal para os dispositivos do dia-a-dia com tecnologia embutida e conectividade à medida que a computação se torne

progressivamente menor e mais potente. Uma tecnologia que permitirá que sistemas computacionais alterem drasticamente o quotidiano de cada ser humano, sem que ele perceba.

A computação pervasiva é uma área recente de pesquisa, considerada o novo paradigma do século XXI, que visa fornecer uma computação onde, quando, o que e como se deseja, através da virtualização de informações, serviços e aplicações. Este ambiente computacional consiste de uma grande variedade de dispositivos de diversos tipos, móveis ou fixos, aplicações e serviços interconectados.

Uma das tecnologias para lidar com este novo ambiente computacional pode ser a computação distribuída em larga escala, objeto foco da computação em grade (*grid computing*). Aplicações computacionais de processamento intensivo, executadas numa infra-estrutura de grade, requerem o uso coordenado e compartilhado de recursos em larga escala oferecidos por diferentes organizações. Como a disponibilização destes recursos é dinâmica, as aplicações que os utilizam precisam ser construídas de forma distribuída e adaptativa ao contexto (recursos e serviços) correntemente disponível.

Assim sendo, um dos principais desafios da computação ubíqua são as aplicações contextuais que implicam na capacidade de ensinar computadores sobre o ambiente corrente e como reagir quando o seu usuário muda de um ambiente para outro. Como exemplo, pode-se citar a ativação da opção de vibra *call* do celular quando seu usuário entra em uma sala de reunião ou conferência. Informações contextuais podem incluir não só a localização do usuário, mas também seu estado físico como temperatura e batimento cardíaco, seu estado emocional, histórico comportamental, entre outros.

O sustentáculo à computação ubíqua implica não só em grandes desafios tecnológicos, mas também em mudanças organizacionais e dos modelos de negócios. Em termos tecnológicos, envolvem computadores de pequeno porte com baixo consumo de potência; sensores e atuadores; redes sem fio e redes de alta velocidade; processamento distribuído; sistemas tolerantes a falhas; interfaces polidas, entre outros. Dependem de uma infra-estrutura de comunicação de alta disponibilidade e segurança. Quando você pensa em computação disponível em qualquer tempo e lugar para uma comunidade global, o provedor de acesso e serviços passa a ser distribuído e surgem novos desafios de segurança principalmente no que se refere à garantia de privacidade e autenticação confiável.

Além disso, mudam-se os modelos de negócios. O relacionamento com o cliente continua sendo baseado nas três premissas básicas: conhecimento, acessibilidade e retorno ágil. O conhecimento deve ser bilateral, isto é, empresa-cliente. Ambos devem ser acessíveis e a empresa deve responder pró-ativamente e adequadamente às necessidades do seu cliente. Tudo isso em um novo cenário, pois com a computação ubíqua, a localização do seu cliente passa a ser a localização do seu negócio. Um dos principais desafios passa a ser como estender os serviços da companhia para toda a localização onde seus produtos podem ser usados e comprados e responder às necessidades do cliente sem invadir sua privacidade.

Do ponto de vista social, devem ocorrer, também, mudanças no modo de se comunicar, colaborar, coordenar, organizar, supervisionar e gerenciar. Tais mudanças já vêm sendo sentidas nas empresas com a adoção de comunidades virtuais, compostas por pessoas geograficamente distribuídas, no desenvolvimento de projetos multinacionais e multifuncionais. Neste novo cenário, têm surgido novos conceitos de gerenciamento de projetos e supervisão de funcionários. Avaliações de funcionários baseadas em presença física e tempo de permanência nos escritórios passam a ficar ainda mais obsoletas, sendo dada ênfase cada vez mais a métricas baseadas em resultados. Como desvantagem, a possibilidade de trabalhar em qualquer lugar em qualquer tempo dificulta a separação entre o local e tempo de trabalho e de lazer.

Todas essas mudanças vêm sendo introduzidas gradativamente com a própria evolução do *e-business* em direção ao *m-business* (computação móvel) e à medida que novos serviços e produtos vão sendo lançados, uma nova cultura vai sendo criada influenciando o comportamento individual, organizacional e da sociedade como um todo e mudando definitivamente a relação cliente-empresa. Assim, modelos de negócios dinâmicos e altamente adaptativos são mais do que nunca essenciais para a sobrevivência neste mercado mutante.

A contribuição deste trabalho para o advento da computação ubíqua é o estabelecimento de um modelo simples de persistência compatível com a natureza de aplicações ubíquas. As aplicações poderão transitar entre os diversos dispositivos e o fluxo de requisições que estas aplicações fazem sobre os dados servirá como um farol para guiar esses dados até os nós onde eles deveriam estar. Com isso, os dados serão replicados nas proximidades das aplicações que os estiverem requisitando.

2.3. Virtualização da memória (*Utility Computing*)

Utility Computing (Utility Computing, 2005), ou *on-demand computing* (computação sob demanda), é um modelo de computação em que é possível pagar pelos recursos à medida que estes são consumidos. A idéia central, com este modelo, é otimizar o gasto em infra-estrutura computacional. Dada uma demanda por recursos, alocam-se os equipamentos necessários e, caso a demanda aumente, aumenta-se o poder de processamento, banda de rede, armazenamento e até mesmo aplicações.

É possível alocar recursos especificamente para um ou para um grupo de usuários, em função de necessidades periódicas. Desta forma, durante os picos de utilização, os recursos são divididos e alocados de maneira mais inteligente do que acontece usualmente.

Imagine a seguinte situação: uma usina hidrelétrica, que gera capacidade de computação. Você tem um centro que gera capacidade de processamento para os outros computadores, e a energia que é gasta se ajusta de acordo com a demanda, ou seja, como na energia fornecida pelas usinas hidrelétricas, você só paga pelo que consome. Por exemplo: o funcionamento do sistema de computação no horário de pico de um banco, quando muitas pessoas precisam ter acesso e muitas operações são realizadas ao mesmo tempo. O *utility computing* ajusta o gasto de energia e a capacidade de operação do sistema na medida em que as demandas vão chegando, para atender aos usuários da melhor maneira possível em questão de tempo de resposta, de disponibilidade e de garantia de segurança.

A idéia central é fazer com que os recursos sejam adquiridos conforme a necessidade. Para isso, é necessário que haja um modelo adequado e que permita aumentar a capacidade dos recursos, de acordo com as necessidades dos serviços. Desta forma, processamento, largura de banda de rede e armazenamento, que são os principais recursos computacionais, precisam estar adaptados a mudanças sazonais de utilização. O aumento ou diminuição da capacidade dos recursos deve ser realizado sem interferir no funcionamento geral da infra-estrutura. Os equipamentos funcionam como cartuchos, que podem ser adicionados e retirados sem que, para isso, seja necessário interromper a execução dos demais módulos.

Existe a hipótese de que o usuário sequer precise adquirir uma infra-estrutura dedicada ao serviço que pretende realizar. Com isso, é possível compartilhar os recursos com outros usuários, minimizando a necessidade de investimentos em hardware e software,

incluindo os básicos, como sistema operacional, ou até mesmo aplicativos de controle operacional e gerencial.

Esta não é uma idéia nova. A constatação dessa necessidade e a idealização de um modelo com estas características foi concebida na década de 60 (ROSS, 2004). Imaginou-se que a informática seria comparável a uma usina fornecedora de energia elétrica ou a uma empresa de telefonia. O paralelo criado coloca a informática como uma concessionária de serviços, de onde se solicita aquilo que é necessário e no momento que se quer utilizar, como fazemos com a energia elétrica: acionamos o interruptor e temos a iluminação. O usuário é "bilhetado" por aquilo que consume e se, eventualmente, for necessário mais energia, a concessionária será responsável por fornecê-la.

Atualmente, o modelo de *utility computing* está restrito a grandes companhias que possam agregar uma grande quantidade de redes de computadores clusterizados e *blade servers* (são computadores multiprocessados que possuem uma ou mais CPUs em uma única placa-mãe) (RAJAMANI, 2003). No entanto, em sua concepção, *utility computing* poderia se estender a toda comunidade de usuários de computador que possa oferecer seus recursos computacionais. Nesta tese, apresentaremos um modelo de virtualização de memória que pode funcionar autonomamente sem a necessidade de supervisão de grandes companhias. Este modelo permitirá que os pequenos usuários subsidiem o custo de seus equipamentos ou até mesmo obtenham uma renda extra com o arrendamento dos seus recursos computacionais.

3. Estado da Arte

Para cada aspecto teórico relatado no capítulo anterior, existem diversas abordagens propostas na literatura. Neste trabalho, procurou-se estudar as propostas mais relevantes e que se apresentam em um estágio maior de desenvolvimento, onde os propósitos de uso são feitos de forma direta ou através da aplicação dos conceitos.

3.1. Trabalhos Relacionados

Cache e Gerência de Objetos Distribuídos

O modelo de gerenciamento do ciclo de vida de objetos no retículo distribuído tem uma analogia ao processo de *caching* de memória e dados. Entre os trabalhos que tratam deste problema podemos citar Chan *et al* (1999), Godart *et al* (2004), Zhou *et al* (2004), Oliveira *et al* (2002) e Hünn *et al* (2003).

Chan *et al* (1999) propõe o uso de *cache* para mitigar a latência de acesso a banco de dados orientado a objeto. Ele propõe o uso de *leasing* para controlar a validade de cópias do *cache*.

Godart *et al* (2004) apresenta o compartilhamento baseado em um repositório centralizado. Ele sugere a replicação de objetos versionados e a separação entre espaço de trabalho local e compartilhado. A atualização do espaço compartilhado é feita periodicamente. Esta separação é defendida para garantir a disponibilidade de objetos caso o nó do usuário desconecte.

Zhou *et al* (2004) argumenta sobre o problema da incompatibilidade de impedância entre objetos e o modelo relacional. Ele propõe uma solução intermediária baseada em um modelo de persistência orientado a objetos. O autor argumenta que algoritmos de *cache* baseado em objetos são mais simples e eficazes do que sua contrapartida em tuplas relacionais.

Oliveira *et al* (2002) afirma que a distribuição de conteúdo na Internet tem migrado de uma arquitetura onde os objetos são armazenados em um único servidor para uma arquitetura onde os objetos são replicados em servidores distribuídos geograficamente. Neste caso, os clientes acessam transparentemente a cópia mais próxima do objeto desejado,

melhorando o desempenho do sistema. Ele apresenta um modelo para o estudo destas redes de distribuição de conteúdo. Utilizando este modelo, avalia quatro heurísticas para a replicação de objetos através de simulação. Segundo o autor, os resultados dos experimentos mostram o ganho significativo de desempenho do sistema ao utilizarmos qualquer um dos algoritmos. Além disso, os melhores resultados foram obtidos quando todos os servidores trabalham cooperativamente para a replicação dos objetos.

Hünn *et al* (2003) ressalta os benefícios obtidos por desenvolvedores Java com a utilização do banco de dados pós-relacional *Caché* (InterSystems, 2001) na criação de sistemas. Ele afirma que um dos principais problemas enfrentados pelos desenvolvedores que trabalham com a linguagem Java é que suas aplicações orientadas a objetos utilizam dados armazenados em sistemas relacionais, o que resulta em muitas horas extras de trabalho e aumento significativo de custo para o mapeamento entre os dois paradigmas. Por esta razão, ele propõe uma solução utilizando o banco de dados relacional *Caché*, considerado pelos profissionais uma ótima opção, já que viabiliza a conversão automática de dados relacionais para o formato objeto.

Ele assegura que a combinação única entre objetos e SQL do *Caché* fornece estrutura de desenvolvimento muito interessante para programadores de orientação a objetos, indo além do que é oferecido por bancos de dados puramente relacionais ou orientados a objetos. Além disso, o *Caché* eliminou o problema de impedância entre objetos e tabelas, pois a sua arquitetura de dados permite o desenvolvimento rápido de aplicações (RAD) de sistemas e aplicações complexas. Além disso, disponibiliza um sistema de banco de dados poderoso, de alta performance, maciço e escalável, que suporta uma grande quantidade de protocolos padrões, linguagens de programação e tecnologias de *middleware*.

Replicação e Migração de Objetos

Barbosa *et al* (2001) discute a mobilidade e replicação em sistemas de objetos distribuídos. Ele argumenta que o uso de sistemas que permitem mobilidade de objetos está aumentando com o uso da Internet, envolvendo assim redes heterogêneas, conectadas por diferentes *links* de comunicação. A replicação em sistemas distribuídos é usada para oferecer segurança e persistência ou para aumentar o desempenho do processamento das informações. Apresenta um modelo de replicação em ambientes de objetos distribuídos que permitem mobilidade desses objetos.

Este modelo tem como objetivo prover um ambiente de execução para apoiar o desenvolvimento de aplicações envolvendo mobilidade explícita e replicação implícita. Assim, o gerenciamento e a consistência das réplicas é feito automaticamente por este modelo. A implementação do modelo foi feita para o ambiente Java. O trabalho traz como principais contribuições o uso de replicação em ambientes que permitem mobilidade e a independência do modelo de mobilidade de objetos, uma vez que esta é explícita, bastando informar apenas a primitiva responsável por mover o objeto.

Em seu trabalho, Richa (RICHA *et al*, 1997) descreve o problema de acesso a recursos distribuídos em uma rede. O objetivo do estudo é projetar um algoritmo eficiente de acesso a objetos compartilhados em uma rede de longo alcance. A quantidade de parâmetros nestas condições pode ser elevada e difícil de levar em conta individualmente. Para superar este problema, o autor estabelece um modelo, onde agrega todos estes valores em um só, que é denominado custo de comunicação. Esta simplificação é suficiente, pois o autor só está interessado em otimizar o desempenho da rede.

Para simplificar ainda mais, somente a operação de leitura foi estudada, uma vez que o interesse era averiguar o princípio de localidade. O algoritmo de localização de recursos está baseado em uma árvore balanceada. Cada nó representa a raiz de uma sub-árvore. A busca dos recursos se dá a partir no nó corrente, navegando nos sub-nós da árvore ou subindo no nó pai se houver necessidade.

O trabalho estima o custo de comunicação baseado no fato que o modelo escolhido de árvore se assemelha com a Internet. Este custo é avaliado em diversas situações envolvendo configurações e operações na árvore, assumindo a complexidade computacional de cada situação. A solução proposta inclui uma rotulação dos nós independente de localidade. A otimização do custo leva em conta uma busca probabilística em torno de esferas que determinam a localidade de acesso.

O proposta tem similaridades com o problema abordado neste trabalho, e muitas idéias podem ser aproveitadas. No artigo, a preocupação é com a performance, neste trabalho a performance tem baixa prioridade, pois se busca principalmente a persistência dos dados. A solução de estrutura de acesso é semelhante à deste trabalho, pois também usa uma árvore. Desta forma, alguns conceitos estudados no artigo poderão ser aplicados nesta proposta, caso em um trabalho futuro se estude a otimização de performance.

Assim como em (RICHA *et al*, 1997), este trabalho tem um modelo de custo, porém ele não é unitário, mas dividido em fatores que possam ser prioritários para cada aplicação como memória, processamento e disponibilidade. O artigo foca em operações de leitura, pois está centrado em performance, mas ao contrário do que diz o título, não aborda diretamente o problema de alocação. Como este trabalho trata da persistência, os problemas mais estudados são a alocação, atualização e sincronização de objetos distribuídos.

A proposta integra o problema de busca dos nós no custo de acesso. Neste trabalho, estas duas vertentes do acesso distribuído estão separadas. A busca se dá no momento de alocação do objeto, e não é o assunto central, pois remete ao trabalho em (SILVA, 2006). A partir desta busca, a árvore de referências é criada, e a operação se dá em cima dela. No artigo, se pressupõe que os objetos já estão alocados e a otimização se dá ao escolher as cópias mais relevantes. No caso deste trabalho, todas as cópias têm que estar envolvidas, para garantir que todas serão sincronizadas. O algoritmo do artigo só serviria para ordenar a seqüência em que as cópias seriam atualizadas, caso se possa postergar a atualização das cópias mais custosas.

3.2. Tecnologias

Computação Distribuída

O aparecimento de redes de computadores permitiu a utilização de um novo paradigma computacional que se mostrou, com o passar do tempo, extremamente poderoso. Estamos nos referindo à possibilidade de distribuição do processamento entre computadores diferentes. Mais do que a simples sub-divisão de tarefas, este paradigma permite a repartição e a especialização das tarefas computacionais conforme a natureza da função de cada computador.

Em sistemas distribuídos, a necessidade de disponibilizar serviços em larga escala, com alta confiabilidade e disponibilidade tem motivado a construção de sistemas com características de tolerância a falhas e bom desempenho. Estes requisitos podem ser favorecidos através da replicação de dados e serviços (objetos). A replicação oferece dois principais benefícios aos sistemas de computação. Com a divisão da carga gerada por requisições de usuários entre os vários objetos replicados e cada usuário buscando acessar a réplica mais próxima, pode-se melhorar o tempo de resposta. O segundo benefício, e o mais

importante, é que com replicação, consegue-se manter o serviço disponível mesmo na ocorrência de falhas locais ou de comunicação.

Por outro lado, a implantação de replicação em um sistema computacional, necessita cuidados com relação à manutenção da consistência dos objetos replicados. A existência de muitas réplicas de um mesmo objeto em diferentes locais implica na utilização de algum método de controle a fim de garantir consistência. Existem vários métodos de controle de réplicas, conhecidos como *Protocolos de Gerenciamento de Réplicas* (PASIN *et al*, 1999).

De maneira geral, a consistência pode ser alcançada se as operações que modificam o estado das réplicas forem executadas em todas as réplicas ou nenhuma (atomicidade) e na mesma ordem (ordenação). A idéia de enviar mensagens a múltiplos destinatários e de ordenação destas mensagens faz parte do modelo de comunicação de grupo. Sistemas que implementam o modelo de comunicação de grupo oferecem ao programador da aplicação as primitivas de *multicast* necessárias para implementar replicação e por isso são utilizados como ferramentas de suporte para esse tipo de aplicações.

Num sistema distribuído, as vantagens obtidas com replicação dependem de vários fatores como: número e localização das réplicas, protocolo de gerenciamento de réplicas utilizado, carga gerada por invocações de clientes, confiabilidade dos componentes do sistema, interdependências entre objetos, etc. Por esse motivo, esses fatores devem ser bem definidos, de acordo com o sistema considerado, para que se alcance os resultados desejados.

Sistemas de Comunicação de Grupo

Os Sistemas de Comunicação de Grupo são mecanismos de *software* responsáveis pelo gerenciamento da troca de mensagens entre processos executados em diferentes computadores de uma rede, oferecendo várias primitivas para ordenação de mensagens, manutenção de grupos lógicos e graus de confiabilidade na distribuição das mensagens. Suas principais aplicações práticas relacionam-se à implementação de memória compartilhada distribuída, serviços de alta disponibilidade, monitoramento de serviços ou servidores, ferramentas colaborativas ou replicação de bases de dados.

