Maurício Emanuel Dourado Cescato

Mentor: Um ambiente para a Distribuição Transparente de Software

Dissertação submetida ao corpo docente do Instituto de Matemática e do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Prof. Carlo Emmanoel Tolla de Oliveira, Ph.D.

Rio de Janeiro

2005

AGRADECIMENTOS

Em primeiro lugar, aos meus pais pelo amor incondicional e pela formação que me possibilitaram. O homem que hoje sou é fruto de um trabalho de muitos anos. Obrigado por tudo!

Meus agradecimentos ao prof. Carlo Emmanoel, pela extraordinária criatividade e experiência demonstradas durante toda a sua orientação. Você foi minha bússola enquanto "navegava em mares revoltos", mostrando que sua sabedoria estava sempre dois passos à frente de meu entusiasmo!

Ao prof. Gilson Tavares, pela semente plantada em mim há quatro anos atrás, e também por insistir em regar uma planta tão rebelde! Você me tornou um profissional melhor, no período em que contigo trabalhei.

À "Tia Deise", à "Zezé" e à Regina pela dedicação, paciência e compreensão demonstradas durante a minha caminhada em busca da realização deste sonho.

Aos colegas do projeto SIGA, pela agradável convivência durante o período em que juntos trabalhamos. Foi importante estar com vocês, pois me senti acolhido no ambiente de trabalho e realizado como profissional, fato que possibilitou meu desenvolvimento, e me proporcionou as bases de conhecimento para realizar este trabalho. E aos meus amigos pelo suporte, nos momentos de dúvida e dificuldade, pois os verdadeiros amigos sempre se fazem presentes nestes momentos.

RESUMO

CESCATO, Maurício Emanuel Dourado. **Mentor: Um ambiente para a Distribuição Transparente de Software**. Rio de Janeiro, 2005. Dissertação (Mestrado em Informática) – Núcleo de Computação Eletrônica, Instituto de Matemática, Universidade Federal do Rio de Janeiro.

Os métodos utilizados por fornecedores de software na distribuição de seus produtos evoluíram pouco. A comunidade de desenvolvimento está consciente de que as aplicações precisam ser atualizadas constantemente para que se mantenham úteis para seus usuários. O número de máquinas que precisam de atualização nas suas aplicações aumenta cada vez mais, na medida que novos dispositivos computacionais com baixo custo de produção se tornam mais disponíveis aos usuários. Alguns fornecedores tentam criar sistemas eletrônicos para atualização de software capazes de solucionar seus problemas particulares, mas não existe convergência de interesses para tentar criar uma solução padronizada que contemple a todos. Em vista das crescentes dificuldades que existem no acesso ao software, uma proposta de sistema de distribuição de software é apresentada: o Mentor. O objetivo é criar uma infra-estrutura para a disponibilização de diversas aplicações e suas atualizações, sem penalizar os usuários com tarefas ligadas à distribuição de software que poderiam ser resolvidas automaticamente. Este trabalho também propõe a colaboração entre as máquinas com o objetivo de dar acesso ao software a qualquer usuário, independente da plataforma e dos requisitos técnicos, ajudando máquinas desprivilegiadas na execução de software que anteriormente não estava acessível. O trabalho utilizou o modelo ponto-a-ponto e uma implementação de um espaço enuplário para permitir o compartilhamento de recursos entre as máquinas, tanto para a distribuição, quanto para a execução de software. Como forma de validação, um estudo de caso foi realizado através de um experimento que simula a utilização real do sistema de distribuição implementado neste trabalho.

ABSTRACT

CESCATO, Maurício Emanuel Dourado. **Mentor: Um ambiente para a Distribuição Transparente de Software**. Rio de Janeiro, 2005. Dissertação (Mestrado em Informática) – Núcleo de Computação Eletrônica, Instituto de Matemática, Universidade Federal do Rio de Janeiro.

Improvements in software distribution methods have hardly been observed in the software industry. The development community is conscious that applications must be updated quite often in order to keep adding value to theirs costumers. The number of computers requiring software updates increases as long as low cost computing devices become more available to users. Although some software houses attempt to create specific electronic systems for software updating in order to solve particular problems, very little effort has been put together in order to create a common solution. Facing an environment with crescent access barriers to software applications, a software distribution system is presented: Mentor. The objective of this work is create an infrastructure that gives access to various applications and updates, without penalizing users with tasks related to software distribution that could be done automatically. This work also proposes machine cooperation with the objective of providing software access to any user, in spite of platform or technical requirements, and help users to execute software previously unaccessible. This work used the peer-to-peer model and an implementation of a tuple space to allow resource sharing between computers to distribute and run software. As a form of validation, a case study was developed throughout an experiment that simulates the use of the actual system implemented during this work.

LISTA DE FIGURAS

LISTA DE SIGLAS

P2P	Peer-to-peer
DLL	Dynamic-link Library
RPM	Red Hat Package Manager
LSB	Linux Standard Base
BSD	Berkeley Software Distribution
APT	Advanced Packaging Tool
RMI	Remote Method Invocation
MVC	Model-View-Controller
MD5	Message Digest
JPF	Java Plugin Framework
XML	Extensible Markup Language
SSH	Secure Shell
J2ME	Java 2 Micro Edition

SUMÁRIO

1 INTRODUÇÃO

1.1 Fornecedores ainda distribuem software de maneira inadequada

1.1.1 Modelos Atuais de distribuição de software: meio físico e meio eletrônico

Os processos de desenvolvimento de software mais recentes se baseiam fortemente no modelo espiral, que é composto por diferentes etapas, realizadas na seguinte ordem: planejamento, análise dos riscos, engenharia e avaliação do cliente (PRESSMAN, 1995). As etapas se repetem a cada volta dada na espiral com o objetivo de agregar novas funcionalidades e fazer correções em funcionalidades entregues anteriormente, baseadas no aprendizado do cliente durante a etapa de avaliação.

A distribuição de um software é realizada através da execução de atividades necessárias para a instalação ou a atualização do mesmo nas máquinas de seus usuários. A distribuição é uma tarefa que ocorre ao final de cada volta na espiral, uma vez que uma nova versão da aplicação foi lançada e está pronta para utilização. Os modelos de distribuição de software existentes diferem em relação ao meio utilizado para o transporte da aplicação até a máquina do usuário, podendo ser por meio físico (mídia) ou por meio eletrônico.

O modelo de distribuição mais difundido hoje faz uso de mídia digital. Ele compreende a gravação do software em disquetes ou cds e seu transporte para as lojas onde podem ser comprados. Apesar de ser o modelo mais utilizado, possui desvantagens importantes:

- A demora para que as alterações sejam disponibilizadas efetivamente nas máquinas de seus usuários;

- Custos de produção, distribuição e armazenamento da mídia usada;

- Instalação manual da aplicação, que pode necessitar da presença de um técnico ou pessoa com capacitação suficiente para a tarefa.

Os modelos de distribuição que usam o meio eletrônico evoluíram bastante com o crescimento e difusão da Internet. A idéia de usar a Internet para distribuir software está em pleno uso. O problema em implementar este modelo é a existência de outros custos que não estavam presentes no modelo de meio físico, como a utilização de máquinas potentes no papel de servidores e também da grande largura de banda passante entre usuários e servidores. Estes custos podem atingir valores elevados com o crescimento do número de usuários atendidos.

Boa parte dos mecanismos de distribuição de software que hoje usam o modelo eletrônico se limitam à distribuição de arquivos apenas, cabendo ao usuário a tarefa de instalá-los. Existem muitos servidores na Internet que disponibilizam software gratuito, mas sua permanência no ar hoje em dia depende de patrocínios e propagandas que pagam seus custos fixos.

O futuro da distribuição de software é com certeza por meio eletrônico e sem sombra de dúvida é o caminho certo a ser seguido. A questão é como realizar esta tarefa e como utilizar o meio eletrônico da melhor forma possível.

1.1.2 Eliminar esforços duplicados no desenvolvimento e manutenção de sistemas de atualização

Existem várias iniciativas de distribuição de software por meio eletrônico. Os autores Ladislav Sobr e Petr Tuma apontam para o aumento do uso da Internet com o objetivo de distribuir software, Eles também comentam que diferentes fornecedores estão tornando esta prática fragmentada, pois cada fornecedor implementa uma solução para os seus problemas particulares de distribuição (SOBR, TUMA, 2005).

Fornecedores de software podem optar por sistemas de distribuição eletrônicos para distribuir uma ou mais aplicações. Uma vez instaladas nas máquinas de

seus usuários, tais aplicações permitem que sejam atualizadas sempre que for necessário. Programas antivírus e editores de texto são exemplos de software que permitem atualizações automáticas. Muitas vezes estas atualizações ocorrem de forma transparente ao usuário, ou seja, não necessitam de sua interação.

A fragmentação descrita por Sobr e Tuma é conseqüência da falta de uma padronização ou da ausência de um sistema de distribuição comum a diferentes fornecedores de software. A distribuição de software é ainda encarada como a distribuição de um produto comum de prateleira. Existe alguma resistência de fornecedores e usuários em enxergar o software como um produto diferente do produzido pela indústria tradicional, como eletrodomésticos, carros e outros produtos duráveis. Fornecedores persistem na utilização de modelos de distribuição caros e trabalhosos uma vez que o software muda com muita facilidade. Usuários resistem em atualizar seu software por encará-lo como um produto industrializado. Porem, até mesmo os produtos industrializados precisam voltar à fábrica para reparos, por exemplo, considere os *recalls* na indústria automobilística.

Muitas vezes a atualização de software é necessária para permitir que o software permaneça funcionando de maneira adequada e agregando valor a seus usuários. A distribuição de um software vai além do transporte de um produto, ela deve garantir ao usuário que o software esteja pronto para uso e resolver, por antecipação, eventuais problemas que possam ocorrer no uso do mesmo. O objetivo da atualização eletrônica é o mesmo de um técnico em informática que vai ao local do cliente e instala o programa, dando a necessária assistência após a entrega do produto.

Existe ainda uma falta de consenso entre os fornecedores de software para entender que o caminho para a distribuição do software será por meio eletrônico. Muitos fornecedores preferem continuar utilizando a mídia magnética uma vez que podem prever com mais facilidade os custos de distribuição. De qualquer forma, a distribuição de um software é claramente uma responsabilidade do fornecedor. Por este motivo é natural que existam

dificuldades para a existência de um esforço conjunto para padronização se muitos deles se consideram os únicos responsáveis pela distribuição de seus produtos. Nenhum fornecedor quer depender de terceiros para uma tarefa tão crítica. Enquanto o consenso entre fornecedores não ocorre, cada um distribui software da sua forma.

1.1.3 Pervasive Computing: o usuário e o novo cenário computacional

Nos últimos anos, importantes progressos tecnológicos ocorreram na área de miniaturização e comunicação de curta distância. Além do desenvolvimento tecnológico, a economia de escala já permite a construção de computadores de um só microchip a custos muito baixos. Estes chips agregam cada vez mais capacidade computacional e permitirão num futuro próximo a comunicação entre quaisquer objetos ou dispositivos presentes no dia-a-dia das pessoas. Celulares e assistentes pessoais digitais (pdas) são exemplos de dispositivos que já fazem parte desta realidade.

Um novo mercado se abre: o Pervasive Computing. Novas oportunidades surgem para fabricantes de hardware e software para explorar este novo cenário computacional em que as pessoas se encontrarão envolvidas no futuro. Torna-se agora possível implementar novos sistemas de informação levando-se em conta a possibilidade da utilização de computadores disponíveis às pessoas. A utilização destes sistemas cria problemas na forma como o software será instalado e atualizado nestes dispositivos (ANDERSSON, 2000).

Todo software sofre atualizações, independentemente do hardware onde ele é instalado. Com tantos computadores disponíveis, maior será a tarefa de atualizar o software neles. Quem deve assumir a responsabilidade pela atualização: o usuário ou o fornecedor?

É bastante provável que os usuários não aceitarão este ônus. A concorrência no comércio de software fará com que este tipo de tarefa seja requisito básico exigido pelo cliente. Uma vez que os primeiros fornecedores disponibilizarem a atualização automática como diferencial competitivo de seu software, em breve

outros fornecedores concorrentes serão forçados a disponibilizar a mesma facilidade. A característica deixa de ser a vantagem de um, e passa a ser requisito indispensável a todos os concorrentes.

O elevado número de dispositivos móveis forçará os fornecedores a abandonar o modelo de atualização de software presencial (o técnico vai ao local), que será extremamente caro em função da logística difícil ou até impossível. O caminho é a automação destas tarefas de atualização de software (ANDERSSON, 2000).

1.1.4 O alto custo da escalabilidade do modelo cliente-servidor

Existem muitas limitações tecnológicas que desafiam a distribuição de software por meio eletrônico:

- Velocidade de processamento;
- Espaço em disco para guardar todas as versões de todas as aplicações disponíveis no mercado;
- Banda passante insuficiente para um volume grande de dados.

De modo especial, os sistemas implementados para a distribuição de software que usam o modelo cliente-servidor (uma ou mais máquinas servidoras atendendo a diferentes máquinas clientes) enfrentam gargalos de acesso criados pelo grande número de usuários acessando simultaneamente estes sistemas e exigindo a transferência de um grande volume de dados.

As limitações e gargalos são resolvidos nestes sistemas através da "força bruta": um grande número de servidores é adquirido e a infra-estrutura das redes é modificada para comportar o grande volume de usuários (TURCAN, SHAHMEHRI, GRAHAM, 2002).

A escalabilidade é a capacidade de um sistema garantir a qualidade de serviço na medida em que o número de usuários do sistema aumenta. Um indicador muito importante para saber se um sistema é escalável é a medição do tempo médio de atendimento. Se o tempo médio de atendimento aumentar muito no momento em que o sistema atende a uma grande massa de requisições, fica bastante claro que o sistema tem dificuldades em escalar e algumas medidas devem ser tomadas para garantir a qualidade do serviço neste cenário de carga.

Solucionar problemas de escalabilidade por meio da "força bruta" funciona e existem inúmeros casos de sucesso que comprovam sua utilização. O grande ponto fraco desta abordagem é o alto custo ou investimento. No momento em que há um maior número de requisições, maior é o investimento e maiores são os custos ao persistir em usar esta solução. Este investimento é necessário para conviver com picos de acesso ao sistema. Picos de acesso são eventos que podem ocorrer em momentos, previstos ou não, e em intervalos de duração variável. Fora estes eventos, boa parte do tempo as máquinas servidoras e o restante da infra-estrutura ficam ociosas. Existe ainda a questão de balanceamento de carga. Se não houver uma boa política de balanceamento de carga, de nada adiantará incorporar mais servidores se as requisições de usuários acabam se concentrando num subconjunto pequeno de máquinas.

Sistemas que seguem o modelo cliente-servidor têm escalabilidade de alto custo. É importante que estudos sejam realizados no desenvolvimento de um modelo que utilize os recursos das máquinas da melhor forma possível. Este trabalho propõe a adoção do modelo ponto-a-ponto na distribuição de software com o objetivo de permitir a utilização de recursos das máquinas existentes de uma maneira mais eficiente.

1. 2 Construir um sistema automatizado para distribuir software

1.2.1 Distribuição transparente: facilitar os papéis de fornecedores e usuários

De acordo com a pesquisa realizada por E. Turcan e outros autores (TURCAN, SHAHMEHRI, GRAHAM, 2002), são muitos os critérios que influenciam a qualidade de um serviço de distribuição de software. Nesta pesquisa, fornecedores e usuários relataram requisitos que consideram importantes. Uma análise indicou que parte destes requisitos são similares, mostrando que, de modo geral, os fornecedores têm uma boa idéia do que seus clientes desejam.

Estes são os requisitos que os fornecedores consideram mais importantes:

- Entregar o software ao cliente com baixo esforço e baixo custo
- Prover uma alta qualidade de serviço, em termos de tempo de resposta, velocidade e tempo de transferência, além de outros atributos
- Assegurar a entrega segura do software;
- Assegurar a escalabilidade do sistema, considerando o tamanho do software e a comunidade de usuários

Os requisitos mais importantes apontados pelos usuários são:

- O sistema deve ser amigável e transparente na forma como funciona
- O sistema deve ser flexível e de fácil configuração, sendo adaptável de acordo com os parâmetros de ambiente do usuário
- O sistema deve usar de forma eficiente os recursos disponíveis, principalmente a largura de banda

- O sistema não deve interferir com o software e hardware operacional da máquina do cliente

- O sistema deve ser capaz de verificar a integridade do software e de sua fonte

Os requisitos apontados pelos usuários indicam claramente a necessidade de um sistema automático em que as tarefas de atualização sejam transparentes aos mesmos. Observou-se ainda, durante o desenvolvimento do trabalho, que as soluções existentes não atendem de forma adequada a estes requisitos ligados à transparência. Provocam muitos conflitos e problemas, deixando para o usuário o encargo de resolvê-los. Por outro lado, os requisitos dos fornecedores levantados pelo trabalho do grupo de Edward Turcan estão relacionados à boa qualidade do serviço de distribuição, à redução dos custos e à manutenção rápida e fácil.

As soluções de distribuição implementadas até hoje, que atendem a diferentes fornecedores de software, como as distribuições do Linux, oferecem caminhos demorados para a disponibilização de versões de terceiros em seus sistemas de distribuição. Cada empresa responsável por uma distribuição particular do Linux tenta evitar que um programa externo, quando distribuído em sua rede, não será nocivo a ela e não provocará erros que afetem a seus usuários. Estes fatos certamente diminuiriam a receptividade do seu sistema operacional perante o público. A idéia deste trabalho é implementar um sistema de distribuição que possa atender a diferentes fornecedores, sem criar entraves para a disponibilização das diversas versões. Se um software disponibilizado pela rede de distribuição não atende aos interesses de um determinado usuário, isto é uma questão a ser resolvida entre usuário e o fornecedor do produto, isentando o sistema que o distribui.

