Universidade Federal de Pernambuco

Centro de Informática

Mobilidade, Autonomia e Distribuição em Agentes para o Gerenciamento Corporativo de Sistemas

por

Hendrik Teixeira Macedo

Dissertação de Mestrado

Recife, fevereiro de 2001

Hendrik Teixeira Macedo

Mobilidade, Autonomia e Distribuição em Agentes para o Gerenciamento Corporativo de Sistemas

Este trabalho foi submetido à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Geber Lisboa Ramalho
Co-Orientador: Prof. Dr. Carlos A. G. Ferraz

Recife, fevereiro de 2001

Resumo

A heterogeneidade e a dimensão das redes de computadores atuais requerem uma atividade de gerenciamento bem mais complexa, para que haja qualidade de serviço para usuários do sistema. As ferramentas de automação existentes são fundamentadas em abordagens cliente-servidor, onde uma estação de gerenciamento (NMS) age como cliente de um conjunto de agentes localizados nos dispositivos a serem gerenciados. Por ser centralizada e utilizar uma noção muito simplista de agentes, essa forma de gerenciamento é pouco flexível e está mais susceptível a falhas e a gerar níveis altos de tráfego de rede. Uma abordagem alternativa promissora baseia-se em agentes inteligentes capazes de tomar decisões com mais autonomia e de migrarem entre dispositivos, possibilitando uma forma de gerenciamento mais distribuída. Entretanto, apesar do potencial de propriedades como autonomia, mobilidade e distribuição, não há ainda um estudo conclusivo sobre como de fato utilizá-las na implementação de um sistema de gerenciamento real.

Esta dissertação discute o impacto que estas três propriedades têm no desenvolvimento de soluções de gerenciamento corporativo de redes baseadas em agentes. Tendo em vista algumas métricas (tempo de processamento, consumo de CPU, consumo de espaço em disco, etc.), foi realizada uma avaliação de diferentes arquiteturas multiagente baseadas na combinação dessas propriedades, tendo como estudo de caso o problema da administração de espaço em disco em máquinas UNIX/NFS. Para realizar os experimentos com maior controle, foi desenvolvido um simulador. Os resultados obtidos servem como guia para elaboração de uma metodologia de construção de sistemas de gerenciamento que se baseiem em agentes autônomos móveis e distribuídos.

Abstract

The diversity and extent of current network computers demand a complex management activity, to provide quality of service to system users. The usual automation tools follow a client-server approach, where a management station acts as client of a set of agents placed in the managed devices. Because it is completely centralized and uses a very simplistic notion of agents, this kind of management has weak flexibility and fault tolerance, and generates higher network traffic. A promising alternative approach is based on intelligent agents capable of taking decisions more autonomously and migrating between devices, allowing a more distributed management. Unfortunately, despite the potential of properties such as autonomy, mobility and distribution, there are not yet conclusive studies about how to effectively use them in the implementation of a management system.

This work discusses the impact that those properties have in the development of corporative networks management solutions based on agents. Considering some metrics (processing time, CPU consumption, space savings, etc.), we have made an evaluation of different multiagent architectures varying the levels of agents' autonomy, agent's mobility and number of agents. As case study, we have chosen the disk space management problem in UNIX/NFS machines. We have developed a simulator to carry out experiments with higher control. The results can be used as a guide to elaborate a methodology for construction of management systems based on mobile, autonomous and distributed agents.

Sumário

Lista de Figuras

Lista de Tabelas

1
Introdução

At present, some intelligence is embedded in computing machines while most is built into people, and there is almost none in the network itself. Using agents, this new model takes advantage of what computers are really good at: remembering and processing large amounts of information; what people are good at: thinking about trends and higher-level understanding; and what networks are good at: maintaining connections and moving information and knowledge around. (W. Cockayne e M. Zyda em [11]).

O crescimento da informação em formato digital e a necessidade de grupos de pesquisa em compartilhar essas informações levou à interligação de computadores. Daí o surgimento, inicialmente, das Redes de Computadores Locais (*LANs – Local Area Networks*) que ligam máquinas de um mesmo laboratório, departamento ou centro, e depois das Redes de Computadores maiores (*WANs – Wide*

Area Networks) que chegam a interligar computadores de todas as partes do mundo (exemplo, *Internet*) [64].

Essa proliferação de redes de computadores apenas reflete a necessidade de comunicação que é inerente ao ser humano e sempre foi uma das grandes responsáveis pelas grandes revoluções: revolução industrial, revolução tecnológica, e das telecomunicações. A tendência é que mais e mais redes de diferentes configurações e diferentes sistemas operacionais interajam para que haja troca de informações e também o compartilhamento de recursos computacionais como impressoras, discos rígidos, CPU, etc. Pode-se imaginar que uma integração desse nível não é simples e essa diversidade de configuração precisa ser bem controlada e gerenciada. Não que não seja necessária a atividade de administração em sistemas isolados. Mas com a evolução da computação centralizada para a computação distribuída, a administração de sistemas tornou-se responsável pela manutenção e interoperabilidade dessas redes compostas por máquinas dos mais diversos fabricantes e sistemas operacionais. A popularidade da Internet acabou por tornar a administração de sistemas também responsável pela segurança das redes.

Desse modo, pode-se ver a administração de sistemas hoje em dia como uma área da Computação responsável por estudar formas, gerar teorias, modelos, técnicas e ferramentas para garantir uma integração confiável e de boa performance para sistemas de computação com plataformas heterogêneas de hardware, software básico e bancos de dados [20].

1.1 Motivação

A automação da gerência de redes significa delegar procedimentos que seriam realizados manualmente pelo gerente da rede[1] a processos computacionais. Mas por que delegar?

O administrador de sistemas é responsável por projetar, operar, manter e atualizar uma rede para que ela seja segura, flexível, robusta e eficiente [20]. Qualquer interrupção na disponibilidade dos recursos computacionais ou inconsistência do estado de transações em rede pode afetar todo o desempenho de uma organização[2] e até causar danos graves aos sistemas. Normalmente, a atividade de gerenciamento é caracterizada por uma grande repetição e continuidade de ações por parte do administrador da rede, que pode acarretar imprecisão, atraso na tomada de decisões e subtilização dos próprios recursos humanos.

Além disso, o crescimento do mercado de informática tem aberto caminhos para o uso de equipamentos de diversos fabricantes, visando custos mais reduzidos e levando a sistemas cada vez mais heterogêneos. Assim, existe uma grande variedade de plataformas a gerenciar e os operadores precisam de conhecimento e habilidade para extrair informações úteis de cada um desses sistemas. A automatização da atividade de gerência onde o comportamento do administrador é mapeado para uma série de situações práticas, de forma a se criar um conjunto de regras que controlem a atuação do sistema gerenciador torna-se cada vez mais importante.

Estas ferramentas devem realizar coleta de dados sobre o estado de vários dispositivos da rede e integrar estes dados, bem como realizar correlação desses dados para indicar problemas e falhas. Uma grande dificuldade para o desenvolvimento é a diversidade de

[1] Em toda a dissertação, termos como “gerente da rede”, “administrador de sistemas”, ou ainda “operadores” são relativos à atividade humana. A ferramenta desenvolvida para automação do processo é referenciada por “software”, “ferramenta”, ou “sistema”.

[2] Universidades, institutos, laboratórios, empresas, indústrias, corporações enfim, qualquer organismo que utilize e/ou dependa de recursos de informática para suas operações.

formatos das informações sobre o estado dos recursos da rede. Assim, para que a necessidade das organizações atuais seja atendida, as soluções para gerência devem atender a requisitos básicos como escalabilidade, redução da complexidade, conformidade com padrões, e oferecer coleta e monitoração distribuídas, filtragem de dados, análise distribuída, consultas dinâmicas, etc.

Muitas dessas características poderiam ser alcançadas utilizando-se técnicas de Inteligência Artificial, particularmente, *Agentes Inteligentes* [57], [76] e [110]. De fato, o conceito de agentes representa uma evolução do paradigma anterior de orientação a objetos, pois permite uma representação mais completa de entidades do mundo (no caso, redes de computadores, máquinas, discos, periféricos, gerentes de rede, etc) onde é possível fazer uma modelagem não apenas do estado, mas do comportamento dessas entidades e eventuais modificações desse comportamento. A provisão de inteligência a um agente permite que possam ser modeladas entidades com conhecimento mais completo do mundo, com mais autonomia e poder de tomada decisões sobre mudanças em seu ambiente de atuação, sem intervenção humana.

Um dos problemas com as soluções de gerenciamento atuais é que elas são construídas segundo abordagens que utilizam uma noção de agentes muito simplista (agentes SNMP, CMIP) (ver Seção2.1.2). Esses agentes, que se localizam nos dispositivos da rede, funcionam simplesmente como processos (*daemons*) que verificam o *status* dos dispositivos e enviam mensagens a uma estação de gerenciamento (máquina utilizada pelo administrador, que é responsável por monitorar os dispositivos, identificar eventuais problemas e acionar comandos para correção dos mesmos) em resposta a alguma mudança repentina de funcionamento. Essa forma de gerenciamento cliente-servidor (CS) é responsável por um alto grau de centralização na solução que sobrecarrega a estação de gerenciamento além de ser pouco tolerante a falhas: qualquer problema com a estação (máquina) ou com a rede em si, interrompe a atividade de gerenciamento. Além disso, há o problema do congestionamento no tráfego de rede. Em períodos de atividade intensa na rede, faz-se necessário o gerenciamento do tráfego

gerado em certas regiões. Entretanto, as operações CS de gerenciamento da estação intensificam o tráfego na região que, por sua vez, demanda por mais intervenção de gerência.

O uso de agentes mais sofisticados nas ferramentas de gerenciamento pode minimizar os efeitos dessa grande centralização. Esta sofisticação viria sobretudo da capacidade do agente de tomar decisões *autonomamente* [57] e da capacidade de *mover* seu código e estado execução para um outro dispositivo [5], [11], [14] e [23]. Além disso, estas duas propriedades, autonomia e mobilidade, dariam melhor suporte a uma maior distribuição do gerenciamento, o que é desejável. Infelizmente, as soluções que tentam implementar agentes mais sofisticados são poucas e apresentam os problemas seguintes:

- pouco proveito se tem tirado da combinação mobilidade-autonomia;
- as soluções são ainda calcadas no controle centralizado, em vez de serem mais distribuídas;
- falta uma metodologia precisa de desenvolvimento, que indique quando, como e em que grau dotar um ou mais agentes de autonomia e mobilidade.

A falta de metodologia que diz respeito à utilização das propriedades dos agentes inteligentes é, ao nosso ver, o problema central do uso efetivo de agentes mais sofisticados na gerência corporativa de redes. Por ser uma tecnologia ainda incipiente, várias questões ainda precisam ser respondidas, como: que grau de autonomia deve ser dado a um agente, dado um problema de gerenciamento? Todos agentes devem ter a mesma autonomia? Em que situação, os agentes devem ser móveis e quantos deles devem sê-lo? Que impacto efetivo tem a autonomia e a mobilidade para a solução de um dado problema de gerenciamento? Quantos agentes devem existir numa solução? É interessante que eles possam cooperar? Todos agentes devem ter as mesmas funcionalidades?

As respostas a essas questões, que têm um impacto direto na performance de uma solução final, ainda não estão claras. O custo/benefício de se prover certas propriedades

pode não ser favorável, visto que a implementação de algumas propriedades mais sofisticadas de agentes podem requer ferramentas de suporte, maior necessidade de intervenção de administrador humano na solução final, maior tráfego de rede, consumo de memória, etc.

1.2 Objetivos

Esta dissertação faz parte de um esforço iniciado com o projeto FLASH [18], realizado no Cin/UFPE, para construção de sistemas para gerência que resolvam algumas classes de problemas da administração de redes. O projeto FLASH foi uma combinação de esforços acadêmicos e empresariais para buscar soluções de problemas complexos e relevantes no espírito da pesquisa cooperativa e multidisciplinar. Ele teve como objetivo buscar inovações científicas e tecnológicas que forneçam novos subsídios para integração de sistemas heterogêneos, desenvolvimento de ferramentas de suporte à administração de sistemas, introdução de novas tecnologias na Internet brasileira, no setor empresarial e no meio acadêmico, e formação de recursos humanos.

Em particular, o objetivo desta dissertação é o de fornecer elementos para a elaboração de uma *metodologia* para o desenvolvimento de soluções inteligentes para a administração de sistemas em rede. Acreditamos que toda atividade de gerência de sistemas em rede deve ser repensada, e a filosofia corrente de uma solução centralizada deve ser substituída por uma filosofia mais distribuída onde o conhecimento das atividades de gerenciamento não resida apenas no administrador humano, ou em uma máquina central, mas em entidades (agentes com autonomia de gerenciamento) distribuídas por todos os dispositivos de rede e que sejam capazes de detectar e corrigir eventuais problemas.

É proposta a discussão de três aspectos: *Mobilidade*, *Autonomia* e *Distribuição*[3] para elaboração de uma solução baseada em agentes. Atualmente é possível encontrar sugestões e pesquisas para desenvolvimento de ferramentas de administração que implementem alguns desses aspectos (ver seção 3.3), entretanto, fica claro que ainda não há um consenso no desenvolvimento: na implementação desses aspectos ou de outras propriedades de agentes inteligentes. Além disso, a idéia da existência de uma estação de gerenciamento controladora ou monitora de todos os dispositivos da rede ainda persiste. Contribuir nesse intento é um dos objetivos do trabalho.

É possível que, se bem trabalhados, bem relacionados, bem quantificados, estes aspectos representem uma boa alternativa ao modelo tradicional de gerenciamento Cliente-Servidor (CS) e a essa maneira *ad hoc* em que é feita a maioria dos projetos e implementações de ferramentas para administração que possuem características de IA embutidas.

1.3 Metodologia de trabalho

Para atingir o objetivo, fez-se necessário definir uma certa metodologia de trabalho. Primeiramente, definimos diferentes *tipos de agentes*, segundo as propriedades de autonomia e mobilidade. Combinando esses tipos de agentes e variando o grau de distribuição deles, elaboramos algumas *arquiteturas de agentes inteligentes*. Instanciamos então os agentes e as arquiteturas definidas para um domínio particular, a administração de espaço em disco, para servir como estudo de caso. Por conta da dificuldade de se obter exclusividade de máquinas para uma grande bateria de testes e instabilidade nos estados dos dispositivos em rede, implementamos um simulador para permitir a realização de *experimentos*. Tendo especificado um conjunto de critérios de avaliação relevantes e usando tal simulador, efetuamos os experimentos, cujos

[3] Distribuição se refere à quantidade dos supostos agentes para a solução a ser desenvolvida.

resultados podem servir de guia para a elaboração de uma solução de administração e da criação de uma futura metodologia.

1.4 Estrutura da Dissertação

Esta dissertação possui sete capítulos. Os dois próximos capítulos falam sobre o Gerenciamento Corporativo de Sistemas (Gerenciamento de Redes e Administração de Sistemas).

O Capítulo 2 faz uma breve introdução à área, falando de conceitos gerais sobre Redes e Sistemas Distribuídos (SD) [13], apresenta as áreas funcionais de gerenciamento de redes OSI além de um levantamento das abordagens tradicionais de gerenciamento existentes na literatura. Ainda nesse capítulo são relacionadas algumas ferramentas de gerenciamento principais. O Capítulo 3 incorpora a utilização de Agentes para o Gerenciamento Corporativo de Sistemas. São abordadas tecnologias como mobilidade, raciocínio e coordenação em sistemas de agentes. Algumas pesquisas sobre inclusão dessas tecnologias são comentadas, por fim.

Os aspectos, *mobilidade*, *autonomia*, e *distribuição* são discutidos no Capítulo 4. Discute-se sobre a influência desses aspectos quando do desenvolvimento de software para administração. Neste capítulo são definidos os tipos de agentes e arquiteturas que servirão para avaliação do impacto de cada um dos aspectos.

No quinto capítulo é explicado o estudo de caso utilizado: o Sistema de Arquivos NFS (*Network File System*) do UNIX. Os tipos de agentes e arquiteturas definidas são adequados a esse estudo de caso.

O simulador de gerenciamento desenvolvido é descrito no Capítulo 6. É mostrada toda a modelagem do sistema, alguns detalhes de implementação do simulador, bem como os códigos de teste para medir os valores reais das operações (exemplo, migração de um agente, chamada de um agente, *multicast* de mensagem, inicialização de ferramentas de suporte, etc) numa rede.

O capítulo 7 traz os resultados e discussões dos experimentos realizados.

As conclusões finais da dissertação são apresentadas no capítulo 8.

2

GCS: Gerenciamento Corporativo de Sistemas

Uma rede de computadores típica é bastante complexa, incluindo uma variedade de plataformas, sistemas operacionais, protocolos de comunicação e arquiteturas. Adicionalmente, administradores de rede se defrontam com requisitos contraditórios: têm de garantir performance ao mesmo tempo em que devem controlar custos; têm de prover segurança e efetivamente gerenciar o crescimento da diversidade de ambientes; garantindo um acesso simples à rede.