Estes sistemas são normalmente implementados utilizando protocolos de rede não confiáveis como UDP (*User Datagram Protocol*), operando não só sobre redes locais como também sobre redes de longa distância onde a latência e perda de pacotes podem ser

significativas. Uma característica dos Sistemas de Comunicação de Grupo é a ordenação das mensagens dentro de grupos lógicos de processos. A diretiva de ordenação total é um recurso para a ordenação global de transações dentro do grupo de servidores que compõem um sistema de banco de dados distribuído. A diretiva de ordenação total garante que todas as mensagens enviadas pelos servidores a um grupo lógico são entregues a todos os componentes em uma mesma ordem.

O JavaGroups (BAN, 1999) é um conjunto de ferramentas para comunicação confiável em grupos. Clientes podem se conectar a grupos, enviar (para todos ou para somente um membro do grupo) e receber mensagens. Ele acompanha cada membro de um grupo, notificando-os quando um novo membro se conecta ao grupo ou quando um membro já existente sai ou quebra. Um grupo é identificado pelo seu nome. Um grupo não precisa ser criado explicitamente, quando um processo se conecta a um grupo não existente, ele é criado automaticamente.

Para se conectar a um grupo, um processo precisa criar um canal (*channel*) e conectar-se a ele utilizando o nome do grupo (todos os canais com o mesmo nome, formam um grupo). Cada canal possui um único endereço e conhece quem são os outros membros daquele grupo (todo membro do grupo possui uma lista com o endereço de cada canal – chamado no JavaGroups de *view*).

Este conceito de Computação em Grupo resolve parte do problema de comunicação requerida. Porém não foi utilizado em favor de uma solução melhor: JXTA.

JXTA

Até recentemente, a tecnologia P2P (*Peer-to-Peer*) esteve sendo utilizada em aplicações como comunicadores instantâneos. Levando o conceito de P2P mais adiante, a *Sun Microsystems* concebeu a idéia do Projeto JXTA (*Juxtapose*) como um meio de integrar a tecnologia P2P ao núcleo da arquitetura de rede.

O Projeto JXTA (WILSON, 2001) é um conjunto de protocolos P2P simples e abertos que habilitam os dispositivos na rede a se comunicarem, colaborarem e compartilharem recursos. No projeto *JXTA* todos os dispositivos são *peers*. Os *peers* JXTA criam uma rede virtual *ad hoc* no topo de redes existentes, mascarando a complexidade existente nas camadas de baixo. Na rede virtual JXTA, qualquer *peer* pode interagir com

outros *peers*, independente de sua localização, tipo de serviço ou ambiente operacional – mesmo quando alguns *peers* e recursos estão posicionados atrás de *firewalls* ou estão em diferentes tecnologias de transporte de rede. Assim, o acesso aos recursos da rede não é limitado por incompatibilidades de plataforma ou restrições da arquitetura cliente/servidor.

No caso de um dos *peers* estar protegido por um *firewall*, é utilizado o protocolo HTTP normal para a comunicação através do *proxy* da rede protegida pelo *firewall*. Assim, é necessário saber à priori o destino da comunicação porque mecanismos baseados em *broadcast* não funcionam através da *firewall*. Neste sentido, um *peer* pode ser um *rendezvous peer*. *Rendezvous Peers* são nós especiais na topologia da rede que atuam como pontos de encontro para nós que não conseguem utilizar mecanismos de descoberta automáticos.

Por exemplo, considerando a Figura 2 como a topologia atual de uma rede JXTA, se o *Peer A* fizer um pedido de alguma informação o pedido é enviado para o *Peer R1*. Este *peer* é de *rendezvous* ou seja, caso não possua a informação pedida encaminha o pedido para os *peers* que conhece (R2 e R3). Estes efetuam o mesmo procedimento: verificar se existe a informação requerida pelo pedido e encaminhar. O funcionamento de cada *peer de rendezvous* é semelhante ao de um roteador da Internet.

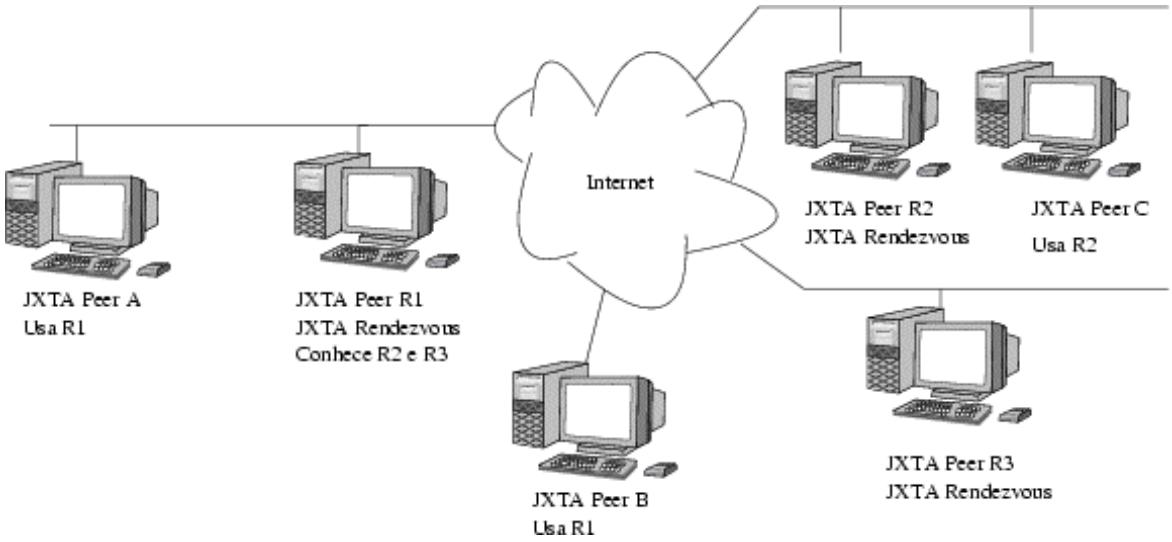

Figura 2 - Exemplo de Topologia de uma rede JXTA.

A tecnologia do Projeto JXTA adota como objetivos: interoperabilidade (entre diferentes sistemas e comunidades P2P), independência de plataforma (diversas linguagens, sistemas e redes), generalidade (qualquer tipo de dispositivo digital) e segurança. A tecnologia JXTA funciona em qualquer dispositivo, incluindo aparelhos celulares, PDAs, sensores eletrônicos, estações de trabalho e servidores. Baseado em tecnologias aprovadas e

padronizadas como HTTP, TCP/IP e XML, JXTA não é dependente de nenhuma linguagem de programação particular, sistema de rede, ou plataforma de sistema e pode trabalhar com uma combinação dos mesmos.

JXTA está posicionada como uma pilha P2P, uma camada localizada acima do sistema operacional ou máquina virtual e abaixo das aplicações e dos serviços P2P. Pode ser descrita simplesmente como uma tecnologia que permite a comunicação entre *peers*. Cada *peer* é associado a um identificador único, um "*peer* ID", e pertence a um ou mais *peergroups*. Dentro dos *peergroups*, os *peers* cooperam e têm funções similares sob um conjunto unificado de capacidades e restrições. JXTA provê protocolos para as funções básicas: criar e encontrar grupos, entrar e sair de grupos, monitorar os grupos, conversar com outros grupos e *peers*, compartilhar conteúdo e serviços – tudo isso é realizado através da publicação e troca de anúncios XML e mensagens entre os *peers*.

Conceitualmente, cada *peer* no JXTA abstrai três camadas: o núcleo, a camada de serviços e a camada de aplicação. Essas camadas estão ilustradas na figura 3

Núcleo – é responsável por gerenciar o protocolo JXTA. Engloba os elementos básicos necessários a uma rede *peer-to-peer*, incluindo *peers*, *peergroups*, descoberta, comunicação, monitoramento e aspectos de segurança relativos. Ou seja, ele contém as funcionalidades e a infra-estrutura suficientes para o desenvolvimento de qualquer aplicação P2P. Esta camada é compartilhada por todos os dispositivos para que a interoperabilidade se torne possível.

Camada de Serviços – É a camada que armazena as funcionalidades comuns que mais de um programa P2P poderia utilizar. Inclui serviços de rede que não são essenciais à rede *peer-to-peer*, mas são comuns ou desejáveis neste ambiente. Exemplos de serviços são busca e indexação, diretórios, sistemas de armazenamento, compartilhamento de arquivos, agregação de recursos, tradução de protocolos e autenticação. Ela provê funcionalidades similares à de uma biblioteca, que pode ser controlada pelas aplicações JXTA através de lógica na camada de aplicação.

Camada de Aplicações – É a camada onde a aplicação P2P realmente reside. Ela pode permitir que o usuário controle diferentes serviços, ou pode ser onde a lógica de uma aplicação autônoma opera. Esta camada abrange mensagens instantâneas, gerenciamento de conteúdo, sistemas de e-mail, sistemas de leilão distribuídos, entre outros. São programas que

utilizam os serviços disponibilizados na camada de serviços. Por exemplo, um simples programa de bate-papo pode ser construído nessa camada, fazendo uso tanto do serviço quanto do núcleo para permitir que os *peers* troquem mensagens.

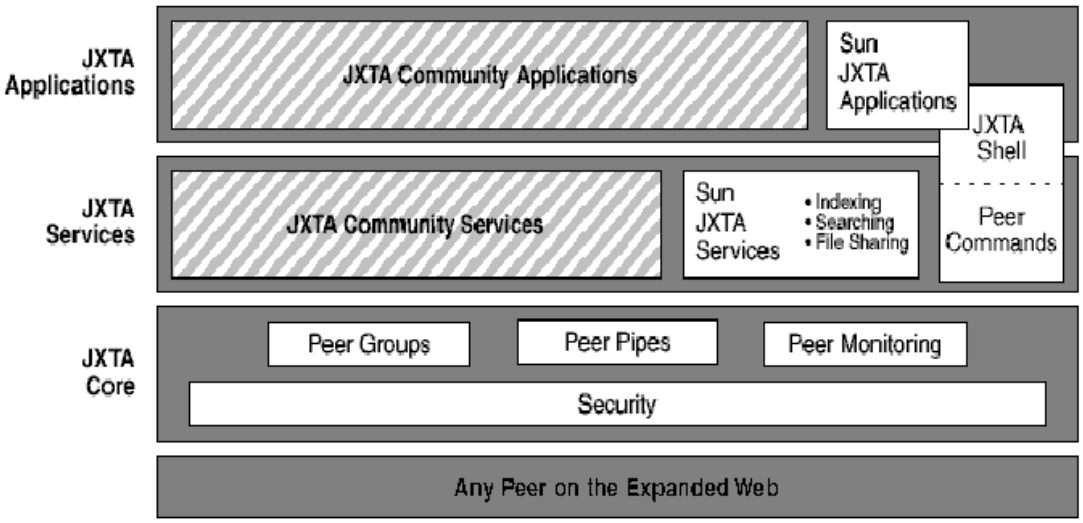

Figura 3 – Camadas do Projeto JXTA.

Para desenvolvedores, o Projeto JXTA provê um conjunto de blocos que permitem uma sólida fundação para aplicações computacionais distribuídas e dão apoio a funções comuns requeridas por qualquer sistema P2P. Utilizando esses recursos, os desenvolvedores podem elaborar suas aplicações P2P mais facilmente. Já estão disponíveis APIs em Java, C++ e outras linguagens. Os protocolos JXTA estão especificados em alto-nível e, portanto, podem ser implementados teoricamente em qualquer linguagem.

A arquitetura JXTA está atualmente em sua segunda versão. Modificações significativas foram realizadas a fim de criar redes P2P de melhor desempenho, maior escalabilidade e facilidade de manutenção.

Prevalência de Objetos

Prevalência de objetos é uma abordagem à persistência de objetos desenvolvida em novembro de 2001 por Klaus Wuestefeld (VILELLA, 2002; BRITO, 2004) e concretizada no projeto *Prevayler*. A utilização de prevalência é uma alternativa ao uso dos bancos de dados. Sua premissa maior é de que os objetos de negócio estarão sempre disponíveis na memória principal do computador, no formato da linguagem utilizada no desenvolvimento da aplicação e há garantia de que eles serão recuperados fielmente caso haja alguma queda de energia ou falha na aplicação.

O mecanismo que mantém persistente os objetos de negócio é baseado no padrão de projetos comando. Este padrão protege todo código da aplicação que interfere no estado atual do sistema. Do estado do sistema entende-se o valor dos atributos de todos os objetos que compõem o sistema, menos os objetos transientes, ou seja, os que não têm vida útil após a execução do sistema.

O Padrão de Projetos Comando

O padrão de projetos comando (GAMMA, *et al.*, 1995) encaminha uma requisição a um a um objeto específico. Ele retém a requisição para uma ação particular dentro de um objeto e dá ao objeto uma interface pública conhecida. Isso dá ao cliente a capacidade de fazer requisições ao objeto sem conhecer nada a respeito da ação que será executada, e permite ao desenvolvedor modificar esta ação sem afetar o cliente sob qualquer aspecto.

Uma forma de assegurar que todo objeto receba os seus próprios comandos diretamente é utilizando o objeto Comando. O objeto Comando sempre possui o método *execute* que é invocado no momento da execução do comando. Um objeto Comando implementa, ao menos, a seguinte interface:

```
public interface Commando {
      public void execute();
}
```

Listagem 1 – Interface Commando.

Então é possível prover um comando para cada ação a ser realizada no objeto alvo. Na invocação do método execute, o objeto alvo pode fornecer ao comando alguma informação, ou alguma dependência. Esta abordagem é utilizada na prevalência de objetos.

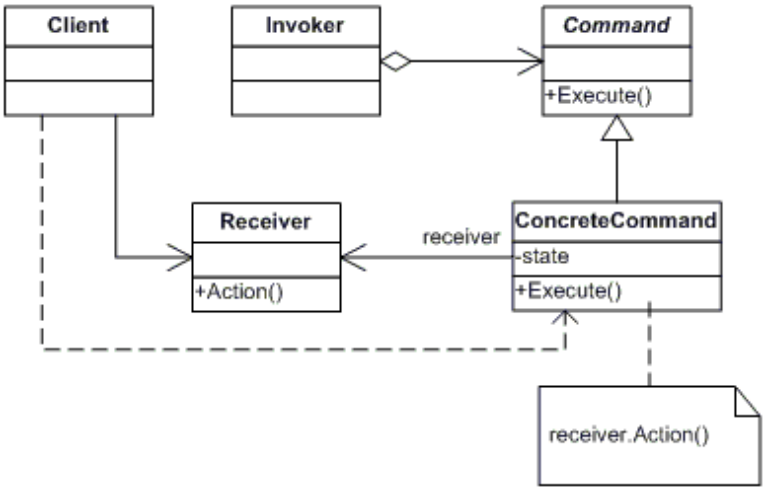

Figura 4 - Padrão de Projeto Comando.

No padrão de projetos Comando (Figura 4, o cliente cria uma instância de um ComandoConcreto. Este comando depende de um receptor que é passado através do seu construtor. O Cliente repassa o Comando ao Invocador, que detém o conhecimento de como e quando invocar a execução do Comando.

Um propósito importante do padrão Comando é manter o cliente e o receptor completamente independentes das ações que iniciam. Ou seja, não é necessário que o cliente conheça os detalhes de acesso ao receptor. O padrão Comando também pode ser utilizado quando é necessário executar uma operação e o sistema deve decidir quando os recursos estarão disponíveis para tal. Nesses casos, o programa está empilhando os comandos para serem executados mais tarde. Por último, os Comandos podem ser utilizados para lembrar operações já executadas de tal forma que possam ser executadas novamente no sistema. É nesta última finalidade que se baseia a prevalência de objetos.

Serialização

Os comandos são armazenados em uma memória duradoura através da serialização. Segundo a Sun Microsystems:

> *"Serialização de objetos suporta a codificação de objetos, e os objetos acessíveis através dele, dentro de uma seqüência de bytes; e suporta a reconstrução complementar do grafo do objeto a partir da mesma seqüência. A serialização é utilizada para persistência leve e para comunicação via sockets ou Invocação*

> *Remota de Método (RMI). A codificação padrão dos objetos protegem dados privados e transientes e suportam a evolução da classe. Uma classe pode implementar a sua própria codificação e será solenemente responsável pelo seu formato externo”.*

Objetos a serem serializados freqüentemente fazem referência a outros objetos. Estes outros objetos serão armazenados e resgatados ao mesmo tempo para manter o relacionamento entre os objetos intacto. Quando um objeto é serializado, todos os objetos acessíveis por ele serão também serializados.

Arquitetura de Prevalência

Cada comando executado no sistema é serializado de forma a manter um registro das ações que levaram a modificações nos objetos de negócio. De tempos em tempos o sistema gera uma fotografia dos objetos, isto é, todo o sistema é armazenado em um único lote, gerando um marco. A partir deste marco, os comandos anteriores a sua gravação podem ser descartados e os comandos subseqüentes serão adicionados ao registro de comandos. A Figura 5 apresenta uma visão geral da arquitetura de um sistema prevalente.

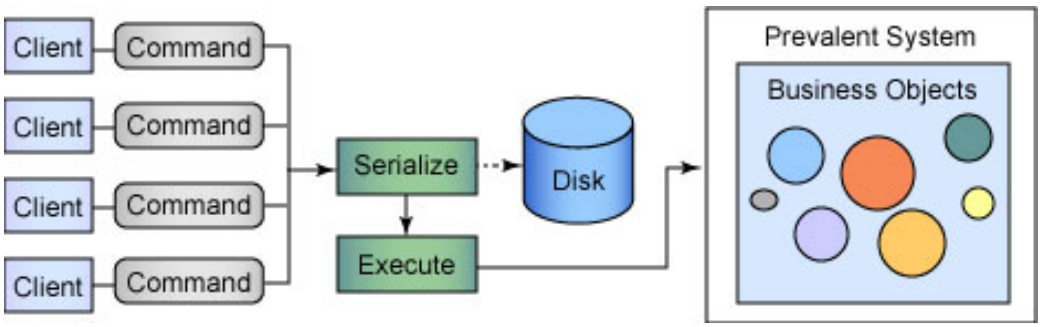

Figura 5 - Arquitetura de prevalência.

A premissa mais extrema do mecanismo de prevalência é o fato de os objetos de negócio estarem alocados todo o tempo em memória. A conseqüência desta premissa é a necessidade de haver memória disponível para alocar todo o sistema de uma só vez. Isto significa que, na arquitetura atual de processadores de 32 bits, o limite virtual de memória pode ser um entrave decisivo para a aplicabilidade desta tecnologia. Entretanto, com o

advento e a proliferação das arquiteturas de 64 bits, este limite não mais existirá para fins práticos.