Este trabalho se destina a propor um sistema para a distribuição de software, facilitando o trabalho de manutenção de fornecedores e mantendo a simplicidade da interação dos usuários de software com o sistema de distribuição. O objetivo é criar uma infra-estrutura para a disponibilização de

diversas aplicações e suas atualizações, sem penalizar os usuários com tarefas ligadas à distribuição de software que poderiam ser resolvidas automaticamente.

1.2.2 Viabilizar a utilização dos recursos existentes adotando o modelo ponto-a-ponto

O levantamento de requisitos feito pelo grupo de estudo de Turcan (TURCAN, SHAHMEHRI, GRAHAM, 2002) mostra que muitos deles são de natureza não funcional e estão relacionados ao desempenho do sistema. Este desempenho depende em parte da forma como utiliza seus recursos. A pesquisa do grupo de estudo analisou diferentes fontes sobre a distribuição de software por meio eletrônico e a conclusão do grupo determinou que os critérios mais usados na avaliação de um sistema de distribuição eletrônico são a largura de banda e a escalabilidade.

A capacidade de processamento das máquinas, o espaço em disco e a largura de banda são recursos que devem ser sempre utilizados da melhor forma possível num sistema de distribuição. A proposta deste trabalho é a de alcançar um melhor aproveitamento destes recursos, com a adoção do modelo ponto-a-ponto para compor a estrutura do sistema de distribuição.

Uma característica importante de um sistema de informação que adota o modelo ponto-a-ponto e que visa o aproveitamento de recursos, é a descentralização do papel de servidor no sistema. Esta situação não ocorre no modelo cliente-servidor, pois neste caso existe um conjunto pré-determinado de máquinas designadas para este papel. No modelo ponto-a-ponto, qualquer máquina pertencente à rede é elegível para dar suporte ao funcionamento do sistema. Qualquer máquina pode contribuir para o sistema, compartilhando seus recursos com outras máquinas e exercendo o papel de servidor por alguns momentos.

Os gargalos de transmissão encontrados em sistemas de distribuição eletrônicos podem ser resolvidos armazenando instalações do mesmo software

em mais de uma máquina, que atuaria como um repositório. Assim, uma máquina, quando disponível, pode atuar como um repositório de um determinado software, servindo como uma opção possível para aqueles usuários que requisitam aquele software. Ao mesmo tempo, a instalação ou atualização de uma aplicação pode utilizar fontes diferentes, desafogando a banda passante de máquinas que disponibilizam o mesmo software.

A determinação do número de fontes usadas e a escolha de fontes geograficamente próximas no momento de uma transmissão dependem de fatores diversos. O número de usuários interessados no software naquele momento e a velocidade de transmissão de cada cliente e dos repositórios são questões relevantes que levam a uma decisão mais acertada sobre o número máximo de pessoas que podem instalar o software em determinados momentos, sem prejudicar a qualidade do serviço prestado pelo sistema de distribuição.

Existem muitas questões técnicas e não técnicas a serem endereçadas na implementação de um sistema ponto-a-ponto (MINAR, HEDLUND, 2001). Como o sistema de distribuição envolve mais de uma máquina, se uma operação precisar de informações ou decisões que dependam de máquinas diferentes, pode haver a necessidade de implementar uma transação distribuída, o que é bastante custoso.

Em situações em que a comunicação entre máquinas demora razoavelmente, podem ocorrer comportamentos assíncronos. Este tipo de comportamento exige um esforço adicional com programação multithread e tratamento de acessos concorrentes de memória para evitar inconsistências (MUGHAL, RASMUSSEN, 2000). Outras questões também são importantes para a implementação do sistema, como a elaboração de heurísticas para decisão de quais máquinas ajudarão nas tarefas do sistema, e para determinar quando elas estarão livres para ajudar. Para facilitar explicações e o entendimento deste trabalho, o termo "ponto" será usado muitas vezes para referenciar uma máquina em um conjunto de máquinas onde opera um sistema no modelo ponto-a-ponto.

1. 3 Problemas de natureza humana persistirão se não forem enfrentados

1.3.1 A inevitabilidade do lançamento de novas versões e suas conseqüências

Existem dois motivos importantes que fazem com que fornecedores lancem novas versões de suas aplicações: o aprendizado humano e a necessidade de correções. O aprendizado humano é agregado ao software a cada alteração e agregação de novas funcionalidades. Usuários podem contribuir no desenvolvimento de um software comentando como o seu uso poderia ser melhorado. Interações diretas ou indiretas entre usuários e equipe de desenvolvimento são essenciais para a construção de sistemas que atendam às necessidades de seus usuários.

Em sistemas feitos sob medida para seus usuários, a interação entre usuário e fornecedor é geralmente mais intensa. O usuário encomenda o sistema de informação com o objetivo de automatizar tarefas que são feitas de forma manual. À medida que o sistema de informação é construído, o usuário vê materializado o sistema que idealizou, e muitas dúvidas e idéias surgem do seu aprendizado em relação ao sistema em construção. Este aprendizado é importante porque serve para forçar o aparecimento de mudanças no sistema que agregarão mais valor aos usuários, incluindo aquelas que ainda são desconhecidas (BECK, 2000).

Muitos sistemas nos dias de hoje não são feitos sob encomenda, muito menos são personalizados. Mesmo nestes casos, as vantagens obtidas em ouvir e implementar o que os usuários desejam têm a mesma importância para a construção de software útil.

Outro motivo importante para o lançamento de versões é a necessidade de fazer correções no software, provocadas pela ocorrência de erros, mudanças de conteúdo, entre outras.

O primeiro motivo não pode ser evitado pelo bem que faz ao usuário, logo deve ser encorajado. O segundo motivo não pode ser totalmente evitado, apenas

amenizado através de maiores estudos na elaboração de processos de desenvolvimento mais eficientes. O problema não é apenas de estudo dos processos, mas de implantação no ambiente corporativo. As empresas fornecedoras devem ser capazes de implantar processos compatíveis com sua organização e com a aplicação a ser desenvolvida ou mantida.

A conclusão é que lançamentos de novas versões de um mesmo software são inevitáveis. Novas versões de um software serão lançadas sempre que o fornecedor estiver disposto a desenvolvê-lo ou mantê-lo, ou seja, enquanto sua comercialização é ainda um projeto financeiramente viável para a empresa. As atualizações de software são necessárias para disponibilizar mudanças e apresentar um software mais adequado às necessidades de seus usuários. Dependendo do sistema de distribuição, estas atualizações podem ser realizadas por uma pessoa ou de forma automática pelo sistema.

Quanto menos automático for o sistema de distribuição, maior será a inércia ou demora na atualização de software. Se não existe facilidade e interesse do usuário em atualizar o software presente em uma máquina, maior será esta inércia. Sistemas de distribuição baseados no meio físico possuem as maiores inércias de atualização se comparadas a sistemas de distribuição eletrônicos.

O lançamento de novas versões gera um encargo ao sistema de distribuição utilizado pelo fornecedor do software, pois este sistema de distribuição será o responsável pela atualização do software. Além do encargo de atualização, outro problema importante para a distribuição de software é provocado também pela mesma necessidade de fornecedores em lançar versões. Este problema é chamado de problema das dependências, e será comentado no próximo tópico.

1.3.2 As dependências entre componentes e os problemas gerados

O paradigma da programação orientada a objetos tornou popular a criação e utilização de componentes de outros fornecedores no desenvolvimento de um software. Uma versão de um software que reutiliza outros componentes possui uma árvore de dependências. Este software foi projetado para funcionar com

versões específicas de outros componentes, e cada um desses componentes dependem de outros componentes, formando uma árvore de dependências.

O mesmo reuso que permite menores custos no desenvolvimento de software é uma das causas da dificuldade em se automatizar o processo de atualização de software. Esta situação será explicada no final deste tópico. Antes disso, diferentes situações serão mostradas para uma melhor análise do problema.

Vários cenários podem ocorrer durante as operações de instalação, remoção e atualização de software numa máquina. Cada cenário tem um desfecho que impede ou dificulta a atualização de um software com dependências numa máquina de um usuário. Estes cenários são demonstrados abaixo. Todos os cenários partem do princípio que será realizada alguma operação de instalação, atualização ou remoção envolvendo um software A que tem dependência para um pacote de software B.

1º cenário: um software A já instalado usa uma versão antiga de B. Se um software C for instalado e uma nova versão de B instalada, A pode parar de funcionar se o componente for substituído.

2º cenário: o software A parou de receber manutenção de seu fornecedor. Se o pacote B for atualizado com uma nova versão por algum motivo, A pode parar de funcionar.

3º cenário: B é um componente que possui uma árvore de dependências e nem o software A nem B foram instalados ainda. A instalação de toda a arvore de B será necessária, e os pacotes serão instalados recursivamente.

4º cenário: o pacote B e sua árvore de dependências foram instalados para atender a um software C. A instalação de A pode resultar na parada do funcionamento de C se a árvore de B for reescrita com outras versões de B e dependências.

5º cenário: o software A será desinstalado, e deve-se pensar na necessidade de desinstalar B e sua árvore de dependências. A desinstalação de B dependerá se o sistema é capaz de saber se outros softwares ainda fazem uso do componente. Como cada software instalado possui sua própria árvore de dependências, podem existir ramos de dependências de diferentes softwares entrelaçados entre si. A árvore de dependências de A não pode ser prontamente removida, pois poderá afetar outras aplicações.

Estes cenários mostram que qualquer atividade de atualização de um software que possui dependências pode gerar problemas sérios se não forem executadas de forma apropriada. Um software sem dependências poderia ser instalado, atualizado e removido sem problemas, mas este não é o caso mais freqüente.

O reuso de componentes é a causa para a existência de dependências de software. Conforme visto nos cenários acima, o reuso combinado com a inevitabilidade do lançamento de versões são as causas para a ocorrência de dificuldades na atualização de software.

Tais problemas independem de limitações tecnológicas e sistemas de atualização precisam propor soluções para minimizá-los. Só após definir uma proposta para as dependências é possível tentar automatizar esta parte do sistema de distribuição, tornando-o menos dependente de interações com o usuário.

1.3.3 O pouco enfoque dado à divulgação do software

O software é um produto que precisa ser divulgado por seus fornecedores para que sua distribuição aconteça com maior intensidade. Existem algumas formas básicas de divulgar um produto: confeccionar e transmitir anúncios em veículos de comunicação e contar com a propaganda boca-a-boca feita por usuários do produto.

As soluções de distribuição de software pesquisadas neste trabalho não estudam formas de melhorar a divulgação de software. Algumas delas até disponibilizam ao usuário mecanismos para obter uma listagem e descrição de softwares disponíveis. Mesmo assim, cabe ao usuário a iniciativa de encontrar aplicações que sejam úteis.

Neste tipo de abordagem, o usuário só procura um software quando ele possui uma necessidade específica. É muito improvável que durante sua procura ele descubra ou se lembre que possui outras necessidades além daquela que motivou sua busca.

Fornecedores de software encaram a divulgação como uma atividade separada do processo de distribuição. Uma das idéias deste trabalho é propor meios de divulgar software através do mecanismo de distribuição. Esta divulgação deve ocorrer de forma mais efetiva, respeitando interesses particulares de cada usuário e somente divulgar informações a quem interessar.

A proposta deste trabalho é diferente da proposta de anúncios pagos que atingem pessoas para atrair interesse. É esperado que o número de pessoas interessadas seja bem menor que número de pessoas atingidas pelo anúncio. A proposta de anúncios pagos não é uma idéia nova. Ela é um meio de propaganda disponível e muito utilizado por fornecedores atualmente. Um dos objetivos deste trabalho é complementar as opções de divulgação já existentes. A divulgação de software promovida pelo Mentor levará em conta os interesses particulares de cada usuário. Uma vez que um usuário define quais são suas áreas de interesse, o sistema de distribuição pode filtrar aplicações de menor importância e somente apresentar aplicações com maiores chances de agregar valor a ele.

O estudo de como descrever interesses de usuários e de como associar estes interesses ao software propriamente dito possibilitará a criação de uma nova forma de divulgação mais precisa, com custos baixos e poupando o tempo de usuários com divulgações improdutivas.

2 REVISÃO BIBLIOGRÁFICA

2. 1 O problema de Classificação e Busca de Componentes de Software

Os autores Rubén Prieto-Diaz e Peter Freeman decidiram criar um mecanismo para incentivar a reutilização de componentes de software (PRIETO-DIAZ, FREEMAN, 1987). A dificuldade de reuso existe quando um grande número de componentes estão disponíveis e não há uma classificação padronizada dos componentes.

Os autores utilizaram uma solução de classificação e busca baseada em conceitos da ciência da informação. Os autores usaram a idéia de facetas em sua solução para busca de componentes. Os usuários desta solução são desenvolvedores de software que desejam reutilizar componentes de seu próprio grupo ou de terceiros. Um componente pode ser classificado por diferentes critérios ou facetas. A classificação de objetos deve se restringir somente a critérios relevantes para o usuário. Cada critério ou faceta possui um conjunto de valores possíveis para a classificação de um objeto.

São três os critérios mais importantes definidos pelos autores na classificação de componentes. Na ordem decrescente de prioridade se encontram o nome da função do componente[1], os tipos dos objetos manipulados pela função e a estrutura de dados usada na função. Uma vez definidas as facetas, seus valores possíveis são cadastrados e todos os objetos são classificados em cada uma das facetas, desenvolvedores de software podem realizar buscas utilizando os mesmos critérios da classificação para obtenção de objetos.

Classificações de objetos podem ser feitas de duas formas: através de facetas ou de enumeração. Os autores afirmam que o método de classificação

[1] Na época, funções eram os únicos componentes disponíveis, muito distante dos componentes orientados a objetos).

utilizando facetas é mais eficiente que o método de classificação tradicional, o enumerativo.

O método enumerativo define uma única árvore de classificação, ao contrário do facetado, que define várias árvores, uma para cada faceta. O método se baseia em organizar todas as classificações numa única árvore onde cada nó é uma classificação mais especializada que seus antecedentes. Objetos são classificados em um dos nós que são folhas da árvore de classificação. De acordo com os autores, o método enumerativo apresenta dificuldades de expansão e manutenção, ao contrário do facetado, que pode criar novos valores para as facetas ou então novas facetas.

A classificação usando facetas deve levar em conta a existência de palavras sinônimas dentro do mesmo conjunto de valores possíveis de uma faceta. Dois objetos cujos valores de facetas são palavras sinônimas deveriam gerar a mesma classificação para ambos. Prieto-Diaz e Freeman concluiram então que era necessário criar um vocabulário de sinônimos. Uma palavra dentre um conjunto de palavras sinônimas era escolhida para representar o significado comum.

Os autores criaram uma tabela para guardar registros representando conjuntos de palavras sinônimas. A tabela foi apelidada de Thesaurus em função do seu grande tamanho. Toda vez que uma classificação de um objeto ou busca é realizada, cada palavra usada como valor de faceta deve ser trocada pela sua palavra sinônima que é determinada pela tabela Thesaurus.

A proposta dos autores inclui também a criação de grafos, um para cada faceta, definindo pesos para medir quão parecidos são os valores possíveis de uma faceta. Estes grafos são utilizados em casos em que uma busca não retorna o resultado esperado. Eles permitem ao usuário a opção de relaxar algum parâmetro usado na busca anterior, trocando um valor por outro parecido e executando a busca novamente.

2. 2 Estudo das soluções para a distribuição de software

2.2.1 Revisão Bibliográfica de Sistemas Operacionais

Alguns sistemas operacionais foram pesquisados, pois estão relacionados de alguma forma à distribuição de software.

2.2.2 Sistema Operacional Windows

O sistema operacional Windows é um exemplo de software distribuído através do meio físico. O sistema de distribuição automático mais conhecido que existe para o Windows é o Windows Update, porém ele serve apenas para atualizar o próprio sistema operacional. O Windows Update não serve para a atualização de aplicações de outros fornecedores de software, porém ele é importante para este trabalho, pois apresenta problemas típicos que devem ser resolvidos num sistema de distribuição de software. Estes problemas são gerados pela existência de diferentes versões de pacotes usados pelas aplicações que são distribuídas.

O sistema operacional Windows disponibiliza um sistema de bibliotecas para promover a reutilização de código entre diferentes fornecedores de software. Essas bibliotecas são chamadas de dynamic-link library, ou DLLs (DLL, 2005). As DLLs foram projetadas para permitir que diversas aplicações instaladas possam compartilhar a mesma biblioteca em memória.

Quando uma aplicação é instalada, muitas vezes o programa de instalação copia para o sistema uma versão diferente de uma DLL já instalada, por não existirem garantias de compatibilidade. Neste caso, duas medidas podem ser tomadas: substituir a cópia anterior pela nova, o que pode provocar problemas com os softwares previamente instalados; ou então instalar a nova DLL, desperdiçando espaço em disco, espaço em memória e também provocando uma demora no carregamento de programas, uma vez que o Windows precisará localizar a versão correta da DLL requisitada.

Este problema agrava-se cada vez que um novo software deseja instalar suas próprias versões de DLLs. Outro agravante é que na desinstalação de

aplicações estas cópias das DLLs não são removidas, pois não se consegue distinguir se as aplicações remanescentes poderão precisar delas. O sistema operacional fica mais lento com tantas bibliotecas com código duplicado presente em memória.

O termo "inferno das DLLs" (DLL, 2005) é uma expressão utilizada coloquialmente para referenciar as dificuldades na gerência de bibliotecas de software DLL. O inferno das DLLs é um efeito de práticas inadequadas de desenvolvimento de software que devem ser evitadas, um vez que sua utilização põe em risco o sucesso na construção de software com qualidade.