O grau de distribuição das redes hoje em dia é tão alto, que fica virtualmente impossível conhecer o que está sendo conectado ou desconectado em *sites* remotos. Dispositivos configurados impropriamente, aplicações não autorizadas ou uso inapropriado podem exaurir recursos. Mesmo *modems* não autorizados representam uma ameaça à segurança de uma organização.

É comum que a infra-estrutura da rede seja responsabilizada por uma eventual baixa performance. O fato é que a gerência de uma rede não é tarefa das mais simples. Se

uma organização não possui grande controle sobre quais dispositivos existem em sua rede, como eles estão conectados, e qual o valor de seus negócios, não é possível gerenciá-la. Se um problema ocorre em um dispositivo de grande importância ou a performance é afetada em um segmento importante da rede, os administradores precisam estar a par do problema, tomar as devidas ações de correção e informar usuários antes das operações serem afetadas.

Este capítulo apresenta o Estado da Arte da gerência e administração de sistemas em rede. Apesar de conceitualmente serem atividades distintas, com o estágio atual de evolução dos negócios em rede, atividades como projeto, operação, manutenção e atualização de uma rede de computadores heterogênea passou a ser tanto tarefa de Administração quanto de Gerenciamento. Essa evolução é responsável pela mudança de enfoque do gerenciamento de recursos pontual para um gerenciamento corporativo, voltado aos negócios da empresa. Assim essas duas atividades passam a fazer parte de um macro gerenciamento denominado de Gerenciamento Corporativo de Sistemas.

2.1 Gerenciamento Corporativo

A presença cada vez maior de ambientes de processamento distribuído como CORBA (ver Seção 6.3.1), tem revelado as limitações das estratégias de gerenciamento de sistemas convencionais. Devido à cooperação existente entre os diversos componentes de um sistema distribuído aberto, qualquer atividade local de gerenciamento pode afetar outros componentes residentes, possivelmente, em uma máquina remota. Sendo assim, faz-se necessária uma coordenação entre as atividades de gerenciamento concorrentes para que sejam evitados efeitos indesejados.

2.1.1 Atividades do Administrador/Gerente

Entre as atividades comumente designadas a administradores de rede estão a criação e gerenciamento de contas de usuários, *backups* de dados, monitoração das atividades do

sistema, manter inventário e instalação de dispositivos da rede, configuração de sistemas, distribuição de software, políticas de segurança dos sistemas, controle de tráfego da rede, etc.

A ISO (*International Standards Organization*) [30] categorizou os requisitos básicos para manutenção e gerenciamento de uma rede de computadores em cinco áreas funcionais de gerenciamento:

- Gerenciamento de Configuração
- Gerenciamento de Falhas
- Gerenciamento de Desempenho
- Gerenciamento de Contabilização
- Gerenciamento de Segurança

Para que seja possível configurar um dispositivo é necessário que o *driver* respectivo esteja instalado em cada máquina que irá usar o dispositivo. Seja considerada, por exemplo, a atividade de instalar uma impressora de rede: os *drivers* requeridos para se interagir com a impressora diferem se a máquina é um Macintosh, um Unix ou um PC, ou se possuem sistemas operacionais diferentes. É função do administrador instalar os *drivers* adequados a cada caso;

O administrador é responsável por detectar, analisar, corrigir e prever operações anormais na rede. Por exemplo, um certo dispositivo pode estar com uma utilização acima da considerada normal, ou um determinado servidor apresentou algum problema, o administrador tem prover, então, um outro servidor para que os usuários da rede não sejam prejudicados.

É função também do administrador avaliar o desempenho geral da rede, a velocidade das transações, tempo de processamento nos servidores e nas máquinas, para verificar se a rede ou uma estação está sobrecarregada, por exemplo. Cabe ao administrador a

decisão de mover ou clonar um servidor sobrecarregado para um ambiente de execução melhor em prol de aumentar a eficiência global da rede.

O administrador é responsável pela manutenção das contas dos usuários da rede, definir cotas de utilização de disco, cotas de impressão, etc. Remoção e compactação de arquivos também pode ser realizado para se evitar o gasto desnecessário de espaço em disco nas máquinas ou em servidores da rede.

Por fim, realizar o controle de acesso ao sistema, segurança de *firewall,* gerenciamento de senhas de usuários, aplicativos de anti-vírus, etc, são medidas de segurança que devem ser tomadas pelo administrador do sistema.

Os administradores de SDs têm de lidar tanto com a crescente heterogeneidade dos componentes de sistema envolvidos quanto com especialização das ferramentas de gerenciamento existentes no mercado. A maioria dessas ferramentas é limitada quanto ao número de plataformas que suportadas. As mesmas atividades de gerenciamento requerem diferentes comandos para diferentes plataformas. A interface gráfica das plataformas de gerenciamento comerciais esconde os detalhes da sintaxe de comando, mas não provê meios suficientes de conectar logicamente as atividades de gerenciamento.

Pode-se imaginar um simples sistema distribuído com uma aplicação cliente-servidor, onde a provisão de serviços pode ser interrompida bruscamente, mas sem perda de dados. Por exemplo, o administrador quer fazer um *upgrade* de um *software* para um servidor. É necessário que antes da instalação do *upgrade*, a versão anterior instalada seja apagada. Para que não haja perda de dados de clientes, o administrador deve desconectar da rede todos os clientes e remover o serviço oferecido. Após a instalação, o servidor reinicia, envia mensagens de *e-mail* para os usuários, e exporta o serviço oferecido para os clientes poderem processar. Pode-se identificar sete diferentes atividades de gerenciamento neste simples exemplo, visualizadas no grafo de dependência da Figura 2.1. Estas atividades possuem dependência de três operações centrais, *upgrade* do *software* (US), migração do servidor (MS), e manutenção da rede

com perda temporária da provisão de serviços (MR). No grafo, as siglas das operações acompanham os nós de atividades com as quais possuem relacionamento de dependência.

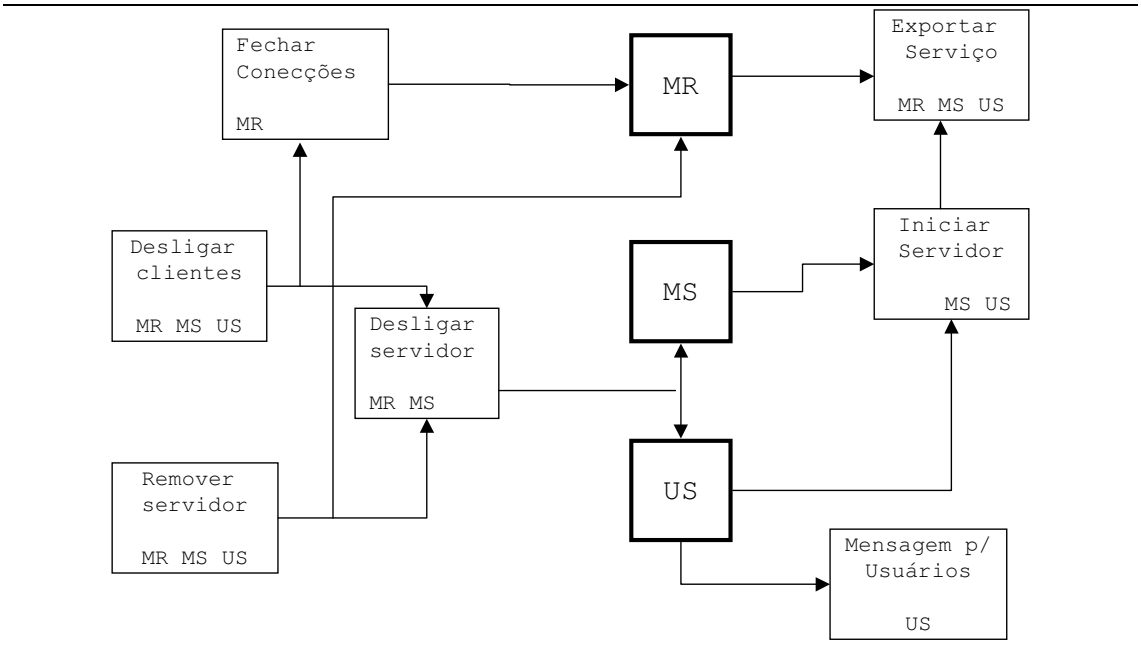


Figura 2.1 - Grafo de dependência de diferentes atividades de gerenciamento.

A automação destas atividades requer um sistema que seja capaz de criar séries de atividades dinamicamente em ambas direções (*backward* e *foreward*), dependendo do estado atual do ambiente (no exemplo acima, pode não ser necessária alguma atividade prévia caso o servidor já esteja desligado na hora do *upgrade*). Essas características sugerem uso de *regras* para modelagem das atividades de gerenciamento. O próximo capítulo trata com mais detalhes este aspecto.

2.1.2 Abordagens Tradicionais

Os sistemas atuais de gerenciamento de redes são baseados num paradigma de centralização: a aplicação de gerenciamento, periodicamente, acessa os dados coletados por um conjunto de módulos de software localizados nos dispositivos de rede,

utilizando um protocolo apropriado. Esses sistemas se baseiam basicamente em dois tipos de módulos: *sistemas gerenciadores* e *recursos*, onde os sistemas gerenciadores monitoram e controlam recursos da rede (Figura 2.2).

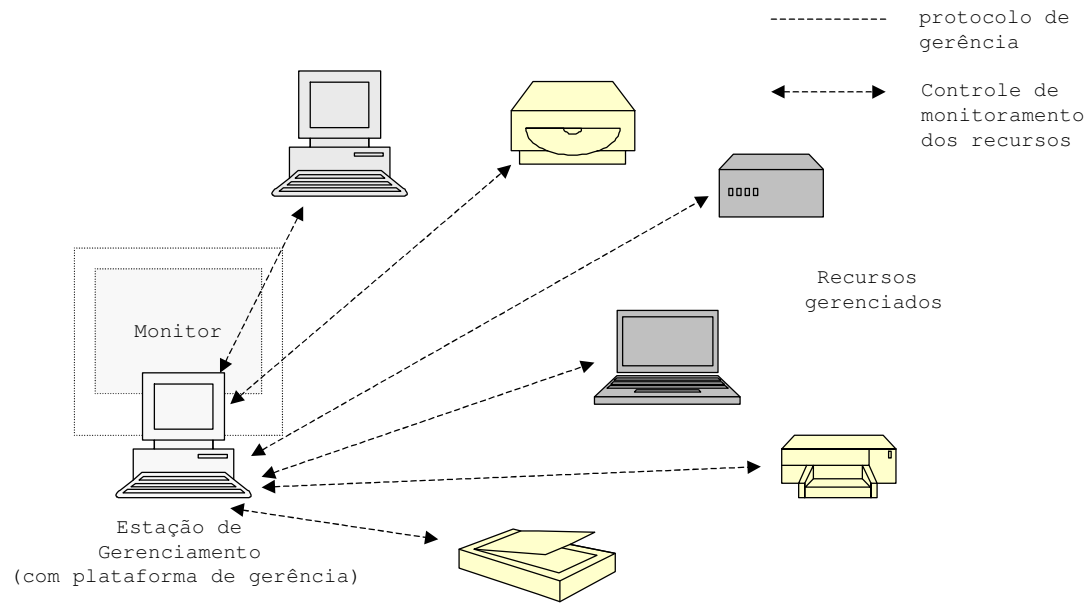


Figura 2.2 - Estação de gerenciamento controla e monitora os recursos da rede.

O sistema gerenciador lida apenas com a funcionalidade do gerenciamento do recurso, não importando os detalhes de implementação. O modelo de programação *baseado em objetos* é um paradigma bastante adequado para definição dos processos de gerenciamento.

Correntemente, a maioria do conhecimento sobre o gerenciamento reside no próprio administrador. Ele conhece as políticas de gerenciamento, as demandas dos usuários do sistema e seus requisitos de qualidade e possuem o conhecimento operacional para transformar estas demandas em ações. O gerenciamento operacional é feito através do monitoramento do status do sistema, análise da situação atual e disparo de comandos apropriados para correção de erros ou otimização da utilização dos recursos. Essa transformação de políticas de gerenciamento em ações corretivas normalmente ocorre apenas e exatamente no instante em que um erro requer a ação correspondente. Além

disso, nem as políticas, nem as ações são documentadas. Uma boa automação das atividades de administração, entretanto, requer uma descrição do conhecimento do diagnóstico e procedimentos de gerenciamento apropriados.

A abordagem mais utilizada[1] de gerenciamento é proposta pela IETF (*Internet Engineering Task Force*) e é baseada no protocolo SNMP (*Simple Management Protocol*) [8] e [7]. Outra abordagem é a proposta pela ISO (*International Organization for Standardization*) para aplicação em redes OSI [64], que é baseada no protocolo CMIP (*Common Management Information Protocol*) [80].

As abordagens são bastante similares, diferindo apenas no modo como operam. Ambas se baseiam totalmente na centralização do gerenciamento, por esse motivo possuem limitações graves, e assumem a presença de uma *estação de gerenciamento* (operada pelo administrador), que interage com os *agentes* em execução em nós da rede (*bridges,* roteadores*, workstations*, etc). A estação de gerenciamento (NMS) é simplesmente uma máquina que faz parte da rede e que possui a plataforma de gerenciamento (espécie de Sistema Operacional para *software* de gerenciamento) onde o administrador trabalha (Figura 2.2). Os agentes SNMP (nome dado aos agentes que falam o protocolo SNMP) podem ser encarados como objetos que utilizam uma interface de gerenciamento exclusiva.

Na abordagem da IETF, cada *snmp-agent,* armazena informação sobre o dispositivo em uma base local de informações chamada *management information base* (MIB) [46]. O software gerente interage com os agentes através do protocolo de gerenciamento que especifica o formato do pacote para um conjunto de operações básicas. As mensagens contendo dados e códigos de operação são transmitidas utilizando-se serviços da camada de transporte [64].

O software de gerência age como um cliente e interage com os agentes SNMP (servidores) que controlam o acesso remoto à sua MIB local. Os softwares gerentes

[1] Por esse motivo optamos em descrevê-la ao invés da abordagem da ISO.

requisitam dos agentes SNMP informações sobre o estado dos dispositivos da rede através de primitivas do protocolo SNMP para troca de mensagens. Essas primitivas constituem em três tipos básicos de operações: operadores *set* para alterar valores de uma variável, *get* para obter o valor de uma variável, e o operador *getnext* para continuar examinando as variáveis seguintes da MIB. Os agentes possuem estrutura muito simples e não executam ações de gerenciamento em seus dados locais. O máximo de iniciativa que chegam a tomar é o envio de *traps* (mensagens ao NMS) não como resultado de uma requisição, mas quando algum evento específico (por exemplo, a mudança repentina do estado de um componente de "ativo" para "inativo") acontece. O agente simplesmente comunica a estação do evento ocorrido e esta fica encarregada de executar a ação de gerenciamento adequada (Figura 2.3). Essa interação CS típica leva à geração de um alto tráfego e sobrecarga da estação de gerenciamento, onde é realizada toda a computação de fato.

Essa forma centralizada de se realizar o gerenciamento da rede descrita acima é bastante apropriada para aplicações com pouca necessidade de controle distribuído, aplicações que não requerem acesso a variáveis da MIB freqüentemente, ou que necessitem apenas de uma quantidade limitada de informações. Por exemplo, o estado da interface de um roteador ou o estado de um link requer a visualização de um número limitado de variáveis da MIB. Entretanto existem aplicações que necessitam de computação sobre uma grande quantidade de informações. Um exemplo seria a computação de uma função que indica o nível de funcionalidade da rede, que deve freqüentemente detectar as variações de um alto número de variáveis da MIB. Nesses casos, o monitoramento e controle deveriam ser um pouco mais distribuídos.

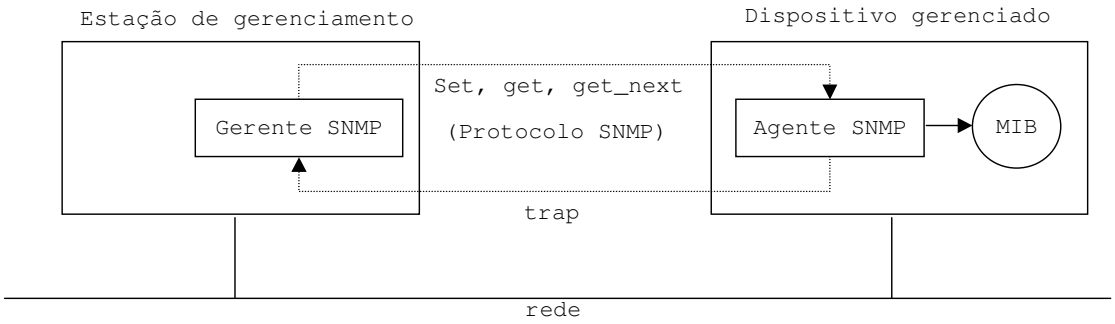


Figura 2.3 - Interação cliente-servidor típica entre um NMS e dispositivo gerenciado.