Os comandos são transações do ponto de vista da aplicação e podem alterar o estado do sistema, persistir esta alteração, ou falhar, cancelando as alterações. A execução de um comando, por si, é uma transação, só que o comando não está limitado por operações SQL, qualquer operação pode ser cancelada e recuperada em caso de falha.

O sistema prevalente é tolerante a falhas no sentido que ele pode recuperar o seu estado anterior à uma queda - forçada ou imprevista - executando o conjunto de comandos armazenados até que o último comando restaure o estado final do sistema. Os comandos são recuperados e aplicados aos objetos de negócio da mesma forma que foram executados da primeira vez, incluindo alguns recursos disponíveis naquele momento, como por exemplo, o relógio do sistema. Dessa forma, os objetos de negócio devem operar de uma forma determinística, isto é, dado um determinado valor de entrada, o valor de saída será sempre o mesmo.

Protocolo SNMP

Devido ao crescimento das redes de computadores, que têm se tornado grandes redes interconectadas (Internet), fez-se necessário a criação de protocolos de gerenciamento que simplificasse, o monitoramento dos equipamentos em uma rede de computadores. Além disso, tornou-se necessário haver integração e comunicação entre os equipamentos a serem gerenciados e o administrador da rede, fornecendo a este as informações necessárias para garantir que a integridade da rede seja mantida, bem como prevenir possíveis falhas.

O SNMP (*Simple Network Management Protocol*) é um protocolo de gestão de rede da camada de aplicação que facilita o intercâmbio de informação entre os dispositivos de rede. É parte do UDP (*User Datagram Protocol*). O SNMP (SNMP Research, 2006) possibilita aos administradores de rede gerir o desempenho da rede, encontrar e resolver problemas de rede, e planejar o crescimento desta.

Agentes são espalhados em uma rede baseada na pilha de protocolos TCP/IP. Os dados são obtidos através de requisições de um gerente a um ou mais agentes utilizando os serviços do protocolo de transporte UDP para enviar e receber suas mensagens através da rede. Dentre as variáveis que podem ser requisitadas utilizaremos as MIBs (*Management Information Base*), podendo fazer parte da MIB II, da experimental ou da privada.

A MIB é o conjunto dos objetos gerenciados, que procura abranger todas as informações necessárias para a gerência da rede. A MIB II, que é considerada uma evolução da MIB I, fornece informações gerais de gerenciamento sobre um determinado equipamento gerenciado. Através das MIB II podemos obter informações como: número de pacotes transmitidos, estado da interface, entre outras. A MIB experimental é aquela em que seus componentes (objetos) estão em fase de desenvolvimento e teste, em geral, eles fornecem características mais específicas sobre a tecnologia dos meios de transmissão e equipamentos empregados. A MIB privada é aquela em que seus componentes fornecem informações específicas dos equipamentos gerenciados, como configuração, colisões e também é possível reinicializar, desabilitar uma ou mais portas de um roteador.

O gerenciamento da rede através do SNMP permite o acompanhamento simples e fácil do estado, em tempo real, da rede, podendo ser utilizado para gerenciar diferentes tipos de sistemas. Este gerenciamento é conhecido como modelo de gerenciamento SNMP, ou simplesmente, gerenciamento SNMP. Por tanto, o SNMP é o nome do protocolo no qual as informações são trocadas entre a MIB e a aplicação de gerência como também é o nome deste modelo de gerência.

Os comandos são limitados e baseados no mecanismo de busca/alteração. No mecanismo de busca/alteração estão disponíveis as operações de alteração de um valor de um objeto, de obtenção dos valores de um objeto e suas variações.

A utilização de um número limitado de operações, baseadas em um mecanismo de busca/alteração, torna o protocolo de fácil implementação, simples, estável e flexível. Como conseqüência reduz o tráfego de mensagens de gerenciamento através da rede e permite a introdução de novas características.

O funcionamento do SNMP, representado pela figura 6 é baseado em dois dispositivos: o agente e o gerente. Cada máquina gerenciada é vista como um conjunto de variáveis que representam informações referentes ao seu estado atual, estas informações ficam disponíveis ao gerente através de consulta e podem ser alteradas por ele. Cada máquina gerenciada pelo SNMP deve possuir um agente e uma base de informações MIB.

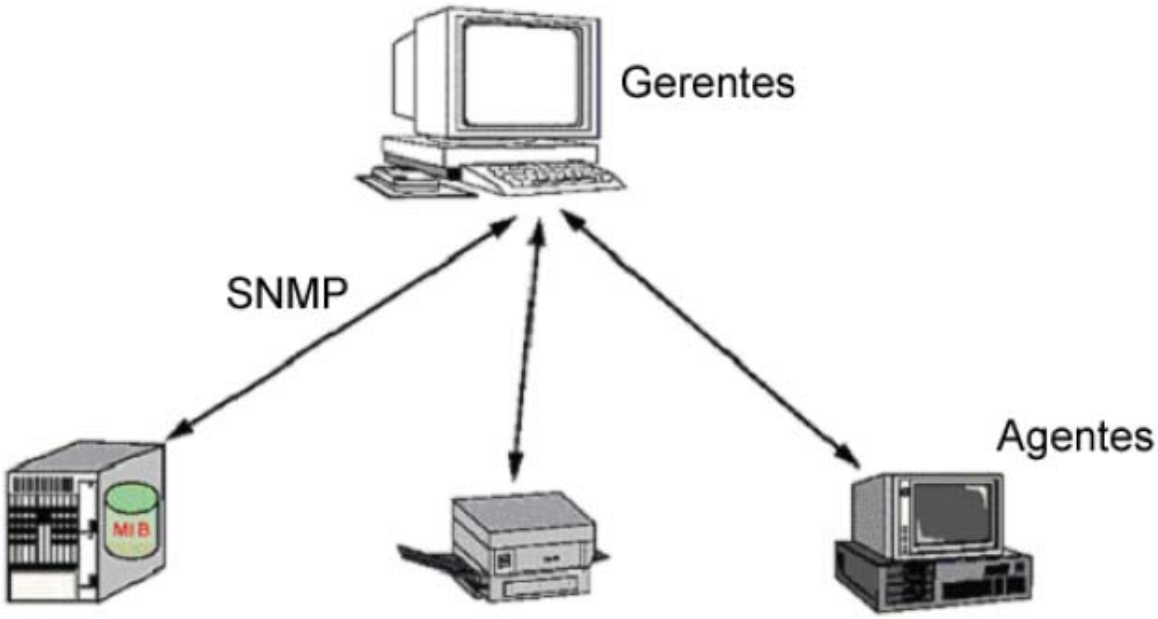

Figura 6 – Funcionamento do SNMP.

O Agente

É um processo executado na máquina gerenciada, responsável pela manutenção das informações de gerência da máquina. As funções principais de um agente são:

- Atender às requisições enviadas pelo gerente.
- Enviar automaticamente informações de gerenciamento ao gerente, quando previamente programado.

O agente utiliza as chamadas de sistema para realizar o monitoramento das informações da máquina e utiliza as RPC (*Remote Procedure Call*) para o controle das informações da máquina.

O Gerente

É um programa executado em uma estação servidora que permite a obtenção e o envio de informações de gerenciamento junto aos dispositivos gerenciados mediante a comunicação com um ou mais agentes. O relacionamento de um gerente com um objeto gerenciado está ilustrado na figura 7

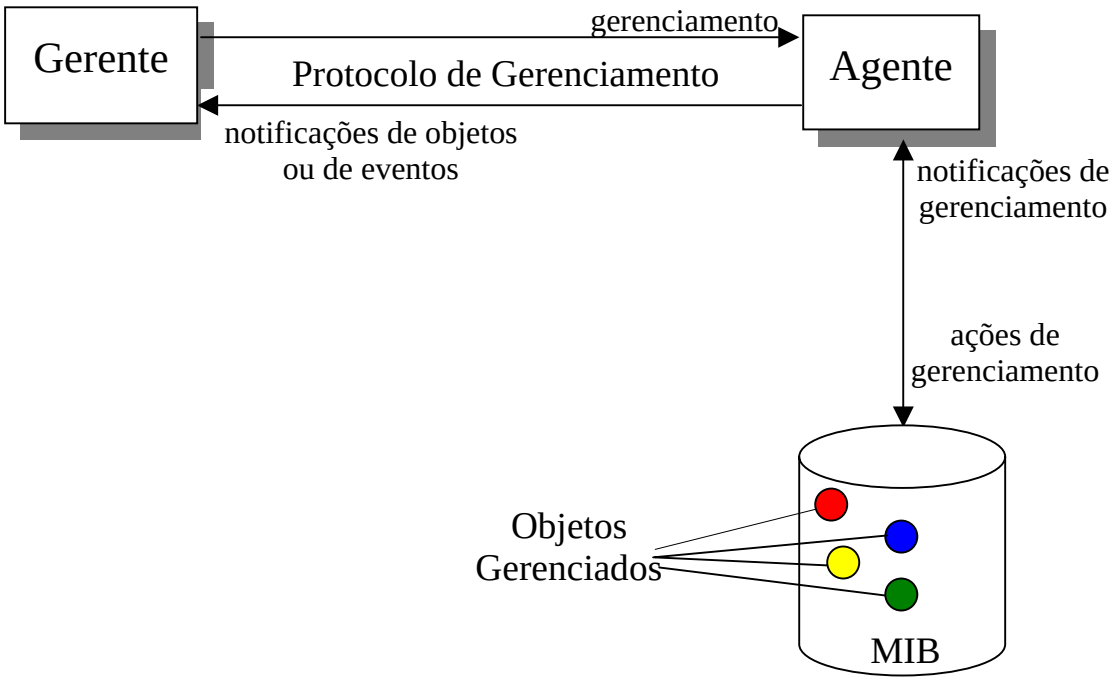

Figura 7 – Relacionamento de um gerente com um objeto gerenciado.

O gerente fica responsável pelo monitoramento, relatórios e decisões na ocorrência de problemas. O agente fica responsável pelas funções de envio e alteração das informações e também pela notificação da ocorrência de eventos específicos ao gerente. O relacionamento entre gerente e agente baseado no modelo TCP/IP está representado pela figura 8

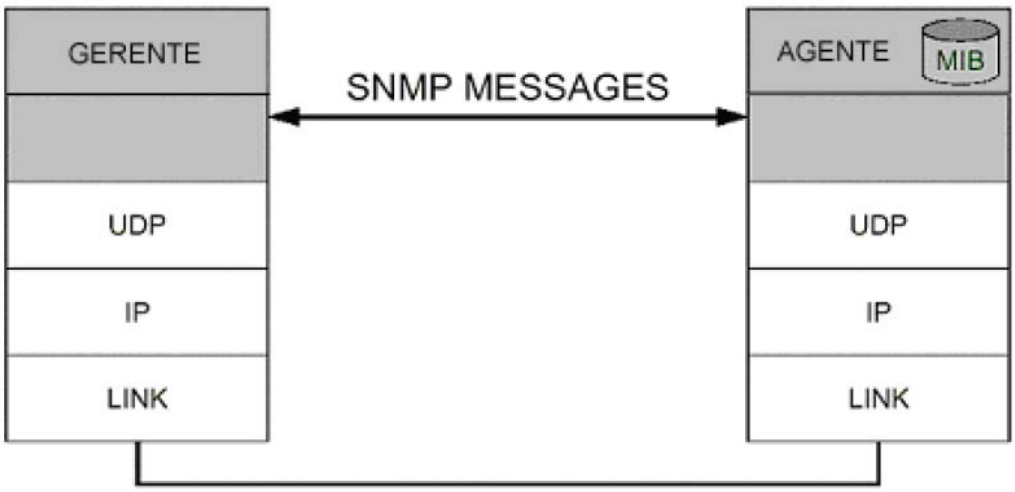

Figura 8 – Relacionamento entre gerente e agente baseado no modelo TCP/IP.

Pyro (Python Remote Objects)

O módulo PYRO (JONG, 2005) implementa a invocação remota de métodos e possui características muito semelhantes a tecnologia Java RMI (*Remote Method Invocation*). Este módulo possui importantes características, dentre as quais destacam-se:

- é totalmente escrito em Python.
- é um módulo pequeno, simples e extremamente portável (pode rodar sem problemas em qualquer máquina com Python (versão 2.1 ou superior) e uma rede TCP/IP disponível).
- tem suporte parcial para Jython (uma implementação da linguagem Python que gera bytecode para máquinas Java).
- o serviço de nomes guarda a localização dos objetos. A localização deste serviço tanto pode ser feita por um mecanismo de *broadcast* quanto por vários outros mecanismos, caso a rede não permita *broadcast*. Existe a possibilidade de persistência dos dados do serviço de nomes.
- os clientes e servidores podem mover objetos, mesmo quando o servidor não os conhece. O módulo PYRO irá automaticamente trasferir o *bytecode* necessário.
- aceita qualquer objeto serializável.
- as exceções ocorridas em um objeto remoto também são repassadas para o cliente.
- o servidor é *multithread*.
- a reconexão automática é realizada em caso de falha na rede.

Padrão de Projeto Proxy

O Padrão *Proxy* (GAMMA, 1995) permite que clientes de um serviço utilizem um representante do componente que oferece o serviço, aumentando a eficiência, a segurança e facilitando o acesso. O *Proxy* pode substituir o servidor no caso de ocorrer problemas com este servidor e permite criar uma independência de endereçamento e implementação do servidor. A figura 9 explica o funcionamento do Padrão de Projeto *Proxy*.

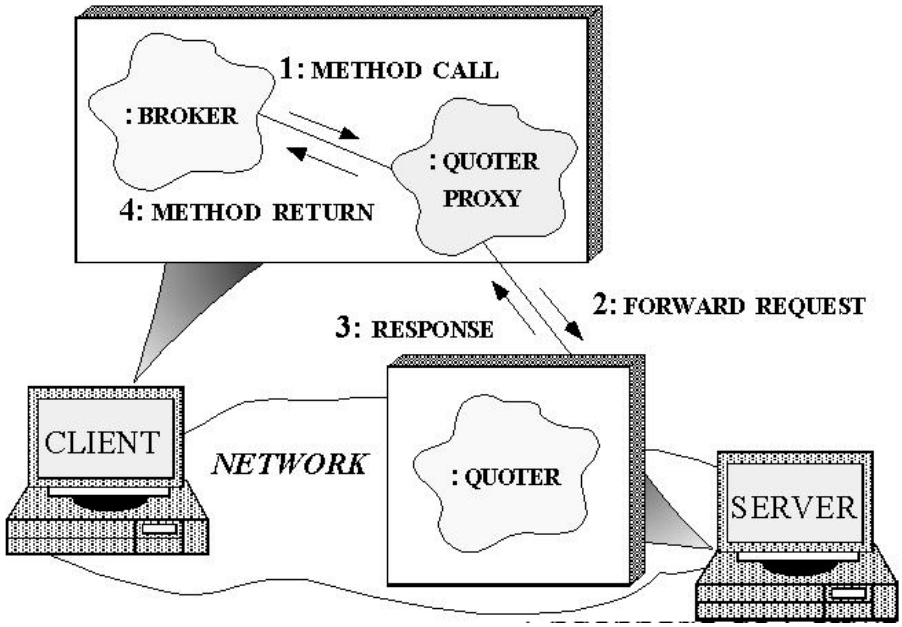

Figura 9 – Funcionamento do Padrão de Projeto Proxy.

Contexto: Um cliente precisa acessar um serviço de um outro componente em um sistema distribuído. O acesso direto é tecnicamente possível, mas pode não ser a melhor opção.

Problemas: O acesso direto pode não ser eficiente em tempo de execução, ter alto custo e não ser seguro. O cliente não precisa ficar dependente de endereço de rede do componente.

Solução: Utilize um representante do cliente que ofereça o serviço de forma idêntica e realize pré- e pós-processamento adicionais para garantir a Qualidade do Serviço.

Afigura 10 representa o diagrama de classes do Padrão de Projeto *Proxy*.

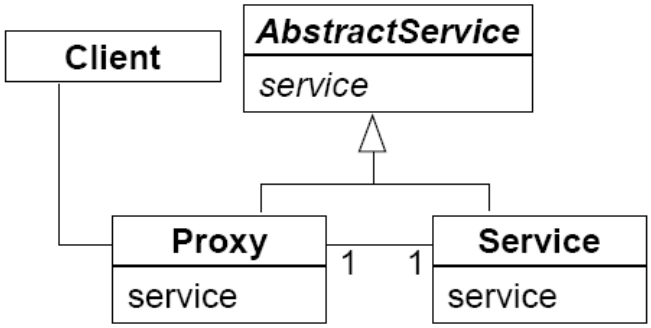

Figura 10- Padrão de Projeto Proxy.

Proxy Dinâmico (Dynamic Proxy)

Nos *proxies* dinâmicos, ao contrário dos *proxies* normais, as classes *proxy* são definidas em tempo de execução e todas as chamadas passam por um único método, denominado Invoke.

Da perspectiva do código que faz a invocação, a interface da classe *proxy* é exatamente igual ao da classe que encapsula. Assim, este tipo de padrão de *software* permite que em tempo de execução se dêem novas funcionalidades aos programas. Por exemplo, não faz muito sentido gastar tempo e energia a desenvolver e manter código de *debugging* e *logging* dentro do código principal da aplicação quando se podem centralizar estas funcionalidades em *proxies*. Utilizando *proxies* dinâmicos é possível fazer com que seja executado código antes e depois das chamadas aos métodos de qualquer classe. Isto possibilita que o código de *debug* ou *log* seja escrito dentro da classe que implementa o *proxy* dinâmico e executado sempre que ocorram chamadas aos métodos das classes encapsuladas. Os *proxies* dinâmicos são generalistas, visto que não são implementados para encapsular uma classe em particular, mas qualquer classe que se deseje, independentemente do seu interface.

O padrão de *software* tradicional de *proxies* obriga a desenvolver uma classe *proxy* por cada classe que se pretende encapsular e reescrever todos os métodos que essa classe implementa no *proxy*. Quando se trata de *proxies* dinâmicos, já não existe a necessidade de escrever um *proxy* por cada classe a encapsular. Como as classes *proxies* só são geradas em tempo de execução, o que o programador tem de escrever é uma classe genérica que deve implementar apenas um método. Este método será executado quando da interseção da chamada a qualquer método das classes encapsuladas. Em tempo de execução, o mecanismo de *proxies* dinâmicos encarrega-se de criar as classes *proxy* com uma interface semelhante à das classes encapsuladas e incluir o código do método previamente definido em todos os métodos do *proxy*.