Existem muitas medidas que podem ser tomadas para amenizar o problema. Uma medida importante é a criação de um órgão centralizado que faria a distribuição das bibliotecas DLLs para diferentes fornecedores de software. Atualmente, as DLLs permitem que mais de um fornecedor possa alterar o mesmo componente, o que deveria ser responsabilidade de um único fornecedor.

O órgão garantiria que apenas um fornecedor fosse o responsável por um componente. Toda mudança feita por um fornecedor deveria ser informada ao órgão. Fornecedores praticariam o reuso procurando por componentes cadastrados no órgão.

A construção da plataforma .NET para desenvolvimento de aplicações para Windows leva em consideração os problemas gerados no passado pelas DLLs. A plataforma .NET da Microsoft possui uma versão própria de um sistema de instalação de pacotes, chamado de Assemblies, que tem uma abordagem diferente para resolver o problema de versões de pacotes.

2.2.2 Distribuições do Linux

O funcionamento genérico de um sistema de pacotes

O Linux é um sistema operacional distribuído por diferentes organizações. Todas as distribuições Linux possuem um sistema de gerenciamento de

pacotes, usado para a instalação de software de diferentes fornecedores. De uma maneira geral, toda aplicação no Linux tem dependências para versões de pacotes que devem estar instaladas no sistema.

Algumas distribuições Linux utilizam a Internet para a instalação de aplicações, e por isso são exemplos de sistemas de distribuição que utilizam o meio eletrônico. Estes sistemas seguem um padrão comum. Um sistema geralmente é composto por uma coleção de ferramentas que servem para automatizar o processo de instalação, atualização, configuração e remoção de pacotes de software de um computador.

Todas as aplicações são distribuídas na forma de pacotes e em geral cada instalação é representada por um arquivo. Este arquivo conterá informações do pacote como seu nome, versão, arquitetura ou processador compatível, dependências com outros pacotes e algum método de paridade para identificar se o arquivo foi corrompido. O arquivo pode conter informações sobre a forma como o pacote deve ser utilizado, como configurá-lo, entre outras. O gerenciador de pacotes do sistema operacional usa estas informações para executar as operações que atualizarão a máquina, a pedido do usuário.

O RPM

O formato de pacotes de instalação mais difundido do Linux é o RPM, sigla para Red Hat Package Manager (RPM, 2005). Além do termo RPM ser usado para referenciar o formato, o RPM é o nome dado ao gerenciador de pacotes que trabalha com o formato de pacotes de mesmo nome.

O gerenciador RPM instala, atualiza e faz verificações em programas. O RPM é formato de pacotes usado como referência pelo Linux Standard Base (LSB), projeto conjunto de várias distribuições do Linux para manter a padronização de qualquer sistema operacional baseado no Linux. O RPM foi desenvolvido pela companhia americana Red Hat para a distribuição Red Hat Linux, e hoje é usado por diferentes distribuições do Linux.

O RPM usa como ferramenta para a instalação de pacotes o comando de linha rpm. O RPM tem sido muito criticado pela comunidade Linux pela falta de consistência que ocorre com nomes de pacotes e conteúdo, tornando difícil o tratamento automático de dependências entre pacotes.

No entanto, este problema ocorre também com outros formatos de pacote pois é gerado pela falta de coordenação entre as principais distribuições Linux que usam o RPM. O tratamento automático de dependências pode funcionar bem quando os pacotes utilizados para atualização de uma máquina foram gerados para uma distribuição do Linux instalada na máquina do usuário. Além disso, ferramentas construídas por cima do RPM devem ser usadas para permitir este tratamento de dependências.

Outros sistemas de pacotes baseados no formato RPM

O YUM é um exemplo de ferramenta de gerência de pacotes que trabalha com a estrutura do RPM. O YUM foi desenvolvido para a distribuição Yellow Dog Linux e gerencia pacotes no formato RPM, podendo ser utilizado em outras distribuições. Ele trata as dependências entre pacotes RPM.

Outro exemplo de gerenciador baseado no RPM é o Urpmi. O Urpmi é o gerenciador de pacotes da distribuição Mandrake Linux que abstrai o usuário dos muitos problemas gerados pelo controle manual das dependências.

Debian APT

O APT ou Advanced Packaging Tool (APT, 2005) é o sistema de gerenciamento de pacotes da distribuição Debian, uma versão do Linux muito utilizada. O APT é uma ferramenta que simplifica bastante a tarefa de instalar e remover software em sistemas Linux. O APT é composto por uma biblioteca de funções escritas em linguagem C++. Estas funções são usadas por diversos comandos de linha utilizados pelos usuários para efetuar operações com pacotes.

Durante a instalação de um pacote de software, o APT detecta os pacotes que faltam e tenta instalá-los automaticamente. Caso exista mais de uma versão de um mesmo pacote ele informa ao usuário quais pacotes são mais recomendados para instalação.

O APT foi originalmente criado para trabalhar com arquivos de extensão .deb. Para aumentar a gama de pacotes que poderiam ser utilizadas no Debian, foi criada uma ponte chamada de Apt4rpm que permite a instalação de pacotes RPM.

Uma das vantagens da distribuição Debian é a disponibilização de um repositório com mais de dez mil pacotes, acessíveis a qualquer máquina ligada à Internet, Estes pacotes podem ser carregados e instalados em qualquer momento.

Muitos desenvolvedores de software disponibilizam suas atualizações de software em repositórios independentes, muitas vezes criados por eles. Os usuários Debian podem usufruir destes repositórios, mas sua disponibilidade não é garantida.

Para facilitar a vida dos usuários, foram criadas interfaces gráficas para evitar o uso direto dos comandos do APT, como a interface Aptitude e a Synaptic (APT, 2005).

Gentoo Portage

O Portage é o sistema de gerenciamento de pacotes da distribuição Gentoo Linux (PORTAGE, 2005). A atualização de pacotes requisitada pelo usuário ocorre através do carregamento do código fonte do software pela rede e de todos os pacotes que ele depende. Todo o código fonte obtido é usado para a compilação do software. A compilação pode ser parametrizada com o intuito de aproveitar a configuração da máquina do usuário, que pode ser limitada.

A instalação de novos softwares ocorre num ambiente protegido, que é uma espécie de caixa de areia. Esta caixa cria um isolamento do que está sendo feito dentro da caixa em relação ao resto do sistema. Dessa forma a instalação não afeta outros softwares já instalados e sendo utilizados pelo usuário no mesmo momento da instalação.

O Portage é um sistema mais avançado de gerência de pacotes se comparado com outras distribuições do Linux. Ele é o responsável pelo Gentoo Linux ser conhecido popularmente como uma meta-distribuição do Linux. Cada vez que o Gentoo é instalado numa máquina, uma nova distribuição de Linux é gerada com base em diversos parâmetros do Portage e das opções de pacotes informadas.

Existem milhares de receitas de instalação de pacotes de software. Estas receitas indicam ao Portage como ele deve compilá-los e instalá-los.

Novas compilações são executadas a cada instalação de novos pacotes. A instalação inicial do sistema é feita através da montagem de um ambiente que permite ao Portage baixar pela rede o código fonte e uma ferramenta para compilar e instalar o núcleo do sistema operacional e qualquer outro software indicado pelo usuário.

O Portage dá suporte ao uso de arquivos binários na instalação. A utilização a partir de arquivos binários é mais indicada quando a compilação e instalação de um pacote ou de um sistema inteiro são penosas para máquinas com hardware limitado. Nestes casos é preferível gerar os binários em máquinas com mais recursos. O tempo de compilação é uma desvantagem do Gentoo Linux.

O funcionamento do Portage é baseado em outro gerenciador de pacotes, o sistema de portas utilizado pela distribuição Linux BSD (Berkeley Software Distribution). Ambos compilam pacotes na instalação a partir do código fonte e gerenciam dependências.

Uma das peculiaridades do Portage é permitir a existência de mais de uma versão do mesmo pacote instalado.

Os problemas gerados pelas dependências entre pacotes

O "inferno das dependências" é uma expressão muito utilizada coloquialmente para designar as situações problemáticas em que usuários de sistemas operacionais Linux se encontram ao tentar instalar pacotes de software que dependem de versões específicas de outros pacotes.

Um gerenciador de pacotes pode se recusar a instalar um determinado software, dado que alguma versão específica de um pacote não tenha sido instalada. É possível que ocorram conflitos com versões mais recentes do que a efetivamente declarada na dependência.

Gerenciadores de pacotes e ferramentas com verificação de dependências podem levar a um longo caminho, onde cada pacote requisitado que possui dependências de outros pacotes é um nó em uma árvore, que sempre se expande em função de novas dependências. De forma geral, gerenciadores de pacotes não conseguem resolver este tipo de conflito, o que leva a uma perda de tempo do usuário para encontrar uma solução satisfatória.

2.2.3 Trabalhos acadêmicos relacionados ao problema da distribuição de software

Foi feita uma pesquisa de trabalhos acadêmicos com o objetivo de encontrar propostas e artigos na área de distribuição de software que pudessem contribuir para este trabalho. A pesquisa concluiu que existe um consenso entre diferentes autores sobre as classificações de distribuição possíveis (todos exceto o autor da proposta do ASMA (YAP, NG, 1999), que será comentada em breve). O sistema pode ser classificado pelo meio utilizado para o transporte do software, através do meio físico ou do eletrônico.

Os autores concordam que sistemas que usam o meio eletrônico têm uma subclassificação que indica qual máquina inicia o processo de atualização de

software da máquina do cliente, podendo ser *push* ou *pull*, se iniciado pelo servidor ou pelo cliente, respectivamente.

Todas as propostas pesquisadas utilizam o meio eletrônico. Todos os autores concordam com a utilização da Internet para a distribuição de software, independente dos fins serem comerciais ou não (software gratuito).

Todos os autores, exceto os responsáveis pelas propostas GDN (BAKKER, STEEN, TANENBAUM, 2001) e SOFANET (SOBR, TUMA, 2005), têm uma clara preocupação com a automatização dos processos de distribuição, evitando quaisquer participações de usuários no sucesso destas atividades.

A proposta do sistema JDRUMS (ANDERSSON, 2000) é permitir a atualização de software escrito em Java no nível de classes e em tempo de execução. O foco do JDRUMS é realizar uma prova de conceito de um mecanismo de atualização dinâmica barato, pois é feita via software ao invés de hardware. A idéia do trabalho é propor uma solução de atualização de classes em sistemas críticos que não podem ficar fora do ar. Não é foco da proposta a atualização de um software por completo. Esta questão não é abordada pelo autor por não ser o objetivo do seu trabalho.

O SOFANET (SOBR, TUMA, 2005) é um sistema de distribuição que concentra seus estudos em formas de realizar e cumprir contratos de uso de software entre consumidores e fornecedores, garantindo que consumidores só utilizem as licenças que têm direito pelo contrato firmado com a empresa fornecedora. Não foi declarada pelos responsáveis desta proposta a intenção de estudar o problema da falta de transparência de utilização do sistema por parte do usuário.

O ASMA (YAP, NG, 1999) é uma proposta de distribuição que usa o conceito de agentes. Os agentes seriam usados para automatizar tarefas de administração de software em redes de computadores numa empresa. O trabalho não enfoca no estudo das razões que fazem tais tarefas de administração darem tanto trabalho.

O SOFTWARE DOCK (HALL, HEIMBIGNER, WOLF, 1999) é a solução mais antiga e consolidada entre as demais, já implementada e testada. Este sistema de distribuição foi projetado para usar o modelo cliente-servidor. Uma das motivações deste trabalho é propor um sistema de distribuição cuja escalabilidade não seja cara.

Todas as propostas previamente mencionadas seguem o modelo cliente-servidor, onde uma ou mais máquinas são definidas previamente como máquinas servidoras. O modelo ponto-a-ponto utilizado neste trabalho tenta aproveitar recursos subutilizados de todas as máquinas em uma rede para ajudar na distribuição de software. Nenhuma máquina participa da rede como cliente sem ter a oportunidade de ajudar, diferentemente do modelo cliente-servidor, onde os papéis de clientes e servidores são predefinidos.

Somente duas propostas dentre as pesquisadas seguem o modelo ponto-a-ponto. Elas foram bastante importantes para este trabalho em função da escolha deste modelo e são apresentadas separadamente.

Muitas idéias foram retiradas das propostas de distribuição ASDA (TURCAN, SHAHMEHRI, GRAHAM, 2002) e GDN (BAKKER, STEEN, TANENBAUM, 2001). As propostas sugerem o compartilhamento de responsabilidades e de recursos entre máquinas, típicas de um sistema ponto-a-ponto, e serviram para validar a proposta do Mentor em relação à melhor utilização dos recursos disponíveis. As contribuições mais importantes para a idealização do Mentor são mostradas a seguir.

Ambas as propostas têm clara preocupação com a escalabilidade, e ambos os autores acreditam que se deve estudar e trabalhar para viabilizar a melhor utilização de recursos ao invés de gastar quantias elevadas persistindo na adoção do modelo cliente-servidor em sistemas de distribuição de software.

Uma das propostas seguindo o modelo ponto-a-ponto é o GDN. O GDN é uma aplicação para distribuição de pacotes de software gratuito, em operação desde o ano 2000. Apesar de qualquer máquina pertencente à rede GDN poder

contribuir como um repositório de software, determinados serviços são indispensáveis ao funcionamento da rede e precisam estar sempre funcionando. Determinadas máquinas devem estar sempre disponíveis para prover tais serviços, o que mostra que o GDN não é um sistema que usa um modelo ponto-a-ponto puro.

O GDN não tem a preocupação em resolver conflitos de instalação, pois ele é um sistema de distribuição de arquivos, sejam de instalação, atualização, pacotes, ou de outros. O que o usuário precisa fazer depois de carregar localmente o arquivo é de sua responsabilidade.

A outra proposta é o ASDA. O ASDA é uma proposta para um sistema de distribuição de software puramente ponto-a-ponto. O trabalho de pesquisa do ASDA contribuiu com o levantamento de requisitos de uma solução de distribuição por meio eletrônico, além de levantar os critérios para a avaliação deste tipo de sistema.

Os requisitos levantados pelos diversos autores destas propostas mostram sua preocupação com a questão da facilidade de uso por parte de usuários e fornecedores. Os critérios mostram a preocupação com limitações de banda que causam gargalos de acesso, além de outros problemas. Até a escrita desta dissertação, não se teve conhecimento de nenhum detalhamento da arquitetura do ASDA, nem de uma possível implementação.

2. 3 Conclusão do estudo das soluções para a distribuição de software

A leitura do material bibliográfico desta dissertação ajudou a esclarecer quais são os problemas reais que ocorrem durante o processo de distribuição de software por meio eletrônico. Isto também ajudou a identificar os autores que se depararam com estes problemas e que tipo de abordagem utilizaram para enfrentá-los.

O estudo desta dissertação ressaltou que algumas propostas focam em questões específicas da distribuição de software, mostrando diferentes

prioridades nestas propostas. Cada autor se limita a abordar o problema de distribuição do seu ponto de vista particular.

Requisitos e questões levantadas pelas diversas propostas foram levadas em consideração no planejamento do Mentor. A existência de propostas que seguem o mesmo modelo do Mentor, o ponto-a-ponto, demonstra que o caminho não é inédito, porém foi pouco pesquisado.

3 PROPOSTA DE UM SISTEMA DE DISTRIBUIÇÃO NO MODELO PONTO-A-PONTO QUE LEVA EM CONTA PROBLEMAS HUMANOS

3. 1 Sistema no modelo ponto-a-ponto que aproveita a oferta de recursos

Esta dissertação defende a proposta de um sistema de distribuição de software onde cada máquina compartilha seus recursos com outras máquinas conectadas à Internet, desempenhando funções de clientes ou de servidores, seguindo o modelo ponto-a-ponto.

O acesso ao sistema de distribuição deve ser fácil, tanto para usuários quanto para fornecedores. Esta facilidade é obtida através do desenvolvimento de um programa cliente cuja instalação e execução é simples. O programa deve ser instalado em todas as máquinas participantes. O programa permitirá a comunicação entre máquinas por uma espécie de protocolo ou linguagem comum entre elas.

A arquitetura do Mentor foi elaborada com o objetivo de permitir que diferentes máquinas pudessem agir como clientes e servidores, através da execução de códigos remotos num modelo parecido com o Java RMI (RMI, 2005), mas sem a necessidade nem a obrigatoriedade de ter conhecimento de qual máquina servidora possui o código remoto a ser executado. O suporte à execução de códigos remotos dá opção a qualquer máquina de delegar o processamento de qualquer tarefa a uma outra, sem obrigar que ela a execute localmente.

A arquitetura do Mentor é dividida em três camadas de alto nível: comunicação, controle e aplicação. A camada de comunicação dá suporte à construção de qualquer ação que pode ser acordada entre máquinas desta rede. Esta camada permite que máquinas se enxerguem numa rede, pois várias instâncias de clientes do Mentor podem estar executando num determinado momento. À medida que uma máquina reconhece a existência de outras, ela

pode executar ações remotas em máquinas diferentes, no momento que julgar necessário.

A tecnologia Jxta (JXTA, 2005) foi escolhida para a implementação da camada de comunicação. Os diferentes canais de comunicação do Jxta, também conhecidos como *pipes*, foram artifícios levados em consideração no planejamento desta camada.

Um fator determinante na escolha do Jxta foi a capacidade da própria tecnologia em permitir a comunicação entre máquinas independente das topologias das redes envolvidas e de diferentes barreiras de comunicação que possam existir, como firewalls. Outras tecnologias foram descartadas pela dificuldade ou incapacidade em permitir a comunicação entre máquinas nestas condições adversas, o que tornaria difícil a configuração e participação de máquinas presentes em outras redes e separadas por barreiras.