Os problemas da abordagem centralizada ficam mais evidentes em períodos de congestionamento da rede, quando a ação do gerente é crucial. Durante esses períodos, as operações de gerenciamento requeridas para solucionar o problema do congestionamento causam, por sua vez, um aumento no tráfego perto da área congestionada da rede. Isto leva a um ciclo vicioso que pode ser nefasto para a rede.

Na busca por uma forma de gerenciamento menos centralizada, a IETF desenvolveu o SNMPv2 [61] onde é introduzido o conceito de *proxy agent*, que provoca uma forma de gerenciamento hierárquico. Um agente desse tipo age como cliente dos agentes dos nós, enviando comandos e coletando as respostas, e como servidor da NMS, respondendo a requisições (Figura 2.4).

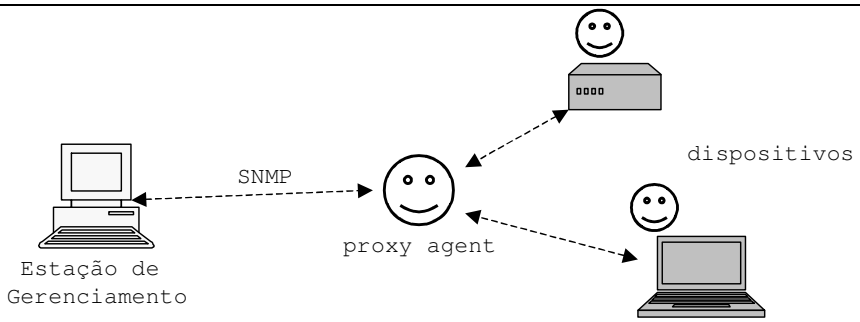


Figura 2.4 - SNMPv2 proxy agents.

Outra abordagem para descentralização proposta pela IETF foi a introdução do RMON (*Remote MONitoring*) [70]. RMON assume a existência de dispositivos apropriados chamados *probes* (ou *monitors*), cuja função é informar sobre o tráfego de rede em uma sub-rede específica, e sobre o estado de alguns dispositivos. Monitorando o tráfego de pacotes e analisando os cabeçalhos, os monitores provêem informação sobre links, conecções entre máquinas, padrões de tráfego, e status dos dispositivos. Por este motivo, RMON pode ser encarado como uma abordagem orientada ao tráfego. Alguns dos processamentos que antes eram feitos nas NMSs, no RMON são executadas por agentes localmente e independentemente da estação. O agente nos *probes* é configurado para capturar amostras estatísticas periodicamente de parâmetros relevantes e, sempre que um *threshold* é ultrapassado, notificar a NMS (Figura 2.5). O grande benefício do RMON é a *compressão semântica* de dados, pré-processando toda a informação coletada, e enviando apenas as que forem significantes para a NMS.

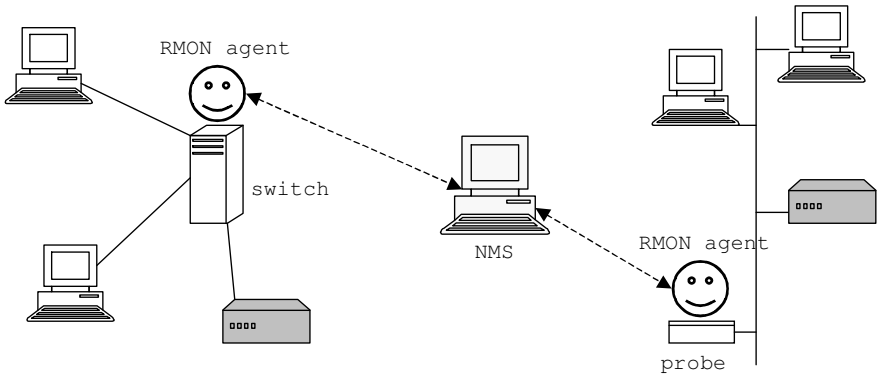


Figura 2.5 - RMON probes.

Recentemente, a Sun Microsystems desenvolveu uma API JAVA para gerenciamento de redes, a *Java Management Extensions (JMX)* [32]. O modelo JMX foi desenvolvido para permitir que a arquitetura fosse escalável nos diversos tipos de plataformas existentes em um ambiente corporativo. O RMI (*Remote Method Invocation*) é utilizado como mecanismo de troca de informações entre os elementos de gerência do sistema (gerentes e agentes).

2.2 A Administração de Sistemas e o Mercado

A tendência de integração de atividades de administração tradicional (como monitoração de sistemas, distribuição de software, *backup* de arquivos, segurança) e de gerenciamento de redes (que conta com uma taxonomia mais definida e mais padronizada dado os esforços de organismos de pesquisa citados na seção anterior), se reflete principalmente na evolução das diversas ferramentas de gerência disponíveis no mercado. Estas soluções fazem parte de um conjunto de ferramentas integradas que realizam desde operações de *backup,* até controle de roteadores, *switches* e outros dispositivos SNMP[2]. Assim, esse *framework* complexo permite monitorar os recursos da rede, distribuir software, fazer correlação de eventos e analisar de mensagens de *log*.

2.2.1 Ferramentas conhecidas

A seguir tem-se uma relação de algumas ferramentas existentes atualmente no mercado da administração de sistemas. A dificuldade de acesso a informações detalhadas sobre esses sistemas, a inexistência de artigos científicos, restando apenas fontes não confiáveis (publicidade), impossibilitaram que fossem feitas comparações mais criteriosas entre as ferramentas. Assim, apenas uma breve descrição é feita, apontando características peculiares de cada uma, contexto de gerenciamento a que se aplica, e relevância atual no mercado de administração.

O *Satool* [49] é uma ferramenta de monitoração de um grande número de máquinas em ambientes distribuídos. É composta por três componentes distintos: um agente SNMP que realiza coleta de dados na MIB das máquinas clientes e uma interface gráfica (GUI) com o administrador. A MIB do *Satool* é uma MIB estendida para serem incluídas variáveis de interesse dos administradores de sistemas UNIX. Entretanto, a introdução de novas variáveis a serem monitoradas requer alteração no código. A ferramenta

[2] Dispositivos SNMP é a denominação dada aos dispositivos de rede que já são fabricados com os agentes que falam o protocolo SNMP.

apresenta uma queda de desempenho quando do aumento do número de máquinas a serem monitoradas.

O sistema *Pong* [24] oferece a monitoração de um conjunto fixo de serviços (NIS, NFS, inetd e ping [20]), não sendo permitido a inclusão de outros recursos e serviços a serem monitorados.

O ambiente *CINEMA* [65] é um mecanismo de apoio à operação de gerenciamento de uma rede, rastreando e resolvendo problemas relacionados à performance, roteamento e segurança. São utilizados agentes SNMP para coleta de informações de gerência nos dispositivos da rede.

Big Brother [45] é um monitor de redes para sistemas UNIX baseado em *Web* e faz monitoração das seguintes áreas: conectividade de rede, uso de disco e CPU, número de usuários, controle de processos, servidores *Web* e mensagens de *log*. As seguintes funcionalidades são encontradas na ferramenta *Big Brother*: visão do estadode todas as máquinas da rede, acesso a dados reais a partir de sistemas remotos, notificação se alguma falha ocorrer, capacidade de acesso às informações a partir de qualquer ponto e capacidade de integrar as informações de outros pacotes. Entretanto a ferramenta não trata da determinação da solução e correção de problemas, e não permite extensão automática; a inclusão de uma nova característica a ser monitorada requer alteração no código e, provavelmente, na arquitetura da ferramenta.

A *Computer Associates' Unicenter TNG* [12] é uma solução de gerenciamento integrada que permite que organizações gerencie os recursos em redes, sistemas, aplicações e base de dados heterogêneas. Engloba as funções de descoberta de redes, topologia, gerenciamento de performance, eventos e monitoração de status, segurança, e distribuição de software. A tecnologia de implementação da solução gerente/agente empregada permite escalabilidade tanto de software quanto de hardware. É uma ferramenta bastante utilizada hoje em dia.

O *SunNet Manager* da Sun Microsystems [63] oferece um ambiente amigável para sistemas independentes do protocolo e soluções de gerenciamento de redes. Apresenta

uma interface de usuário gráfica aberta e intuitiva. Oferece, também, um conjunto de ferramentas gráficas para descoberta automática, isolamento de falhas, diagnóstico e análise de desempenho. Agentes *Proxy* utilizam *Remote Procedure Call* (RPC) para se comunicar com os agentes de coleta localizados nos dispositivos gerenciados e permitem a compatibilidade de diferentes protocolos de gerência, traduzindo-os para o protocolo de gerenciamento da rede em questão. O *SunNet Manager* é baseado em redes TCP/IP e roda no ambiente operacional UNIX.

Ou ferramenta de gerenciamento bastante importante é a *HP OpenView* [27]. É um pacote de gerenciamento para redes TCP/IP, Netware e SNA (*System Network Architecture*) da IBM. Realiza mapeamento de toda a rede, incorpora todos os dispositivos com suporte a SNMP, configura ações automaticamente, define prioridades em categorias de eventos, possui uma interface gráfica de gerenciamento, e realiza relatórios de administração em tempo real. O poder da ferramenta, sem dúvida, está na grande base de aplicações de terceiros e grande desempenho.

Por fim, o *TME 10* (*Tivoli Management Environment*) [66] é um conjunto de ferramentas agrupadas modularmente de acordo com as necessidades de gerenciamento do sistema, isto é, funções de gerenciamento desejadas. O *TME 10* possui funções de gerenciamento de sistemas de computação nas seguintes categorias: *gerenciamento de implantação* (distribuição de software, configuração e instalação), *gerenciamento de disponibilidade* (monitoração centralizada do ambiente, automação de procedimentos e a avaliação de desempenho), *gerenciamento de segurança* (acesso seguro dos usuários a múltiplos sistemas, banco de dados, serviços e aplicações através de visões unificadas dos usuários), *operação e administração* (facilidades para escalonamento de processos, backup, recuperação de dados e gerenciamento de impressão) e *integração* (meios para integração com aplicações de gerenciamento de outros fabricantes).

2.2.2 Problemas

O grande problema dessas ferramentas de administração é que elas se baseiam em uma abordagem extremamente centralizada, sobrecarregando em demasiado o administrador humano. Normalmente, a estação de gerenciamento, operada pelo administrador, é responsável por realizar o monitoramento do *status* de funcionamento de todos os dispositivos em rede, e tomar ações de prevenção e de correção de eventuais problemas. A maior parte dessas ações são ainda extremamente dependentes da intervenção do administrador na estação.

Essa característica de controle centralizado também causa problemas de tráfego de rede, gera sobrecarga da estação de gerenciamento, provê pouca robustez e é mais susceptível a falhas [2]. Ela é conseqüência direta dos protocolos (SNMP e CMIP) sob os quais as ferramentas são construídas.

Por fim, como essas abordagens utilizam um conceito de *agentes* muito simplista, os problemas citados acima limitam as possibilidades de soluções. Os agentes, que se localizam nos dispositivos da rede, funcionam simplesmente como processos (*daemons*) que verificam o *status* dos dispositivos e enviam mensagens à estação de gerenciamento em resposta a alguma mudança repentina de funcionamento dos mesmos, por exemplo. Talvez, dotados de maior poder de raciocínio e de outras propriedades, os agentes poderiam dar uma maior contribuição para uma administração mais distribuída, flexível e menos penosa para o gerente.

Existem outros problemas mais específicos. O primeiro deles é que fabricantes já fornecem incorporada ao sistema operacional, alguma ferramenta de administração proprietária destinada a usuários inexperientes, e contemplam apenas redes homogêneas. Algumas ferramentas desenvolvidas por terceiros, em ambientes acadêmicos e comerciais, contemplam aspectos específicos da administração de sistemas em ambientes heterogêneos, tais como administração de usuários, distribuição de software, gerenciamento de recursos, segurança, monitoração e análise de desempenho e configuração dinâmica. Entretanto, são projetadas de forma totalmente

ad hoc, sem qualquer tipo de padronização ou integração, tornando-se difíceis de serem expandidas e integradas. Outro problema é a existência de poucas ferramentas de monitoração que utilizam a Web, impossibilitando a gerência através de um *host* qualquer da Internet. Além de todos esses detalhes de projeto e implementação peculiares, o custo para comprar e instalar esses sistemas e treinar administradores para operá-los costuma ser bastante alto.

2.3 Conclusões

O gerenciamento corporativo de sistemas é uma tarefa de extrema importância hoje em dia, da qual pode depender o futuro dos negócios de uma organização e é fato que a gerência de uma rede não é das tarefas mais simples.

Entretanto as ferramentas de administração atuais são caracterizadas por um controle muito centralizado do gerenciamento, sobrecarregando tanto o administrador quanto a rede. Essa característica pode ter como umas das causas o fato do conceito de agentes utilizado ser muito simplista. Uma abordagem de agentes mais sofisticados, que possibilite mobilidade entre os dispositivos da rede e maior grau de autonomia, pode ser interessante e tem já sido investigada como será discutido no próximo capítulo.

3

Gerenciamento Corporativo de Sistemas baseado em Agentes

> ***agent*** n. *a person who acts on behalf of another person, business or government, etc.* (The Collins English Dictionary)

O conceito de *agente* utilizado até então é designado para definir entidades que possuem funções simples de acesso a bases de dados para verificação do estado de algumas variáveis e informá-los ao sistema gerente (modelo de gerenciamento com agentes SNMP descrito na seção 2.1.2). Esses agentes são extremamente simples, tendo como a reatividade de suas ações, a característica mais importante.

É importante se utilizar uma noção de *agentes* mais completa que englobe características como raciocínio, mobilidade, coordenação, planejamento, etc. Agentes com tais características podem auxiliar bastante o administrador da rede, provendo um

maior grau de distribuição do gerenciamento nas ferramentas. Por exemplo, um agente pode ficar responsável por monitorar o nível de utilização da CPU de uma determinada máquina e ter o conhecimento especialista necessário e autonomia suficiente para tentar resolver um eventual problema de super utilização localmente, sem necessidade de intervenção da estação de gerenciamento. Isso diminuiria o tráfego de rede gerado e a sobrecarga de processamento na estação.

Neste capítulo serão aprofundados os conceitos de *mobilidade, raciocínio* e *coordenação* em agentes e como estas três características podem auxiliar a construção de sistemas de agentes mais poderosos que contribuam para uma maior descentralização do GCS. Trabalhos de pesquisa com sistemas multi-agentes especialistas, agentes móveis, ou que de alguma forma procuram contribuir nesse sentido, com técnicas de Inteligência Artificial [57], também são abordadas neste capítulo.

3.1 Mobilidade

O modelo clássico de comunicação entre computadores de uma rede é o modelo cliente-servidor (CS). Neste modelo, as entidades envolvidas possuem papéis fixos e bem definidos; um servidor oferece serviços e um cliente faz uso desses serviços (Figura 3.1). O mecanismo de comunicação utilizado entre clientes e servidores pode ser feito tanto através de protocolos de passagem de mensagens quanto por RPC, ou Objetos Remotos.

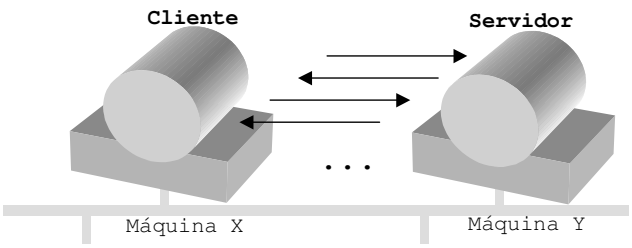


Figura 3.1 - Modelo cliente-servidor (CS).

O conceito de *mobilidade de código* surgiu como uma nova alternativa ao modelo CS. A idéia básica é a de conseguir diminuir o alto número de interações entre máquinas clientes e servidoras, provendo melhorias no que diz respeito a pelo menos dois fatores: *flexibilidade do servidor e uso da largura de banda.*

Em aplicações CS, o conjunto de serviços oferecidos pelo servidor é fixo, definido *a priori* pelo designer da aplicação, e acessível através de interfaces definidas estaticamente. Por exemplo, para o caso específico do gerenciamento corporativo de sistemas, se o protocolo, a estrutura da base de informação, ou a funcionalidade do agente for modificada, o agente tem de ser estendido e reconstruído.

Aplicações que envolvem interação CS contínua e intensa através da rede podem causar consumo de largura de banda. Novamente, este é um dos grandes problemas do micro gerenciamento com SNMP onde a NMS consulta continuamente o agente SNMP, consumindo largura de banda, e aumentando a sobrecarga da rede.