4. Proposição de uma Prevalência Distribuída

Prevalência é um modelo de persistência transparente de objetos em memória. Em um sistema prevalente (que utiliza o modelo de prevalência para persistência de objetos), todos os objetos são mantidos na memória principal do computador. Uma classe específica é utilizada para garantir que todas as alterações efetuadas nesses objetos não sejam perdidas. Consultas são executadas recuperando objetos em memória, tornando seu acesso mais rápido do que através de consultas executadas diretamente em disco (VILELLA, 2002).

Na prevalência, duas restrições são feitas em relação aos tipos de dados que podem ser persistidos ou não: as classes devem ser serializáveis e determinísticas. A serialização é a transformação da estrutura do objeto em memória para uma seqüência de bytes, podendo a mesma ser armazenada e depois restaurada ao estado anterior do objeto. No caso da prevalência, ela é o processo que salva o estado do objeto, e deve ser aceita pela linguagem que implementará a prevalência.

Os objetos também devem ser determinísticos, ou seja, devem sempre chegar ao mesmo estado quando exposto ao mesmo conjunto de comandos. Essas duas restrições não implicam diretamente em um baixo nível de ortogonalidade (persistência independente da utilização do dado), já que dentre as classes de negócio que modelam o problema, e que realmente devem ser persistidas, as duas características podem ser atendidas.

Como o armazenamento principal dos objetos é feito em um meio físico volátil (memória principal do computador), surge a questão de como a prevalência garante a persistência dos objetos após várias execuções do sistema. Utilizando os mecanismos de *log* de transação e de imagem, a prevalência garante que, mesmo se o sistema for desligado ou parar de executar, nenhum objeto armazenado em memória será perdido. Isso acontece porque os dois mecanismos salvam o estado de um ou mais objetos em um meio físico não-volátil (disco), podendo restaurar o seu valor caso alguma falha ocorra.

O *log* de transação, que neste trabalho chamaremos de *audit*, é implementado pela utilização de classes que representam as alterações em objetos, serializando-as e salvando-as em um meio físico não-volátil. A forma de imagem utilizada é o "*snapshot*", responsável por salvar o estado de todos os objetos da memória principal para o disco, tirando uma espécie de "fotografia" do estado dos objetos naquele dado momento. A persistência do estado dos

objetos só é possível devido ao mecanismo de serialização, que deve ser suportado pela linguagem na qual foi implementada a prevalência.

Apesar de terem a mesma função (salvar o estado do objeto em disco), esses mecanismos são complementares devido às suas características e devem ser utilizados em conjunto para que a persistência seja garantida. Isso ocorre porque a geração da imagem (*snapshot*) pode se tornar lenta, caso o modelo de objetos ocupe muito espaço em memória. Sendo assim, utilizá-lo a cada vez que o modelo fosse alterado poderia ter grande impacto no desempenho do sistema.

No caso da restauração de uma falha ou reinício da aplicação, o sistema de prevalência procura pela última imagem gravada, restaurando o estado do sistema para o momento em que a gravação foi feita. A seguir, são re-executados todos os *audits* das alterações que foram efetuadas no sistema, posteriores à imagem tirada, para que o sistema volte ao estado anterior à falha ou ao desligamento do sistema.

Embora esses mecanismos utilizem o disco rígido, o armazenamento principal do modelo de prevalência ainda é a memória principal do computador, pois os objetos residem e são manipulados nela. O formato de gravação do objeto em disco é resultante da serialização de seu estado, não podendo ser acessado ou alterado enquanto estiver nesse formato. Enquanto em um SGBD o esquema de recuperação está relacionado à ocorrência de falhas, na prevalência, o *audit* de transação e a imagem funcionam tanto no caso de falhas, quanto no caso de reinício do sistema.

O paradigma prevalente implica no desenvolvimento do conceito de prevalência reticulada, que permite uma preservação robusta, escalável e adaptativa do domínio computacional. Essa prevalência provê a dispersão da camada de persistência, controlando a alocação e replicação da informação de modo distribuído. A incorporação dessa prevalência nos sistemas computacionais atuais resultará na redução significativa da impedância que limita o pleno aproveitamento da tecnologia. Isso permitirá o uso pervasivo de toda a capacidade computacional instalada, incluindo sistemas embutidos e plataformas móveis. Com a queda da impedância, se espera um aumento considerável no desempenho e flexibilidade dos sistemas computacionais. A baixa impedância também facilitará o desenvolvimento de sistemas autônomos e robustos, uma vez que a redução da complexidade das partes componentes beneficia os algoritmos de diagnose e replicação computacional.

O retículo é a camada virtual do sistema, onde o objeto se encontra em sua forma latente, ou seja, nenhuma transformação está operando nele. Essa região representa a coleção de todos os nós de uma subrede lógica que participam de uma dada aplicação distribuída. Por isso, sobre ela recaem os requisitos de escalabilidade do sistema. A escalabilidade e disponibilidade do retículo são providas por um mecanismo de migração e alocação de objetos distribuídos e autônomos baseado na oferta e demanda de cada nó.

O retículo seria dinamicamente alocado para respeitar as flutuações na capacidade de armazenamento de cada nó e otimizar a condição de latência de resposta. A gerência do retículo será plenamente distribuída e autônoma, onde cada nó é responsável pela política de aceitação, passivação e migração de objetos. O retículo também é responsável pela resiliência do sistema, promovendo a replicação de dados dentro do critério de responsividade e disponibilidade da informação. A implementação é baseada no roteamento da corrente de auditoria prevalente, através de espaços enúplarios (*tuple space* (FIGUEIREDO, 2001), *javaspace* (BISHOP *et al*, 2002)) federativos e distribuídos.

A operação do retículo provê mecanismos para o controle de ciclo de vida dos objetos, garantindo a replicação, restauração, passivação e ativação de objetos sob demanda. O modelo de prevalência exige que o conjunto de objetos ativos esteja todo em memória, porém com a imagem de memória replicada em um meio persistente. O modelo de replicação persistente nas implementações atuais usa o mecanismo de serialização do Java. Isto implica na geração de um grande arquivo monobloco contendo todo o conteúdo do conjunto ativo, que pode montar a dezenas de gigabytes. Para agilizar a geração e restauração da imagem de memória, o processo de transferência aplica um mapeamento Objeto-Relacional (Hibernate, 2006; MURTA *et al*, 2001; ARAÚJO, 2006). Uma vez que a imagem está armazenada em uma base de dados, o processo de restauração e replicação não precisa gravar todo o conjunto ativo, podendo se resumir aos objetos referenciados pelas ações prevalentes. As ações prevalentes são a base do mecanismo de persistência do *Prevayler*, que marcam os objetos envolvidos em cada transação. Esta marcação permite assinalar também a região da imagem que deve ser restaurada em caso de falha na transação.

4.1. Arquitetura

Atualmente, os sistemas distribuídos têm sido amplamente utilizados nos sistemas de computação em geral. Este fato deve-se, principalmente, à grande difusão do uso de redes de computadores. O objetivo é prover um aumento considerável no poder de processamento.

O armazenamento dinâmico de objetos entre os nós de processamento é estabelecido a partir da análise de recursos de cada nó (memória, processamento, disponibilidade na rede). Essa solução permite desacoplamento de objetos, que torna mais fácil sua distribuição dinâmica. O posicionamento dinâmico de objetos permite processamento mais uniforme e de maneira mais eficiente. A distribuição alcançada é também dinâmica e determinada pelo comportamento do sistema.

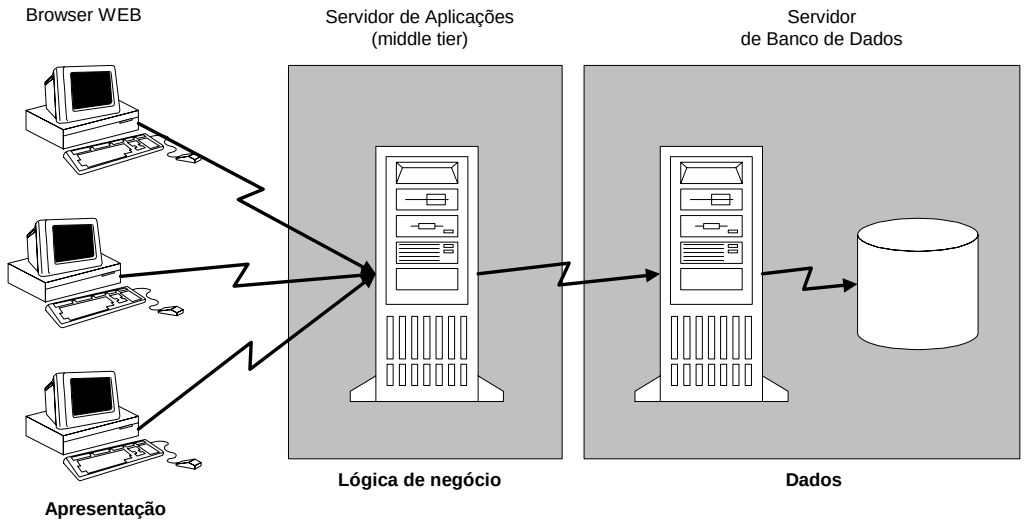

Figura 11 - Arquitetura 3-camadas: Apresentação – Lógica de Negócio – Dados

Na abordagem utilizada neste trabalho de dissertação, os sistemas OOD passariam de um modelo cliente-servidor ou 3-camadas (Figura 11), para um modelo verdadeiramente distribuído ponto-a-ponto (Figura 12). Neste caso, a distribuição dos objetos é feita de maneira dinâmica, segundo um conjunto de estratégias baseadas na observação do comportamento dos nós que participam do processamento e que levam o sistema a ser processado de maneira uniforme.

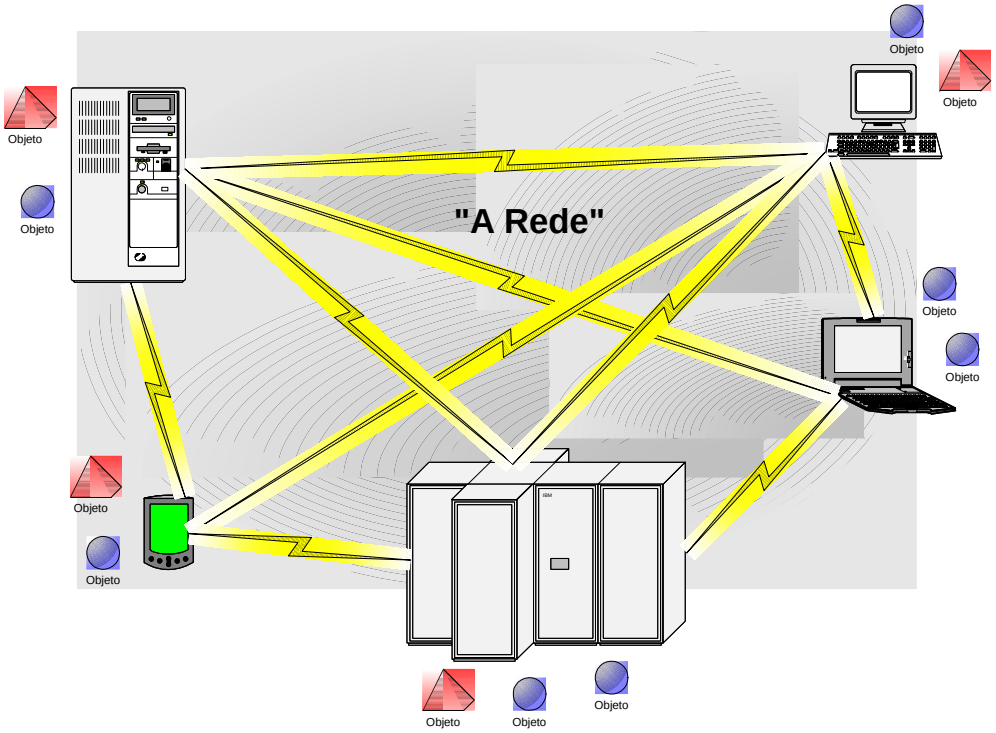

Figura 12 - Objetos distribuídos ponto-a-ponto: distribuição uniforme.

O propósito do projeto Pantheon é possibilitar o desenvolvimento de sistemas de informação de larga escala usando objetos distribuídos. Existem situações cotidianas que tornam necessário a distribuição de dados em diferentes locais geograficamente separados. Os objetos devem permanecer em memória e devem poder ser usados de qualquer parte da aplicação em qualquer momento. Para isso é necessário garantir a persistência e a disponibilidade dos objetos, além da escalabilidade do sistema.

A persistência foi realizada através da integração com o arcabouço de prevalência *Prevayler*. O *Prevayler* usa uma forma diferenciada de armazenamento de dados baseado em objetos chamado prevalência, derivado do fato que os objetos têm que permanecer em memória. Desta forma, ele oferece compatibilidade total à orientação a objetos, sem a necessidade de camadas de mapeamento para o modelo relacional.

Entretanto, a implementação do *Prevayler* não atende os requisitos de um sistema coorporativo: disponibilidade 24x7 e uma base de dados muito volumosa, pois existe a limitação da quantidade de memória utilizada pela máquina virtual. Esses problemas podem ser resolvidos com uma arquitetura distribuída, inexistente no *Prevayler*.

A arquitetura propõe uma redefinição do *Prevayler*, incluindo a possibilidade de armazenar objetos em máquinas dispostas em locais diferentes. Desta maneira, os problemas citados no parágrafo anterior são sanados. Assim, os objetos podem ser armazenados nas diversas máquinas que compõem a rede.

O *Prevayler* realiza periodicamente uma cópia dos dados, chamada *snapshot*, que consiste na serialização dos objetos em um arquivo (*audit*). Isso quer dizer que toda manipulação de objetos é registrada nesse *audit* de comandos, o qual é escrito imediatamente no disco. Dessa forma, quando o sistema for reiniciado, os objetos serão recompostos a partir da imagem (*snapshot*) e do arquivo de log (*audit*).

Entretanto, o *Prevayler*, em sua proposta original, mantém todas as instâncias persistentes na memória de uma única máquina. Isso gera uma superutilização da memória, o que acarreta em um desperdício, pois o limite de capacidade de expansão de memórias RAM é limitado pela placa-mãe do computador. A figura 13 ilustra o modelo de persistência adotado pela implementação original do *Prevayler* – Sistema Prevalente sem distribuição.

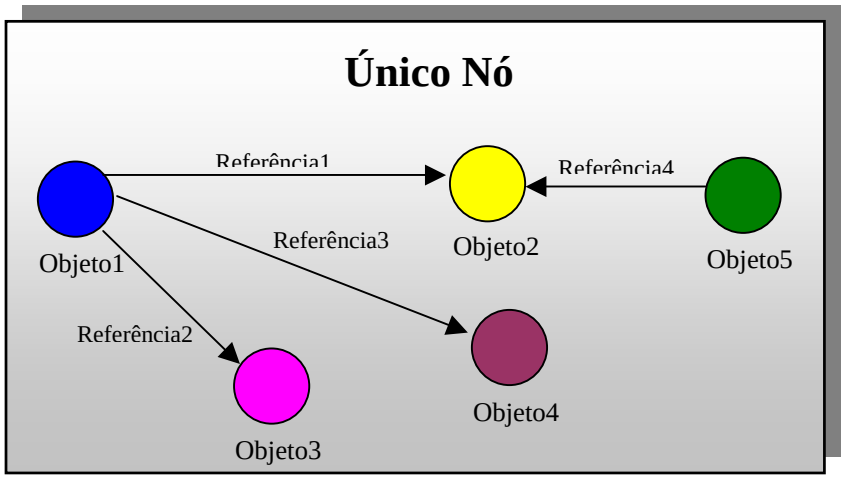

Figura 13- Sistema Prevalente sem distribuição.

Com o propósito de otimizar a utilização de memória, foi introduzido na arquitetura o conceito de distribuição. Distribuição no sentido de que os objetos agora não ficarão armazenados apenas na memória de uma única máquina. Estratégias de armazenamento dos objetos entre os nós que fazem parte da rede, como veremos mais adiante, foram adotadas afim de que este problema de limitação de memória seja resolvido.

A figura 14 esclarece o tipo de sistema proposto no Pantheon - Sistemas Prevalentes Distribuídos sob demanda.

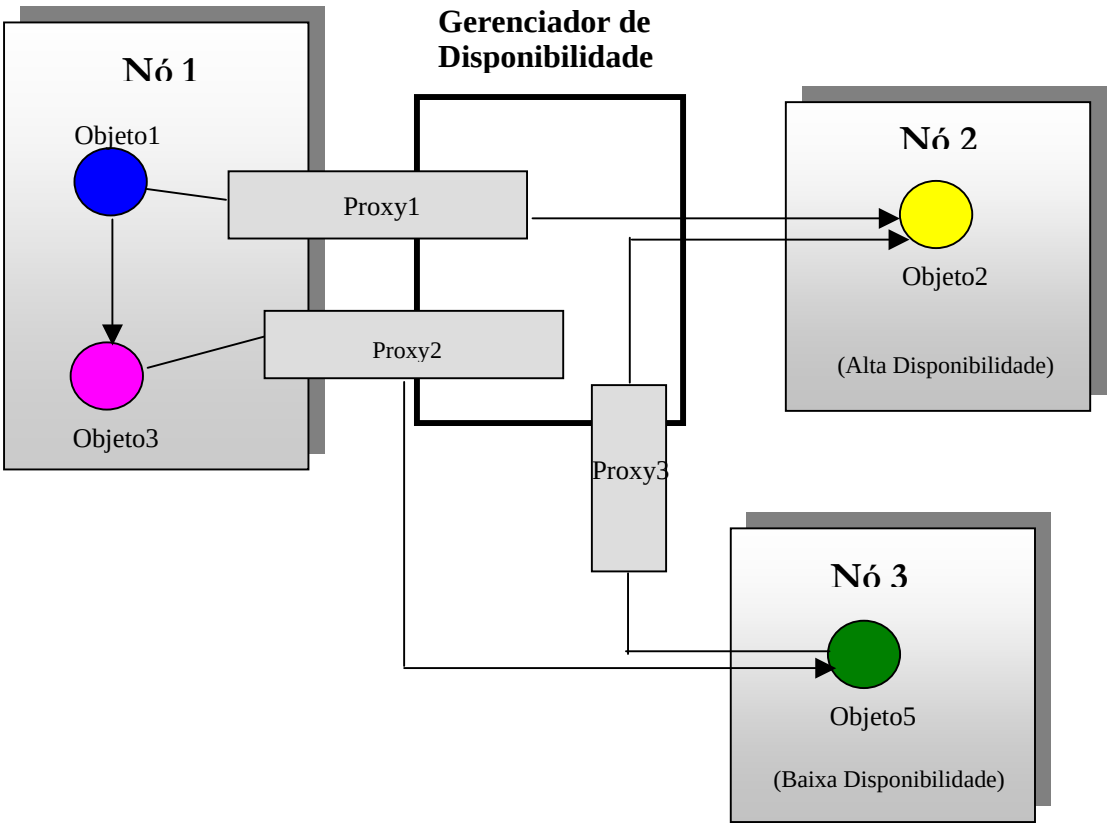

Figura 14 - Sistema Prevalente Distribuído sob demanda.