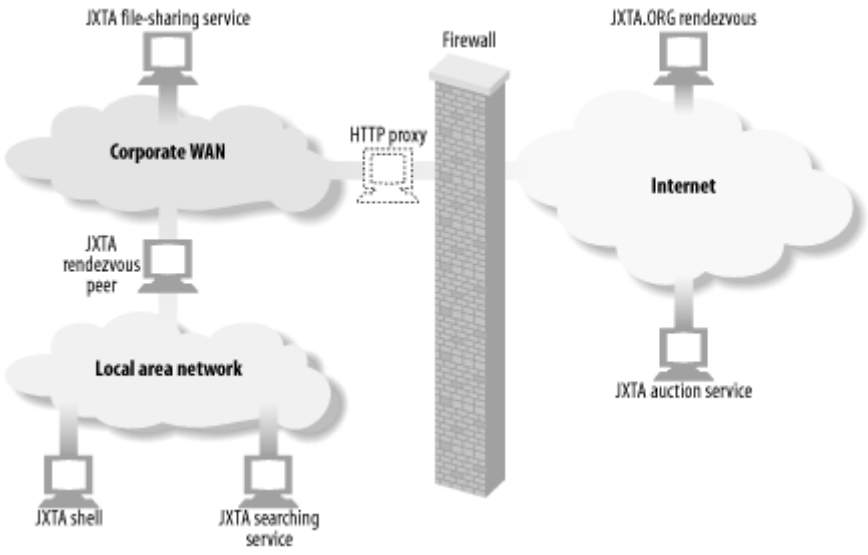

Figura 3.1: Um exemplo de máquinas com aplicações diferentes que se comunicam entre si através do Jxta, desprezando topologias de rede e firewalls
(OAKS, TRAVERSAT, GONG, 2002)

A segunda camada, a de controle, representa o conjunto de ações efetivamente implementadas usando o ferramental disponibilizado pela camada de comunicação. Estas ações foram criadas para serem executadas por

qualquer ponto na rede, sendo fundamentais para o funcionamento do sistema de distribuição de aplicações.

A última camada, a das aplicações, representa todo software, escrito em linguagem Java (JAVA, 2005), que pode ser distribuído pelo Mentor. A princípio, nem toda aplicação distribuída pelo Mentor precisa do suporte a comunicação ponto-a-ponto disponibilizada pela camada de comunicação, pois muitas delas funcionam localmente. Em geral, aplicações pertencentes a esta camada não precisam ter conhecimento das camadas anteriores.

A estrutura em camadas permite que cada camada em uma máquina possa se comunicar com a camada de mesmo nível presente em outra máquina, sem ter conhecimento de como as camadas em níveis inferiores são implementadas. A comunicação entre camadas de mesmo nível acontece de forma indireta.

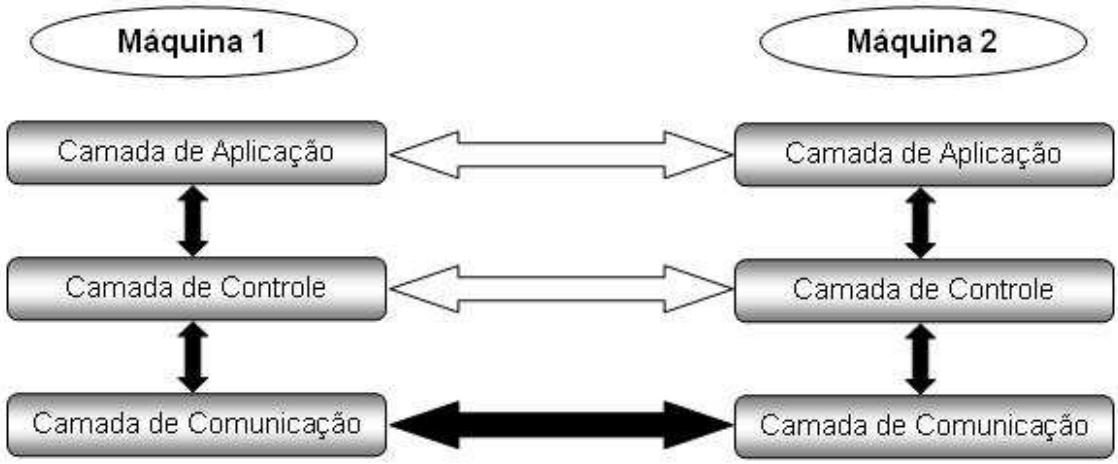

Figura 3.2: Comunicação entre camadas de forma direta ou indireta, representada por setas pretas e brancas, respectivamente.

3. 2 Soluções propostas para os problemas de natureza humana.

3.2.1 Proposta para as dependências de componentes

A linguagem Java foi escolhida para ser usada no desenvolvimento de aplicações distribuídas pelo Mentor. Ela foi escolhida pela sua portabilidade, permitindo que aplicações pudessem ser disponibilizadas em qualquer

dispositivo. Independente da linguagem em que aplicações são escritas, problemas são gerados pela dependência entre componentes presentes na máquina do usuário, que é conseqüência do reuso. Em função da escolha específica do Mentor pelo Java, é natural esperar que ocorram problemas gerados por dependências, conseqüências da forma como o reuso é permitido por esta linguagem. Para entender os problemas gerados por dependências específicos da linguagem Java é necessário ter um breve resumo de como a linguagem funciona e como ela possibilita a reutilização de código.

Java é uma linguagem interpretada. É necessário um programa para a interpretação deste código, chamado de Máquina Virtual Java. Qualquer plataforma que possui implementação deste programa pode executar programas Java, o que dá a grande vantagem dos programas Java poderem funcionar em qualquer plataforma. Esta vantagem é chamada de portabilidade.

Uma das principais razões da escolha da linguagem Java neste trabalho é permitir que qualquer máquina tenha acesso ao sistema de distribuição e em conseqüência tenha acesso às aplicações distribuídas pelo sistema. Se um programa cliente fosse desenvolvido para uma plataforma específica, isto excluiria outros usuários com outras plataformas de ter acesso ao sistema de distribuição, limitando o público atingido.

A linguagem Java permite a programação orientada a objetos. Tudo em Java deve ser implementado usando classes e objetos, incentivando o desenvolvimento de aplicações no paradigma de programação orientada a objetos. O código de uma classe que está pronto para interpretação por uma máquina virtual é chamado de bytecodes. O reuso de componentes é permitido em Java através da utilização de arquivos compactados, os chamados arquivos JAR. O arquivo JAR é formado por um conjunto de classes na forma de bytecodes, podendo ser gerado por qualquer fornecedor de software que deseja disponibilizá-lo para reuso.

Problemas gerados pelo reuso aparecem quando a situação descrita a seguir ou alguma variante ocorre. Este cenário é representado por uma aplicação A

que utiliza dois componentes de terceiros B e C, representados pelos arquivos B.JAR e C.JAR, respectivamente. Ambos os componentes têm dependência para um terceiro componente D, representado pelo arquivo D.JAR.

Como os componentes estão sempre mudando, o fornecedor da aplicação A pode mudar alguma versão dos componentes B e C utilizados. Em determinados intervalos do ciclo de vida da aplicação A, ela pode necessitar de versões diferentes do mesmo arquivo D.JAR. Quando este cenário se apresenta, problemas podem ocorrer.

A linguagem Java não disponibiliza um meio imediato de permitir que mais de uma versão de um mesmo arquivo JAR possam coexistir carregados em memória pela mesma máquina virtual. Uma classe fundamental do Java, o ClassLoader, é a classe responsável pelo carregamento de todas as outras classes utilizadas por uma aplicação. O ClassLoader identifica uma classe de forma única usando o caminho de pastas onde ela está presente, mais o nome desta classe.

O ClassLoader não pode garantir e muito menos informar, qual versão de uma mesma classe foi a última a ser carregada em memória, dado que mais de uma versão desta classe se encontra em arquivos JAR diferentes, com mesmo caminho de pastas e nome. Ambos tem a mesma identificação.

O programa cliente do Mentor precisa gerenciar diferentes versões de pacotes. O programa é uma aplicação escrita em Java e que executa numa máquina virtual própria. Ele é capaz de executar qualquer aplicação distribuída pelo sistema de distribuição. Estas aplicações executam na mesma máquina virtual do cliente Mentor, compartilhando o mesmo ClassLoader e conseqüentemente as classes carregadas por ele. Duas aplicações distribuídas pelo Mentor podem ter como dependência duas versões diferentes de um mesmo componente. Se estas aplicações forem executadas, vários problemas podem ocorrer em função da execução de códigos de uma versão errada da mesma classe.

Uma pesquisa foi realizada com o intuito de descobrir formas de permitir que mais de uma versão de uma classe pudesse coexistir na máquina virtual onde o Mentor opera. Descobriu-se que dois trabalhos usaram uma abordagem para resolver este problema utilizando diferentes classes estendidas do ClassLoader para o carregamento de classes em memória. Os trabalhos são o container de páginas dinâmicas Jakarta Tomcat (TOMCAT, 2005) e o autor Don Schwarz (SCHWARZ, 2005).

A solução para os problemas de dependências é obtida substituindo a classe padrão ClassLoader por uma classe que seja responsável pela gerência de diferentes objetos instanciados de uma classe estendida do ClassLoader padrão do Java. Cada instância desta classe estendida terá a tarefa de carregar em memória as classes necessárias a uma das aplicações instaladas. Quando uma aplicação for executada pelo Mentor, a classe gerente criará um objeto da classe estendida e pedirá que carregue os arquivos JAR que são as dependências da aplicação. Pode ser executada mais de uma aplicação pelo Mentor, sem gerar o tipo de problema descrito, pois cada aplicação possuirá um ClassLoader particular.

3.2.2 Proposta de divulgação de software

O objetivo de usar o Mentor como ferramenta de divulgação levou a uma pesquisa sobre como esta divulgação poderia acontecer.

Os autores Rubén Prieto-Diaz e Peter Freeman defendem uma solução de classificação e busca baseada na ciência da informação (PRIETO-DIAZ, FREEMAN, 1987). O autor Prieto-Diaz precisava criar um mecanismo para incentivar a reutilização de componentes em sua empresa. Prieto-Diaz e Freeman escreveram alguns artigos sobre o assunto.

Os autores usaram a idéia de facetas para a solução. O objeto usado no problema, o componente de software, pode ser classificado por diferentes critérios. Cada critério ou faceta possui um conjunto de valores possíveis para a classificação de um objeto. Uma vez que as facetas foram definidas e todos

os objetos classificados, buscas podem ser feitas usando os mesmos critérios da classificação para obtenção de objetos.

A divulgação de software é um problema parecido com o problema de Prieto-Diaz e Freeman. Ambos podem ser resolvidos através da classificação e busca de objetos. A principal diferença entre os dois são os critérios usados na classificação, pois no primeiro caso o usuário do mecanismo é um desenvolvedor, e no outro, um usuário de software.

A classificação de objetos deve restringir somente a critérios relevantes ao usuário na busca. A ordem de prioridade entre critérios também é relevante. Com o objetivo de propor uma classificação de software para o Mentor, outra pesquisa foi realizada. Chegou-se à conclusão que sítios na Internet especializados em distribuição de arquivos de software, como o SourceForge, já usam critérios bastante consolidados por seus usuários.

A idéia é aproveitar os critérios relevantes destes sítios e propor outros que possam completar a classificação de aplicações. Todo desenvolvedor que utilizar o sistema de distribuição deve classificar sua aplicação nos critérios definidos pelo Mentor. A princípio, toda aplicação distribuída pelo Mentor será visível a seus usuários. A divulgação de software será mais eficiente à medida que os usuários definam uma ou mais queries de busca que limitarão as aplicações as quais desejam ter conhecimento de sua existência.

3. 3 As três classificações de software: A, B e C

Hoje existem computadores com especificações de hardware diferentes, desde celulares e assistentes pessoais digitais até microcomputadores de mesa. Uma observação conservadora conclui que nem todos os computadores podem executar as aplicações existentes no mercado.

Apesar da grande disponibilidade de computadores portáteis e cada vez mais usuários terem acesso a este recurso computacional, a gama de aplicações disponíveis para este tipo de hardware será sempre dependente de suas

limitações técnicas. Máquinas com menor poder de processamento e memória não permitirão que seus usuários tenham acesso a determinados tipos de serviços.

A proposta do Mentor é permitir a execução de aplicações em qualquer categoria de hardware inclusive os mais limitados, desde que as tarefas que demandam mais recursos sejam executadas por outras máquinas colaboradoras. Qualquer máquina deve ter acesso a aplicações disponibilizadas pelo sistema de distribuição, independente da plataforma do computador e de suas limitações.

Uma aplicação será instalada em um computador baseando-se em requisitos de hardware que indicarão a qual categoria de hardware ela pertence. A aplicação funcionará em cada máquina usando um regime específico, dependendo de suas limitações.

O Mentor define três classificações de hardware possíveis, A, B e C. Fornecedores podem desenvolver aplicações para serem executadas nas três categorias disponíveis. Apesar de não ser condição do sistema de distribuição, a implementação nas três categorias permitirá que qualquer usuário tenha acesso à aplicação, independente do hardware utilizado por ele.

A classificação B compreende a categoria de máquinas capazes de executar uma aplicação sem a ajuda de outras máquinas. No momento da instalação, todos os arquivos necessários à execução são guardados na máquina do cliente, incluindo bibliotecas de software e outros arquivos. Em breve será explicado como uma máquina de classificação B pode aumentar a disponibilidade de determinado software servindo a máquinas de classificação A e C.

A classificação A compreende a categoria de máquinas que não podem executar aplicações por não possuírem os requisitos mínimos de hardware exigidos pelas mesmas. Máquinas com hardware muito limitado precisam da ajuda de outras máquinas que farão grande parte do trabalho.

Uma aplicação na classificação A se divide em dois componentes. O primeiro componente é uma casca leve que executa do lado do cliente, com complexidade suficiente apenas para gerar a interface do programa que é mostrada ao usuário. O outro componente é parte predominante da aplicação e executa numa máquina suficiente para a tarefa requisitada. A comunicação entre os dois lados ocorre através do envio de comandos e respostas, no sentido do primeiro componente para o segundo.

A classificação C compreende a categoria de máquinas que podem até ajudar na execução da aplicação que está requisitando, mas que não são capazes de executá-la sozinha. Uma vez que este regime é escolhido, existe uma separação das tarefas de processamento necessárias ao funcionamento da aplicação, entre o cliente e qualquer máquina colaboradora.

A máquina C pode ter acesso a uma aplicação apenas com a presença de um número suficiente de máquinas C na rede, cada uma se responsabilizará por um pedaço da aplicação. No regime C, as responsabilidades são quebradas em pedaços menores para permitir que máquinas de menor capacidade e classificadas como C possam colaborar com outras, aumentando a disponibilidade de atendimento das aplicações distribuídas na rede do Mentor. É importante enfatizar que máquinas nas classificações A e C nunca teriam acesso a determinadas aplicações, característica que só é possível através do Mentor.

O Mentor é um subprojeto pertencente ao projeto Labase (LABASE, 2005). O Labase é um esforço conjunto de diferentes pesquisadores cujo intuito é promover a automação em sistemas de informação. O Mentor distribui aplicações no regime C, mas não estuda a separação efetiva das tarefas necessárias à execução de uma aplicação no regime C. Esta separação permitiria que fornecedores não precisassem lidar diretamente com as três classificações no código da aplicação a ser desenvolvida. Esta separação deve ser transparente ao desenvolvedor e é objeto de estudo de outros subprojetos pertencentes ao Labase, constituindo um trabalho posterior ao Mentor.

O subprojeto Hercules do Labase propõe uma separação no nível de modelo (PAIS, 2004). Aquele estudo criou uma separação de classes de uma aplicação, apelidado de MVC recursivo, onde o componente vista é uma outra tríade MVC responsável pela implementação dos casos de uso necessários à aplicação. Esta separação em nível de classes possibilitaria que parte dessas classes pudessem ser executadas em outras máquinas. O desenvolvimento futuro do Hércules é um caminho de estudo importante para permitir que fornecedores desenvolvam aplicações sem levar em conta a existência de classificações de hardware.

O subprojeto Hydra (LABASE, 2005) do Labase pretende estudar formas de migrar tarefas de processamento de uma máquina cliente para outras máquinas colaboradoras. O Hydra lida com questões muito complexas para permitir este comportamento e é um projeto de estudo a parte.

4 A ARQUITETURA DO MENTOR

4.1 Introdução

A arquitetura do Mentor foi pensada com o objetivo de suportar a distribuição e utilização de aplicações nas três classificações de software propostas, A, B e C. Apesar da implementação existente do Mentor ainda não tratar de questões como a resolução de dependências entre pacotes e a utilização de facetas para a divulgação, o projeto da arquitetura considerou estas questões, o que permitirá que tais mudanças sejam facilmente agregadas ao Mentor futuramente.

Um usuário que deseja ter acesso a aplicações de outros usuários ligados à rede do Mentor precisa ter um programa cliente instalado em sua máquina. O cliente foi escrito em linguagem Java. Uma vez que o cliente é inicializado, uma interface é mostrada ao usuário.

Alguns procedimentos precisam ser executados antes que o usuário possa efetivamente ter acesso a qualquer aplicação disponibilizada pelo Mentor. O mais importante deles é a inicialização de um serviço que representa a camada de comunicação, indispensável para a participação da máquina na rede do Mentor. Esta camada de comunicação é chamada de GoP2P.

Uma vez que o serviço inicia com sucesso, a máquina ou ponto pode se comunicar com outros pontos, tanto enviando ações para serem executadas por outros, quanto executando ações recebidas de outros pontos. A máquina pode funcionar como cliente e servidor em relação a outras máquinas, e por esta razão o Mentor pode ser classificado como um sistema que segue o modelo ponto-a-ponto.