Paradigmas de código móvel não fazem ligação estática de códigos responsáveis por serviços a um ou mais *hosts,* mas permitem que esse código *migre* (possivelmente com o estado de execução) para um *host* diferente, fazendo com que toda troca de informações entre um cliente e um servidor seja feita localmente, independente de comunicação via rede.

Pelo menos três paradigmas de código móvel podem ser identificados na literatura: *Remote Evaluation (REV)* [62]*, Code on Demand (COD)* [21], *Mobile Agent (MA)* [11]. A diferença entre eles está na localização dos diferentes componentes antes e depois da execução do serviço, no componente computacional que é responsável pela execução do código, e no local onde a computação é realmente executada (Tabela 3.1).

Na tabela, M1 e M2 representam duas máquinas distintas de uma rede, Cx e Cy representam dois componentes computacionais distintos, KH (know-how) é o código em si, R representa os recursos (arquivos, *driver* de dispositivo de rede, *driver* de impressora, etc.) necessários para se realizar o serviço. Os componentes computacionais responsáveis pela execução real do código estão em negrito.

Componentes em realce são os que migraram. Um exemplo de componente computacional é um processo ou *thread*. Eles são caracterizados por um estado, que inclui dados privados, o estado de execução, e ligações com outros componentes, em particular com código (conhecimento necessário para execução) e recursos.

Paradigma	Antes						Depois					
	M1			M2			M1			M2		
CS			Cx	KH	R	Cy			Cx	KH	R	Cy
REV	KH		Cx		R	Cy			Cx	KH	R	Cy
COD		R	Cx	KH		Cy	KH	R	Cx			Cy
MA	KH		Cx		R					KH	R	Cx

Tabela 3.1 - Paradigma CS e paradigmas de Código Móvel

Para um melhor entendimento das diferenças entre os três paradigmas, Carzaniga [6] faz uma analogia bastante interessante com um cenário da vida real: duas pessoas interagem e cooperam para fazer um bolo de chocolate. Para que o bolo (resultado do serviço) possa ser feito, é necessária uma receita (código do serviço), ingredientes (os recursos que podem facilmente ser movidos), um forno (recurso que tem dificuldade em ser movido), e uma pessoa para misturar os ingredientes seguindo a receita (componente computacional responsável pela execução do código). É claro que para que seja possível preparar o bolo (executar o serviço), todos os componentes têm de estar na mesma casa (máquina). A Figura 3.2 ilustra como as duas pessoas cooperariam seguindo a filosofia de cada um dos paradigmas (CS, COD, REV, e MA), respectivamente.

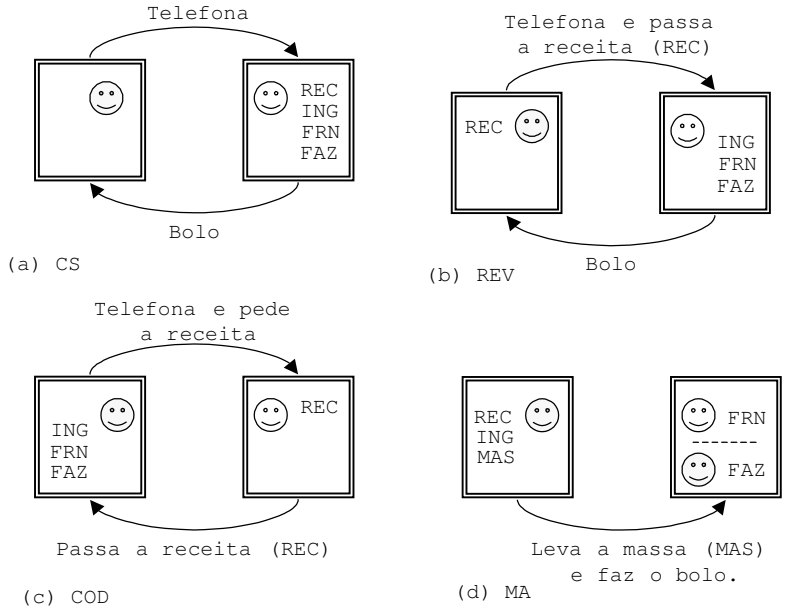


Figura 3.2 - Metáfora para paradigmas de código móvel e CS[1].

3.1.1 Agentes Móveis (MA)

Em particular, a tecnologia dos agentes móveis é de maior interesse para a dissertação, dado o grande número de trabalhos existentes sobre como se abordar o gerenciamento de redes com seu uso.

Agentes móveis podem ser considerados como processos computacionais que estão "livres" das restrições impostas por modelos de comunicação como CS ou qualquer um outro baseado em mobilidade de código[2] listado na seção anterior. Pode-se dizer que modelos CS assumem que o importante é *migrar os dados* para o local onde se encontra o programa que irá processá-los, enquanto que o paradigma dos MAs assume

[1] Duas pessoas interagem e cooperam para fazer um bolo de chocolate. A figura ilustra como o processo funcionaria em cada dos paradigmas de código móvel e também no modelo CS tradicional. REC = Receita, ING = Ingredientes, FRN = Forno, FAZ = onde o bolo é feito.

[2] Tanto na abordagem REV quanto na COD, ainda há uma certa dependência de um controle mais centralizado.

que o ideal é *migrar o programa* para o local onde se localizam os dados. Eles podem, indefinidamente, migrar para o local do serviço desejado (Figura 3.3) levando ambos os dados coletados e estado de execução, e o código de execução do serviço. Se forem comparadas a Figura 3.3 e a Figura 3.1, percebe-se a diferença entre os dois modelos de comunicação: as interações deixam de ser feitas remotamente, gerando tráfego de rede, por vezes, acentuado, para serem feitas localmente onde se localiza o serviço.

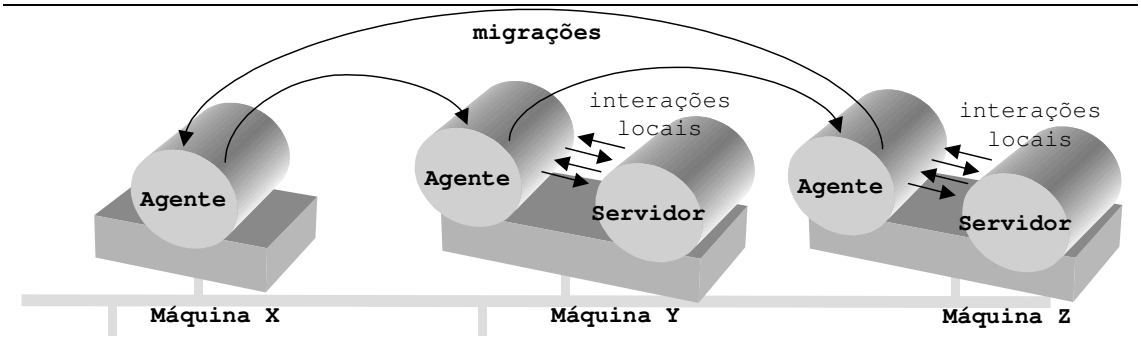


Figura 3.3 - Modelo de comunicação baseado em Agentes Móveis.

Uma vez que as interações ocorrem localmente nas máquinas onde se localizam os serviços, não apenas o tráfego de rede é reduzido, mas a sobrecarga de trabalho no servidor também diminui. Além disso, MAs podem rapidamente se adaptar a mudanças tanto no estado do programa quanto no ambiente da rede onde executam (*hosts* podem ser desconectados, por exemplo) para mudar seu itinerário. Essa característica pode ser extremamente importante para o caso de redes não muito estáveis, ou para utilização de laptops, por exemplo, que são conectados e desconectados com freqüência.

Bieszczad [5] faz um levantamento de alguns benefícios potenciais que se pode ter do uso de agentes móveis. A Tabela 3.2 mostra um compacto desses benefícios e respectivas justificativas.

Harrison [23] discute mais sobre vantagens e desvantagens em se utilizar agentes móveis para o desenvolvimento de aplicações distribuídas, e sobre os requisitos básicos

(comunicação, persistência, autenticação, segurança, privacidade, etc) que devem ser atingidos por sistemas que os implementam.

O uso de agentes móveis, além de vantagens técnicas individuais, pode prover um ambiente de trabalho generalizado, aberto e difuso para o desenvolvimento e personalização de novos serviços em rede.

Benefícios	Por que?
Eficiência	Consumo limitado de CPU, já que MAs executam em apenas um nó por vez.
Espaço em Disco	Consumo de recurso é limitado, pelo mesmo motivo anterior. Um MA carrega consigo a funcionalidade evitando, assim, a duplicação de código. Objetos Remotos provêem benefício similar, entretanto o existe o *drawback* do *middleware* necessário [68].
Tráfego de Rede	Código é geralmente menor que dados a serem processados, portanto a migração de MAs para a fonte dos dados gera menos tráfego que transferir os dados. Objetos Remotos podem prover benefício similar, mas envolvem *marshalling* de parâmetros (que podem ser grandes).
Interação Assíncrona	MAs podem ser delegados para executar determinadas tarefas mesmo que a entidade responsável não permaneça ativa.
Tolerância a falhas	Se um SD começar a dar problema, um MA pode ser usado para aumentar a disponibilidade de certos serviços. Por exemplo, a densidade de agentes para detecção ou reparação de falhas pode ser aumentada.
Suporte a ambientes heterogêneos	MAs são separados dos *hosts* em que executam apenas pelo *framework* de mobilidade[3].

Tabela 3.2 - Possíveis benefícios do uso de Agentes Móveis.

3.1.2 *Framework* para Agentes Móveis

Para que se possa fazer uso de Agentes Móveis, uma rede de computadores tem de incorporar um *Framework para Agentes Móveis* (MAF). Cada máquina da rede tem de prover ambiente de execução seguro e de proteção aos agentes móveis. Estes *servidores de agentes móveis* executam o código do agente e provêem operações primitivas para programadores de agentes, como as que permitem que agentes possam migrar, se

[3] O custo de se rodar um JVM (*Java Virtual Machine*) [42] (que é a base para muitos *frameworks* de mobilidade) está diminuindo.

comunicar, ou acessar recursos de um *host*. Uma rede lógica de servidores de agente constitui um MAF (figura 3.4).

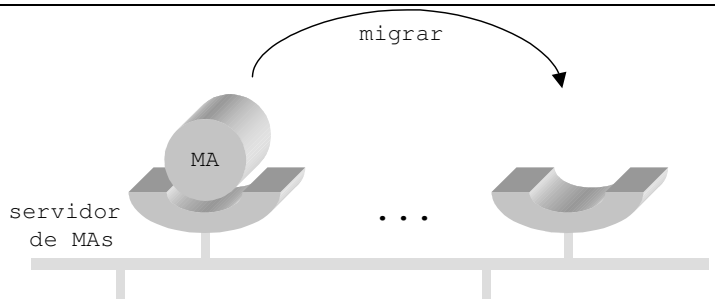


Figura 3.4 - MAF: Rede lógica de servidores de agentes móveis.

Esta seção resume algumas questões que devem ser levadas em consideração quando do design de sistemas de agentes móveis: *suporte à mobilidade*, *esquema de nomeação*, *questões sobre segurança* e *questões sobre linguagens de programação de agentes*.

Suporte à mobilidade

Uma característica básica de um agente móvel é a capacidade de *migrar* de um *host* para outro. Um agente pede que seu servidor o transporte para um outro *host* remoto, e o servidor deve então desativar o agente, capturar seu estado, e transmiti-lo ao servidor da máquina remota. O servidor de destino deve então, restaurar o estado do agente e reativá-lo, completando a migração. O estado de um agente inclui seus dados e o estado de execução de seu *thread,* que é representado pelo seu contexto de execução e a pilha de chamadas. Se isso puder ser capturado e transmitido com o agente, o servidor de destino pode reativar o *thread* no exato ponto onde a migração foi iniciada, o que pode ser interessante para sistemas tolerante a falhas. Entretanto, a maioria dos MAF utiliza as máquinas virtuais ou ambientes de linguagens, que não provêem captura do estado ao nível de *thread*.

Outra questão na implementação de mobilidade de agentes é a transferência do código do agente. Numa abordagem, o agente carrega todos os seus códigos quando faz a

migração, o que permite que ele rode em qualquer servidor apto a executar o código. Outros sistemas não permitem que o código seja transferido e requerem que todo o código necessário seja pré-instalado no servidor de destino. Numa terceira abordagem, o agente não carrega seu código, mas contém uma referência para sua base de códigos – servidor que provê o código quando o agente requisitá-lo[4].

Nomeação

É necessário um esquema de *nomeação* para identificar unicamente as várias entidades constituintes de um MAF – agentes, servidores de agentes, recursos, e usuários. Por exemplo, um agente deve ter um nome exclusivo para que o agente servidor que controla uma eventual migração sua possa comunicar-se com ele durante a viagem. O sistema deve prover um mecanismo para encontrar a localização atual de uma entidade, dado o nome desta – *mecanismo de resolução de nomes*. Alguns MAFs utilizam um esquema de nomeação dependente da localização - o que facilita a implementação de resolução de nomes, mas dificulta o trabalho da aplicação em rastrear um agente quando duma migração, já que seu nome vai ser modificado [28], [22], [34]. Outros implementam um esquema de transparência de localização a nível de aplicação, provendo *proxies*[5] locais para entidades remotas, ou ainda, um *serviço de nomes* que mapeia um nome simbólico para a localização atual da entidade [52].

Questões sobre segurança

A introdução de mobilidade de código em uma rede provoca uma série de questões sobre segurança [69]. Agentes maliciosos[6] podem causar consumo desordenado de recursos, danificando a utilização dos mesmos por outros agentes, da mesma forma que

[4] Esta abordagem se assemelha muito ao mecanismo COD descrito anteriormente.

[5] Espécie de representante local de uma entidade remota.

[6] Denominação comumente encontrada na literatura para designar agentes que possuem códigos não confiáveis, análogos a vírus ou Cavalos de Tróia.

hosts maliciosos podem causar prejuízos a eventuais agentes. Questões de segurança são relacionadas em três categorias principais:

- *Privacidade e integridade do agente*: Um agente de compras[7], por exemplo, carrega o número do cartão de crédito de um usuário da Internet. O protocolo de transporte do agente precisa prover privacidade, para prevenir que outros usuários da rede tomem conhecimento das informações;

- *Autenticação de agentes e de servidores*: Quando um agente tenta se transportar para um servidor remoto este precisa averiguar a identidade do destino, para poder decidir sobre os direitos e privilégios a garantir ao agente. Da mesma forma, quando um agente migra para um servido, ele precisa de certeza sobre a identidade do servidor antes de revelar-lhe seus dados. Sistemas de *assinatura digital* [58] estão sendo usados e integrados a protocolos de transporte de agentes;

- *Autorização e controle de acesso:* Restrições de acesso[8] devem ser impostas aos agentes pelos *hosts* nos quais esses irão executar, ou mesmo codificadas no próprio estado do agente.

Questões sobre linguagens de programação de agentes

Sendo agentes entidades que podem migrar entre máquinas de ambientes heterogêneos, inclusive de sistemas operacionais distintos, a portabilidade do código do agente é um requisito primordial. Conseqüentemente, a maioria dos MAFs são baseados em linguagens de programação interpretadas que provêem máquinas virtuais para execução do código dos agentes. Linguagens que suportam *type checking*, *encapsulamento*, e *restrição de acesso à memória* são particularmente adequadas para implementação de servidores de agentes. MAFs podem diferir significantemente no modelo de programação utilizado para codificar os agentes. Em alguns casos, o programa agente é apenas um script, com pouco ou nenhum controle de fluxo. Em outros, a linguagem de script possui características da programação orientada a objetos e suporta

[7] Comum em implementações de solução para comércio eletrônico.

[8] Restrições do tipo: "o agente só possui acesso de leitura e escrita a um arquivo particular", ou "o agente não pode criar conecções de redes", ou ainda, "o agente pode utilizar 1 Mbyte de espaço em disco e criar conecções apenas em hosts no domínio xxx.com".

extensivamente um controle de fluxo procedimentar. Outros sistemas modelam a aplicação baseada em agentes como um conjunto de objetos distribuídos que interagem, cada qual com seu próprio *thread* de controle, permitindo, assim, a migração através da rede.