No modelo prevalente descrito na figura 14, a persistência se dá em diversos nós. Sendo assim, a escalabilidade do sistema deve ser garantida, isto é, o sistema continua eficaz quando há um aumento significativo do número de recursos e utilizadores (nós), não sendo necessário alterar a implementação dos componentes e da forma de interação das mesmas. Para que essa escalabilidade seja possível, os recursos envolvidos devem aumentar linearmente com a dimensão dos sistemas e o aumento na dimensão dos recursos não deve levar a uma diminuição significativa do desempenho.

Para garantir a integridade das referências dos objetos e que os objetos estejam sempre disponíveis para serem utilizados, existe o **gerenciador de disponibilidade**, presente na figura 14. Por exemplo, o nó 2 é de alta disponibilidade e por isso é um forte candidato para que os objetos mais referenciados migrem para este nó. Entretanto, outras características são levadas em consideração ao se armazenar um objeto em um nó. Tais características são: quantidade de memória disponível e capacidade de processamento.

Os módulos que compõem gerenciador de disponibilidade estão ilustrados na Figura 15.

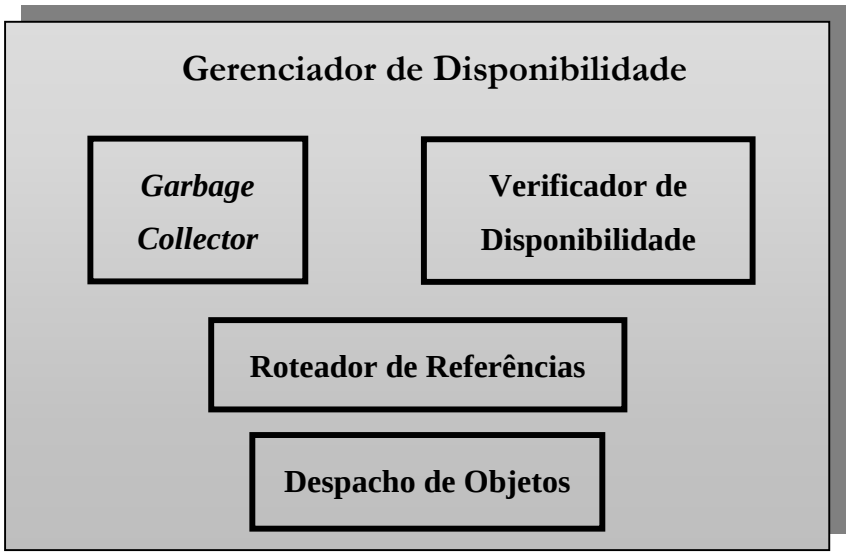

Figura 15 - Gerenciador de Disponibilidade.

O gerenciamento de memória, em termos de aplicações, envolve a quantidade de memória necessária para estruturas de dados, armazenamento de objetos e também a liberação de recursos/memória para reuso em outras transações. O gerenciamento de memória está relacionado a duas tarefas:

- alocação: quando é requisitado um bloco de memória, o gerenciador disponibiliza esse bloco para a alocação;
- reciclagem: quando um bloco de memória foi alocado, mas os objetos não são requisitados já há algum tempo e esse bloco pode ser reusado para outra requisição.

O gerenciamento automático de memória, conhecido como ***Garbage Collector*** ou simplesmente coletores, é um serviço que automaticamente libera blocos de memória que não serão mais usados em uma aplicação. No Pantheon, ele observa o decaimento do *audit* para retirar os objetos da memória. Com este arquivo (*audit*), responsável por guardar as transações (alterações) realizadas sobre os objetos, basta que seja observado quais os objetos que não possuem mais referência, isto é, quais objetos não foram requisitados por algum tempo e a partir daí retirá-los da memória.

O **verificador de disponibilidade** cria uma estatística das máquinas mais e menos disponíveis na rede e o **despacho de objetos** move os objetos mais utilizados (observando o

audit) para nós mais disponíveis, verificando as condições de migração (disponibilidade dos nós). O objetivo é garantir uma alta disponibilidade desses objetos de forma que estejam sempre acessíveis. Com essa migração, os *proxies* precisam ser redirecionados para esses objetos movidos, o que é tarefa do **roteador de referências**. A localização das instâncias dos objetos é feita de forma transparente, pois esses objetos podem estar na memória de outra máquina.

Em uma aplicação distribuída orientada a objetos, múltiplos objetos estão cooperando para executar uma tarefa. Se um nó falhar, os objetos armazenados neste nó deixam de responder. Isso pode fazer com que toda a computação distribuída venha a falhar. Depender de um nó da rede que saiu do ar por questões de falta de energia elétrica, manutenção, falha do hardware ou da rede é algo muito inconveniente, ainda mais se ocorrer perda ou corrupção dos dados. Assim, se um nó cair, ficar fora do ar ou da rede, o sistema não pode perder informações e nem ficar inacessível total ou parcialmente.

Além disso, o usuário não precisa saber como isso foi implementado e nem como funciona. Isso deve ser totalmente transparente para ele. Ele simplesmente requisita um objeto, e o sistema deve entregá-lo, mesmo que algum dos nós falhe ou caia. Para isso é necessário implementar transparência a falhas.

Para resolver esse tipo de problema, a arquitetura do Pantheon foi desenvolvida baseando-se na solução onde os objetos são replicados. A replicação é uma estratégia utilizada para distribuição de dados compartilhados, permitindo que várias cópias da entidade computacional residam em diferentes memórias locais, ou seja, o mesmo objeto está em diferentes máquinas que participam da rede. Dessa forma, conseguimos resolver dois problemas consideráveis: distribuição da carga causada pelo acesso aos objetos nos vários nós que compõem o sistema; e mesmo que algum nó falhe e fique fora da rede, outros nós poderão fornecer os mesmos objetos.

Duas formas de replicação foram estudadas: replicar todos os objetos em mais de uma máquina, ou granularizar um pouco ao replicar somente os objetos mais acessados. A primeira opção gera um desperdício grande de memória, já que todos os objetos estariam replicados em diversas máquinas, mesmo aqueles pouco referenciados. No contexto dessa dissertação, a segunda opção foi escolhida. Logo, somente os objetos que forem bastante requisitados sofrerão replicação. Após ser feita a replicação, essa cópia deverá ser armazenada em um outro nó que compõe a rede.

O uso de sistemas que permitem mobilidade de objetos está aumentando com o uso da Internet, envolvendo assim redes heterogêneas, conectadas por diferentes *links* de comunicação. Essa mobilidade tem como principal vantagem manter a localidade dos objetos que trocam muitas mensagens, diminuindo assim o tráfego na rede. Com a replicação, várias cópias de uma mesma entidade computacional residem em diferentes máquinas do sistema, podendo haver acesso simultâneo de diferentes nós à mesma entidade computacional. Porém, um problema fundamental é a necessidade de manter a consistência da informação, isto é, se um objeto sofre alteração, todas as máquinas que possuem cópia deste objeto devem ser notificadas e as réplicas atualizadas.

A replicação de um objeto somente é útil se este é freqüentemente consultado por um nó remoto. Em sistemas distribuídos, a replicação é usada para oferecer segurança e persistência ou para aumentar o desempenho do processamento das informações. Sendo assim, a confiança e a eficiência dos serviços é generosamente aumentada. Eficiência em termos de tempo de resposta, carga dos nós e tráfego de rede. Confiança caso um determinado nó caia ou fique indisponível, os objetos ainda podem ser acessados por possuírem cópias em outro ponto da rede.

Deste modo, essa solução provê tolerância a falhas, já que o usuário nem sequer percebe que o nó que ele estava usando caiu e que outro entrou no lugar para prover o recurso que ele estava usando. Logo, o sistema também deve oferecer transparência de replicação, pois o usuário não precisa saber como o sistema cuida da replicação desses objetos.

A solução distribuída utiliza comunicação em grupo, que consiste em quando ocorrer uma alteração por algum dos nós, este manda um *broadcast* para os outros nós dizendo que o objeto foi alterado. Estes, por sua vez, podem alterar esse objeto imediatamente ou somente quando forem utilizá-lo.

A replicação foi usada nesta proposta como uma técnica viável para garantir alta disponibilidade de objetos na presença de falhas em ambientes distribuídos. O mecanismo de comandos para a persistência pode ser extensivo à replicação dos objetos pela rede distribuída do Pantheon.

Um outro problema existente é a limitação de memória dos nós. Quando uma máquina necessita realizar algum tipo de processamento e não possui a quantidade de memória necessária, os objetos menos referenciados que residem neste nó são enviados

(migrados) para uma máquina que atenda os pré-requisitos exigidos por esses objetos (disponibilidade, quantidade de memória e processamento requisitados). A migração de objetos é uma característica que proporciona a obtenção de uma melhor eficiência em sistemas distribuídos.

No âmbito deste trabalho de dissertação foram estudadas algumas heurísticas a fim de realizar essa migração de objetos entre os nós que participam da rede. Foi criada uma heurística que simula uma roleta, onde só é levada em consideração a quantidade de memória exigida pelo objeto a ser armazenado. Todas as máquinas que atenderem essa exigência, isto é, tiverem uma quantidade de memória disponível maior ou igual à estabelecida pelo objeto, participarão desta roleta. Apenas uma delas é sorteada e o objeto é migrado para a máquina vencedora.

Tanto a migração quanto a replicação de objetos entre os nós de processamento são feitas dinamicamente e são estabelecidas a partir da análise de recursos de cada nó. O resultado disso é o desacoplamento dos objetos, ou seja, a referência ao objeto é substituída por uma identificação do mesmo que é utilizada para invocá-lo. Um ganho direto do desacoplamento entre objetos é que o objeto não necessita estar em um mesmo processo para ser invocado, bastando haver uma relação entre sua identificação e o local onde se encontra. A conseqüência imediata disso é que mesmo migrando os objetos entre máquinas, o objeto continua passível de ser invocado.

Esta redundância provê uma solução de alta disponibilidade a um custo baixo. Utilizando-se métricas do custo de processamento, tais como disponibilidade do nó na rede, quantidade de memória disponível, capacidade de processamento, pode-se tomar a decisão de migrar ou replicar o objeto para locais onde os recursos são menos escassos ou há menos concorrência por eles.

Para que os engenhos de migração e replicação fossem possíveis, foi necessário propor um mecanismo de sincronização desses objetos. A sincronização é necessária porque, como os objetos são dinâmicos, existe a possibilidade de alteração por outros nós. Logo, quando um nó, contendo um conjunto de referências a objetos, que estava desconectado, conectar-se ou quando uma referência a um objeto que está persistido em mais de um lugar for modificada, terá que ocorrer uma sincronização das modificações de forma a garantir a consistência da mesma.

Este mecanismo é o mesmo utilizado pelo *Prevayler*, ou seja, cada modificação realizada em um objeto é encapsulada na forma de um comando. Este comando é enviado pela rede aos pontos que possuem cópia do objeto. O gerenciador de modificações do Pantheon recebe o comando e se encarrega de reproduzi-lo no seu nó local, para que o mesmo seja sincronizado com as demais cópias. Exatamente da mesma forma como o *Prevayler* realiza a carga dos dados de um sistema ao inicializá-lo.

No modelo de prevalência distribuída sob demanda, as referências cruzam as fronteiras via *proxies*. O *proxy* tem a função de substituir o nó no caso de ocorrer problemas com o mesmo. Desta forma, os objetos estarão sempre disponíveis, mesmo em caso de falha.

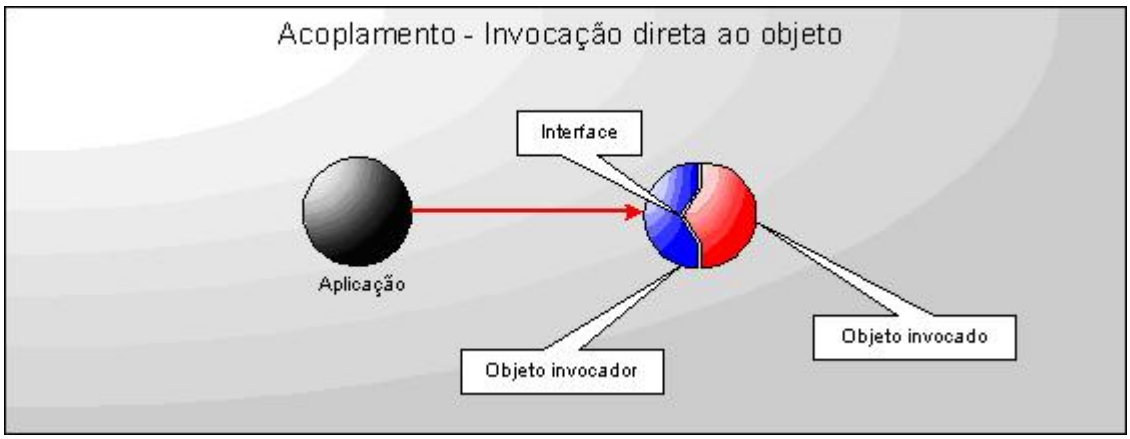

Figura 16 - O objeto que invoca o método acoplado ao objeto invocado.

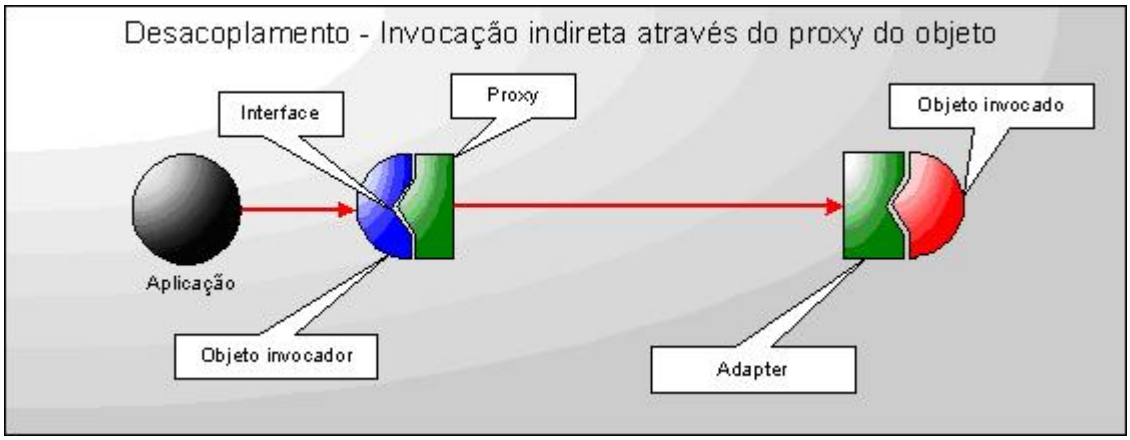

Figura 17 - Desacoplamento com o proxy: implementa a mesma interface do objeto.

O *proxy* promove o desacoplamento entre o objeto que invoca o método e o objeto que tem seu método invocado. Numa aplicação orientada a objetos, que não utiliza a

distribuição de objetos, a invocação de um método é feita diretamente ao objeto (Figura 16). O objeto possui a referência do objeto invocado e faz o acesso ao método diretamente. Isso faz com que haja um grande acoplamento entre os objetos, não permitindo que eles se "distanciem", o que significa que não é possível movê-los para nós diferentes e quebrar o acoplamento entre eles. O *proxy* soluciona esse problema, uma vez que ele representa um objeto que pode estar instanciado localmente ou remotamente (Figura 17).

Uma vez que um objeto possui referência a outro, ele pode invocar diretamente seus métodos sem nenhuma interferência intermediária, conforme mostrado na Figura 16, entre o objeto invocador e o invocado. O resultado disso é que não é possível, por exemplo, que o objeto seja destruído ou movido para outro nó, por algum outro mecanismo, e fazer com que suas referências sejam atualizadas. Com isso, o padrão de projeto *proxy* aparece como uma solução definitiva para a criação de componentes mais desacoplados. Um *proxy* deve possuir a mesma interface do objeto a ser invocado e é mantido localmente como referência pelo objeto que o invoca. Uma vez invocado o método, o *proxy* se encarrega de repassar a invocação ao método apropriado do objeto a que se destina a invocação, retornando o resultado, como visto na Figura 17.

O *proxy* possibilita o desacoplamento total entre o objeto que invoca e o que é invocado, até mesmo permitindo a invocação de objetos que ainda nem estão instanciados ou referenciados, técnica conhecida como *late binding*. O *proxy* pode implementar o protocolo que for necessário para conseguir se comunicar com o objeto, no caso do *Python*, o *Pyro*, que utiliza o TCP/IP como protocolo físico de rede.

Uma vez que o próprio *proxy* filtra através dele todas as mensagens, ele tem que auditar todas essas mensagens. Logo, o *audit* é o combustível de todos os algoritmos que foram utilizados para otimizar a idéia do Pantheon (otimizar a robustez, a elasticidade do sistema). Ou seja, baseado nesse *audit* é possível observar quais são as informações dos objetos que têm o maior valor econômico. É o que chamamos de uma economia de mercado, quanto mais aquela informação é “comprada”, maior o seu valor econômico. Esse valor econômico é inferido através desse “volume de venda”.

A contabilização do *audit* é responsável por estimar esse valor econômico, de acordo com o seu “volume de venda”, ou seja, existem muitas mensagens chegando naquele objeto. Isso significa que ele é muito útil para uma determinada computação. Logo, o *audit* serve para separar que objetos têm valor econômico alto dos objetos que têm valor econômico baixo.

Tudo isso é feito sobre uma plataforma. Uma vez que esses diversos valores econômicos estão mapeados no modelo do Pantheon, é possível criar diversas estratégias de mercado, de forma a chegar em um sistema otimizado.

4.2. Implementação

A computação distribuída consiste em adicionar o poder computacional de diversos computadores interligados por uma rede de computadores ou mais de um processador trabalhando em conjunto no mesmo computador, para processar colaborativamente determinada tarefa de forma coerente e transparente, ou seja, como se apenas um único e centralizado computador estivesse executando a tarefa. A união desses diversos computadores com o objetivo de compartilhar a execução de tarefas é conhecida como sistema distribuído.