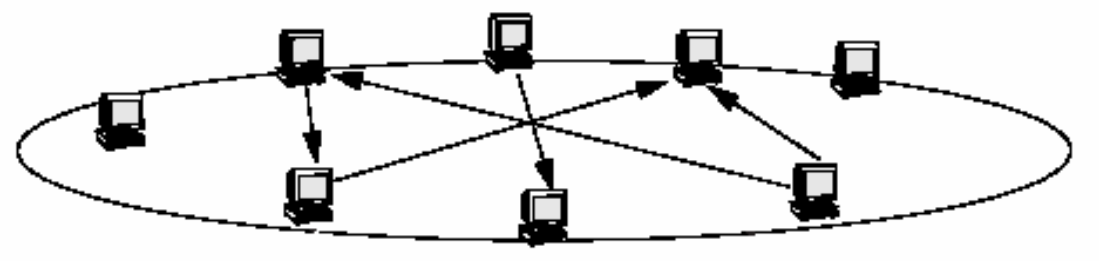

Figura 4.1: Máquinas de uma rede ponto-a-ponto onde não existem clientes e servidores predefinidos

Uma vez que o serviço de comunicação está operacional, ele automaticamente tenta descobrir quais máquinas estão conectadas naquele momento através de um código identificador que representa a rede ou grupo específico de máquinas. Máquinas precisam conhecer o identificador de um grupo para poder participar dele. Enquanto o cliente do Mentor está executando em uma máquina, outras máquinas podem sair e entrar no grupo, cabendo ao serviço a tarefa de atualizar a listagem de pontos que estão funcionando.

Uma vez que a inicialização do cliente ocorre, o usuário pode iniciar um programa que já havia sido instalado em sua máquina ou então requisitar a execução de qualquer outro programa da rede.

Toda vez que um ponto é descoberto pelo serviço de comunicação de um outro ponto, a camada de controle é utilizada para verificar quais aplicações estão sendo disponibilizadas pelo novo ponto. A camada de controle será explicada detalhadamente em breve.

Todo ponto que carregou localmente a instalação do programa passa a ser um repositório possível para instalação da mesma aplicação em outras máquinas. A máquina que faz o papel de repositório compartilha seu espaço em disco para que outras possam instalar estas aplicações no futuro.

Uma vez que uma ação requisitando as aplicações instaladas em determinado ponto chega ao seu destino, a camada de controle do novo ponto repassa estas informações à camada de gerência de aplicações, que é responsável por

saber quais aplicações estão disponíveis na rede. A camada de gerência de aplicações será explicada posteriormente.

A lista de aplicações disponíveis é finalmente mostrada ao usuário, através de uma interface gráfica, como um menu de opções ou uma tabela. O usuário pode escolher executar um programa. Caso o programa esteja instalado, ainda é possível que outra versão mais recente esteja disponível na rede. Independente se existe ou não um programa instalado, o cliente do Mentor perguntará ao usuário se ele deseja instalar a nova versão. Em caso positivo, a camada de gerência de aplicações que é responsável também pela execução das aplicações descobre um ponto colaborador que pode transmitir a versão da aplicação desejada. Uma vez que existe algum colaborador, o cliente requisita uma ação para este ponto. O objetivo desta ação é carregar localmente a instalação da aplicação. Cada aplicação é instalada numa pasta separada na máquina do cliente.

Uma vez que a aplicação está instalada, o usuário poderá utilizá-la. Dependendo da classificação de hardware da máquina do usuário, ela pode depender da colaboração de outras máquinas. A forma como a colaboração ocorre será explicada em tópicos posteriores.

4.2 Camada de Comunicação GoP2P

A camada de comunicação GoP2P foi desenvolvida com o objetivo de facilitar a construção de aplicações distribuídas seguindo o modelo ponto-a-ponto. A camada é composta por um serviço, que deve ser inicializado em cada ponto que deseja se comunicar com outros, e por um conjunto de classes e interfaces que criam uma abstração sobre como funciona efetivamente a comunicação entre os pontos. A abstração permite a rápida construção de aplicações escritas em linguagem Java que se comunicam numa rede ponto-a-ponto.

Numa rede ponto-a-ponto, cada ponto pode desempenhar o papel de cliente e de servidor dependendo da ocasião. Quando um ponto deseja que determinada ação seja executada em um ponto qualquer, ele age como cliente

e o segundo ponto age como seu servidor. A ação é executada remotamente no ponto servidor, e a resposta é devolvida para o ponto cliente, que pode tomar medidas em função da resposta recebida.

A execução de ações remotas é possível desde que elas sejam previamente acordadas entre os pontos envolvidos, ou melhor, as bibliotecas que possuem tais ações precisam ser carregadas em memória em ambos os pontos.

A arquitetura do Mentor se baseia fortemente na camada de comunicação, pois ela possibilita que qualquer comportamento que envolva a interação de dois pontos seja implementado com facilidade. Um dos requisitos fundamentais da arquitetura era que ela fosse bastante extensível, porém sem exigir grandes custos de desenvolvimento. A extensibilidade de baixo custo ocorre na camada de controle da arquitetura, que pode receber novos comportamentos sem precisar de grandes esforços.

Além de permitir a extensibilidade da camada de controle, a camada GoP2P permite que mais de um ponto possa contribuir para a realização de uma tarefa particular que necessite de algum processamento paralelo. A listagem de aplicações disponibilizada pela camada de controle de cada máquina é um exemplo de aplicação que envolve processamento em diferentes pontos entram em operação.

A camada GoP2P também foi projetada com o objetivo de facilitar o descobrimento de outros pontos. Ela provê um jeito simples para um ponto obter referências a outros pontos na rede. Não existe necessidade prévia de saber endereços fixos de cada máquina que irá participar da rede, até mesmo porque máquinas entram e saem da rede a todo momento. Os endereços das máquinas são dinâmicos e variam dependendo da ocasião. Tecnologias que dependem da utilização de IPs estáticos foram descartadas para a implementação da camada de comunicação. A tecnologia Jxta foi escolhida para a implementação da camada, pois sua abordagem era a que mais se assemelhava ao propósito da camada de comunicação.

4.3 Camada de Controle ou Camada Servidora

A camada de controle ou camada servidora é constituída pelas ações comuns a todos os pontos e que podem ser executadas por um ponto a pedido de outros conectados na mesma rede. Tais ações são implementadas utilizando a camada de comunicação.

As ações recebidas por um ponto através da camada de comunicação representam todo o comportamento que o ponto pode desempenhar como servidor. Todas as ações que um ponto pode executar fazem parte da sua camada de controle. A camada de controle se divide em dois conjuntos de ações: as ações fundamentais e as não fundamentais.

As ações fundamentais são indispensáveis ao funcionamento do sistema de distribuição. Chegou-se à conclusão que determinadas ações são necessárias para suportar o funcionamento do sistema. Para possibilitar a implementação de um sistema de distribuição no modelo ponto-a-ponto, todo ponto deve ser capaz de:

- Indicar quais aplicações disponibiliza e quais versões possui e;

- Transmitir arquivos de instalação de uma aplicação quando requisitados por outros pontos.

Além das ações fundamentais, outras ações podem ser executadas por um ponto. Estas ações constituem o conjunto das ações não fundamentais. Máquinas de classificação A e C precisam da colaboração de outras máquinas para poder utilizar uma aplicação. Estas máquinas fazem uso de ações não fundamentais e que são específicas da aplicação requisitada. Uma vez que colaboradores se prontificaram a ajudá-las, estas máquinas A e C pedem que tais ações sejam executadas por eles.

4.4 Camada de Gerência de Aplicações

A camada de gerência de aplicações é responsável por todas as atividades que dão suporte à utilização de aplicações dentro do Mentor. A camada é requisitada toda vez que uma aplicação precisa ser instalada, executada, atualizada e desinstalada.

A camada é responsável por verificar a disponibilidade de recursos de outras máquinas promovendo a colaboração na distribuição e utilização de aplicações. A camada procura por pontos que possam transmitir pacotes de instalação requisitados ou que possam executar pedidos de processamento, caso o próprio ponto possua limitações de hardware que impedem que a aplicação seja executada de maneira independente.

Se um usuário desejar atualizar um software ou instalá-lo pela primeira vez, a camada é requisitada para desinstalar a versão antiga quando existir alguma, e instalar a nova versão. Quando um ponto requisita uma aplicação a outro ponto, um pacote de instalação específico será transmitido para o requisitante, dependendo de sua classificação de hardware.

Após a transmissão do pacote, é necessário iniciar o processo de instalação da aplicação. O processo de instalação disponibilizado pelo Mentor apenas suporta a simples descompactação de um pacote gerado pelo algoritmo ZIP. Geralmente, uma aplicação escrita em Java não precisa que arquivos sejam instalados em outros diretórios, como programas que usam DLLs, por exemplo.

A opção feita foi de implementar um mecanismo simples de instalação, tentando manter uma camada flexível que possa futuramente disponibilizar mecanismos de instalação mais complexos. Existem outras frentes de pesquisa que tentam representar de forma descritiva o processo de instalação de aplicações (HOFF, PARTOVI, THAI, 1997) (DTMF, 1994), mas esta questão não se encontra no escopo deste trabalho.

Máquinas que já possuam alguma aplicação instalada podem servir como repositório para outras máquinas que desejam instalar ou atualizar aplicações, uma vez que os pacotes transmitidos não são apagados após a instalação.

A camada de gerência de aplicações deve também ser capaz de inicializar uma aplicação quando for requisitada pelo usuário. Além das atividades de atualização e inicialização de aplicações, a camada também tem como responsabilidade manter registros de todas as aplicações instaladas no cliente, assim como todas as aplicações disponibilizadas por outros pontos. Cada registro mantem referência para o ponto que disponibiliza a aplicação em questão e sua versão. Toda vez que se descobre que um ponto “entrou” ou “saiu” da rede, a camada de gerência de aplicações deve atualizar a listagem de aplicações disponíveis ao usuário.

4.5 Interface gráfica

A interface gráfica é necessária para facilitar a interação do usuário com o sistema de distribuição, assim como a utilização efetiva de aplicações. Ela deve permitir ao usuário um meio de visualizar quais aplicações estão disponíveis respeitando os critérios de busca estabelecidos por ele e relativos à divulgação de software (queries de busca).

Independentemente do software se encontrar instalado ou não na máquina do usuário, o mesmo pode requisitar a utilização de qualquer aplicação, cabendo à interface gráfica transferir este pedido à camada de gerência de aplicações. A camada de gerência de aplicações cuidará para que o usuário tenha acesso à aplicação desejada.

A interface foi implementada em linguagem Java usando a biblioteca de componentes gráficos Swing (SWING, 2005). A implementação da interface usando Java segue o objetivo da proposta do Mentor de permitir que qualquer usuário tenha acesso ao sistema de distribuição e às aplicações distribuídas.

4.6 Espaço de tuplas

O espaço de tuplas ou espaço enuplário (FREEMAN, HUPFER, ARNOLD, 1999) é usado na arquitetura do Mentor para possibilitar o compartilhamento de recursos entre máquinas como processamento e banda passante. Este compartilhamento de recursos pode ocorrer tanto nas atividades de distribuição quanto na execução efetiva de aplicações. Máquinas podem colaborar com outras quando se julgarem livres para isso. A forma como este compartilhamento ocorre é explicado a seguir.

O espaço de tuplas cria uma área virtual de memória compartilhada entre diferentes processos ou máquinas permitindo que elas possam enxergar tuplas existentes no espaço. Uma tupla representa um grupo de informações, na forma de um conjunto de pares de chaves e valores. As máquinas podem fazer operações de adição, leitura ou remoção de tuplas presentes no espaço.

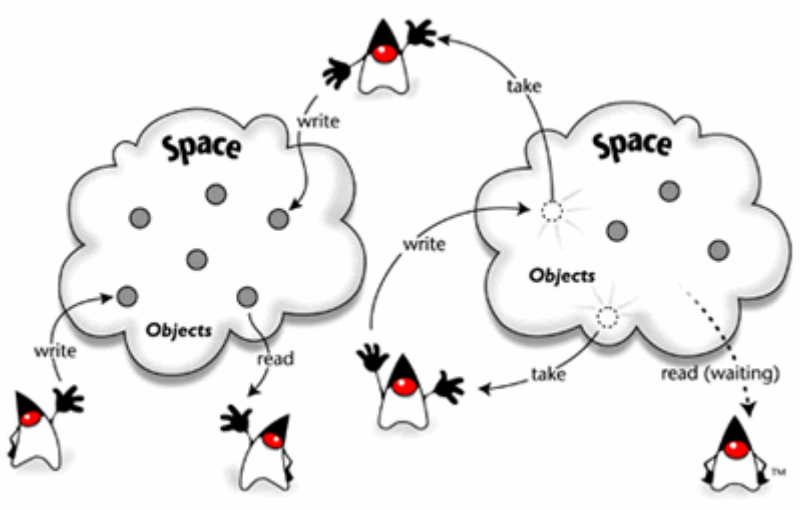

Figura 4.2: Operações possíveis realizadas por máquinas ou processos num espaço enuplário (FREEMAN, HUPFER, ARNOLD, 1999)

Máquinas que se comunicam através da camada de comunicação são consumidores e produtores de tuplas deste espaço comum. Cada tupla presente no espaço representa uma requisição ou uma resposta de requisição feita por uma máquina.

Máquinas podem precisar de algum recurso eventualmente. A princípio uma máquina desconhece quais colaboradores possuem o recurso de que necessita e qual deles o ajudará efetivamente.

O cliente ou solicitador que deseja fazer uma requisição a uma máquina colaboradora e até então desconhecida por ele cria uma tupla com os dados descrevendo esta requisição e coloca a tupla no espaço. O solicitador aguardará por uma outra tupla que será a resposta colocada no espaço por outra máquina que resolveu atendê-lo.

Todos as máquinas podem colaborar. Colaboradores usam uma heurística particular para determinar se estão livres para atender requisições de outros. Esta heurística pode usar diferentes critérios no momento de avaliar se a máquina está ocupada ou não para atender requisições. Máquinas com configurações de hardware diferentes podem e devem usar heurísticas diferentes, a avaliação de sua disponibilidade depende dos tipos de recursos envolvidos na requisição em questão.

O uso de heurísticas permite que colaboradores escolham o momento de atender requisições, ao invés de recebê-las sem pedir e serem forçados a respondê-las. Colaboradores agem como servidores que escolhem o momento de atender seus clientes ou solicitadores.

Em sistemas com múltiplos servidores e que recebem um grande número de acessos, o tempo de resposta ou tempo de atendimento de um cliente é geralmente obtido através da soma de dois tempos: o tempo de espera na fila de atendimento e o tempo de processamento de sua requisição. O tempo de resposta médio de um sistema é a média dos tempos de respostas de cada requisição feita a cada servidor. O tempo de resposta médio é um dos requisitos de sistema mais importantes exigidos por usuários.

O tempo de resposta médio de um sistema pode ser muito alto em determinados sistemas porque alguns servidores são sorteados mais vezes para atender requisições do que outros, aumentando desnecessariamente os

tempos de espera de clientes nas máquinas sobrecarregadas e por conseqüência o tempo médio. É possível fazer uma distribuição mais natural de atendimento usando o espaço de tuplas e heurísticas.

Uma vez que a heurística usada pelo colaborador indica que o mesmo está livre para atendimento, ele observa o espaço de tuplas para ver se existe alguma requisição que possa atender. Caso exista alguma tupla de requisição, ele retira a tupla do espaço. A retirada da tupla do espaço indica que o colaborador está preparado para atender o pedido do cliente, e impede que outros colaboradores livres atendam ao mesmo solicitador, ao mesmo tempo.

Clientes que criam tuplas de requisição observam o espaço procurando por alguma tupla que possa ser a resposta de sua requisição. Esta resposta apenas se limita a dizer se algum colaborador está disponível, nenhum processamento ou recurso será utilizado até este momento. Uma vez que a tupla de resposta foi colocada no espaço pelo colaborador, o cliente retira a tupla deste espaço e descobre qual colaborador irá lhe atender.

A tupla de resposta traz um código, utilizado pelo colaborador como seu identificador único na rede de comunicação do Mentor. O identificador é usado pelo solicitador para obter referência ao colaborador através da camada de comunicação. Uma vez que esta referência é obtida, ações podem ser executadas pelo colaborador a pedido do solicitador, através da camada de comunicação.

Os recursos são compartilhados entre máquinas de maneira eficiente, pois cada colaborador escolhe a hora de atender seus solicitadores. A chance de um colaborador ficar superocupado é menor em comparação a um sistema com múltiplos servidores que usa sorteio na escolha do servidor para o atendimento. Antes que a máquina fique sobrecarregada, a heurística dirá a ela para não atender mais ninguém.

Máquinas com classificação A só colaboram com outras servindo como repositório de pacotes usados nas instalações de suas aplicações.

Uma máquina é escolhida para ser o servidor deste espaço de tuplas e mantê-lo funcionando. A máquina deve executar um programa implementado em Java com este objetivo.

4.7 A distribuição e utilização de aplicações de diferentes classificações

A proposta para promover um melhor compartilhamento de recursos entre pontos se baseia em usar o espaço de tuplas e heurísticas para que cada colaborador escolha a hora que deseja ajudar outros pontos. No momento em que um usuário decide usar uma aplicação dentro do Mentor, diferentes questões devem ser analisadas para escolher em que regime a aplicação rodará na máquina do cliente. Uma classificação mais compatível com as limitações da máquina e de seu cliente deve ser escolhida. A partir dai o ponto sempre executará a aplicação neste regime.

Uma aplicação utilizada por um cliente usa um roteiro de distribuição e utilização. Cada máquina com classificação A, B ou C utilizará um roteiro específico. Todos os roteiros usam o espaço de tuplas e heurísticas para promover o compartilhamento de recursos.

Roteiro para classificação A

O cliente nunca utilizou uma aplicação e decide requisitá-la.

Ele coloca uma tupla no espaço que representa o seu interesse na aplicação. O cliente fica bloqueado esperando que alguma resposta seja colocada no espaço.