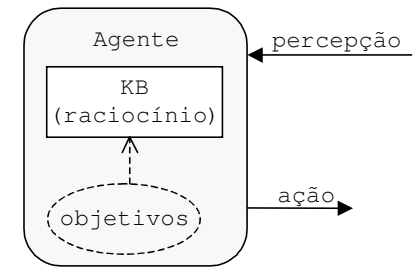
Outra questão importante é das primitivas de programação para implementação de aplicações baseadas em agentes. São necessárias primitivas para o g*erenciamento básico de agentes* (criação, despacho de agentes, clonagem, e migração), c*omunicação e sincronização entre agentes,* c*ontrole e monitoramento de agentes* (consultas ao status de um agente, *recall* e término de execução), *tolerância a falhas* (checkpoint, tratamento de exceção, e auditorias), e *segurança* (criptografia, assinatura, proteção de dados). Uma discussão mais completa sobre todas essas questões é feita na literatura [35], [15] incluindo um conjunto de mecanismos, abstrações e conceitos de linguagens de programação que podem ser utilizados para analisar, comparar, ou desenvolver linguagens de código móvel.

3.2 Raciocínio

> ***raciocínio:*** Filos*. Processo discursivo pelo qual de proposições conhecidas ou assumidas se chega a outras proposições a que se atribuem graus variados de verdade.* (Novo Dicionário Aurélio)

Ainda não existe uma definição consensual de agente inteligente. Para os propósitos da dissertação, agente inteligente é definido como uma entidade computacional que tem a capacidade de reagir a mudanças de estado de seu habitat (ambiente de execução), tomando decisões de acordo com seu conhecimento sobre o mundo. Isto é, o agente é uma entidade com capacidade de raciocínio que utiliza a lógica para manter uma descrição do que percebe de seu ambiente e deduzir um curso de ações para atingir seus objetivos.

Esse agente tem sido classicamente implementado através de mecanismos de inferência dedutiva (ou indutiva[9] [47]), que permitem que o agente escolha as ações mais apropriadas a partir de suas *percepções*, de seu *objetivo* e de seu *conhecimento* [57] (Figura 3.5). Esse conhecimento é descrito por um grupo de sentenças declarativas em uma linguagem de representação do conhecimento e armazenado em uma *base de conhecimento* (KB). Um grupo de pré-condições deve ser satisfeito para que ações de correção sejam disparadas de acordo com regras, denominadas de *Regras de Produção* O mecanismo que encontra a ação de correção a partir da informação adicionada é implementado por um *Motor de Inferência* [57] (ver seção 5.3.4).




```
Objeto agente {

   atributos
      KB, t (momento);

   métodos
      fato <- percebe(ambiente,t);
      ação <- raciocina(KB,fato,t);
      age(ação,ambiente,t);


}
```

Figura 3.5 - Agente Genérico - percebe o ambiente, raciocina e dispara a melhor ação.

Não cabe no escopo deste trabalho detalhar mais as formas, propriedades e implementações de mecanismos de inferência, visto que se trata de um tema vastamente coberto na literatura clássica da IA.

Essa capacidade de raciocinar logicamente é uma característica extremamente importante para se obter uma maior grau de *autonomia* para o agente. Dando-se autonomia ao agente para que aja segundo suas próprias decisões, pode-se potencializar as vantagens da mobilidade de agentes no GCS de várias maneiras:

[9] Aprendizagem. A aprendizagem é um resultado da interação entre o agente e o mundo (sobre o qual possui conhecimento incompleto), e de sua própria experiência prévia de tomada de decisões nesse

- Reduzindo a necessidade da inteligência humana ou qualificações pessoais durante a instalação e operação da solução de gerenciamento;
- Agentes móveis autônomos podem solucionar os problemas dos dispositivos localmente sem a necessidade de qualquer contato com a NMS;
- O agente pode, por exemplo, decidir que seqüência de nós deve visitar baseado num relatório de suas viagens prévias;
- Decidindo que nós podem permanecer um tempo maior sem ser gerenciado, reduzindo o número de eventuais migrações. Se um determinado dispositivo, historicamente, possuir uma baixa incidência de problemas, este dispositivo pode ser visitado com menor freqüência.

A autonomia potencializa o poder da mobilidade já que agentes móveis devem operar normalmente mesmo não havendo conexão com a NMS. Além disso, muitos problemas não são muito bem ajustados a soluções procedimentais. Agentes podem precisar se mover de dispositivo em dispositivo, acumulando informações, e decidindo o destino imediato baseado nestas informações.

3.2.1 Coordenação

Para que um sistema de automação de gerência de rede siga a tendência de descentralização que hoje é altamente recomendada [2], seria preciso pensar não apenas em um único agente, mas em um grupo deles, os chamados *sistemas multiagentes* [41]. De fato, uma das situações em que se pode optar por uma solução multiagente, em detrimento do agente único, é quando o problema requer a ação de entidades fisicamente distribuídas, o que é o caso da administração de sistemas em redes. Por exemplo, uma ferramenta de gerenciamento de uma rede com diversos dispositivos distintos, tem de implementar agentes inteligentes com conhecimento particular sobre os dispositivos onde irão atuar. Esses agentes, não necessariamente, precisam cooperar ou mesmo ter conhecimento da existência de outros. Suas funções são específicas e eles possuem todo o conhecimento necessário para cumprir seus objetivos.

---

mundo.

Em alguns casos, entretanto, os agentes necessitam de *cooperação*. Essa necessidade de interação entre agentes, normalmente, ocorre quando os agentes são designados para resolver problemas que são interdependentes, seja porque necessitem concorrer por determinados recursos computacionais, seja por causa dos relacionamentos entre os subproblemas. Esses relacionamentos resultam de duas situações básicas referentes à decomposição natural da solução para o problema domínio em subproblemas. A primeira situação é quando os subproblemas são do mesmo tipo, mas agentes diferentes possuem métodos ou dados alternativos que podem ser utilizados para gerar a solução. A outra ocorre quando dois subproblemas são parte de um problema maior cuja solução requer que certas restrições existam entre as soluções para seus subproblemas; é o caso, por exemplo, quando os resultados de um subproblema são necessários para que o outro possa ser resolvido. A resolução de uma equação matemática é um exemplo bastante simples, mas válido dessa segunda situação (Figura 3.6).

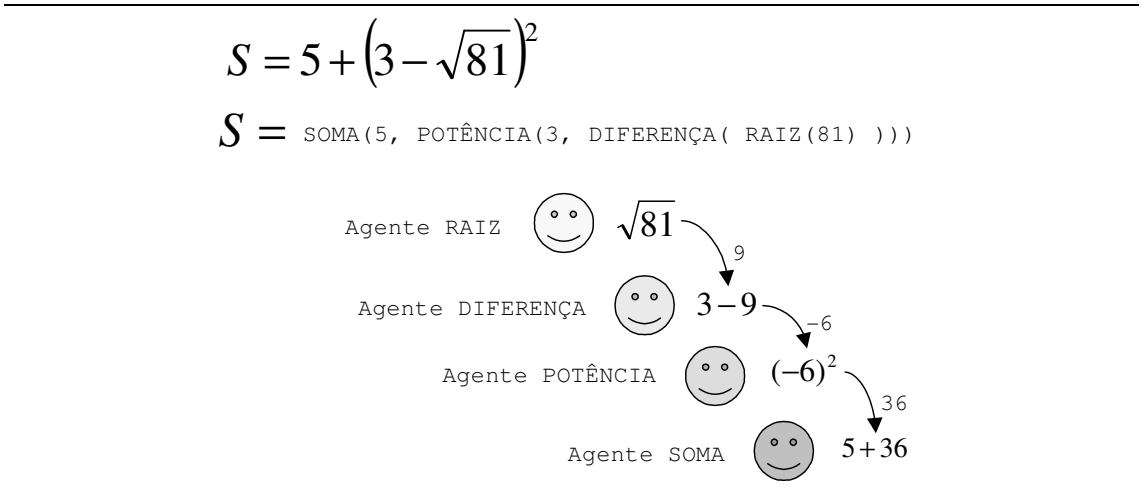


Figura 3.6 - Agentes cooperam para resolver uma equação matemática.

Para que agentes possam cooperar, fazem-se necessários mecanismos de *coordenação* do grupo. Um dos problemas chave em sistemas multi-agentes cooperativos é como coordenar os agentes para que o objetivo global do sistema seja alcançado.

A coordenação dos agentes é baseada na natureza dessas interdependências, no estado corrente do agente durante a resolução do problema, e no estado do ambiente de execução do agente. A falta de coordenação ou a coordenação inapropriada pode contribuir para que grupos de agentes gerem soluções que são sub-ótimas, consumindo tanto recursos computacionais quanto de comunicação desnecessários, com resultados redundantes ou, simplesmente, não consigam chegar a resultado interessante algum.

Muitas pesquisas científicas focam o desenvolvimento de princípios computacionais e modelos para construção, descrição, implementação e análise dos padrões de interação e coordenação em sistemas multiagentes, de acordo com protocolos de interação embutidos dentro dos próprios agentes [29]. Propostas recentes consideram essa interação como uma questão chave no desenvolvimento de sistemas multiagentes [78], a ser enfrentada a partir da fase de análise preliminar.

Coordenação pressupõe *comunicação* e muitos esforços na área lidam atualmente com o problema de estabelecer um bom meio de comunicação entre os agentes: *Agent Comunication Languages* (ACLs) [60] (como KQML [17]ou FIPA [19]), permitindo que o conhecimento seja transferido de um agente para outro, componentes *middleware* (mediadores [73]) e infraestruturas (como ORBs), provendo comunicação com algum grau de transparência de acordo com ontologias, serviço de nomes de agentes (ANS), roteadores de mensagens (AMR) [33]e [31], etc.

Outros trabalhos possuem uma visão mais holística do sistema e introduzem noções como *social agency* [60], *socialware* [25], e *social laws* [59]. Essas abordagens assumem que para se ter um completo entendimento de sistemas multiagentes faz-se necessário uma modelagem não só dos agentes que compõem o mundo, mas do próprio mundo como uma sociedade onde agentes convivem e se relacionam, e que possuem regras sociais a serem seguidas, ou seja, uma abordagem baseada em modelos de coordenação. Ciancarini [10] mostra como uma abordagem desse tipo pode ajudar no desenvolvimento de sistemas multi-agentes.

Um exemplo bem claro de sistema multiagente cooperativo é a implementação de um time de futebol. Cada jogador possui um conhecimento individual e limitado (posição em campo), características diferentes das dos demais (atacante, zagueiro, goleiro, etc.) e não pode resolver o problema sozinho (visto que futebol é um esporte eminentemente coletivo). Os jogadores agem de maneira autônoma e assíncrona, mas em busca de um único objetivo global que é de conhecimento de todos e que está acima de qualquer objetivo individual, a vitória da equipe (normalmente). Da mesma forma que acontece em uma partida real de futebol, o time deve possuir uma disciplina tática e posicionamento em campo que pode ser modificado com comunicação entre os agentes. Cada agente tem uma função específica (atacante joga pra driblar e fazer gols, goleiro tenta defender as bolas, zagueiro tenta impedir a ação do ataque adversário) que pode ser encarado como um objetivo individual em um dado momento de jogo, mas é necessário que haja uma responsabilidade de cooperação entre estes de modo que, caso seja necessário, um agente zagueiro possa auxiliar o ataque ou um agente atacante desarmar uma investida do oponente. Recentemente, foi instituído um torneio de futebol virtual anual, onde pesquisadores de universidades desenvolvem seus times segundo certas regras e restrições comuns, mas com liberdade de uso de tecnologias para implementação [37]e [79].

3.3 Modelos em andamento

Esta seção discute as pesquisas acadêmicas em curso que visam modelar, construir e/ou usar agentes mais sofisticados no gerenciamento corporativo de redes.

3.3.1 MIT

Uma das áreas principais do gerenciamento de sistemas em rede é a atividade de se *modelar a rede*. O conhecimento da topologia de uma rede de computadores é pré-requisito básico para que se possa ter uma boa interação entre seus nós. Com o

crescimento das redes de computadores em termos de escala, escopo, e importância, as topologias tornaram-se, naturalmente, mais dinâmicas e a complexidade de configuração, administração e utilização também aumentou.

O grupo de Software Agentes do MIT Media Lab[10] desenvolveu um projeto de pesquisa intitulado "Mobile Agents for Routing Discovery" que utiliza MAs para fazer o mapeamento da topologia de redes dinâmicas [50]. São apresentados resultados de uma simulação de agentes móveis que "habitam" uma rede de nós interconectados e trabalham cooperativamente para construir um mapa da rede. O modelo é implementado com um motor de simulação baseado em eventos discretos. A cada passo de tempo da simulação o agente faz três coisas: aprende sobre todas as conecções que o nó, onde está no momento possui, troca informações com outros agentes que estejam no mesmo nó que o seu, e migra para um outro nó. Os agentes podem migrar aleatoriamente entre os nós da rede ou podem utilizar de certas heurísticas para migração, como migrar sempre para um nó adjacente que ele nunca tenha visitado (similar à busca em profundidade [57]) ou que tenha visitado menos recentemente, ou ainda migrar para nós que ainda não tenham sido explorados a partir de informações compartilhadas com outros agentes. Foi simulada uma rede de 250 nós e 2522 conecções ao todo (Figura 3.7). A simulação termina quando todos os agentes têm a informação completa sobre cada nó e conecções no grafo de comunicação; o sistema então convergiu. É avaliado o impacto do número de agentes, a colaboração entre os agentes, e os diferentes algoritmos de migração sobre os resultados da simulação.

As conclusões tiradas da análise das simulações realizadas apontaram que os agentes que cooperam podem coletivamente resolver o problema mais rapidamente que agentes trabalhando sozinhos. Entretanto, fica difícil otimizar o sistema para ter a máxima performance. Estratégias que aparentam serem mais vantajosas sob a perspectiva de um agente em particular, não são necessariamente eficientes para o sistema como um todo.

[10] A home page dos projetos desenvolvidos e em desenvolvimento do grupo é http://agents.www.media.mit.edu/groups/agents/projects/

Talvez um dos maiores desafios de desenvolvedores de sistemas multi-agentes seja encorajar a diversidade de comportamento em uma população de agentes cooperativos.

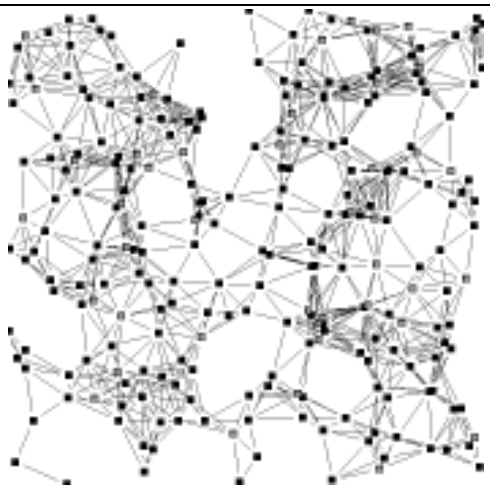


Figura 3.7 - Simulação de uma rede de 250 nós e 2522 conecções ao todo.

Este trabalho mostra um exemplo de como agentes móveis podem ser utilizados para se conseguir alta distribuição e descentralização de uma atividade de gerenciamento em particular: a modelagem da topologia da rede. Não há a idéia de uma estação central que monitora ou controla os agentes. Eles, simplesmente, realizam suas funções, cooperando ou não uns com os outros. As limitações do trabalho é que não leva em consideração os custos, por agente, do processamento de CPU ou consumo de largura de banda, por exemplo. Também não são simuladas problemas de comunicação em rede ou mudanças dinâmicas da topologia da rede.

3.3.2 Deglets e Netlets

Código móvel também pode ser um método conveniente para atividades de descobrimento automático de dispositivos de rede. Uma das técnicas de descobrimento mais comumente utilizadas é o envio de *pings* e uma visão da rede é construída a partir das respostas recebidas. Ao invés disso, um agente móvel pode ser criado e injetado na rede com a simples função de enviar o identificador de um nó visitado para seu criador.

O fim da atividade pode ser determinado heuristicamente dentro do agente, por exemplo, a partir de um número médio de visitas a um mesmo nó. A *delegação de autoridade* para a criação e manutenção de um modelo de rede é feita usando dois agentes chamados *deglets* e *netlets* [4], [72]. Um *netlet* (agente de descobrimento) pode ser injetado numa rede pela NMS para visitar os componentes (Figura 3.8). O *netlet* traz consigo o endereço IP do *host* que requer o modelo da rede.