Os bancos de dados distribuídos (BDD's) são muito utilizados dentro da área de sistemas distribuídos. Eles consistem de uma coleção de vários bancos de dados logicamente inter-relacionados, distribuídos por uma rede de computadores. Neles os arquivos podem estar replicados ou fragmentados e esses dois tipos podem ser encontrados ao longo dos nós do sistema de BDD's. Quando os dados se encontram replicados (BURETTA, 1997), existe uma cópia deles em cada nó, tornando as bases iguais (ex: tabela de produtos de uma grande loja). Já na fragmentação (OZSU, 1999), os dados se encontram divididos ao longo do sistema, ou seja, em cada nó existe uma base de dados diferente.

O trabalho de dissertação aqui proposto também utiliza os conceitos de replicação e fragmentação (que aqui chamaremos de migração). Entretanto, no Pantheon os dados não residem no disco, mas na memória. Para isso, foi utilizado o arcabouço chamado *Prevayler*. Entretanto, como dito no capítulo de arquitetura, a limitação da quantidade de memória utilizada pela máquina virtual e o fato de não atender aos requisitos de um sistema coorporativo constituem problemas na implementação do *Prevayler*. A solução adotada no Pantheon foi a reimplementação desse sistema, adotando uma arquitetura distribuída, tornando possível a persistência de objetos na memória em máquinas residentes em ambientes distintos.

O trabalho do Pantheon é selecionar máquinas, que irão garantir uma persistência distribuída robusta, ou seja, mesmo que algumas máquinas caiam, a aplicação continuará no ar. Não se levou em consideração a performance, mas sim a robustez, usando a rede como um computador virtual. Não é exigido que exista um servidor, mas sim que, se o dado existe, ele

deve estar disponível em algum lugar. Estratégias, que serão vistas mais adiante, foram criadas para garantir essa disponibilidade.

Para que fosse possível compreender quais os requisitos necessários para o desenvolvimento da plataforma proposta nesse trabalho, foi utilizado como estudo de caso o Crivo de Eratóstenes (WEISSTEIN, 1999), por possuir informações determinísticas, isto é, valores coletados do sistema como dados de entrada, diferente dos sistemas de informação, que lidam com informações não determinísticas.

O Crivo de Eratóstenes consiste em encontrar todos os números primos numa lista de números inteiros pequenos. Tendo em conta que a multiplicação é uma operação mais fácil de realizar do que a divisão, Eratóstenes de Cirene teve a brilhante idéia de organizar estas computações em forma de crivo ou peneira.

Para encontrar todos os números primos pequenos, por exemplo, os menores que 20, basta fazer uma lista com todos os inteiros maiores que um e menores ou igual a *n* e riscar os múltiplos de todos os primos menores ou iguais à raiz quadrada de *n* ($n^{½}$). Os números que não estiverem riscados são os números primos.

Vamos determinar, por exemplo, os primos menores ou iguais a 20:

1. Inicialmente faz-se a lista dos inteiros de 2 a 20.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

2. O primeiro número (2) é primo. Vamos mantê-lo e riscar todos os seus múltiplos. Desta forma, obtemos:

	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~

3. O próximo número "livre" é o 3, outro primo. Vamos mantê-lo e riscar seus múltiplos:

	2	***3***	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~

4. O próximo número primo é 5, porém não é necessário repetir o procedimento porque 5 é maior que a raiz quadrada de 20 ($20^{½}$ = 4,4721). Os números restantes são primos, destacados abaixo:

	2	***3***	~~4~~	***5***	~~6~~	***7***	~~8~~	~~9~~	~~10~~
11	~~12~~	***13***	~~14~~	~~15~~	~~16~~	***17***	~~18~~	***19***	~~20~~

Os números primos encontrados foram {2, 3, 5, 7, 11, 13, 17, 19}.

Com o Crivo, foi possível levantar todos os requisitos necessários para atender o modelo não determinístico. Se um usuário deseja saber quais são os números primos, então é oferecido um conjunto os números inteiros e o Pantheon fornece todos os números primos dentro desse conjunto. Pode existir mais de um usuário desejando obter os números primos. Logo, os números gerados para o primeiro cliente são aproveitados para todos os clientes seguintes.

Toda vez que um objeto é instanciado, o Pantheon cria um *proxy* deste objeto. Logo, cada objeto possui seu *proxy*. Os clientes pantheon se comunicam com esse *proxy* e este se comunica com os outros objetos, que podem estar na mesma máquina ou não. Sendo o *Python* a linguagem escolhida para o desenvolvimento deste trabalho, este *proxy* lança mão de artifícios que a linguagem possui, tais como:

- o método getattr, que permite fazer um *rebind* dos métodos do objeto a tempo de execução. O protocolo getattr dita como atributos são transparentemente localizados em uma hierarquia de classes e suas instâncias, e segue a seguinte receita:

1. Ao acessar um atributo de uma instância (por meio de uma variável qualquer ou *self*) o interpretador tenta localizar o atributo no estado da instância.

2. Caso não seja localizado, busca-se o atributo na classe da instância em questão. Por sinal, este passo é o que permite que métodos de uma classe sejam acessíveis a partir de suas instâncias.

3. Caso não seja localizado, busca-se o atributo entre as classes base definidas para a classe da instância.

4. Ao atribuir uma variável em uma instância, este atributo é sempre definido no estado local da instância.

o metaclasses, que já oferece toda a idéia de *proxy*. Possuem as informações relacionadas às classes. São usadas como *templates* para construção de classes. Assim como uma classe é usada como "forma" para construir objetos, metaclasses são a "forma" para construção de classes. Com o conceito de metaclasse, as classes podem ser manipuladas como objetos. Podem-se criar classes dinamicamente, modificar atributos e métodos.

Após o objeto ser criado, ele é inserido no Teseus. O Teseus (SILVA) é um espaço de tuplas baseado no *JavaSpaces* (FREEMAN *et al*, 1999). No escopo deste trabalho de dissertação, foi implementada uma versão simplificada do Teseus, com o objetivo de atender às necessidades do Pantheon. Algumas diferenças foram implementadas para permitir que o mesmo seja completamente distribuído, possa ter uma maior flexibilidade nas pesquisas por objetos e coleções deles no espaço e implementar um modelo de sincronização que possa ser utilizado para diminuir o tráfico na rede e permitir interconexões com baixo acoplamento.

O objeto, para ser inserido no Teseus, é encapsulado em um token, que além de possuir o objeto, possui também informações sobre esse objeto (identificador e tamanho), conforme é mostrado na figura 18.

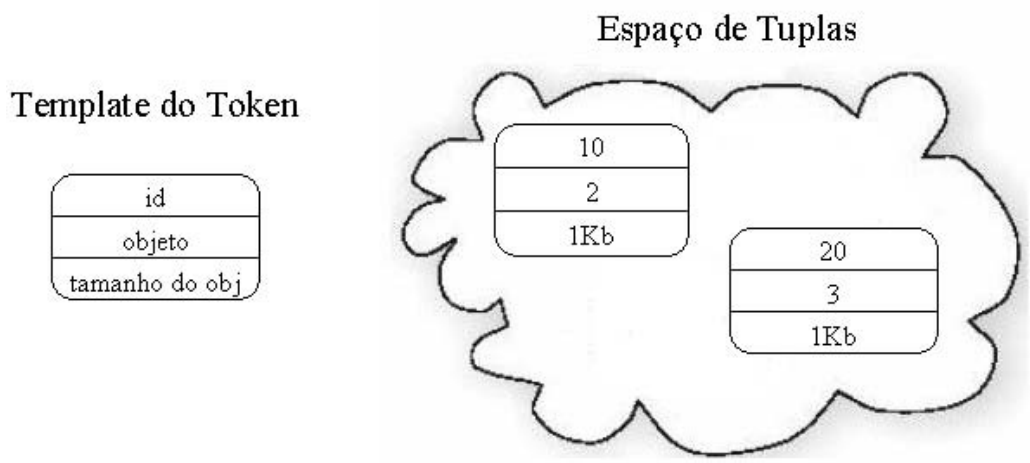

Figura 18 –Tokens no espaço de tuplas. Exemplo usando o estudo de caso: números primos.

Quando um nó (*peer*) deseja obter um recurso do espaço de tuplas, ele faz uma chamada ao método de leitura do espaço para solicitar um recurso e informa ao espaço seu dado sobre quantidade de memória disponível. Ao chegar ao espaço, o mesmo seleciona os objetos que possuem uma quantidade de memória menor ou igual ao disponível no nó, fazendo assim uma pré-seleção dos objetos que participarão da roleta. Nessa roleta é feita uma escolha aleatória de qual objeto será armazenado naquele nó.

Os gráficos 1 e 2 mostram duas heurísticas (estratégias) de roleta. A primeira não leva em consideração a quantidade de memória existente na máquina. Logo, essa heurística não testa se as máquinas possuem memória suficiente para armazenar o objeto. Simplesmente é sorteada uma máquina para onde o objeto migrará.

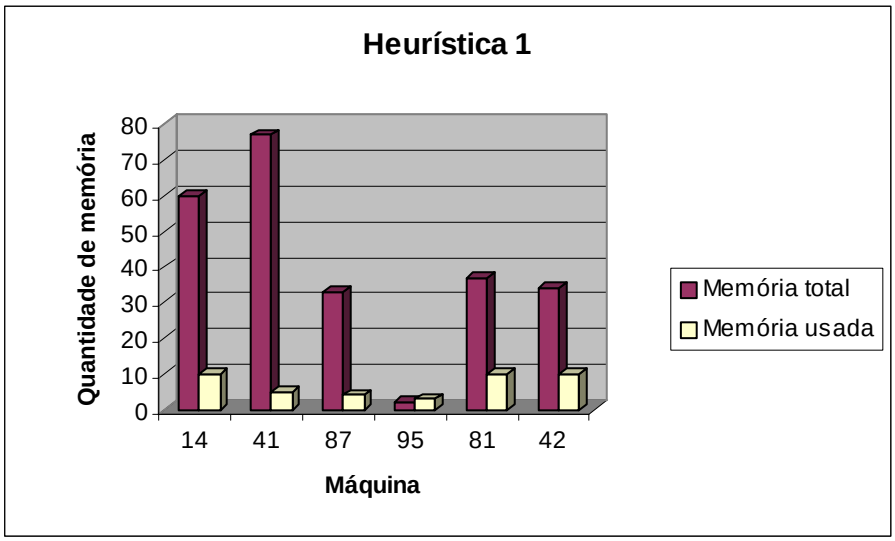

Gráfico 1 – Heurística que não leva em consideração a quantidade de memória da máquina.

Note que a máquina 95 possui uma memória total inferior a usada. Isso quer dizer que o objeto armazenado nela é maior que a sua capacidade de memória.

A segunda heurística, adotada neste trabalho, leva em consideração a quantidade de memória da máquina, conforme foi explicado anteriormente.

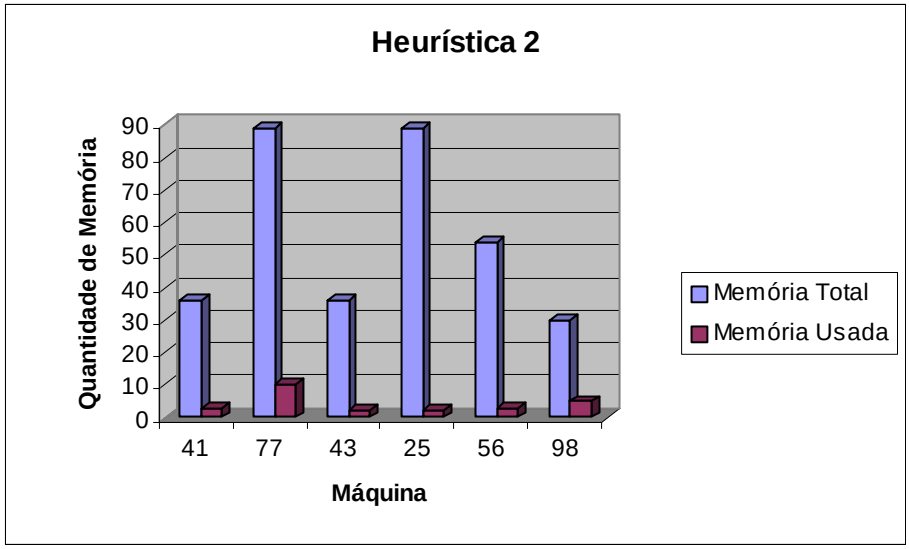

Gráfico 2 – Heurística que leva em consideração a quantidade de memória da máquina.

Com os objetos no espaço de tuplas, todas as máquinas são investigadas afim de que sejam verificados os recursos que cada uma possui (quantidade de memória, disponibilidade na rede e capacidade de processamento). Essa investigação é de responsabilidade do Agente Olimpo, que alocará a memória da máquina para que o objeto seja migrado para ela. Logo, o Agente Olimpo é responsável oferecer “terra nula” (muita memória, alta disponibilidade na rede, alto processamento). Então a máquina que possui este Agente Olimpo poderá ser beneficiada, caso possua os dados que o usuário precise para realizar o processamento.

Essa memória alocada pelo Agente Olimpo pode nunca ter sido usada para armazenar dados ou código, que serão utilizados pelo usuário naquele momento. Logo, essa máquina tem uma chance maior de armazenar o dado ou código do que a máquina que possui memória que já foi utilizada.

Suponha que existe um número primo para ser armazenado. O Agente Olimpo que tiver o programa do número primo poderá se oferecer para armazenar esse número. Suponha que uma determinada máquina tenha o algoritmo de fatoração, mas o Agente Olimpo desta máquina percebe que não tem nenhum número primo para fatorar. Então, os números primos

serão buscados em algum lugar (outra máquina ou no espaço de tuplas) para serem fatorados. Suponha que exista uma máquina que tem um monte de números primos. Logo é possível que esta máquina possa fatorar esses números.

Logo, é o Teseus que faz a conexão entre o Pantheon e o Agente Olimpo. Ou seja, toda a parte de sincronização é feita no Teseus, utilizando o padrão arquitetural chamado *Blackboard* (BUSCHMANN e*t al*, 1996), uma estrutura de dados utilizada para comunicação geral e mecanismo de coordenação para os múltiplos agentes e é gerenciado por um componente *controller*.

O Teseus implementa a arquitetura do *blackboard* de forma distribuída. O *blackboard* desse projeto é um espaço onde cada nó tem permissão de escrever e ler em apenas uma parte do quadro negro. De tempos em tempos, uma atualização ocorre de forma a disponibilizar para todos os espaços a informação de apenas um deles (sincronização frouxa). Esta ação é denominada *beacon*. Depois do último *beacon*, todos os nós calculam um tempo fixo e adicionam um valor aleatório. O nó com o menor tempo será o que executará o próximo *beacon*.

Quando um nó entra na rede ele passa por um período de observação. Este nó espera até receber o primeiro *beacon* e então ele passa a poder executar o cálculo para os próximos *beacons*.

O modelo de quadro negro do Teseus é mostrado na figura 19.

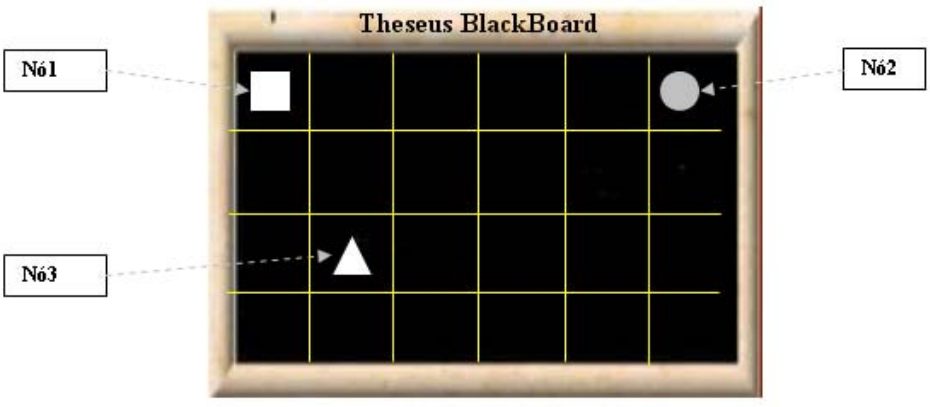

Figura 19 - Os nós escrevem em apenas um pedaço do Blackboard(t1).

Cada nó escreve objetos (representado acima por figuras geométricas) no espaço. Os nós vêem diferentes informações até que o *beacon* ocorra (t1). Por exemplo, se o nó2 envia um

beacon (figura 19) todos os demais passam a ter seus objetos mais os objetos existentes no nó2. A figura 20 mostra o *blackboard* após o *beacon*.

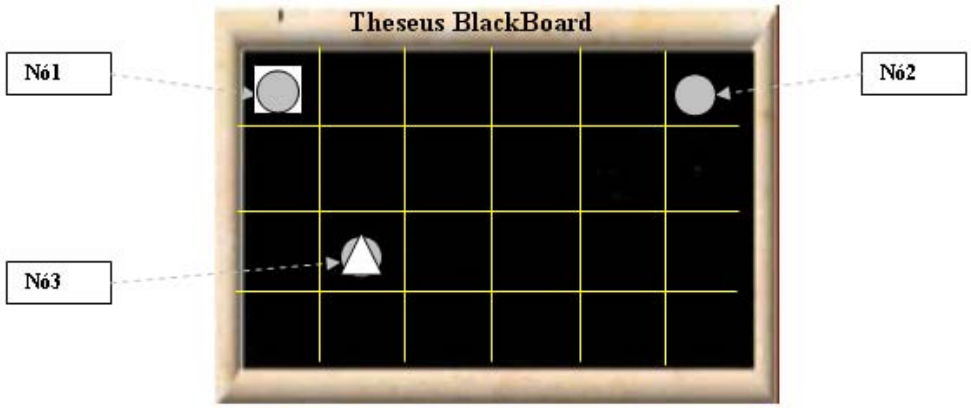

Figura 20 - Os nós tem uma visão de partes do blackboard(t2).

Nesse *blackboard* são escritas as requisições, que são disponibilizadas para toda a rede e é enviada uma mensagem para todos os Agentes Olimpo notificando sobre essas requisições. O Pantheon faz uma requisição, que será transitada pelo Teseus. O Teseus é responsável por disponibilizar essas requisições para todos os Agentes Olimpo para que este possa tentar cumprir aquela requisição. Uma vez que um Agente Olimpo se disponha a cumprir, ele novamente pede para customizar o processo, pois os Agentes Olimpo colocarão as suas ofertas no Teseus e será realizado um “leilão”, onde quem oferecer o melhor benefício pelo melhor preço será “comprado”. Então o Teseus fará esse trabalho de “leiloar”, de modo a se ter um modelo econômico.

Para que se possa garantir essa robustez, as informações são duplicadas (clonadas), para que a sobrevivência da informação seja garantida. E por isso, os clones precisam observar as informações (objetos) originais. Sendo o ponto-a-ponto o modelo adotado, cada cópia observa todas as outras.