Uma máquina colaboradora descobre que está livre e decide olhar o espaço. Ela possui um conjunto de aplicações que pode disponibilizar para outros pontos, e percebe uma tupla no espaço que é uma requisição de uma aplicação que possui.

O colaborador decide atender o solicitador retirando a tupla do espaço e colocando outra no lugar, indicando ao solicitador que irá atendê-lo.

O solicitador deixa de estar bloqueado ao perceber a existência de uma resposta. Ele retira a tupla do espaço, obtém o identificador do colaborador e pede que uma ação seja executada pelo colaborador, através da camada de comunicação.

A ação é um pedido de transferência de um arquivo para o solicitador. O arquivo é um pacote para instalação da aplicação no regime apropriado para a máquina. Como a aplicação rodará no regime A, apenas serão instaladas na máquina do solicitador classes de interface com comandos de comunicação. Durante a utilização da aplicação, nenhum processamento pesado será feito do lado do cliente, apenas processamento relativo à atualização da interface são executados.

Toda interação do usuário com a interface que necessitar da execução de alguma regra de negócio deverá ocorrer num ponto colaborador. Desta forma, cada interação do usuário com a interface implicará na busca de um colaborador disponível e na execução de uma ação em algum colaborador.

A escolha de um colaborador ou de diferentes colaboradores pode ser estática ou dinâmica. No modelo estático, o primeiro colaborador encontrado será usado em futuras interações do usuário. No modelo dinâmico, sempre que ocorrer uma interação, uma nova busca por um colaborador deve ocorrer.

O modelo dinâmico permite uma distribuição de carga mais eficiente, porém tem implementação difícil. A dificuldade é implementar a migração do estado do cliente entre colaboradores que atenderem o mesmo cliente em diferentes momentos. O subprojeto Pantheon, do Labase, endereça este tipo de questão.

Caso não existam colaboradores livres e o tempo de espera definido pelo solicitador para aguardar por respostas no espaço de tuplas se esgotar, o usuário será alertado da indisponibilidade de colaboradores.

Roteiro para classificação B

O roteiro para a classificação B é parecido com o roteiro da classificação A. A semelhança com o outro roteiro termina no momento em que o pacote de instalação é trazido para o solicitador. Decidiu-se que a máquina é suficientemente rápida para executar a aplicação sem depender de ninguém.

A aplicação é instalada por completo no cliente, e todos os recursos necessários à execução da aplicação estarão presentes na máquina. A única colaboração possível entre pontos ocorre na transmissão do arquivo de instalação, quando o colaborador utiliza sua banda de transmissão para esta tarefa. Uma vez que a aplicação é instalada no cliente, ela será executada localmente.

Roteiro para classificação C

O roteiro para a classificação C é igual ao roteiro da classificação A, pois ambos permitem que máquinas de hardware limitado tenham acesso a aplicações mais pesadas. A diferença principal entre os dois regimes é que no regime C, parte da responsabilidade do processamento é do solicitador, ao invés dele delegar todo o processamento para o colaborador.

No regime C, qualquer processamento da aplicação pode ser dividido em três partes, executadas de forma seqüencial: uma lógica que é executada no solicitador antes da execução da ação no colaborador, a lógica da ação que roda no colaborador, e a lógica final que roda no solicitador dependente dos dados gerados pela ação executada no colaborador.

Além das classes de interface, algumas classes relacionadas à regra de negócio da aplicação serão instaladas na máquina do solicitador.

4.8 A colaboração estendida para tratar a indisponibilidade de execução

Máquinas que possuem uma aplicação podem colaborar transmitindo o pacote de instalação da mesma e processando ações para outras máquinas com classificações A e C.

Apesar desta colaboração, há momentos em que uma aplicação se torna tão requisitada que pedidos de execução de clientes são colocados no espaço de tuplas e permanecem por lá por não haverem colaboradores disponíveis no momento.

Normalmente, todo ponto possui um ciclo para colaboração. Para cada aplicação instalada localmente, o ponto observa o espaço de tuplas procurando por algum outro ponto que deseja a aplicação, seja transferindo ou executando a aplicação para este ponto. O ciclo se repete após se colaborar com outro ponto que requisita transferência ou execução da última aplicação instalada.

Com o objetivo de ajudar a resolver a indisponibilidade de execução, o Mentor define que pontos podem colaborar mesmo que não possuam uma aplicação particular. No final de um ciclo de colaboração é dada uma chance a um pedido de execução de uma aplicação qualquer.

Se no final de um ciclo existir no espaço de tuplas um pedido de execução de uma aplicação que um ponto não possui, ele tentará aumentar a disponibilidade de execução desta aplicação transmitindo e instalando-a localmente. Dependendo da classificação do hardware da máquina em questão, o pacote de instalação apropriado será escolhido. Máquinas com classificação de hardware B e C podem ajudar aumentando a disponibilidade de execução de aplicações que a principio não possuem. O Mentor é uma prova de conceito de que é possível a distribuição e utilização de aplicações pesadas independente das limitações de hardware das máquinas envolvidas.

A proposta de colaboração estendida feita por máquinas C dentro do Mentor é limitada, pois atualmente o Mentor só dá suporte a separação de uma

aplicação em dois pacotes de instalação classe C apenas, onde um ponto C será o cliente e outro o servidor. Máquinas com classificação C podem instalar o módulo servidor para ajudar na disponibilidade de execução. Se a máquina já possui o módulo cliente instalado, não poderá colaborar na execução, pois já estaria com sua capacidade esgotada.

Uma evolução natural da colaboração estendida de pontos de classificação C é estudar como uma aplicação pode ser quebrada em um número maior de módulos, não sendo parte do escopo de estudo deste trabalho. Cada parte do sistema seria responsabilidade de um ponto C.

A quebra de uma aplicação em vários módulos aumentaria a sua disponibilidade usufruindo-se da existência de máquinas mais limitadas com potencial não utilizado. No modelo C proposto pelo Mentor, mesmo a metade de um aplicativo pode ser grande demais para executar numa máquina limitada.

5 A IMPLEMENTAÇÃO DO MENTOR

5. 1 Camada de Comunicação GoP2P

Alguns padrões de projeto foram usados na implementação da camada de comunicação GoP2P, como o Observer, o Template Method e o Singleton (GAMMA, HELM, JOHNSON, VLISSIDES, 1995). A camada de comunicação GoP2P estabelece um contrato de comunicação entre pontos clientes e pontos servidores ligados em rede. Este contrato estabelece uma forma como os desenvolvedores devem implementar objetos que representarão as ações executadas remotamente. A interface RemoteAction cria este contrato ao forçar a implementação de seus três métodos. Toda ação remota deve implementar esta interface.

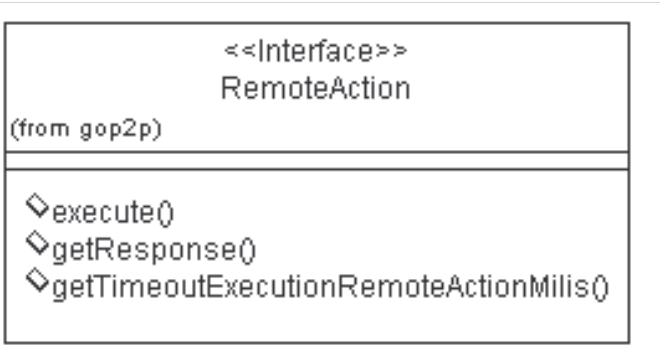

Figura 5.1: Contrato definido pela interface RemoteAction

Uma ação pode ser vista como um procedimento remoto. Todo procedimento é formado por parâmetros de entrada, o corpo do procedimento com o código a ser executado e os parâmetros de saída. Dependendo dos parâmetros de entrada, haverá uma saída diferente.

Um objeto descendente de RemoteAction possui comportamentos importantes, implementados por seus métodos. O método execute() possui o código a ser

executado pelo ponto servidor. O método getResponse() retorna a resposta gerada pela chamada de execute(), sendo chamado após o objeto ir ao ponto servidor, ser executado por ele e voltar ao ponto inicial.

A execução remota da ação só é possível se a transmissão deste objeto ocorrer sem problemas. A camada GoP2P utiliza a serialização de objetos da linguagem Java para transmitir ações entre camadas GoP2P operando em pontos diferentes. Sempre que um objeto precisa ser transmitido entre dois pontos, a camada de um dos lados serializa o objeto ação, envia os dados pela rede e deserializa do outro lado, recriando o objeto com mesmo estado antes da transmissão. O estado de um objeto é representado pelos valores de suas propriedades em determinado momento.

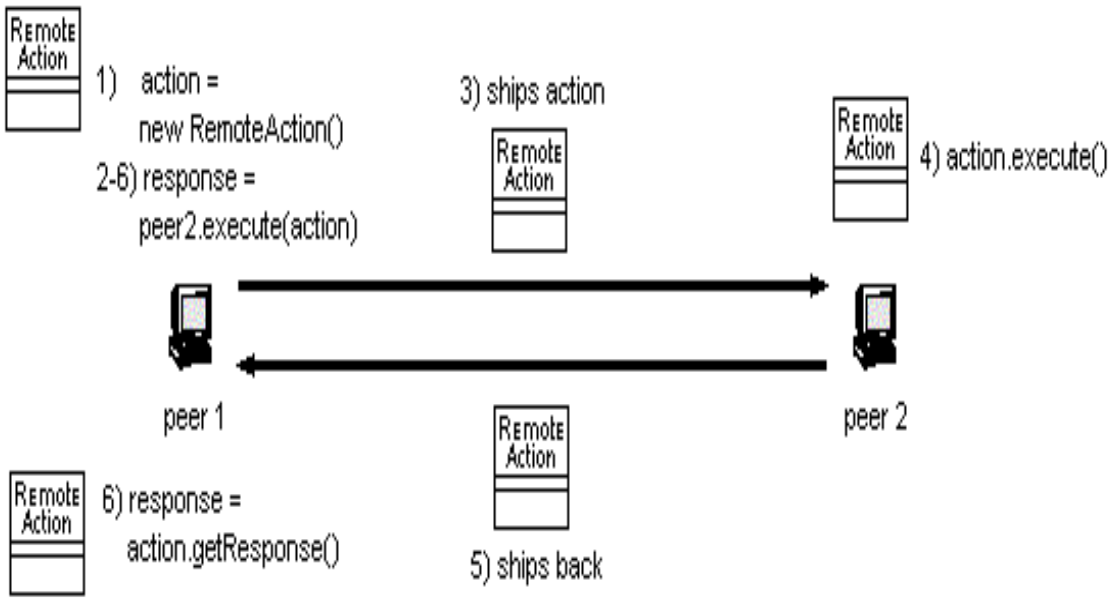

Figura 5.2: Passos na execução de uma ação remota

Um serviço foi projetado e implementado com o objetivo de obter uma abstração simples na forma como uma instância de uma aplicação qualquer consegue fazer requisições a outras instâncias. Todo ponto que deseja se comunicar com outros pontos através de ações precisa inicializar um serviço local. Este serviço é responsável pela comunicação efetiva entre os pontos,

chamado de PeerPublishingService. O serviço é uma thread escrita em linguagem Java, que precisa estar sempre em funcionamento.

O serviço PeerPublishingService permite que uma aplicação qualquer obtenha referências a pontos que rodam o mesmo serviço. Ele é responsável por avisar quando pontos entraram ou sairam da rede, além de possibilitar que pontos façam requisições a outros. O serviço PeerPublishingService é composto por quatro subserviços internos: ActionExecutionServer, PeerPublish, PeerDiscovery e PeerCleaning.

O subserviço ActionExecutionServer é responsável por atender requisições de outros pontos. Sempre que o serviço entra no ar, um endereço novo é criado no qual o ponto receberá requisições.

O subserviço PeerPublish é responsável por enviar para outros pontos, de tempos em tempos, informações que indicam que o ponto está no ar e atende a quaisquer requisições no endereço determinado pelo subserviço ActionExecutionServer.

O subserviço PeerDiscovery é responsável por descobrir quais pontos estão na rede naquele momento, lendo pacotes enviados pelo subservico PeerPublish de outros pontos.

O subserviço PeerCleaning é responsável por eliminar as referências de pontos que ficaram muito tempo sem se comunicar com o ponto onde o serviço principal executa. Este procedimento evita o fracasso na execução de ações em pontos que provavelmente estão desligados e não atenderão.

Aplicações que utilizam a camada GoP2P precisam primeiramente iniciar o serviço PeerPublishingService antes de poder executar ações em outros pontos. O objeto PeerPublishingService possui comportamentos importantes, implementados por seus métodos. Os métodos startService() e stopService() iniciam e param o serviço. O método getConnectedPeers() retorna referências para os pontos conectados no momento.

Os métodos addPeerDiscoveryListener() e removePeerDiscoveryListener() servem para avisar (ou não) ao objeto passado como parâmetro de entrada no método sobre eventos de aparecimento de novos pontos na rede ou de desaparecimento. O objeto indicado deve implementar a interface PeerDiscoveryListener para que possa receber os avisos e tomar as medidas necessárias, medidas estas presentes no corpo do método.

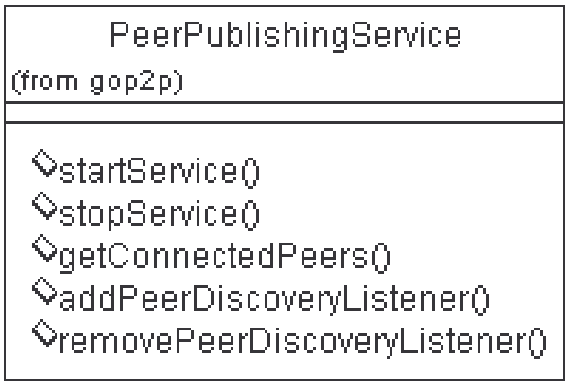

Figura 5.3: Classe PeerPublishingService

Uma vez que um ponto obtém uma referência para um ponto remoto, ele pode fazer requisições ao outro. A referência para outro ponto é representada por um objeto da classe Peer, que possui métodos para fazer requisições remotas assíncronas ou síncronas. A escolha entre os dois tipos de execução depende da natureza do processamento. Se existe boa chance da execução remota demorar um tempo razoável, a escolha da execução assíncrona é recomendável para evitar o bloqueio de processamento no lado do ponto cliente.

Na escolha da execução assíncrona, um objeto deve ser alertado do término da execução remota, permitindo a continuação do processamento que depende da resposta da ação remota. O objeto indicado deve implementar a interface AsynchronousRemoteActionListener. A implementação do método notifyEndOfExecution() representa as medidas a serem tomadas pelo ponto após receber a resposta da ação. Os métodos que pertencem à classe Peer e

que iniciam a execução síncrona e assíncrona de ações são chamados de execute() e asynchronousExecute(), respectivamente.

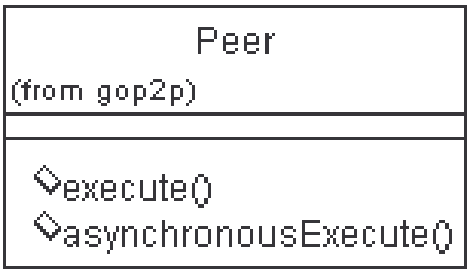

Figura 5.4: Classe Peer

O conjunto de classes e interfaces disponibilizadas pela camada GoP2P permite a construção rápida de aplicações usando distribuição ponto-a-ponto. A aplicação que é cliente da camada utiliza seus serviços, despreocupando-se com a implementação que possibilita a comunicação entre os pontos.

5. 2 Camada de Controle

A camada de controle de um ponto possui todas as ações remotas que este ponto pode executar no papel de servidor. Estas ações são implementadas usando o ferramental da camada de comunicação. Duas ações fundamentais utilizadas pelo sistema de distribuição merecem maior atenção.

A primeira ação é utilizada quando um ponto deseja obter uma listagem de aplicações disponibilizadas por um ponto particular. Esta ação é sempre chamada quando a camada de comunicação informa à aplicação cliente de que um ponto novo entrou na rede, pois neste caso o ponto novo pode possuir alguma aplicação ou versão que o ponto desconhecia. Pontos usam a camada de comunicação para publicar suas aplicações para outros.

A implementação da ação de publicação é simples. Quando a ação chega ao ponto de destino, o trabalho de obter as aplicações disponíveis é delegado à camada de gerência de aplicações.

A segunda ação essencial ao mecanismo de distribuição de software serve para iniciar o transporte efetivo de pacotes de instalação. O transporte acontece quando uma aplicação é requisitada por algum ponto da rede, uma vez que outro ponto confirmou ter a posse e disponibilidade para transmissão.

A ação é um pedido para abertura de um soquete do lado que possui o arquivo. O soquete é uma implementação disponibilizada pela tecnologia Jxta. Este soquete é um componente que encapsula um pipe Jxta, facilitando sua utilização pelos desenvolvedores. Ao contrário dos soquetes de rede tradicionais que dependem da definição do IP e da porta da máquina servidora, o soquete Jxta usa um identificador particular. Este identificador é obtido na criação de um pipe Jxta do lado da máquina que possui o arquivo.

A transferência efetiva do arquivo começa quando o cliente recebe o identificador pela resposta da ação e abre um soquete para leitura de dados. O soquete Jxta foi escolhido ao invés da implementação padrão de soquete de rede. Esta decisão foi tomada em função da capacidade da tecnologia Jxta em superar barreiras de comunicação que existem entre redes na Internet, impossibilitando a transmissão de arquivos envolvendo redes protegidas.

O Mentor usa uma implementação de um algoritmo de message digest (MD5, 2005). A implementação é usada para obter uma seqüência de caracteres que representará um arquivo de instalação de alguma das classificações. O message digest ou MD5 é um identificador único do arquivo. Este identificador é gerado a partir do conteúdo do arquivo em disco. Independente do algoritmo ser executado em máquinas diferentes, o mesmo identificador é obtido.