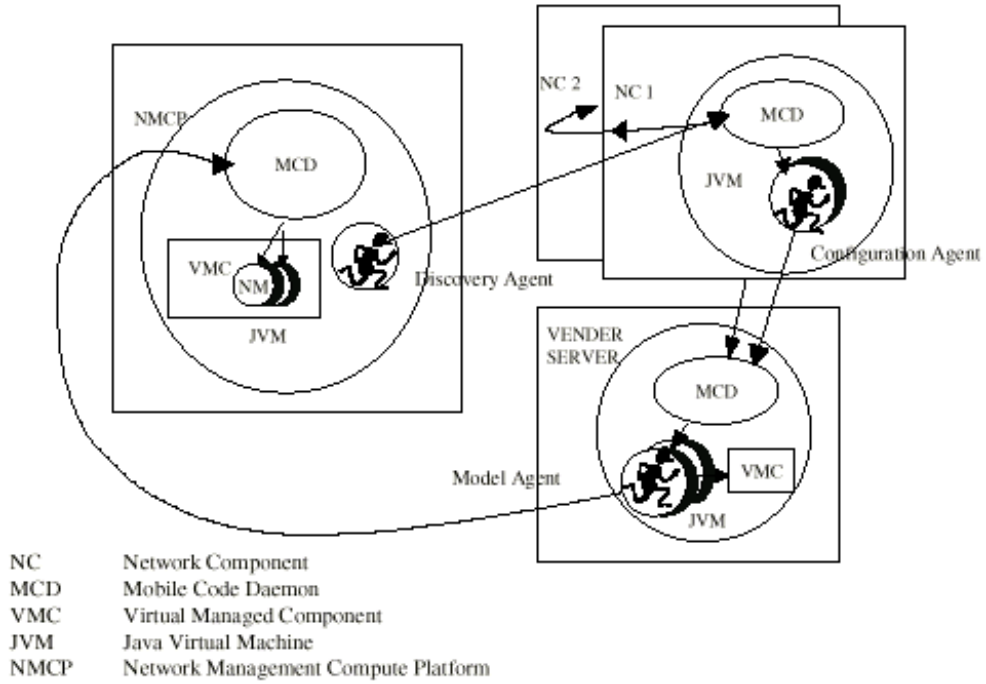


Figura 3.8 - Criação de um modelo de rede com Agentes Móveis.

Em cada componente da rede, o agente interage com o VMC[11] para descobrir a URL que mantém informações de dispositivos para o componente que acabou de ser

[11] *Virtual Managed Component* é a única interface para recursos a serem gerenciados e garante-lhes acesso seguro. O VMC provê facilidades de get, set, eventos e notificação (similar ao agente SNMP) com o uso de um mecanismo de lista de controle de acesso para reforçar a segurança. VMCs são desenvolvidos para conter procedimentos de reparo, abastecimento, uma MIB e informações relacionadas ao fabricante do componente.

descoberto. Uma vez que a URL do fabricante foi estabelecida, o *netlet* gera um *deglet* (agente de configuração) e copia informação do estado do componente para este. O d*eglet* também recebe o endereço IP e a porta associada do MCD[12] do *host* que requer a construção do modelo de rede. Enquanto o *netlet* continua a explorar o resto da rede, o *deglet* visita o *site* do fabricante e requisita que um outro *deglet* (agente de modelo) seja enviado para o *host* de gerenciamento (NMS). Este *deglet* interage com o VMC na NMS para construir um componente de rede virtual para o modelo de rede criado. O VMC armazena todo o modelo de rede criado e todos os acessos ao modelo de rede são feitos através de uma interface de aplicação provida pelo VMC. Dessa forma, o modelo de rede é um recurso compartilhado disponível para todos os agentes e aplicações que rodem numa JVM.

O principal trabalho da criação do modelo de rede é realizado pelo agente "provedor de modelo" (APM). O APM visita a estação de gerenciamento e instancia um VNC interagindo-se com o componente de rede de fato. Sendo o APM e o VMC no componente de rede providos pelo vendedor, a natureza do protocolo utilizado na comunicação entre eles não é importante e pode ser puramente proprietário. Apenas a API para as aplicações que utilizam o modelo de rede precisam ser publicadas. Ao contrário de modelos de rede tradicionais, o APM instala o estado e o comportamento do VNC no modelo de rede.

Este trabalho descreve uma abordagem alternativa para a criação e manutenção de modelos de rede com o uso de agentes móveis e que prevê mudança de estado e comportamento da rede. Isto é, em suma, o trabalho é uma nova proposta tanto para padronização da forma de provisão das informações de configuração de dispositivos de rede pelos fabricantes quanto para a forma com a qual redes heterogêneas podem ser modeladas.

---

12 Mobile Code Daemon recebe agentes com assinatura digital e faz checagens de autenticação antes de permitir que estes executem no componente.

Esta nova abordagem, potencialmente, elimina a necessidade de utilização de protocolos padrões (como SNMP ou CMIP) já que todas as interações ocorrem via o VNC e o fabricante implementa tanto o VNC quanto o VMC.

Os autores desenvolveram um *browser* gráfico em JAVA para permitir a visualização do modelo de rede pelo gerente na NMS.

O trabalho é bastante interessante e, apesar de ainda utilizar o conceito de estação de gerenciamento, representa uma grande iniciativa para a descentralização da atividade, uma vez que adota a mobilidade de agentes em detrimento do uso de protocolos de comunicação CS padrões.

3.3.3 Frameworks

Alguns autores [54] desenvolveram uma aplicação baseada em agentes para realizar operações de gerenciamento básico numa rede. Os agentes realizam coletas de informação sobre o estado da rede e realizam o gerenciamento de dispositivos de rede através do monitoramento de *funções de saúde* definidas pelo usuário. Por intermédio de uma interface gráfica, são selecionados os agentes de gerenciamento, é feito o controle da lista de agentes ativos na rede, e são avaliadas as informações coletadas pelos agentes que terminaram suas atividades. Foram implementados três tipos básicos de agentes para o gerenciamento: *browser agent*, *daemon agent*, *messenger agent*, e *verifier agent.* O primeiro coleta variáveis da MIB de um conjunto de nós especificados pelo usuário do sistema (administrador) e as guarda em sua estrutura interna, retornando para a estação em seguida. O segundo monitora uma "função de saúde" definida pelo administrador. Este agente migra para o nó gerenciado e compara esse valor da função com um *threshold* fixo pré-definido; caso seja maior, uma mensagem de notificação é enviada para o servidor de onde o agente saiu. A função do *messenger agent* é visitar todos os nós especificados e coletar as ações do *daemon agent*, obtendo uma visão global do estado da rede. O último agente é responsável por obter informações sobre propriedades específicas de sistema (por exemplo, verificar a versão

de um software, ou avaliar o espaço em disco disponível, ou verificar alguns arquivos de log, etc.) sobre nós da rede e retornar com uma lista dos nós que atendem àquela propriedade (ver Figura 3.9).

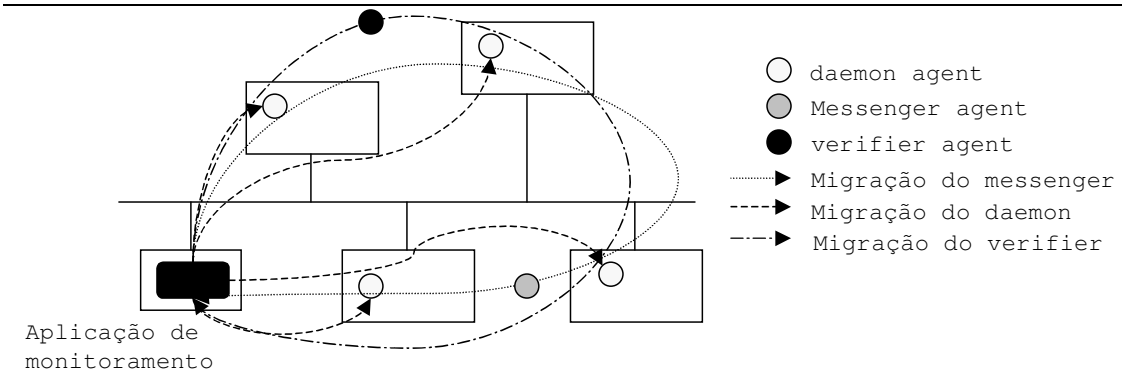


Figura 3.9 - Agentes cooperam para realizar operações de gerenciamento básico numa rede.

Na verdade, esta aplicação pareceu mais um protótipo para validação de um MAF que os autores desenvolveram previamente. Não existe metodologia para escolha dos tipos de agentes criados, o trabalho é pobre em resultados, e indica a tendência que ainda persiste em se imaginar soluções de gerenciamento dependentes de um controle central e baseadas em protocolos clássicos (SNMP, etc.). Também não fica claro se os agentes são providos de autonomia.

Nick Parkin [9] propõe uma redefinição da arquitetura de gerenciamento de redes tradicional para incorporação da tecnologia de agentes inteligentes. A arquitetura, ilustrada na Figura 3.10, propõe a inclusão explícita de um componente inteligente ao agente residente nos nós. Sendo assim, o novo agente inteligente passaria a ser composto por dois "*plug-ins*" escaláveis (componentes inteligente e de gerenciamento) mais o componente "cola", que é a única parte que é específica ao *hardware* e à funcionalidade de um nó particular. O componente inteligente pode ser implementado na forma de regras de produção [57], lógica *fuzzy* [38] ou redes neurais [26]. O componente de gerenciamento é o componente de protocolo de gerenciamento

responsável por lidar diretamente com a MIB. Ele recebe os *gets*, *sets* e envia os *traps* tanto da estação de gerenciamento quanto dos componentes "cola" e inteligente.

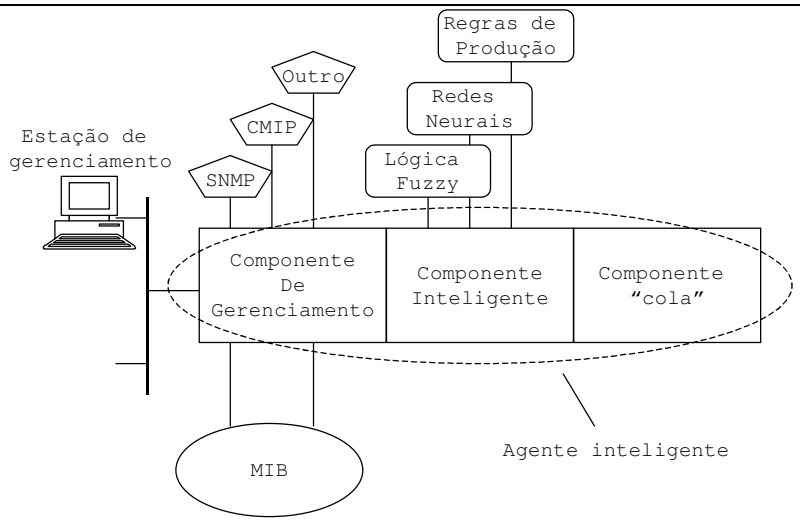


Figura 3.10 - Incorporação da tecnologia de agentes inteligentes.

A arquitetura foi aplicada a um estudo de caso onde um nó controla um número de dispositivos conectados, cada qual com um número de sinais discretos que devem ser monitorados. Caso Os requisitos desse nó é que ele utilize protocolos de gerenciamento SNMP, possua inteligência local e execute em hardware de baixo custo com poder de processamento limitado. A inteligência foi implementada com lógica *fuzzy*. O componente "cola" examina as portas do hardware com o qual os dispositivos são conectados e atualiza as variáveis MIB. As definições MIB modelam os sinais que estão associados com posições de bits na porta particulares.

A idéia por trás da proposta é a de que se deve prover um grau de autonomia aos agentes nos nós gerenciáveis de acordo com os recursos computacionais (poder de processamento, memória disponível, etc.) do nó em questão para que não haja sobrecarga. Na verdade, o tratamento aqui é relacionado apenas ao agente localizado nos nós. O trabalho não comenta sobre como estariam relacionados os agentes dentro

de uma solução global de gerenciamento da rede. Ainda assim, toda a proposta é montada sobre a utilização de protocolos de comunicação clássicos.

3.3.4 Críticas

Apesar do grande potencial do uso de agentes verdadeiramente inteligentes para o GCS, as pesquisas em curso parecem não estar colaborando muito. Alguns trabalhos apenas descrevem cenários onde algumas atividades de gerenciamento podem ser efetuadas com a utilização de agentes e não apresentam detalhes de arquiteturas. Outros apresentam soluções de agentes inteligentes apenas para uma questão particular de gerenciamento, mas não podem ser utilizadas em uma abordagem integrada de gerenciamento. Alguns outros descrevem arquiteturas gerais, com exemplo de uma aplicação para uma função de gerenciamento específica (falha, configuração, performance, etc.) [9].

A verdade é que ainda não existe um guia para utilização coerente de propriedades de agentes como mobilidade e autonomia para construção de soluções de GCS. Essa forma com a qual as pesquisas vêm sendo conduzidas não contribui efetivamente para o desenvolvimento de uma técnica. Elas ainda não são muito bem coordenadas, e pesquisadores, principalmente no meio acadêmico, utilizam essas propriedades de agentes ainda de forma não muito consensual.

Outro problema é que parece ficar claro a não preocupação em adotar uma postura realmente distribuída para o gerenciamento. Mesmo com a inclusão de agentes autônomos em soluções, a idéia de uma estação de gerenciamento na rede ainda permanece.

Essa nova abordagem para o GCS, onde o gerenciamento é verdadeiramente distribuído e independente de um controle central potencializa a necessidade de mudança de procedimento sobre a utilização das propriedades de agentes. Faz-se necessária, a criação de um guia para utilização coerente dessas propriedades (como autonomia e

mobilidade) em soluções de GCS baseada nessa filosofia de distribuição de gerenciamento total.

3.4 Conclusões

O uso de agentes inteligentes para implementação de soluções de administração pode ter vantagens consideráveis em relação ao modelo tradicional CS apresentado no capítulo 2. A implementação de agentes móveis pode auxiliar na descentralização da atividade de gerência de redes e a autonomia pode potencializar as vantagens da mobilidade reduzindo a necessidade da inteligência humana ou qualificações pessoais durante a instalação e operação da solução de gerenciamento. Além disso, agentes móveis autônomos podem solucionar os problemas dos dispositivos localmente sem a necessidade de qualquer contato com a NMS. Em suma, a utilização organizada de técnicas de IA em soluções de GCS pode prover um alto grau de automação e flexibilidade que pode diminuir a necessidade de interação constante do administrador humano no gerenciamento.

Entretanto ainda não existe um guia para utilização coerente de propriedades de agentes como mobilidade e autonomia para construção de soluções de GCS. As pesquisas ainda não são muito bem coordenadas, e pesquisadores, principalmente no meio acadêmico, utilizam essas propriedades de agentes ainda de forma não muito consensual.

4
A Abordagem

"The network is the computer[1]" (Sun Microsystems)

Nos dois capítulos anteriores foi visto o estado da arte do gerenciamento corporativo de sistemas e as tendências de pesquisas com agentes inteligentes para um gerenciamento mais distribuído, flexível, e menos dependente da figura do administrador humano.

Este capítulo discute três propriedades de agentes importantes para concepção de arquiteturas para o gerenciamento corporativo de sistemas: a *mobilidade*, *autonomia* e *distribuição*. Baseados nessas propriedades, são definidos alguns tipos de agentes e arquiteturas multiagentes baseadas na combinação desses tipos.

---

[1] Slogan usado durante uma campanha.

4.1 Requisitos

Achamos que as discussões sobre o desenvolvimento de abordagens menos centralizadas, onde entidades mais autônomas são responsáveis pela gerência de um dado recurso, devem ser transladadas para um novo contexto, onde a *rede deve ser encarada como um desses recursos* (RR^2) a ser explorado e gerenciado.

Mas qual a relação dessa nova visão RR com o advento de pesquisas sobre aplicação da tecnologia de agentes inteligentes no GCS? A resposta para a questão está na nova postura do administrador humano e, conseqüentemente, na complexidade da ferramenta de automação.

Com o RR, o administrador necessita agora de uma ferramenta que lhe passe informações sobre o estado global de cada rede de computadores - já que a rede é o recurso - e não mais de um recurso em particular. Essa nova postura vai de encontro, inclusive, à forma já padronizada de gerência onde se tem uma NMS central que fala protocolos específicos de rede e interage constantemente com os recursos gerenciados. Por que não se ter um sistema de gerenciamento distribuído a ponto de não existir uma estação central e onde a inteligência simplesmente emane de um conjunto de agentes inteligentes que se relacionam para resolver o problema de um determinado RR?

Se por um lado, a distribuição do gerenciamento dos recursos de uma rede diminui a necessidade de intervenção humana, por outro, aumenta a necessidade de organização dos agentes inteligentes. Isto é, é de grande importância que seja feito uma avaliação da inclusão de propriedades de agentes como mobilidade, autonomia, por exemplo. O uso dessas propriedades deve ser feito de maneira coerente.

---

[2] A fim de se evitar confusões terminológicas, adotaremos o termo "Recurso-Rede" (RR) para representar o recurso Rede de Computadores, e "recurso"continuará se referindo a dispositivos de uma rede – impressoras, modems, scanner, etc.

4.2 Histórico: Mobilet

No primeiro semestre de 1999, foi desenvolvida, como projeto conjunto de duas disciplinas do mestrado (*Métodos de Computação Inteligente* e *Sistemas Distribuídos*), uma ferramenta para administração de espaço em disco de uma rede UNIX/NFS (a Seção 5.1 explica a organização de arquivos NFS da Sun).