A replicação consiste na manutenção de cópias idênticas armazenadas na memória em diferentes máquinas. A replicação é extremamente útil para a melhora na disponibilidade dos dados, visto que quando um nó sai do ar a cópia dos dados poderá ser acessada impedindo que o sistema como um todo saia do ar. Além disso, aumenta o paralelismo visto que vários nós podem vir a consultar o mesmo dado em diferentes locais.

A auditagem é a base para se verificar o valor econômico de cada um dos objetos existentes. No caso específico dos números primos, o valor econômico é computado de

acordo com quantas vezes o número é requisitado. Quando o número é requisitado muitas vezes, sua importância é aumentada, logo esse número não poderá ser perdido. Então esse número é clonado (replicado) e armazenado em algum nó que atenda os seus requisitos. Dessa forma, a alta disponibilidade desse número é garantida.

Essa replicação é realizada segundo a função $F(N) = \log_2(N)$, onde N é o número de invocações ao objeto e F é o número de clones. Logo, um objeto é replicado a cada quatro invocações feitas a ele. A escolha desta função se deve ao fato de evitar que o número de cópias cresça desordenadamente, o que aconteceria se tivesse sido utilizada uma função linear.

Existem duas possibilidades para a realização da clonagem. Essas possibilidades estão representadas nas figuras 21 e 22 abaixo. A primeira opção, representada pela figura 21, mostra que quando o número primo 2 é clonado, todas as referências a sua frente também sofrerão clonagem. Isso acarreta em um desperdício de memória, já que existirão réplicas desnecessárias, mesmo daqueles números pouco referenciados.

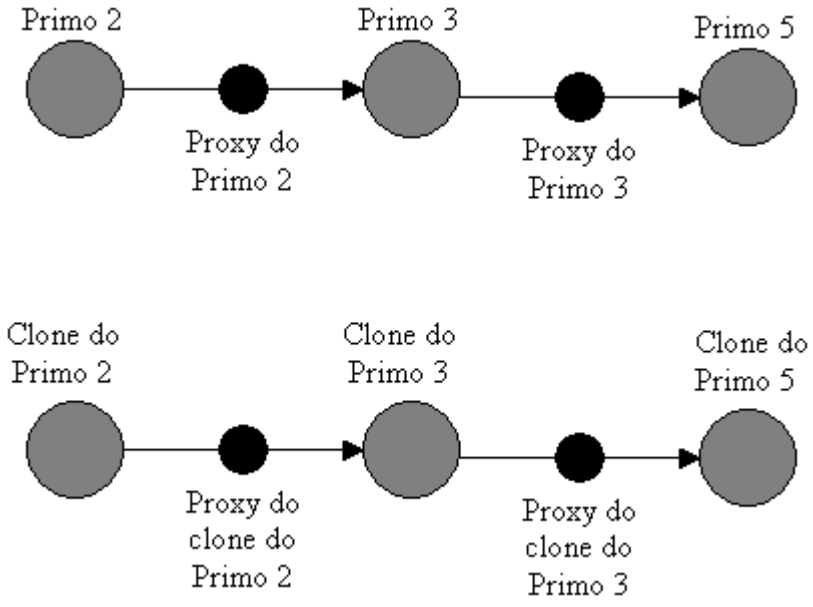

Figura 21 – Todos os números são replicados.

A segunda opção, adotada como solução neste trabalho de pesquisa e representada pela figura 23, mostra que quando o número primo 2 é clonado, este clone apontará para todas as referências do objeto original. Dessa forma, somente os objetos muito referenciados naquele momento sofrerão replicação.

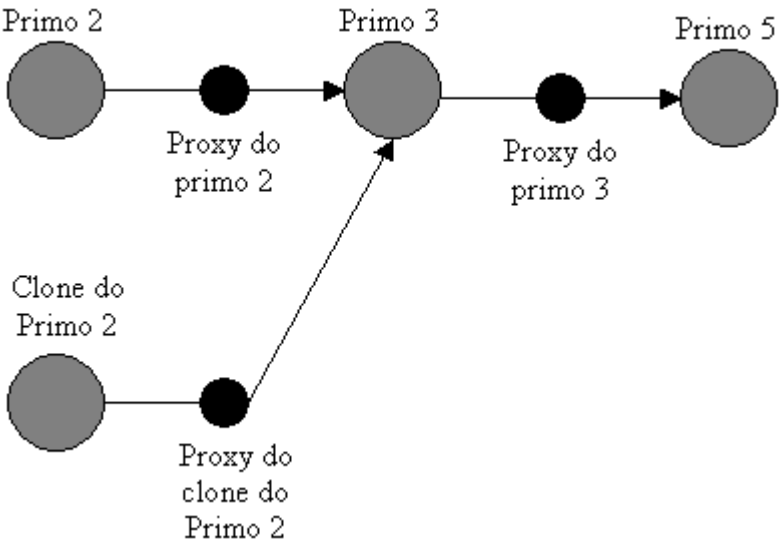

Figura 22 – Só os números mais invocados são replicados.

Portanto, cada máquina possui um Pantheon (o cliente) e um Agente Olimpo (que oferece os recursos). O Pantheon, o Agente Olimpo e o Teseus formam um modelo econômico de computação, conforme mostrado na figura 13.

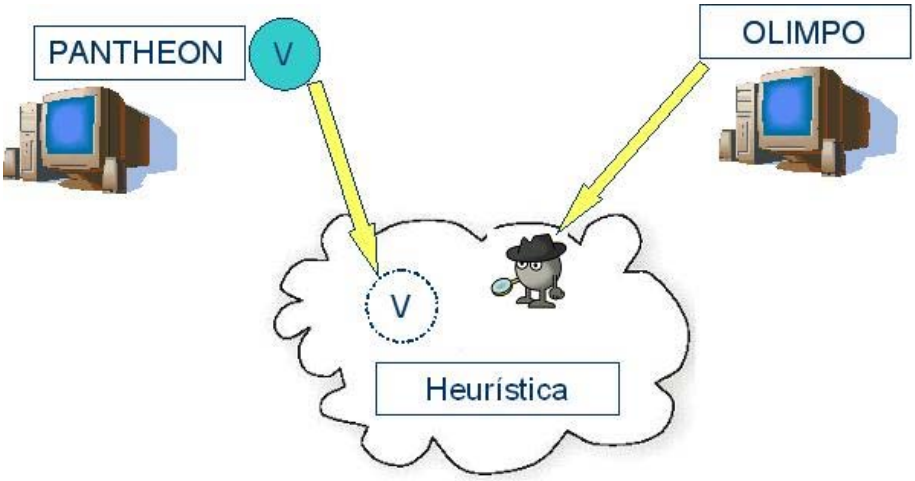

Figura 23 - Gerenciando Oferta e Procura por Recursos.

Esse modelo econômico está sendo desenvolvido por uma Tese de Doutorado (MOURA). Nesse modelo, usuários que necessitam de recursos negociam seu uso com os provedores desses, e pagam pelo serviço obtido. Entretanto, essas soluções visam à criação de rede, onde qualquer tipo de aplicação pode ser executado.

Para a execução de uma aplicação que necessite de comunicação intensa entre as tarefas, por exemplo, o cliente precisará da alocação de um conjunto de recursos que atenda a

um nível de qualidade de serviço (QoS). O usuário precisará tanto de máquinas que tenham uma determinada capacidade de processamento e disponibilidade quanto que seu canal de comunicação possua um mínimo de largura de banda disponível. Essas exigências tornam necessária a auditoria do serviço prestado pelo provedor, pois um cliente precisa de garantias de que só pagará o valor do serviço realmente obtido. Os nós provedores de serviço, por outro lado, precisam de garantias de que os clientes pagarão este valor, tornando necessários também mecanismos de moedas e bancos eletrônicos.

As tecnologias necessárias para viabilizar o uso de mecanismos de dinheiro e banco eletrônicos e a auditoria de serviços computacionais não estão disponíveis comumente. Também não é uma tarefa trivial torná-las altamente disponíveis. A ausência de mecanismos mais simples do que os baseados na em modelo economia impede usuários de aplicações mais simples de usufruir a computação em rede P2P hoje.

Em modelos econômicos, existem duas formas de alocar recursos entre agentes competidores: a economia baseada em trocas e a economia baseada em preços. Na primeira, cada agente possui um conjunto de recursos e faz trocas destes recursos com de outro agente que o interessam. Na abordagem baseada em preços, uma moeda é utilizada para expressar os custos dos recursos. Cada agente possui uma riqueza e a utiliza para comprar acesso aos recursos que necessita de outro agente. Os preços dos recursos são obtidos da negociação entre consumidores e provedores, regulando oferta e demanda.

É possível, a partir de formas de interação bem definidas nos modelos econômicos, automatizar o processo de negociação e obtenção do acesso aos serviços em uma rede P2P. Além disso, dependendo do modelo econômico utilizado, é possível cobrar dos usuários algum retorno para o provedor do serviço.

Hoje, a maioria dos sistemas construídos com base nesse modelo econômico utiliza a abordagem baseada em preços. A utilização de uma moeda dá uma flexibilidade ao sistema e possibilita o uso de mecanismos bastante elaborados para a obtenção do equilíbrio no mercado. Entretanto, utilização de uma moeda faz necessários mecanismos de controle como cobrança, segurança entre outros. Porém, prover as garantias necessárias para o funcionamento correto destes mecanismos em um ambiente computacional de participantes não confiáveis, distribuídos e grandes é uma tarefa muito complexa.

Dentro dessa perspectiva, *Economic Cnossos* (MOURA, 2007) propõem um mecanismo P2P de compartilhamento de recursos e da definição de políticas de confiança. Esse mecanismo tem como base a utilização de hipóteses capaz de satisfazer as necessidades de um conjunto de usuários. São elas:

1. Todos os usuários do sistema são provedores de serviços;

2. Os provedores de serviço são capazes de dar garantia de qualidade de serviços.

Nessa estrutura, todos os serviços e recursos são disponibilizados e negociados em um mecanismo onde são definidos os seus valores. Esses valores são determinados de acordo com mercado, conforme a quantidade de serviços e recursos ofertados e a procura dos mesmos. Feita essa definição, o valor é dado em cima de uma moeda única especificada pelo modelo. Mecanismo esse similar a uma bolsa de mercadoria e futuros, onde são oferecidos produtos para serem negociados.

Toda implementação do projeto Pantheon foi realizada utilizando a linguagem de programação Python. A estrutura da implementação (pacotes e módulos) será explicada a seguir.

Pacote maquina

Módulo maquina: instancia cada máquina e seu respectivo Agente Olimpo.

Módulo simulador: realiza a simulação de entrada e saída das máquinas da rede.

Pacote numberGenerator

Módulo numberGenerator: fornece números dentro de um intervalo para serem testados se são primos.

Pacote primo

Módulo primo: verifica se o número recebido é primo. Se o número for primo, então é gerada uma nova referência do objeto para o mesmo. Para isso, é utilizada o setReferenceObject que aqui está representado por um método (newInstance), mas deve ser encapsulado em uma classe.

```
def isPrime(self, number):
  if number % self.__value != 0:
    if self.next != NullPrime():
      obj = self.next
      obj.isPrime(number)
    else:
      logger.addLog('proximo', number)
      self.next =
         pantheonProxy.newInstance(PrimeNumber, number)
```

Listagem 2 - Trecho de código do módulo primo.

Módulo logger: armazena o número primo e quem é o seu próximo. Por exemplo: foi gerado o número 5. Ele então será testado se é primo. Logo, será verificado se ele é divisível por 2 e depois por 3 (o número 3 é o próximo do 2).

Pacote catalogo

Módulo catálogo: mapeia os números primos para uma determinada máquina e lista todos os números armazenados nas máquinas.

Pacote olimpo

Módulo strategyRoleta: representa os algoritmos da roleta.

```
#-------------------------------------------------------------
#Roleta que Soma todos os dados do agente olimpo
#-------------------------------------------------------------
class RoletaSoma(StrategyRoleta):

  def roleta(self,memoria,disco,cpu,disponibilidade):
     self._setaEstrategia(memoria)
     return
       ((cpu+disco+disponibilidade)/(memoria+disponibilidade))
```

Listagem 3 – Trecho de código do módulo StrategyRoleta.

Módulo agenteOlimpo: verifica a configuração das máquinas (disponibilidade na rede, quantidade de memória e disco e capacidade de processamento) e executa o algortimo da roleta. Além disso, lê o token existente no Teseus e verifica se poderá se oferecer para armazenar o objeto. Também escreve em um arquivo de log informações como: id e tamanho do objeto e executa a rotina de mapeamento primo-máquina.

```
soma =
self.roleta.roleta(self.memoria,self.disco,self.cpu,self.dispo
nibilidade)
```

```
original = soma
while self.go:
   time.sleep(soma)
   tupleSpace = TupleSpace()
   element = tupleSpace.read()
   now = time.localtime(time.time())
   self.arquivo.write(time.strftime("%Y-%m-%d,%H:%M:%S", now),
         str("%7.0f"%(time.time()*100)), str(self.id),
         str(self.memoria), str(element['tam']),
         str(element['freq']))

    Catalogo().mapeiaNumero(self.id, element['value'])
```

Listagem 4 - Trecho de código do módulo agenteOlimpo.

Pacote proxy

Módulo pantheon: cria uma instância do objeto, registra esse objeto (chama o módulo PantheonOid para gerar um id para o objeto) e chama o módulo resposável por criar o proxy do objeto.

```
def newInstance(self, clazz, *args):
   try:
       self.pantheonOid.registraClasse(clazz)
   except: RegistroException

   self.clazz = clazz
   obj = Proxy(clazz,
             self.pantheonOid.proximoOid(clazz), *args)
   return obj
```

Listagem 5 - Trecho de código do módulo pantheon.

Módulo pantheonOid: gera um id para o objeto.

Módulo proxy: cria o proxy do objeto, repassando os métodos desse objeto para o *audit* do proxy e o coloca no espaço (Teseus). Também é responsável por replicar os objetos.

```
def __getattr__(self, name):
  x = getattr(self.obj, name)

  if callable(x):   # retorna true quando for um método
    def proxy(*args, **kwargs):
      for stub in self.stubs:
        c = getattr(stub, x.im_func.func_name)
        y = c(*args, **kwargs)
      proxyLogger.addLog('elemento', 0)
      self.audit.addLista(self.getOid(), x.im_func.func_name)
      conjuntosDosN = range(2,10)
```

```
      #um clone é criado a cada 4 referências do objeto
      if ((log(self.audit.returnFrequencia(self.getOid()),2)
in conjuntosDosN) and condition(stub) ) :
        self.stubs += [self.createCloneProxy(self.obj, stub)]

       return y
    return proxy
  else:
    return x
```

Listagem 6 - Trecho de código do módulo proxy.

Pacote teseus

Módulo space: é o espaço de tuplas, onde os objetos são inseridos pelo Pantheon e retirados pelo Agente Olimpo, seguindo a estratégia da roleta.

```
def readRoleta(self,estrategiaOlimpo):
  # pré-seleciona os objetos da roleta que tenham a memória
  # maior ou igual ao desejado
    objSelecionado = ''
    objPreSelecionado = {}
    for numPrimo in self.objetos.itervalues():
      if(numPrimo['tam'] <=  estrategiaOlimpo.memoria):
        objPreSelecionado[id(numPrimo)]=
              RoletaObjetosPrimos(numPrimo)
        objSelecionado = numPrimo

    #roda a roleta para escolher o objeto
    maior = 0
    for objPrimos in objPreSelecionado.itervalues():
      roleta = RoletaTeseus(estrategiaOlimpo,objPrimos)
      valor=roleta.confronto()
      if (valor >= maior):
        objSelecionado = objPrimos.objeto
        maior=valor

    return objSelecionado
```

Listagem 7 - Trecho de código do módulo space.

Pacote log

Módulo audit: adiciona os objetos em uma estrutura de dados e contabiliza a freqüência com que esses objetos são referenciados.

```
def addLista(self, elemento, metodo):
  self.lista.append([elemento, metodo])
  self.addDicionario(elemento)

def addDicionario(self, elemento):
  self.frequencia = 1
```

```
  if self.dicionario.has_key(elemento):
    self.frequencia = self.returnFrequencia(elemento) + 1

  self.dicionario[elemento] = self.frequencia

def returnFrequencia(self, elemento):
  return self.dicionario[elemento]
```

Listagem 8 - Trecho de código do módulo audit.

Módulo manipulaArquivoLog: gera um arquivo de log, pedido pelo agenteOlimpo, com informações como id e tamanho do objeto.

5. Avaliação

Para atingir a meta de tornar os sistemas computacionais mais confiáveis, duas técnicas complementares podem ser seguidas: prevenção de falhas e tolerância a falhas. A primeira objetiva a confiabilidade do sistema através de uma fase de especificação e estruturação cuidadosa, seguida de uma extensa bateria de testes antes da implantação do sistema. No entanto, esta técnica não é suficiente, pois sempre haverá falhas residuais que poderão culminar com a interrupção da execução de um ou mais módulos do sistema. Para tratar destes resíduos e aumentar a integridade, a disponibilidade de recursos e a segurança do sistema, deve-se utilizar técnicas de tolerância a falhas que permitam ao sistema continuar a executar corretamente, mesmo na presença de falhas.

Mecanismos de tolerância a falhas são fundamentais para aplicações em sistemas distribuídos. A confiabilidade e a distribuição andam de mãos dadas, porque os melhores sistemas confiáveis são os construídos a partir de técnicas de tolerância a falhas distribuída. Isto é, mesmo que as nossas necessidades não incluam a distribuição, devemos construir o nosso sistema como sendo distribuído, se desejarmos elevada confiabilidade. E assim fazendo, obteremos quase sempre as restantes vantagens dos sistemas distribuídos, nomeadamente: um preço reduzido, escalabilidade e expansibilidade incremental.

O conjunto de técnicas mais importantes para conseguir confiabilidade enquadra-se no que se denomina tolerância a falhas, isto é, a capacidade de um sistema para resistir a falhas dos seus componentes sem que o serviço seja afetado. Para isso, é implementada a técnica de replicação. Este trabalho baseou-se em redundância de objetos. Réplicas idênticas de um mesmo objeto distribuídas pelos nós que fazem parte da rede aumentam a disponibilidade e a confiabilidade desse sistema. Entretanto, essas réplicas precisam manter o mesmo estado para poder responder com o mesmo resultado ou executar o mesmo serviço. Foram estudadas estratégias de controle de consistência de réplicas assim como a criação, eliminação e deslocamento dinâmico de réplicas de objetos em um sistema distribuído para aumentar o grau de confiabilidade e disponibilidade do sistema, mas sem comprometer em demasia o desempenho.