O identificador MD5 é usado tanto para encontrar o arquivo de instalação nas pastas onde as aplicações se encontram instaladas quanto para validar se a transferência do arquivo ocorreu com sucesso. O ponto cliente que deseja uma aplicação deve informar na ação que inicia a transferência o identificador md5 do arquivo desejado, para que seja possível encontrá-lo e após a transmissão, checar se chegou íntegro.

É possível obter uma melhoria na transmissão de aplicações com a checagem de disponibilidade e utilização de mais de um colaborador como fonte para a transmissão de arquivos. O uso de mais de uma fonte na transmissão de arquivos é uma idéia muito usada em redes ponto-a-ponto para compartilhamento de arquivos e em gerenciadores de transmissão de arquivos na Internet.

5. 3 Camada de Gerência de Aplicações

A gerência de aplicações se baseia na idéia de plugins. Um plugin nada mais é do que um subprograma que é executado dentro de outro programa principal seguindo determinadas condições. O cliente do Mentor é o programa principal neste caso. Ele possui um diretório onde se encontram todas as aplicações instaladas, cada uma delas na forma de um plugin diferente.

O uso de um mecanismo de plugins permite que aplicações particulares possam ser altamente extensíveis e customizadas. A aplicação com suporte a plugins permite que seus usuários definam quais plugins desejam instalar, situação em que se encontra o programa cliente do Mentor.

O programa cliente do Mentor pode ser instalado e funcionar normalmente sem plugin nenhum, assim como pode ser distribuído com plugins mais conhecidos e utilizados. À medida que o usuário decide usar determinadas aplicações, elas serão gerenciadas pela camada de gerência de aplicações.

Aplicações podem ser instaladas no Mentor de duas formas: através do mecanismo de distribuição do Mentor ou sendo copiadas manualmente pelo usuário para o diretório de plugins do cliente. Na última forma, estas aplicações serão disponibilizadas ao usuário após nova inicialização do programa cliente.

A biblioteca Java Plugin Framework (JPF, 2005) foi utilizada para fazer o controle dos plugins. Esta biblioteca foi desenvolvida em Java e disponibilizou um meio de descrever informações de plugins através de arquivos XML, assim como um meio de executá-los quando requisitados pelo usuário.

Qualquer aplicação pode ser instalada durante o funcionamento do cliente do Mentor, permitindo que o usuário não precise reiniciar o programa principal. Esta facilidade cria um problema, pois na linguagem Java a definição de quais bibliotecas de classes serão carregadas é geralmente feita na inicialização da máquina virtual. O carregamento das classes deste plugin não poderia ocorrer, pois as classes não foram previamente informadas ao ClassLoader padrão do Java na inicialização do programa cliente do Mentor. O JPF implementa uma solução para o problema, pois possui as descrições das bibliotecas de cada plugin, carregando as classes destas bibliotecas quando necessário.

Pontos podem colaborar com outros pontos transmitindo pacotes de instalação ou processando ordens de execução. A implementação da arquitetura usa um espaço de tuplas cuja implementação se baseou na tecnologia Jxta. Este espaço de tuplas é chamado de Jxtaspaces (JXTASPACES, 2005).

Objetos encontrados no espaço podem ser classificados pelo tipo de requisição e pelo tipo de produtor do objeto. Requisições podem ser de transmissão de aplicações ou de processamento. Objetos colocados no espaço podem ser produzidos por pontos que fazem requisições ou pelos colaboradores, quando colocam respostas no espaço para alguma requisição.

Durante o planejamento da arquitetura considerou-se que operações ao espaço deveriam ser limitadas, evitando operações desnecessárias que pudessem produzir algum overhead. Uma tupla presente no espaço pode representar um pedido ou uma resposta de algum ponto. Se o consumidor da tupla é um colaborador, a retirada da tupla implica que ninguém mais colaborará a não ser este que a retirou do espaço. Se o consumidor da tupla é um ponto que precisa de colaboração, a retirada da tupla representa que ele está ciente de que outro ponto colaborará com sua necessidade. Ambas as operações são de retirada, pois na retirada o ponto interessado obteve a informação que desejava.

No caso de colaboração estendida, um colaborador retira a requisição de execução do espaço para obter a chave da aplicação que está indisponível no

momento que a retirada ocorre. A chave é necessária para que possa instalá-la e servir ao usuário que a requisitou. O único caso em que o objeto de requisição é devolvido ao espaço acontece quando não existe disponibilidade de transmissão.

Tuplas encontradas no espaço são de dois tipos: de requisição e de resposta. Uma tupla de requisição é colocada no espaço por clientes. Ela possui quatro atributos possíveis. Todos eles são necessários para que um colaborador decida se pode atender a requisição: um dos atributos é um identificador oid de requisição, que indica ao cliente uma forma de saber qual tupla no espaço é uma resposta de um colaborador para o seu pedido de mesmo identificador; o identificador da aplicação em questão; a versão da aplicação desejada; e a classificação de hardware do cliente.

Uma tupla de resposta é colocada no espaço por colaboradores. Ela possui dois atributos importantes. Esses atributos são necessários para informar ao criador da requisição como ele fará para obter a colaboração que pediu anteriormente. O primeiro atributo é o identificador oid de requisição informado quando ele fez uma requisição ao espaço. O segundo atributo é um código que identifica o ponto colaborador na camada de comunicação GoP2P. Este código será utilizado pelo ponto para fazer um pedido direto ao ponto colaborador, através de uma ação remota passada a esta camada.

Quando o tipo de requisição feita por um ponto é de transmissão de pacote de instalação, a tupla de resposta conterá mais três atributos. Estes atributos são informações necessárias à transmissão, como nome do arquivo do pacote, tamanho e código MD5 do arquivo. O nome é usado para recriar o arquivo no ponto que o solicitou. O tamanho do arquivo é usado na transmissão do mesmo. Como a transmissão ocorre através da leitura de bytes de um soquete, o lado cliente precisa do tamanho do arquivo para saber quando parar de ler. O último atributo é o código MD5 que será usado para validar se a transmissão do pacote foi bem sucedida ou não.

Pontos podem ser clientes e colaboradores de outros. Para cada um dos papéis que um ponto pode desempenhar é criada uma thread Java que fará operações no espaço. O uso de threads é necessário para evitar que a thread principal do cliente do Mentor fique bloqueada em determinados momentos e impedir sua utilização pelo usuário.

6 ESTUDO DE CASO - A DISTRIBUIÇÃO E UTILIZAÇÃO DE UMA APLICAÇÃO

6. 1 Introdução

Um estudo de caso foi realizado mostrando que é possível promover a colaboração entre máquinas na distribuição e execução de aplicações. Este estudo de caso não se limita a provar que o Mentor possibilita esta colaboração. Seu objetivo principal é mostrar em que medida se dá esta colaboração. Uma aplicação foi especialmente desenvolvida para este estudo de caso, de forma a suportar a sua utilização em máquinas pertencentes às três classificações de hardware A, B e C propostas por este trabalho.

Elegeu-se a realização de dois experimentos para este estudo de caso, cada um servindo para simular uma situação específica de utilização do Mentor e coletar os dados necessários à observação de grandezas importantes para quantificar a colaboração entre máquinas.

O primeiro experimento se destinou a observar o comportamento da grandeza "volume total de bytes transmitidos". O experimento consistiu em observar a distribuição de uma aplicação particular à medida que máquinas que não a possuíam passaram a requisitar sua transmissão. No segundo experimento observou-se o comportamento das grandezas "tempo médio de atendimento" e "indisponibilidade", à medida que as máquinas pediam a colaboração de outras para executar uma aplicação.

6. 2 A aplicação escolhida para os experimentos

Uma aplicação bastante simples foi escolhida para o estudo de caso. Sua única funcionalidade permite o cálculo do próximo número de uma seqüência de Fibonacci.

Números de uma seqüência de Fibonacci são gerados pela fórmula:

$$F(n)= F(n-2) +F(n-1), \quad \text{onde } F(1)=1 \quad e \quad F(2)=1$$

Um número de Fibonacci é sempre a soma dos dois números anteriores da seqüência, exceto o primeiro e segundo números que são iguais a um.

Uma versão da aplicação de Fibonacci foi desenvolvida e pacotes de instalação foram gerados para as três classificações de hardware. Máquinas nas classificações A e C dependem da colaboração de outros pontos para executar o processamento da aplicação. A camada de comunicação GoP2P foi utilizada na aplicação de Fibonacci para permitir que máquinas nas classificações A e C pudessem invocar ações remotas em pontos que podem colaborar no processamento da aplicação.

O pacote usado na instalação da aplicação em máquinas A possuirá apenas classes de interface. A aplicação Fibonacci de classificação A guarda o estado da aplicação em um ponto colaborador que é escolhido para esta tarefa. O estado da aplicação é representado pelos dois últimos números gerados da seqüência. Além do estado, o colaborador será o responsável pela geração do próximo número. A geração do próximo número é uma regra de negócio apenas conhecida por pontos colaboradores.

Máquinas C possuem maior responsabilidade que máquinas A na execução de aplicações. Dessa forma, o pacote usado na instalação de C, conhecido como módulo cliente, conterá além das classes de interface, as classes para guardar o estado da aplicação na memória local. A máquina C precisa de outra máquina B ou C para a geração do próximo número.

Um outro pacote para instalação da aplicação em pontos C foi criado para permitir que máquinas nessa classificação pudessem colaborar com outras máquinas com o módulo cliente instalado. Este outro pacote é chamado de módulo servidor, e permite que pontos C possam promover a colaboração estendida comentada no tópico referente à arquitetura.

Por último, o pacote usado na instalação em máquinas B possuirá todas as classes da aplicação e, além disso, possuirá os pacotes de instalação de máquinas A e dos módulos cliente e servidor de C.

Máquinas com a aplicação B instalada podem servir a pontos A e C, pois conhecem as ações remotas de ambas. Uma máquina C com módulo servidor instalado conhece a ação remota usada pela máquina C com módulo cliente instalado e por isso pode servir a ela.

6. 3 A preparação inicial para a realização dos experimentos

Cada experimento se baseou em executar instâncias do cliente do Mentor em máquinas diferentes numa rede. Foi utilizado o laboratório do Curso de Mestrado, localizado no Núcleo de Computação Eletrônica da UFRJ, para a realização de ambos os experimentos. Até a elaboração deste documento, o laboratório apresentava uma taxa de ocupação bastante significativa. Além de ser utilizado individualmente por alunos, também era freqüentemente alocado pelos professores para ministrar aulas práticas com o auxílio de computadores. Os experimentos foram realizados em horários diferentes aos das aulas práticas e contaram com um subconjunto menor de máquinas, num total de seis, com o objetivo de não atrapalhar os demais usuários do laboratório.

A instalação de programas indispensáveis à realização dos experimentos nas máquinas foi garantida. Foram instaladas uma distribuição do sistema operacional Linux, uma versão do Windows e uma máquina virtual Java, quando não estavam presentes na máquina. Com o objetivo de facilitar a execução do experimento, o programa cliente do Mentor foi alterado para permitir sua execução livre de qualquer interação humana. Ele passou a aceitar parâmetros que indicam a forma como ele deve ser executado automaticamente. Os componentes gráficos do programa foram desabilitados, uma vez que a interação do usuário foi simulada pelo computador.

As máquinas utilizadas no experimento podem rodar em modo colaborador ou modo cliente, conforme o caso. Máquinas colaboradoras tentarão atender às

requisições dos usuários. Em modo usuário, a máquina tentará executar alguma tarefa, dependendo do experimento em questão. No primeiro experimento (teste da distribuição), todas as máquinas configuradas para o modo cliente tentarão obter uma aplicação que não possuem, requisitando seu arquivo de instalação. Uma vez que uma máquina obtém a aplicação, ela servirá de repositório para outras que ainda não a possuem. No segundo experimento (teste da execução), o cliente tentará gerar números de Fibonacci, um após o outro. Conforme mencionado, a geração de um número de Fibonacci depende dos dois números anteriores. Por este motivo um usuário só pode fazer uma requisição de cada vez, o que é o comportamento normal da aplicação.

Uma idéia inicial para realizar os experimentos era inicializá-los remotamente utilizando-se um arquivo bash no Linux, que executaria processos paralelos em máquinas diferentes via SSH. Estes processos são instâncias do Mentor configuradas por parâmetros que determinam os intervalos de tempo em que cada máquina participará do experimento. A configuração de um cenário particular e inicialização do experimento seria mais simples, bastando-se criar um único arquivo que determina quais processos seriam iniciados e seus parâmetros. Este idéia não foi levada adiante em virtude da opção pela utilização do laboratório de mestrado. O laboratório possuía algumas máquinas com sistema operacional Windows, sistema que não dá suporte imediato ao SSH. As máquinas que possuiam o sistema Linux instalado não possuíam o programa servidor de SSH configurado. Estas dificuldades levaram ao abandono do uso do SSH na inicialização dos experimentos.

6. 4 A execução e conclusões do experimento para medir a colaboração na distribuição de aplicações

O Mentor promove a colaboração entre máquinas para a instalação de aplicações. Uma máquina que possui uma aplicação serve como repositório para que outras possam obtê-la. A idéia do experimento foi observar como a grandeza “volume total de bytes transmitidos” reage conforme aumenta o número de máquinas colaboradoras que possuem uma determinada aplicação.

Nenhum modelo que representa uma possível demanda de software foi usado na execução do experimento. O experimento se baseou em representar o pior cenário possível de demanda, em que todas as máquinas ligadas na rede decidem baixar uma determidada aplicação ao mesmo tempo. O experimento servirá para mostrar como o sistema de distribuição reagirá a tal demanda.

O experimento se baseou em coletar o volume de bytes transmitidos para as máquinas que requisitaram a aplicação, separando-os em intervalos de tempo de tamanho fixo. Foram utilizados intervalos ao invés de instantes de tempo pois bytes só podem ser coletados em intervalos de tempo. A implementação do Mentor utiliza um soquete Jxta para a transmissão de arquivos entre máquinas. As coletas ocorrem em máquinas que requisitaram a aplicação. A cada iteração do loop que faz a leitura de bytes do soquete, o número de bytes lido é guardado na memória. A cada término de intervalo, são computados os bytes lidos durante o intervalo, e a soma de bytes é guardada num arquivo texto. Podem ocorrer intervalos de tempo em que nenhuma coleta é feita, logo não há contribuição da máquina no gráfico final de volume nestes intervalos.

No início do experimento, existe uma única máquina que possui o pacote necessário para a instalação da aplicação. A tarefa das máquinas restantes será obter o pacote de instalação e instalá-lo localmente. Como só uma máquina possui o pacote, a primeira máquina que começar a transferir utilizará toda a sua banda de transmissão, e as demais máquinas deverão aguardar o término da transmissão do pacote para serem atendidas. Uma vez que o primeiro arquivo foi transmitido, o número de repositórios disponíveis aumenta para dois. O pacote de instalação foi alterado para ter um tamanho bastante significativo, por volta de nove megabytes. Neste experimento a aplicação de Fibonacci não foi utilizada porque ela é muito simples, o que tornaria muito complicada a observação da grandeza volume, em conseqüência do pequeno tamanho do arquivo.

O experimento foi executado usando um intervalo fixo de cinco segundos para contabilizar as coletas. A cada término de intervalo, todos os bytes transmitidos no período são somados e contabilizados. O experimento gera quatro arquivos

texto com as coletas de cada máquina que demandou o arquivo de instalação. A função volume total de dados transmitidos é uma soma de funções, onde cada função é o volume parcial de dados transmitidos para cada uma das máquinas que solicitou o arquivo. A tabela de cada uma das funções foi extraída de cada arquivo.

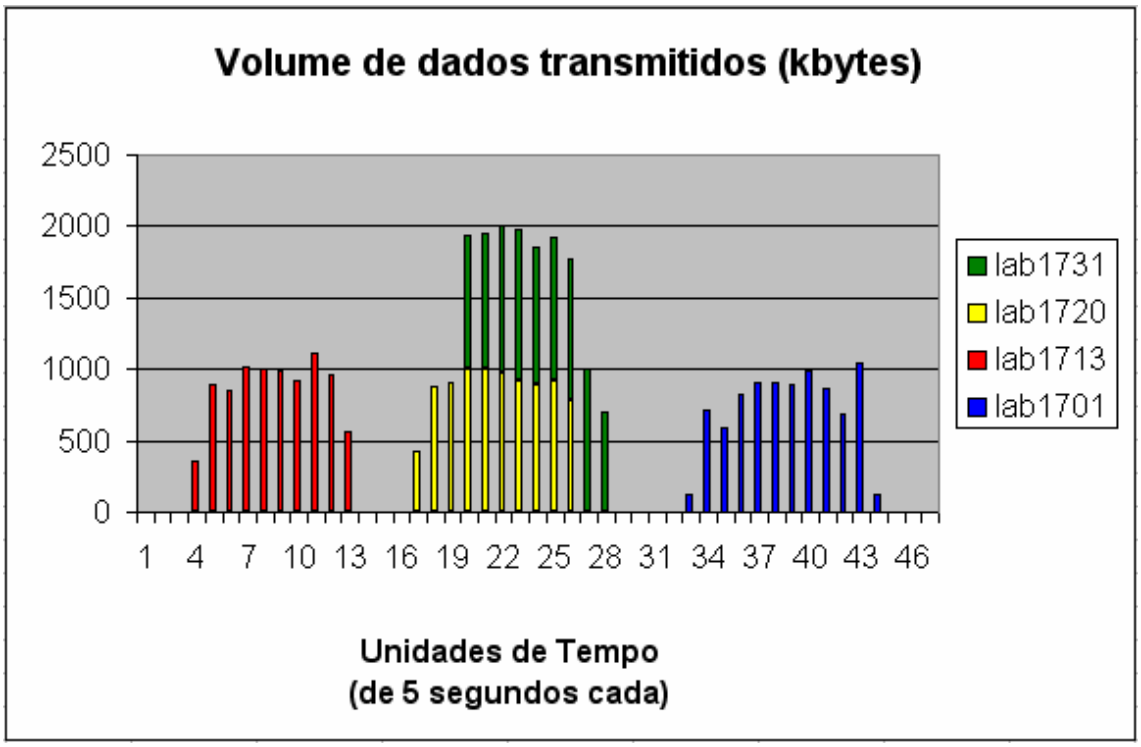

Figura 6.1: Gráfico do Volume total de bytes transmitidos

A avaliação do gráfico na Figura 8.1 mostra que existem três intervalos de tempo importantes. O primeiro intervalo ocorre entre as unidades de tempo de números quatro e treze. Neste período, a máquina "lab1713", que foi a primeira a ser atendida, recebeu o arquivo de instalação da aplicação. A partir do intervalo quatorze, existem dois repositórios com o arquivo. O segundo intervalo importante ocorre entre as unidades de tempo de números dezessete e vinte e oito. Neste intervalo as máquinas "lab1720" e "lab1731" receberam o arquivo. A partir do intervalo vinte e nove, o número de repositórios passa a ser quatro. No terceiro intervalo, entre trinta e três e quarenta e quatro, somente a

máquina "lab1701" ainda não possui o arquivo da aplicação, sendo a última a obtê-la, na unidade de tempo quarenta e quatro.