A princípio, o objetivo principal do projeto era o de mostrar como a tecnologia de agentes móveis poderia ser aplicada como uma alternativa igualmente funcional à tecnologia dos objetos remotos em sistemas distribuídos e na inteligência artificial. A opção pela administração do espaço em disco de máquinas em rede ocorreu, por ser esta uma atividade custosa para o administrador, de extrema importância para o bom funcionamento da rede, e pela possibilidade de se construir um software real para administração da rede do CIn/UFPE.

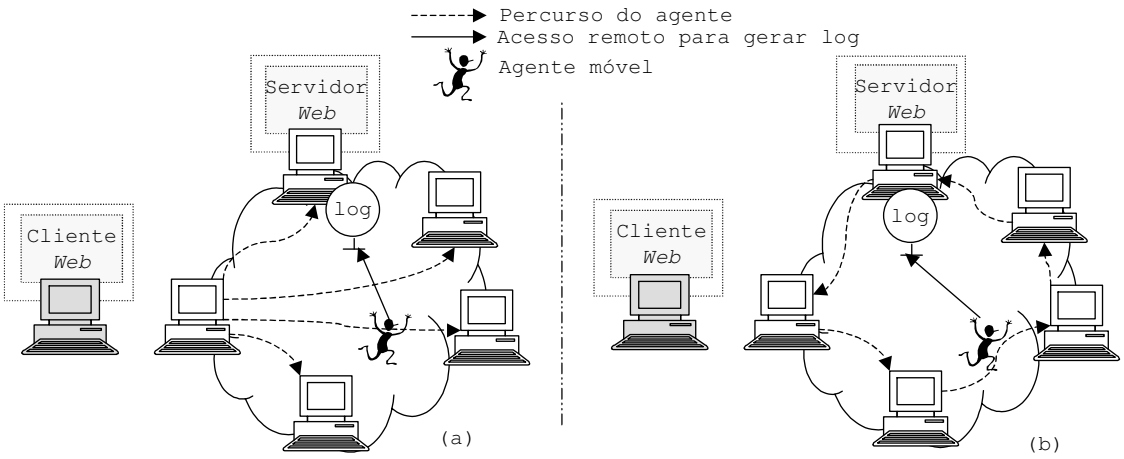


Figura 4.1 - Arquiteturas Mobilet. (a) Mobilet, (b) Mobilet2 (Mobilet Ring).

O *Mobilet* (*Mobile* + *Net*) é um agente desenvolvido com conhecimento especializado sobre as operações de gerenciamento de espaço em disco comuns ao administrador. O agente "viaja" para cada uma das máquinas da rede – seguindo planejamento prévio especificado pelo administrador – e monitora o estado das partições (*filesystems*) dos

discos destas máquinas. Por exemplo, uma determinada partição pode estar com 90% do espaço ocupado quando, segundo o conhecimento do administrador, o ideal seria que estivesse com menos de 80%. O agente ao monitorar o disco desta máquina, irá detectar o problema de super-utilização e efetuar operações sobre arquivos presentes na partição para reduzir a área ocupada. Essas operações podem ser remover e compactar arquivos, mover arquivos para uma partição de uma outra máquina e criar referências para os mesmos na máquina de origem, etc. Quando for apresentado o estudo de caso no próximo capítulo, irão ser mostrados maiores detalhes de como estão organizadas as partições de disco em uma máquina UNIX.

A Figura 4.1(a) ilustra a arquitetura do *Mobilet*. O agente é inicializado em uma máquina da rede que funcionaria também como a estação de gerenciamento. Ou seja, a máquina é encarregada de ficar enviando o agente para cada uma das outras máquinas da rede. O agente faz o monitoramento das máquinas e gera um *log* das operações em uma outra máquina específica via operações remotas CORBA. O administrador pode acessar o arquivo de *log* através de um *applet* Java numa página *Web*. Para viabilizar a migração do agente, cada máquina deve prover um servidor de agentes móveis (ver Seção 3.1.2), bem como um *middleware* de suporte à distribuição.

O *Mobilet* [44] apresentou resultados bastante interessantes e estimulou o desenvolvimento e implementação de arquiteturas alternativas para resolução de problemas do mesmo domínio. Ele suscitou várias questões, mostrando o quão era difícil decidir sobre o grau de autonomia, mobilidade e o número de agentes presentes numa arquitetura. Assim, o objetivo passou a ser o de fazer uma avaliação preliminar dos prós e contras de arquiteturas diferentes baseadas ou não em código móvel, com menos ou mais autonomia aos agentes, mais distribuídas ou mais centralizadas, etc.

O *Mobilet2* (ou *Mobilet Ring*) [43], ilustrado na Figura 4.1(b), é semelhante em funcionalidade ao anterior com a diferença de que o agente ganha mais autonomia e independe de um controle central responsável por enviá-lo para cada uma das

máquinas. Ao contrário, o agente realiza suas atribuições de gerência numa máquina e decide a próxima para a qual irá migrar.

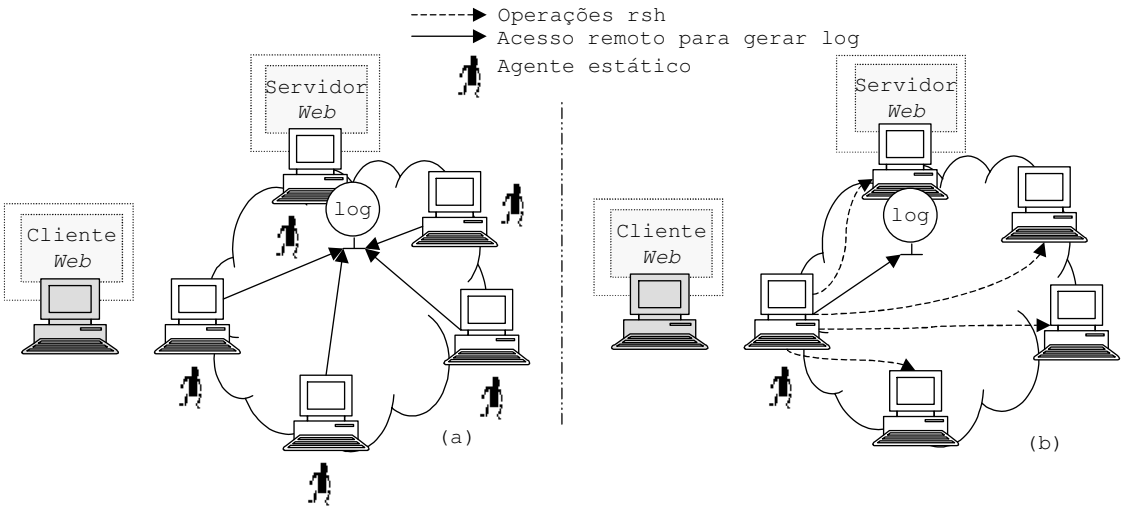


Figura 4.2 - Arquiteturas de agentes estáticos (a) locais (b) um único agente.

Desenvolvemos mais duas arquiteturas para administração do espaço em disco [43]. Uma delas é constituída de vários agentes estáticos distribuídos entre as máquinas a serem gerenciadas. Os agentes não trocam qualquer tipo de informação e são responsáveis por operações apenas em suas respectivas máquinas. Essa arquitetura tem como característica principal a independência de necessidade de comunicação em rede; que fica sendo necessária apenas para o caso de se gerar os arquivos de *log* das operações na máquina responsável (ver Figura 4.2(a)).

Finalmente, a quarta arquitetura implementada é constituída por apenas um agente estático residente na NMS que realiza o monitoramento e controle das partições das máquinas de rede através de operações *rsh* (*remote shell,* ver Seção 5.1). O agente possui todo o conhecimento especializado do domínio, traz o estado das partições das máquinas para a NMS, infere as ações de correção necessárias e as executa remotamente. Fica evidente, que esta arquitetura é extremamente dependente de

comunicação de rede. A idéia era a de simular operações CS como as de gerência de redes tradicionais como SNMP. A Figura 4.2(b) ilustra a arquitetura.

Foram feitas muitas comparações entre as arquiteturas implementadas. Algumas necessitam de suporte à distribuição (*framework* de mobilidade, JVM, etc). Algumas se mostraram mais dependentes de intervenção humana. Algumas necessitam de comunicação de rede, etc. O espaço em disco disponível necessário para executar a ferramenta é menor em algumas, o tráfego de rede gerado é maior em outras, etc. Algumas tiveram menor tempo de processamento em algumas etapas da gerência e pior em outras. Ou seja, tornou-se inevitável se atentar para a questão de qual a influência da mobilidade, ou comunicação, ou grau autonomia para a performance global de uma ferramenta de administração que ambiciona independer totalmente de intervenção humana. Percebeu-se, entretanto, que outras mais variações de arquiteturas seriam necessárias para que os resultados das comparações fossem conclusivos.

4.3 Questões Abertas

O *Mobilet* e arquiteturas conseqüentes, ao mesmo tempo em que se revelaram como soluções potenciais para o GCS, deixaram indícios de que se faz necessária uma metodologia mais bem definida para a confecção de softwares RR. Essa metodologia pode ser definida a partir do juízo sobre algumas questões que se apresentam em aberto sobre o uso de IA no GCS.

Na verdade, essa discussão vai muito mais além do domínio do gerenciamento de redes. Apesar dos significantes avanços do desenvolvimento de sistemas baseados em IA, existe ainda muito pouco investimento em tentar se entender como fazer a engenharia desses sistemas [75].

Nas próximas seções são discutidas questões relativas à mobilidade de agentes dentro de arquiteturas para GCS, grau de autonomia que deve ser dado aos agentes, número de

agentes presentes na solução e como estão distribuídos, e tipos de agentes existentes, funcionalidades diferentes, etc.

4.3.1 Agentes Móveis ou Estáticos?

Como foi discutido na seção 3.1, o uso de agentes móveis em soluções para o GCS possibilita uma forma de controle mais descentralizado da execução da computação distribuída. Entretanto, pode ser que existam circunstâncias em que uma solução de agentes mais estáticos seja mais bem adequada. De fato, apesar de haver um certo consenso na literatura de que a mobilidade de agentes traz vantagens para as soluções de administração em particular, a utilização de mobilidade é circunstancial. O grande desafio é descobrir quais são essas circunstâncias.

Situações onde não se faz necessário um gerenciamento freqüente, talvez o uso de mobilidade não seja compensatório. O custo de se prover o suporte necessário à migração (seção 3.1.2) pode ser maior do que os benefícios trazidos. Nesses casos, talvez o melhor fosse uma solução baseada em um agente estático que gerenciasse a rede via operações CS.

Já nos casos onde há uma grande variação de *status* dos dispositivos a serem gerenciados, onde problemas ocorrem com grande freqüência, talvez uma solução com agentes móveis seja mais adequada. Agentes estáticos locais também poderiam ser utilizados, mas talvez não fossem interessantes sob o ponto de vista de economia de espaço em disco, por exemplo.

Problemas de complexidade não muito grande, que não demandem agentes implementados com milhares de linhas de código, talvez pudessem ser resolvidos com mobilidade. Mas e se esses problemas aconteçam com bastante freqüência numa rede bastante instável, agentes móveis continuariam a ser uma boa opção? E para o caso de redes onde alguns dispositivos necessitam de um gerenciamento muito mais freqüente do que outros? Seria o caso de se utilizar agentes estáticos nesses dispositivos e móveis para o resto da rede?

Todas essas hipóteses sugerem a *mobilidade* como uma primeira propriedade de agentes que merece ser avaliada. Um agente pode ser móvel ou estático. Sendo estático, pode ser local no dispositivo ou estar conectado a este via operações remotas (ver Figura 4.3).

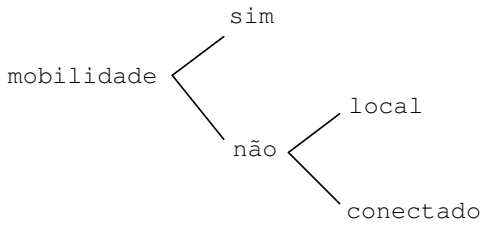


Figura 4.3 - Mobilidade de agentes.

4.3.2 Que Grau de Autonomia?

O grau de autonomia de agentes em sistemas de gerenciamento de redes é um fator de fundamental importância para uma maior descentralização das operações. Como foi mostrado (seção 2.1.2), nas abordagens tradicionais os agentes são apenas responsáveis por notificar a NMS de um evento e não possuem meios de corrigir o problema; todo o conhecimento da atividade de administração reside na NMS.

Mas quanto de autonomia deve ser dada a um agente numa solução de GCS?

Se por um lado, a autonomia pode potencializar as vantagens da mobilidade, por outro, pode ser que um alto grau de autonomia signifique um agente mais "pesado", com mais regras codificadas.

Seria interessante então, existir agentes com especializações diferentes, isto é, diferentes funcionalidades?

Em situações onde as atividades são bastantes complexas e podem ser quebradas em sub-atividades, esquemas de agentes especializados também poderia ser interessante. Quando se faz necessário uma resposta rápida, onde há um grande volume de dados,

poderia ser interessante, por exemplo, um agente responsável apenas por processar esses dados (percepção). Da mesma forma, um agente ficaria responsável por monitorar o dispositivo (percepção), um outro determinaria as ações cabíveis (decisão) e um outro executaria tais ações (execução). Agentes deste tipo seriam menores e menos complexos que um agente completo.

A *autonomia* é, portanto, uma outra propriedade que deve ser analisada. O agente pode ter total autonomia dentro da solução (chamado de agente completo) ou ter uma autonomia restrita a alguma funcionalidade em particular (agente especializado) (ver Figura 4.4). Além disso, um agente pode ser especializado tanto a título de subproblema (por exemplo, agente *perceptor*, *resolvedor* e *executor*), como a título de funcionalidade. É possível que para determinadas soluções de gerenciamento faça-se necessário um agente com função exclusiva de *cooperar* ou de *delegar* atividades ou ainda de *criar* outros agentes, etc.

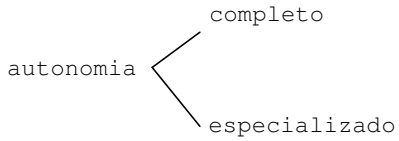


Figura 4.4 - Autonomia de agentes

4.3.3 Quantos Agentes?

Outra questão importante é sobre o número de agentes necessários para se implementar a solução desejada.

Por exemplo, numa solução baseada em agentes móveis completos, quantos agentes idênticos devem coexistir? Em um domínio de gerência onde haja uma certa freqüência de variações de estado de dispositivos, talvez seja interessante um número maior de agentes. Os agentes podem ser espalhados em uma rede e ficarem responsáveis, cada

um, por uma determinada sub-rede. Por outro lado, mais agentes pode significar uma maior necessidade de coordenação.

O número de agentes numa solução também pode estar diretamente relacionado com a questão da autonomia. Agentes com poder de autonomia restrito, mais especializados, necessitam cooperar com outros agentes para resolver eventuais problemas: outro problema de coordenação.

Coordenação sugere suporte à comunicação entre os agentes. Ou seja, tem de estar bem definida uma forma de comunicação seja via ACLs (*Agent Comunication Language*) ou chamadas remotas. No caso de comunicação entre agentes móveis, o cenário pode se tornar um pouco mais complicado. Cada plataforma de mobilidade provê seu próprio mecanismo de solução para comunicação bem como a interface entre um agente e o *host* onde executa. Isso pode gerar problemas de compatibilidade com outros sistemas [40].

Essas observações suscitam a existência de uma terceira propriedade a ser levada em consideração quando da escolha de uma arquitetura multiagente para uma solução RR: *distribuição*. A solução pode conter apenas um único agente, ou agentes distribuídos pela rede (Figura 4.5).

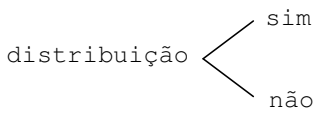


Figura 4.5 - Distribuição de agentes

4.4 Metodologia do Projeto de Mestrado

As questões expostas acima tornam clara e evidente a grande dificuldade em se projetar e desenvolver sistemas de agentes para soluções GCS dentro desse novo contexto RR. Faz-se necessário um certo método para construção desses sistemas.

Um dos objetivos desse projeto de mestrado é o de contribuir com "guias" que auxiliem o desenvolvedor no momento de decidir sobre mobilidade de agentes, a complexidade dos agentes, e o número de agentes a ser implementado.

A metodologia empregada para se chegar a esses guias está ilustrada na Figura 4.6.