Vários testes foram feitos com o intuito de avaliar a confiabilidade do Pantheon. Para isso, medimos a resiliência do sistema em várias situações. Olhando no dicionário, a palavra "resiliência" significa:

“a capacidade de pessoas resistirem à adversidade, valendo-se da experiência assim adquirida para construir novas habilidades e comportamentos que lhes permitam sobrepor-se às condições adversas e alcançar melhor qualidade de vida.”

No âmbito do trabalho em questão, resiliência é a capacidade que o sistema tem de continuar executando mesmo que algum ou alguns nós desconectem da rede, valendo-se das cópias dos objetos armazenados em outros nós.

Os testes foram realizados utilizando várias máquinas em um ambiente distribuído. A geração de números primos foi empregada como estudo de caso, onde a amostragem vai de 0 a 100. Todos os testes foram realizados tomando como experimento o número primo 2, isto é, o nó que caíra (ou os nós) será o que possui este número. A escolha deste primo se deve ao fato dele ser o número mais requisitado no algoritmo de descoberta de números primos.

No primeiro teste, os objetos não possuíam cópias, isto, é, não eram replicados nos nós. Logo, assim que o nó que contém o primo 2 cair, o sistema quase que instantaneamente também cairá. A tabela 1 mostra número de requisições a cada número primo.

Primo	Número de requisições
2	97
3	48
5	31
7	24
11	21
13	19
17	18
19	17
23	16
29	15
31	14
37	13
41	12
43	11
47	10
53	9
59	8
61	7
67	6
71	5
73	4
79	3
83	2
89	1
97	0

Tabela 1– Simulação sem a utilização de réplicas dos objetos.

No segundo teste, os objetos foram replicados utilizando a função $F(N) = \log_2 N$, onde N é o número de requisições ao objeto (no caso o número primo). Logo, a cada 4 requisições feitas ao objeto, uma cópia do mesmo é criada. A tabela 2 mostra o resultado deste teste.

Primo	Número de requisições	Número de cópias
2	97	24
3	48	12
5	31	7
7	24	6
11	21	5
13	19	4
17	18	4
19	17	4
23	16	4
29	15	3
31	14	3
37	13	3
41	12	3
43	11	2
47	10	2
53	9	2
59	8	2
61	7	1
67	6	1
71	5	1
73	4	1
79	3	0
83	2	0
89	1	0
97	0	0

Tabela 2 - Simulação utilizando a função F(N) = $\log_2 N$.

Podemos concluir que quando um dos nós que contém o primo 2 cair, o sistema continuará rodando, pois existem réplicas deste objeto que continuarão atendendo às requisições. Entretanto, se todos os nós que contém o primo 2 caírem, o sistema cairá em um curto espaço de tempo.

No terceiro teste, os objetos foram replicados seguindo a função F(N) = $\log_4 N$. A tabela 3 ilustra o resultado deste teste.

Primo	Número de requisições	Número de cópias
2	97	6
3	48	3
5	31	1
7	24	1
11	21	1
13	19	1
17	18	1
19	17	1
23	16	1
29	15	0
31	14	0
37	13	0
41	12	0
43	11	0
47	10	0
53	9	0
59	8	0
61	7	0
67	6	0
71	5	0
73	4	0
79	3	0
83	2	0
89	1	0
97	0	0

Tabela 3- Simulação utilizando a função F(N) = $\log_4 N$.

Nota-se que o número de réplicas será menor que no caso anterior, pois agora a cada 16 requisições feitas ao objeto, uma cópia do mesmo é criada. Logo, a probabilidade do sistema cair é maior.

O quarto teste é semelhando ao segundo e terceiro. Foi utilizada função F(N) = $\log_8 N$ para determinar o número de replicas a serem criadas. Agora o número de replicas é ainda menor que no teste anterior, pois a cada 64 requisições. A tabela 4 demonstra o resultado do teste.

Primo	Número de requisições	Número de cópias
2	97	1
3	48	0
5	31	0
7	24	0
11	21	0
13	19	0
17	18	0
19	17	0
23	16	0
29	15	0
31	14	0
37	13	0
41	12	0
43	11	0
47	10	0
53	9	0
59	8	0
61	7	0
67	6	0
71	5	0
73	4	0
79	3	0
83	2	0
89	1	0
97	0	0

Tabela 4 - Simulação utilizando a função F(N) = $\log_8 N$.

Nos testes 2, 3 e 4 se um nó que contém o primo 2 cair e ainda não existirem réplicas desse objeto, o sistema também cairá. Observando as tabelas geradas nas simulações, concluímos que o melhor resultado obtido foi o teste aplicando a função f(N) = $\log_2 N$, pois foi nesta simulação que o sistema obteve uma maior resiliência.

6. Conclusão

Sistemas distribuídos têm como função principal permitir que aplicações e dados possam ser utilizados independentes uns dos outros, mas que tenham a capacidade de serem enxergados como uma única informação de forma consistente. Compartilhar informações e recursos em sistemas distribuídos é fundamental para a existência dos mesmos, permitindo o aumento do poder de processamento dos equipamentos que dele participam.

O principal objetivo deste trabalho foi o desenvolvimento de um modelo de prevalência distribuída P2P, onde os dados são armazenados nas memórias de diversas máquinas que fazem parte da rede. O modelo de prevalência previamente existente utiliza um modelo distribuído “mestre-escravo”, onde existe a predominância de um determinado nó (“mestre”). Dessa forma, se o nó mestre falhar, todo o processamento pára.

Como existe a limitação de memória das máquinas para armazenar todos os dados, foram utilizadas técnicas de replicação e fragmentação (aqui chamada de migração), empregadas em banco de dados distribuídos. Dessa forma a alta disponibilidade dos dados é melhorada, pois a probabilidade de serem acessados é otimizada por um algoritmo distribuído.

O paradigma de memória compartilhada distribuída (chamado “espaço”) foi implementada de forma simplificada neste trabalho de dissertação, permitindo que usuários disseminassem informações para todos os participantes. Com a distribuição dos dados, é possível minimizar os gastos com as redes, o processamento e a capacidade de armazenamento dos nós, dividindo o processamento e o tráfego de informações na rede.

A proposta aumenta a autonomia local. Tendo em vista os dados armazenados, cada nó é responsável pelo gerenciamento dos mesmos. Com isso, há um escalonamento autonômico distribuído, onde não existe um nó central. Cada máquina toma sua própria decisão quanto ao armazenamento dos dados, através de heurísticas que rodam independente em cada nó. E essa decisão se reflete automaticamente no escalonamento. Como estes dados encontram-se replicados, caso um dos nós se torne indisponível, o funcionamento do sistema não é inviabilizado. Como conseqüência, o sistema se torna mais robusto.

A contribuição deste trabalho é o modelo de prevalência distribuído, onde estudou-se diversas heurísticas, que foram sucessivamente experimentadas e melhoradas para tornar o

processo de distribuição de disponibilização mais robusta. Além do estudo, foi produzida uma pequena implementação de referência que está integrada ao projeto Dedalus. O Dedalus é uma plataforma P2P do projeto LABASE, que além deste módulo Pantheon, integra outros módulos que estão sendo construídos por alunos de Mestrado, como Teseus (SILVA), Atlas (GONÇALVES) e Cnossos (MOURA). Para realizar as simulações em algumas etapas do teste, este trabalho também contribuiu com simuladores simplificados do engenho de quadro-negro (Teseus) e da plataforma de mensagens (Atlas). Como contribuição teórica, este trabalho estabelece o conceito de prevalência distribuída P2P e comprova que ela pode ser construída através da implementação em *software* de uma prova de conceito.

Apesar de ter apresentado uma proposta de prevalência distribuída P2P, por problemas de tempo, alguns aspectos não foram discutidos e implementados. Na próxima seção foram listados alguns destes aspectos para serem tratados em trabalhos futuros.

Trabalhos Futuros

A implementação original do *Prevayler* salva o estado dos objetos da memória principal (*snapshot*) para o disco. Uma alternativa de implementação seria que este *snapshot* fosse salvo em um banco de dados relacional. A idéia seria monitorar os objetos que estão há muito tempo sem ser acessados e dar um *passivate* neles, ou seja, tirá-los da memória e colocá-los no banco de dados.

Em muitos casos, os objetos armazenados são muito grandes – e recuperá-los por completo a cada consulta não é uma boa idéia. A técnica conhecida como *lazy loading* (carregamento tardio) poderia ser utilizada para garantir que os objetos somente serão carregados do banco de dados para a memória à medida que forem solicitadas pelo usuário.

Um outro problema é a duplicação de referências dos objetos carregados do banco de dados para a memória, coisa o que no *Prevayler* não se consegue detectar. Essa duplicação acarreta um desperdício muito grande, visto que não existe a necessidade de ter duas ou mais referências ao mesmo objeto na memória. A solução seria a utilização de *Dynamic Proxies*, onde o modelo de domínio é baseado em interfaces e onde foram implementadas as transações. O *Dynamic Proxy* é responsável por fazer o roteamento dos relacionamentos entre os nós. Além disso, adotando o mecanismo de *lazy loading*, os objetos seriam carregados na memória somente quando utilizados, mesmo que haja em dependência entre dois ou mais objetos.

Referências

ARAÚJO, M.; SILVA, J.; DAIBERT, M.; ALMEIDA, V.; JULIO, A. - **Estratégias de Persistência em Software Orientado a Objetos: Definição e Implementação de um *Framework* para Mapeamento Objeto-Relacional**, 2006. Faculdade Metodista Granbery Bacharelado em Sistemas de Informação. Juiz de Fora – MG. Disponível em: http://re.granbery.edu.br/artigos/si/artigo_si_001.pdf. Último acesso em: 23 maio. 2007.

BAN, B. - **JavaGroups: A Toolkit for Reliable Multicast Communication**, 1999. Disponível em: http://www.jgroups.org. Último acesso em: 10 novembro. 2005.

BARBOSA, D.; VARGAS, P.; BARBOSA, J., Geyer, C. – **ReMMoS: Um Ambiente de Suporte à Replicação em Sistemas com Mobilidade Explícita de Objetos Distribuídos**, 2001. In: Conferencia Latinoamerica de Informática (CLEI).

Barry & Associates, Inc. – **Object Relational Mapping Articles**. Disponível em: http://www.object-relational.com/articles/object-relational_mapping.html. Último acesso em: 06 maio. 2005.

BATTISTI, J. – **Criando Aplicações em 3, 4 ou n Camadas**, 2003. Disponível em: http://www.juliobattisti.com.br/artigos/ti/ncamadas.asp. Último acesso em: 09 maio. 2005.

BRITO, N. ***Prevalencia: otro enfoque a la persistencia.*** Disponível em: http://www.fnbrito.com/articulos/prevayler.htm. Último acesso em 5 junho. 2004.

BURETTA, M.: Data Replication – **Tools and Techniques for Managing Distributed Information** - Jonh Wiley & Sons, Inc., 1997.

BUSCHMANN F. *et al* - "Pattern-Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996.

CHAN, Y.; SI, A.; LEONG, H. *et al* - **MODEC: A Multi-Granularity Mobile Object-Oriented Database Caching Mechanism, Prototype and Performance.** Distributed and Parallel Databases. v. 7(3), pp. 343 – 372, July 1999, Springer.

FREEMAN, E.; HUPFER, S.; ARNOLD, K. - "JavaSpaces(TM) Principles, Patterns and Practice", Addison-Wesley Pub Co, June 1999.

GAMMA, E.; HELM, R.; JOHNSON, R. - ***Design Patterns: Elements of Reusable Object-Oriented Software*.** 1. ed. Addison-Wesley, 1995. ISBN 0201633612.

GODART, C.; MOLLI, O. *et al* – **The ToxicFarm Integrated Cooperation Framework for Virtual Teams**. In: Distributed and Parallel Database, v. 15(1), pp. 67 – 88, January 2004, Kluwer Academic Publishers.

GONÇALVES, F. - **Atlas: Uma Plataforma Peer-to-Peer para Provisão de Serviços Móveis.** Tese de Mestrado. Rio de Janeiro: NCE/UFRJ (em preparação)

HELMERSEN, P. - **Human Factors in Emerging Markets: First World Solutions Addressing Third World Needs**, 2006. In: Human Factors and Telecommunications HFT 2006, Sophia Antipolis, France.

HIBERNATE REFERENCE DOCUMENTATION VERSION 2.1.7, JBoss Inc., 2006. Disponível em: http://www.hibernate.org/hib_docs/reference/en/pdf/hibernate_reference.pdf . Último acesso em: 23 maio. 2007.

HÜNN, M.; OCH, C. – **Desenvolvedores Java utilizam Caché para agilizar criação de sistemas**, 2003. Disponível em: http://www.intersystems.com.br/cgi-bin/nph-mgwcgi?MGWLPN=ISC&wlapp=ISC&PAGE=NEWSLETTER&NewsId=229. Último acesso em: 07 maio. 2005.

InterSystems – **InterSystems Caché**, 2001 - Disponível em: http://www.intersystems.com.br/isc/CacheTecnoIntro.csp?CSPCHD=0000000000023ge2k60k003010886609. Último acesso em: 15 março. 2007.

JONG, I. DE – **PYRO: Python Remote Objects**, 2005. Disponível em: http://pyro.sourceforge.net/. Último acesso em: 28 agosto. 2006.

KELLER, W. - **Mapping Objects to Tables: A Pattern Language**. In: Proceedings of the 2nd European Conference on Pattern Languages of Programming (Europlop '97). Siemens Technical Report 120/SW1/FB. Munich, Germany: Siemens, 1997. Disponível em:http://www.riehle.org/community-service/hillside-group/europlop-1997/p13final.pdf. Último acesso em: 23 maio. 2007.

LERMEN, A. - **Um Framework para Mapeamento de Aplicações Orientadas a Objetos a Bancos de Dados Relacionais/Objeto-Relacionais**. In: Semana Acadêmica de CPGCC, 3., 1998. Porto Alegre: Curso de Pós-206 Graduação em Ciência da Computação - Instituto de Informática – Universidade Federal do Rio Grande do Sul, 1998. Disponível em: http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana98/lermen.html. Último acesso em: 23 maio. 2004.

MOURA, L. – **Modelo Econômico do Cnossos**. Rio de Janeiro: NCE-UFRJ, 2007 (Relatório Técnico, Rt-02/2007)

MOURA, L. – **Cnossos: Um Micro Kernel P2P distribuído para Grid Computing**. Tese de Doutorado. Rio de Janeiro: COPPE-UFRJ (em preparação)

MURTA, L.; VERONESE, G. *et al* – **Mor: Uma ferramenta para mapeamento Objeto-Relacional em Java**, 2001. In: *XV Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas*, pp.392-397, Rio de Janeiro, RJ, Brasil, outubro 2001. Disponível em: https://sety.cos.ufrj.br/prometeus/publicacoes/mor.pdf . Último acesso em: 22 de agosto. 2005.

OLIVEIRA, A.; GOMES, L. *et al* – **Algoritmos para Replicação de Objetos em Redes de Distribuição de Conteúdo**. Belo Horizonte: Departamento de Ciência da Computação – UFMG, 2002.

ORFALI, R.; HANKEY, D. Hankey, EDWARDS J. - **The Essential Distributeds – Objects**, 1996. In: Survival Guide, John Wiley & Sons, Inc.

OZSU, M. T.; VALDURIEZ, P. - **Principles of Distributed Database Systems -** 2nd Edition Prentice Hall, 1999.

PASIN, M.; WEBER, T. S. - **Protocols for server reintegration in a distributed system providing file replicas** - In: Internacional Symposium on Computer and Information Sciences, ISCIS, 14., 1999. Bornova: Ege University, 1999. p.1037-1039.

RAJAMANI, K.; LEFURGY, C. - **On evaluating request-distribution schemes for saving energy in server clusters**, 2003 –In: Performance Analysis of Systems and Software., Austin, TX, USA. p. 111- 122.

RICHA, A. W.; PLAXTON, G.; RAJARAMAN, R. - **Accessing Nearby Copies of Replicated Objects in a Distributed Environment**, Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), June 1997.

ROSS, J.; WESTERMAN G. - **Preparing for utility computing: The role of IT architecture and relationship management Source**. In: IBM Systems Journal, Volume 43, Issue 1 (January 2004), Pages: 5 – 19, ISSN:0018-8670

SILVA, A. – **Migrando Bancos de Dados Relacionais para Tecnologia Objeto-Relacional**, Dissertação de Mestrado, Departamento de Ciência da Computação da Universidade Federal de Minas Gerais (UFMG), 1999. Disponível em: http://www.dcc.ufmg.br/pos/html/spg99/anais/alecia/alecia.htm. Último acesso em: 06 maio. 2005.

SILVA, I. Da, - **Teseus: Escalonador de Espaço de tuplas distribuído para processamento em GRID**. Tese de Mestrado. Rio de Janeiro: NCE-UFRJ (em preparação).

SNMP Research – The SNMP Protocol - Disponível em: www.snmp.com. Último acesso em: 12 dezembro. 2006.

STONEBRAKER, M.; MOORE, D. - **Object-Relational DBMSs: The Next Great Wave**. Morgan Kaufmann Publishers, 1996. ISBN 1-55860-397-2

Utility Computing – Disponível em http://www.utilitycomputing.com/. Último acesso em: 20 de agosto de 2006.

VILLELA, C. – **An introduction to object prevalence**, 2002. Disponível em: http://www-106.ibm.com/developerworks/web/library/wa-objprev. Último acesso em: 05 maio. 2005.

WEISER, M. – **The Computer for the 21st Century**, Scientific American, Vol. 265 No. 3, September 1991, pp. 94-104.

WEISSTEIN, E. W. - **Sieve of Eratosthenes**, 1999 - From *MathWorld* - A Wolfram Web Resource.

WILSON, B. J. - **JXTA**, 2001. Disponível em: http://www.jxta.org. Último acesso em: 10 de novembro de 2005.

WUESTEFELD, K. – **Prevayler**, 2001. Disponível em: http://www.prevayler.org/wiki.jsp. Último acesso em: 05 maio. 2005

WUESTEFELD, K., ROUBIEU, J. – **Prevayler: Objetos Java Invulneráveis**, 2003. http://www.objective.com.br/artigoprevayler.html. Último acesso em: 05 maio. 2005.

ZHOU F., JIN C., WU, Y., ZHENG W. - **TODS: cluster object storage platform designed for scalable services.** In: **Algorithms and Architectures for Parallel Processing, 2002**. pp. 36- 43.

ZIMBRÃO, G. - **Mapeamento Objeto-Relacional – Transforme um Modelo de Classes em um Modelo Relacional**. In: Revista SQL Magazine, Rio de Janeiro: Neofício Editora v.5, ano 1, p. 28-33, Edição 5, 2003.