Conforme visto no gráfico, o Mentor duplica o número de repositórios disponíveis no término de dois dos três intervalos importantes. No terceiro intervalo, a demanda pela aplicação caiu, e por conseqüência, o volume total de transmissão. Uma extrapolação razoável é considerar que a execução do mesmo experimento usando mais máquinas, num total de oito ou mais demandando a aplicação, gerará um gráfico parecido ao do experimento realizado, onde a função volume total de bytes transmitidos é crescente durante o primeiro intervalo e decrescente no intervalo final, como conseqüência da queda da demanda.

Outra conclusão importante pode ser tirada. Observou-se que a transmissão de arquivos entre máquinas através do soquete Jxta é lenta. Uma conta rápida indica que uma máquina levou, aproximadamente, dez intervalos de cinco segundos para transmitir o arquivo com tamanho aproximado de nove megabytes. A velocidade de transmissão do soquete é de aproximadamente cento e oitenta kbytes por segundo. Este valor é muito baixo quando comparado à velocidade de transmissão de uma rede local, cujo valor pode atingir cerca de cem megabits por segundo em condições ideais. Por outro lado, apesar de menos eficiente, o soquete Jxta possui a vantagem de funcionar independentemente de topologias e barreiras de rede, situações em que um soquete de rede normal não poderia operar.

6. 5 A execução e conclusões do experimento para medir a colaboração na execução de aplicações

O segundo experimento simulou a utilização da aplicação de Fibonacci por um número fixo de usuários com máquinas que dependem da colaboração de terceiros. A idéia do experimento foi observar como as grandezas “tempo médio de atendimento” e “indisponibilidade” reagem na medida em que aumenta o número de colaboradores. As duas grandezas são funções do número de colaboradores em operação. O experimento se baseou em coletar

os tempos de atendimento e número de indisponibilidades de cada cliente durante diferentes intervalos de tempo.

Quando um cliente faz uma requisição de execução e o tempo limite para encontrar um colaborador disponível se esgota, ocorre um fracasso de acesso. A indisponibilidade de uma aplicação é definida como a razão entre a soma de fracassos de cada cliente e a soma de tentativas de acesso de cada um, expressa na forma de uma porcentagem.

No início do experimento, existe um único colaborador disponível com a aplicação classe C instalada e três clientes com a aplicação classe A. O número de clientes é fixo para que a carga de acessos em cada intervalo seja semelhante durante todo o período de realização do experimento. A primeira conclusão pode ser tirada antes da realização do experimento através da análise da aplicação de Fibonacci. A aplicação de Fibonacci tem processamento simples e por essa razão considera-se que o tempo de processamento de cada requisição é praticamente zero. O tempo médio de atendimento obtido no experimento para a aplicação Fibonacci é o tempo mínimo de atendimento assumido pelo Mentor para qualquer aplicação. Qualquer aplicação mais complexa, executada nas mesmas condições, deve usar este limite inferior como referência para estimar o tempo médio de atendimento. A aplicação de Fibonacci também ajuda na estimativa da indisponibilidade de outras aplicações executadas nas mesmas condições, pois define um limite inferior para a indisponibilidade.

Os arquivos de coleta gerados foram utilizados para montar tabelas e conseqüentemente os gráficos das grandezas observadas. A análise dos gráficos mostra que as duas grandezas não sofreram alteração significativa quando mais de um colaborador se encontrava disponível.

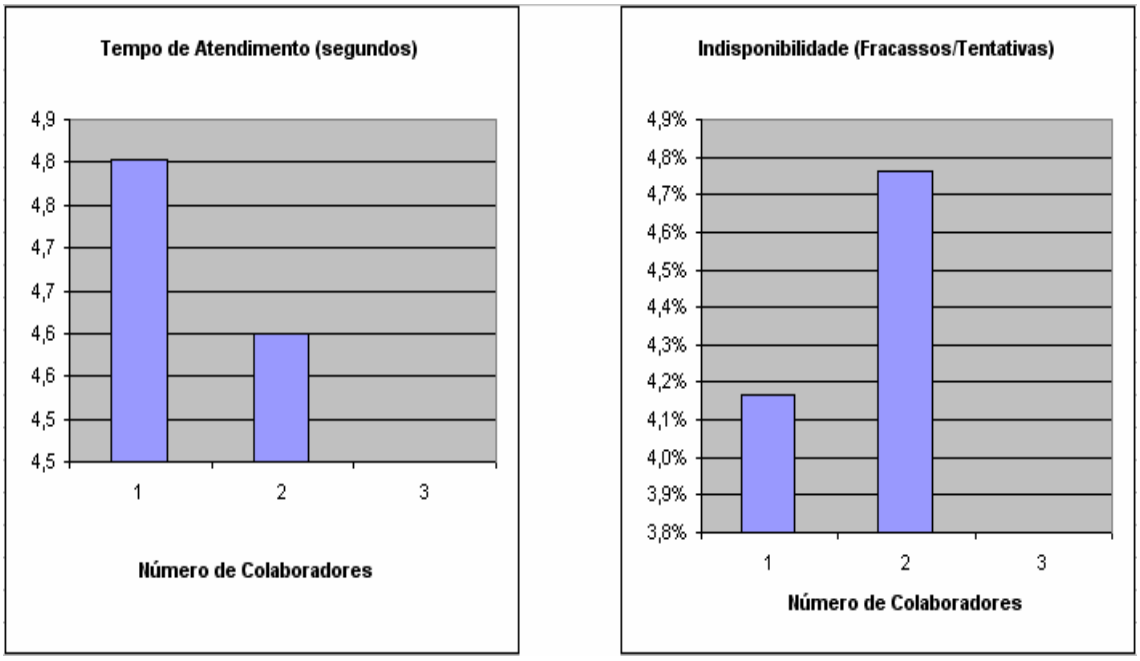

Figura 6.2: Gráficos do Tempo de atendimento e da Indisponibilidade

Devido à natureza de processamento da aplicação de Fibonacci, a geração de números é um processamento com dependências, onde as tarefas de gerar números não podem ser paralelizadas em função de dependências de números anteriores. Apesar do número de clientes ser significativo, cada máquina cliente ficou muito tempo esperando a resposta de disponibilidade de um colaborador, além de esperar pela execução da ação remota que gera efetivamente o número. Um colaborador é mais do que suficiente para atender um número bastante elevado de clientes desta aplicação, não havendo alterações nas grandezas observadas. Como o número de máquinas usadas no experimento é limitado, a escolha da aplicação de Fibonacci não ajudou a demonstrar melhora nos tempos de atendimento e diminuição da indisponibilidade. O experimento serviu apenas para demonstrar a colaboração de máquinas na execução de aplicações promovida pelo Mentor, lembrando que as máquinas em questão nunca teriam acesso à aplicação em função do hardware limitado. Um experimento mais representativo deve utilizar uma aplicação cujo tempo de processamento no colaborador é suficiente grande para afetar a disponibilidade da mesma.

7 CONCLUSÃO

7. 1 Introdução

O trabalho de pesquisa, elaboração e implementação do Mentor atingiu seu objetivo de criar uma proposta transparente de distribuição de software. O trabalho apontou falhas e levantou questões importantes sobre uma área que, até a apresentação deste documento, foi pouco pesquisada e discutida. Um dos objetivos é mostrar que um caminho importante a ser desbravado é utilizar o potencial de máquinas dispostas a colaborar, para permitir acesso ao software a pessoas ou organizações com menores recursos. A adoção do Mentor, ou de parte de suas idéias, em sistemas de distribuição atualmente em operação, pode contribuir para a democratização do acesso ao software, levando-se em conta somente requisitos de hardware. A classificação de hardware em três grupos diferentes, conforme sua capacidade de processamento, possibilita que máquinas com hardware suficiente possam ter acesso a qualquer aplicação, desde que existam máquinas colaboradoras mais robustas, disponíveis e ligadas em rede.

7. 2 Idealização e implementação da camada de comunicação

Uma contribuição importante deste trabalho foi o êxito na implementação da camada de comunicação GoP2P. Ela possibilitou a construção de aplicações no modelo ponto-a-ponto com baixo custo de implementação. A camada é a base para a implementação do sistema de distribuição e foi razoavelmente testada desde que passou a ser utilizada. Além de contribuir para o sistema de distribuição, a camada foi utilizada na construção da aplicação escolhida para o estudo de caso, que também adota o modelo ponto-a-ponto.

Apesar de ter sido bastante testada, algumas melhorias ainda podem ser feitas na camada. Uma delas é implementar um mecanismo de execução de ações que permita que mais de uma ação remota seja executada em paralelo.

Atualmente um único pipe bidirecional é oferecido pela camada, o que formaria uma fila de atendimento se ações com efeito bloqueante fossem requisitadas. A solução para a má utilização da capacidade de processamento é obtida implementando um mecanismo para a criação de um pipe bidirecional para cada ação atendida em paralelo. Para este trabalho, não houve necessidade de implementar um mecanismo de atendimento de múltiplas ações em paralelo, mas ele será necessário para evitar a subutilização de máquinas com maior capacidade de atendimento.

7. 3 A abordagem do Mentor para a divulgação de software

A proposta de divulgação de software do Mentor é relativamente original. Apesar do método de classificação por facetas ser antigo, a idéia de utilizar o próprio mecanismo de distribuição para também divulgar software, pode ser aplicada a outros sistemas de distribuição eletrônicos, principalmente aqueles que distribuem software gratuito. A classificação e busca de software usando critérios permitirá que usuários só sejam comunicados de aplicações de seu interesse, de maneira seletiva. Além disso, a implementação permitirá que fornecedores de pequeno porte possam divulgar suas aplicações de forma gratuita.

O Mentor é uma proposta de sistema de distribuição que ainda não foi colocada em operação. A criação das facetas, e de seus valores possíveis, deve levar em conta o feedback de usuários e fornecedores com a utilização real de um sistema de distribuição. Antes de colocar o sistema em operação, será necessário também alterar a implementação do programa cliente do Mentor, para possibilitar a filtragem de aplicações indesejadas pelos usuários.

7. 4 Trabalhos Futuros

Existem muitas questões a serem equacionadas antes que usuários e fornecedores possam adotar o Mentor como uma solução alternativa para a distribuição de software não comercial. O Mentor não pode ser considerado ainda uma solução definitiva, pois ainda existe bastante trabalho a ser feito.

Um trabalho importante a ser desenvolvido é o aperfeiçoamento do suporte dado aos fornecedores no desenvolvimento de aplicações para serem distribuídas e executadas na rede do Mentor. A separação da aplicação em três classificações distintas deve ser feita de forma transparente ao fornecedor. Atualmente a separação é de responsabilidade do fornecedor e deve estar presente no código fonte da aplicação. O fornecedor precisa implementar classes especiais e separá-las em pacotes de instalação para cada classificação e, durante todo o processo de desenvolvimento, ele deve levar em conta esta separação. A melhoria deste suporte é fundamental para ajudar na divulgação do Mentor aos fornecedores.

A segurança do ambiente de distribuição Mentor não foi estudada neste trabalho por estar fora do seu escopo. No entanto, esta questão é de grande importância e deve ser tratada de maneira adequada em trabalhos posteriores.

Uma melhoria importante, a ser aplicada na camada de comunicação GoP2P, está relacionada ao encapsulamento da tecnologia Jxta. Atualmente o encapsulamento não é total. Ainda existe um trabalho de configuração da própria tecnologia Jxta que precisa ser feito pelo usuário da aplicação que usa a camada. Esta configuração é fundamental para que máquinas que executem o serviço GoP2P consigam se comunicar. Um trabalho futuro deve ser realizado para que a configuração da camada seja transparente ao usuário, detectando barreiras na rede em que a máquina se encontra e decidindo a melhor configuração para o seu caso particular.

Atualmente a implementação do Mentor não dá suporte ao uso de aplicações em dispositivos limitados como os celulares e pdas. Um cliente mais leve do Mentor precisa ser desenvolvido para a máquina virtual da plataforma Java 2 Micro Edition (J2ME), o que necessitará um trabalho de conversão da arquitetura para rodar neste tipo de hardware.

8 REFERÊNCIAS BIBLIOGRÁFICAS

PRESSMAN, R. S. Engenharia de Software. Mc Graw Hill, 1995. 1056 p.

SOBR, L., TUMA, P. SOFAnet: Middleware for Software Distribution over Internet. In: IEEE Proceedings of the 2005 Symposium on Applications and the Internet (SAINT'05).

ANDERSSON, J. A Deployment System for Pervasive Computing. In: IEEE Proceedings of the International Conference on Software Maintenance (ICSM'00) on October 2000.

TURCAN, E., SHAHMEHRI, N., GRAHAM, R. L. Intelligent Software Delivery Using P2P. In: IEEE Proceedings of the 2002 Second International Conference on Peer-to-Peer Computing (P2P'02).

MINAR, N., HEDLUND, M. Peer-to-Peer: Harnessing the Power of Disruptive Technologies, Chapter 1, O'Reilly, 2001. 448 p.

MUGHAL, K., A., RASMUSSEN, R., W. A Programmer's Guide to Java Certification: A Comprehensive Primer. Addison-Wesley, 2000. 754 p.

PRIETO-DIAZ, R., FREEMAN, P. Classifying Software for Reusability. Article reprinted with permission from IEEE Software, 1987.

DLL. Literatura disponível em: http://msdn.microsoft.com. Último acesso: 2005.

RPM. Literatura disponível em: http://www.rpm.org. Último acesso: 2005.

APT. Literatura disponível em: http://www.debian.org. Último acesso: 2005.

PORTAGE. Literatura disponível em: http://www.gentoo.org. Último acesso: 2005.

YAP, M., NG, W. ASMA: Autonomous Software Management Agent System. In: IEEE Proceedings of the International Conference on 31 Oct.-3 Nov. 1999, p. 630-637.

HALL, R. S., HEIMBIGNER, D., WOLF, A. L. A Cooperative Approach to Support Software Deployment Using the Software Dock. In: ACM Proceedings of the 21st international conference on Software engineering on May 1999.

BAKKER, A., STEEN, M. V., TANENBAUM, A. S. A Law-Abiding Peer-to-Peer Network for Free-Software Distribution. In: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01) on October 2001.

BECK, K. Extreme Programming Explained: Embrace Change. 1. ed. Reading, MA: Addison-Wesley, 2000. 190 p.

RMI: Especificação da Arquitetura. Disponível em: http://java.sun.com/products/jdk/rmi. Último acesso: 2005.

JXTA. Disponível em: http://www.sun.com/jxta. Último acesso: 2005.

OAKS, S., TRAVERSAT, B., GONG, L. Jxta in a Nutshell. O'Reilly, 2002. 416 p

JAVA. Disponível em: http://java.sun.com. Último acesso: 2005.

SCHWARZ, D. Managing Component Dependencies Using ClassLoaders. Disponível em: http://www.onjava.com/pub/a/onjava/2005/04/13/dependencies.html. Último acesso: 2005.

TOMCAT. Disponível em: http://jakarta.apache.org/tomcat. Último acesso: 2005.

Projeto LABASE. Literatura disponível em: http://labase.nce.ufrj.br. Último acesso: 2005.

PAIS, A.P.V. Arquitetura de Controle Hércules: a Base para a Geração Automática de Sistemas de Informação com Ênfase na Camada de Controle. 2004. 132 p. Dissertação (Mestrado em Informática) - Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

HOFF, A., V., PARTOVI, H., THAI, T. The Open Software Description Format (OSD), 1997. Disponível em: http://www.w3.org/TR/NOTE-OSD.html. Último acesso: 2005.

DTMF, Desktop Management Task Force, Enabling your product for manageability with MIF files, Nov. 1994. Disponível em: http://www.dmtf.org. Último acesso: 2005.

SWING: The Swing Tutorial. Disponível em http://java.sun.com/docs/books/tutorial/uiswing. Último acesso: 2005.

FREEMAN, E., HUPFER, S., ARNOLD, K. JavaSpaces: Principles, Patterns, and Practice. Addison-Wesley, 1999. 368 p.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. Design Patterns. Addison-Wesley Professional, 1995. 395 p.

Fast MD5 Implementation in Java, Timothy W Macinta. Disponível em: http://www.twmacinta.com/myjava/fast_md5.php. Último acesso: 2005.

JPF: Java Plugin Framework. Disponível em: http://jpf.sourceforge.net, Último acesso: 2005.

JXTASPACES. Disponível em: http://jxtaspaces.jxta.org, Último acesso: 2005.