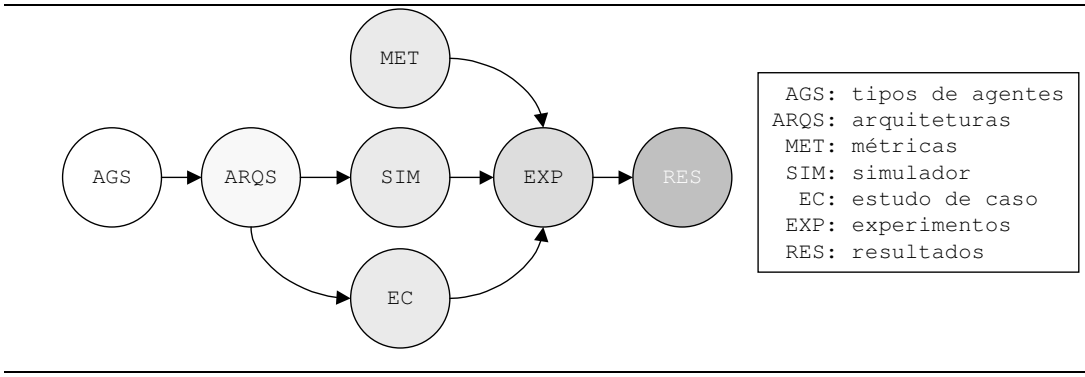


Figura 4.6 - Metodologia do projeto de mestrado.

Alguns tipos de agentes devem ser definidos (seção 4.4.1) e arquiteturas devem ser elaboradas combinando-se esses tipos de agentes (seção 4.4.2). Métricas devem ser estabelecidas (seção 4.4.3) e deve ser feito um estudo de caso onde as arquiteturas são adaptadas (capítulo 5). Um simulador (capítulo 6) deve ser desenvolvido para se poder realizar experimentos (capítulo 7) com o estudo de caso escolhido e seguindo as métricas estabelecidas. Os resultados dos experimentos poderiam assim servir de guias para o desenvolvedor de uma eventual solução. Futuramente, essas guias poderiam ser organizadas para criação de uma metodologia de construção de sistemas de agentes inteligentes para o GCS.

4.4.1 Tipos de Agentes

As propriedades *mobilidade* e *autonomia* de agentes discutidas anteriormente podem ser combinadas:

A árvore resultante dessa combinação de propriedades pode ser reorganizada sob a ótica do agente (Figura 4.7):

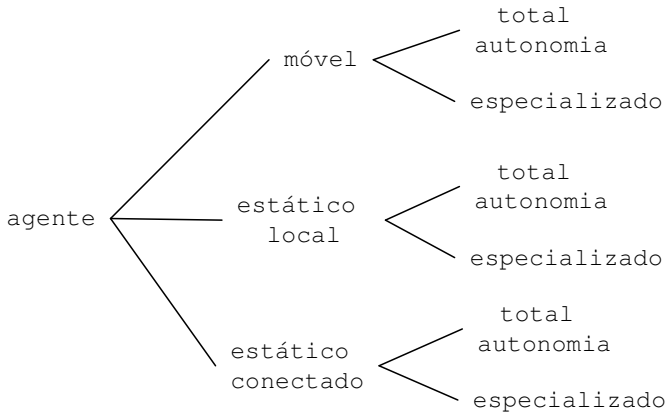


Figura 4.7 - Árvore de propriedades de um agente

Dessa forma, podem-se definir pelo menos seis tipos diferentes de agentes. Se forem levadas em consideração as variações de especialização este número pode subir ainda mais. Para o trabalho, resolvemos utilizar três especializações básicas: *percepção*, *decisão* e *execução*, já mencionadas na seção 4.3.2; com estas especializações o número de tipos de agentes sobe para doze (ver Figura 4.8). Dependendo da atividade de gerenciamento a que vai ser aplicada a solução, eventualmente outras especializações podem ser definidas.

As propriedades de agentes podem ser representadas por dois conjuntos (“Mobilidade” e “Autonomia”) e os tipos de agentes pelos elementos (E1, E2, ..., En) resultantes do produto cartesiano desses dois conjuntos (Figura 4.8).

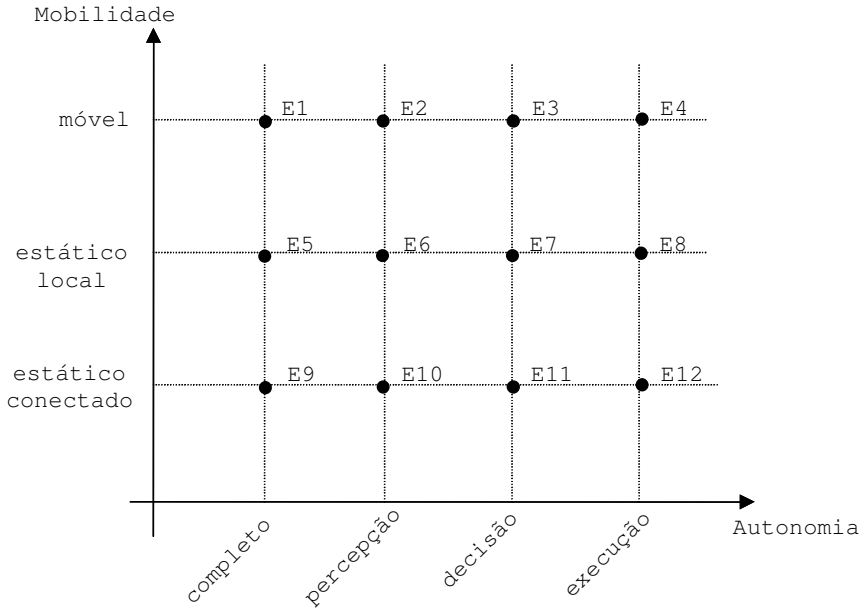


Figura 4.8 - Tipos de agentes no plano de propriedades.

O seguinte esquema de nomeação da Tabela 4.1 será utilizado para facilitar identificações futuras.

MobEntire (= Mobilet)→ E1	MobSentinel → E2	ModDecide → E3	MobDoit → E4
Entire → E5	Sentinel → E6	Decide → E7	Doit → E8
FarEntire → E9	FarSentinel → E10	FarDecide → E11	FarDoit → E12

Tabela 4.1 - Tipos de agentes definidos

O agente do tipo MobEntire possui todo o conhecimento necessário para a atividade de gerência em questão e é capaz de migrar de um local para outro.

O agente do tipo MobSentinel também realiza migrações (rondas), mas sua autonomia está restrita à capacidade de identificar e alertar sobre alterações de funcionamento em dispositivos.

Um agente MobDecide é um agente migratório, mas só possui autonomia para determinar as ações de administração cabíveis.

A função de execução dessas ações pode ficar por conta de um agente móvel do tipo MobDoit.

Um agente Entire é um agente estacionário que se localiza em um dispositivo e possui autonomia total para detectar, inferir e executar as ações.

A função de um Sentinel é a de apenas detectar alterações de comportamento do dispositivo onde reside.

O agente do tipo Decide é também um agente estacionário cuja autonomia se restringi a decidir sobre as ações cabíveis para solução do problema.

Um agente Doit, também estacionário, pode ficar responsável por executar essas ações.

Os agentes do tipo Far (FarEntire, FarSentinel, FarDecide e FarDoit), são os agentes estacionários conectados. Suas funcionalidades são as mesmas das de seus parentes, mas ao contrário daqueles, eles agem remotamente.

É importante deixar claro, que para o caso das especializações, nada impede que tipos de agentes sejam associados em um único. Por exemplo, um agente estacionário pode ser responsável por decidir por ações e por executá-las também. Esse agente seria do *tipo associado:* DecideDoit.

4.4.2 Arquiteturas Multiagentes

Agentes podem ser combinados para se formar arquiteturas multiagentes. Dessa forma, segundo a terceira propriedade, *distribuição*, uma arquitetura pode consistir de apenas um único agente ou de agentes distribuídos. Neste último caso, os agentes podem ser de mesmo tipo (idênticos) ou de tipos diferentes (distintos).

O número de combinações entre tipos de agentes que se pode ter para composição de uma arquitetura multiagente é muito grande. Para que se tenha idéia do número de possibilidades, suponha o exemplo de uma mini-rede de apenas três nós gerenciáveis.

A Figura 4.9 (a) ilustra 6 arquiteturas possíveis. Da esquerda para direita e de cima para baixo, a primeira seria composta por dois agentes do tipo MobEntire, a segunda seria composta por um agente do tipo MobSentinel e um do tipo associado MobDecideMobDoit, a terceira composta por um agente do tipo FarSentinel e um do tipo associado FarDecideFarDoit, respectivamente, a quarta composta por três agentes do tipo Entire (E), a quinta composta um agente do tipo MobSentinel e dois do tipo associado MobDecideMobDoit, e a sexta composta por três agentes do tipo Sentinel e um do tipo associado MobDecideMobDoit, respectivamente.

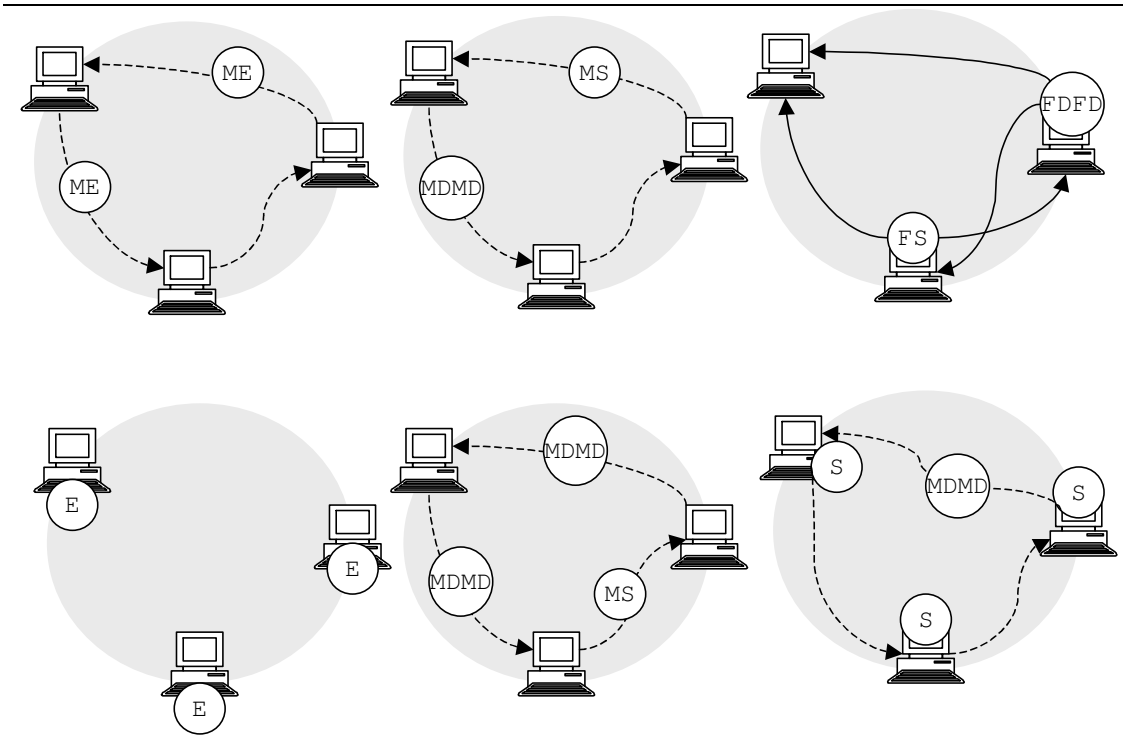


Figura 4.9 - Diferentes composições de arquiteturas multi-agente[3].

Esse pequeno exemplo é suficiente para ilustrar a grande dificuldade que é se determinar os tipos de agentes adequados para uma eventual solução e o número de

[3] Onde ME = MobEntire, MS = MobSentinel, MDMD = MobDecideMobDoit, FS = FarSentinel, FDFD = FarDecideFarDoit, E = Entire e S = Sentinel.

agentes que devem estar presentes na mesma. Para que se tome uma decisão coerente, o tipo de atividade de gerência, os número de nós gerenciáveis presentes na rede, e algumas métricas (ver seção 4.4.3) devem ser levados em consideração.

Como não dá para tratar todas as possibilidades, fizemos uma análise de cada um dos tipos de agentes definidos e identificamos algumas arquiteturas que seriam mais coerentes (Figura 4.10). O número de agentes na arquitetura depende de outros fatores, como já foi explicitado anteriormente.

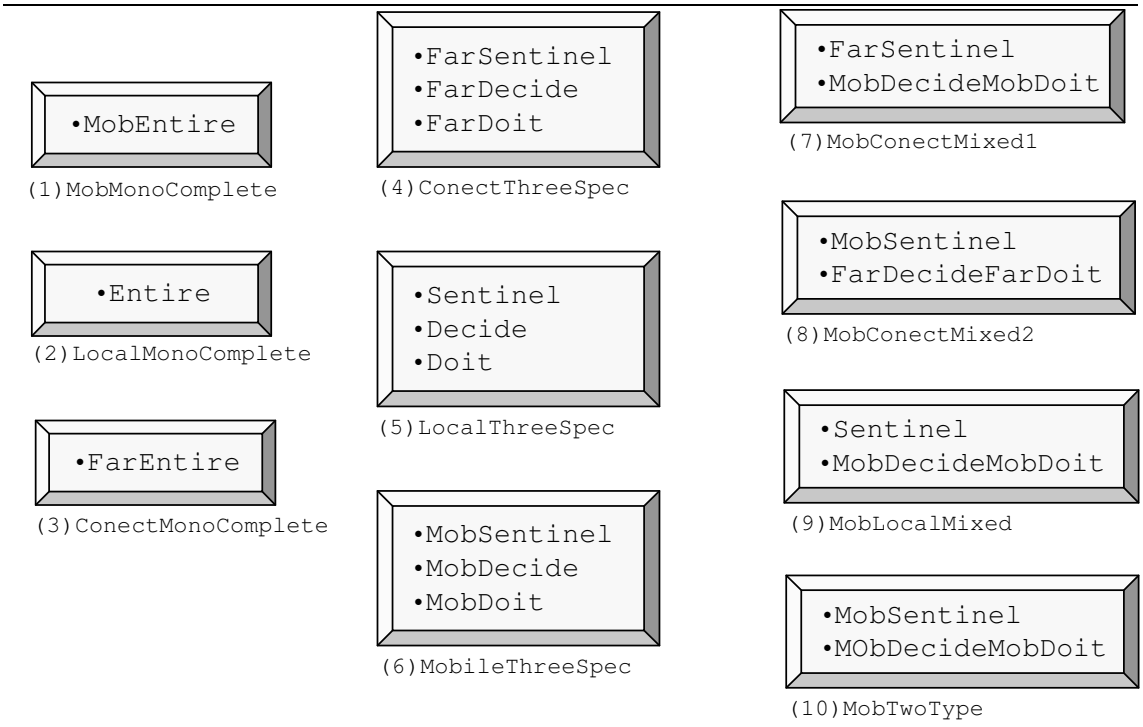


Figura 4.10 - Arquiteturas multi-agentes.

As arquiteturas 1, 2 e 3 da figura são arquiteturas mono-agentes com os agentes tendo total autonomia para gerenciar a rede em questão. As arquiteturas 4, 5 e 6 são compostas, cada uma, de três agentes com especializações básicas: *percepção*, *decisão* e *execução*. A arquitetura 7 é composta de agente(s) FarSentinel que monitora(m) todos os dispositivos remotamente e quando algum problema é detectado, aciona(m) agente(s) MobDecideMobDoit para inferir as ações e executá-las. A arquitetura 8 é

composta de agente(s) móvel(is) MobSentinel que monitora(m) os dispositivos e informam o(s) agente(s) estacionário(s) conectado(s) FarDecideFarDoit de algum eventual problema. Na arquitetura 9, agentes Sentinel localizados estaticamente em cada um dos dispositivos, ao detectarem algum problema, chamam por agente(s) MobDecideMobDoit. Enfim, na arquitetura 10, agente(s) móvel(is) MobSentinel monitoram dispositivos e quando detectam algum problema chamam por agente(s) MobDecideMobDoit.

4.4.3 Métricas

Para que se possa avaliar o impacto de tipos agentes e número de agentes presentes na arquitetura de uma solução RR, faz-se necessário estabelecer alguns critérios de comparação (métricas).

Por exemplo, pode ser interessante utilizar agentes móveis, mas será compensador do ponto de vista da sobrecarga que uma MAF pode causar ao sistema? Agentes estacionários conectados não necessitam do suporte de nenhuma plataforma MAF, mas talvez operações remotas não sejam interessantes do ponto de vista de tráfego de rede ou tempo de processamento. Agentes do tipo estático local não necessitam de MAF, não realizam operações remotas, e não dependem de robustez da rede, em compensação todos os dispositivos a serem gerenciados necessitam da presença de um agente desse tipo, e, conseqüentemente, consumo do espaço em disco de todos os dispositivos.

Em suma, podem-se identificar algumas métricas:

- Consumo de CPU;
- Tempo de processamento;
- Tráfego de rede;
- Consumo de espaço em disco;
- Robustez da rede.

- Consumo de memória;
- Flexibilidade de uma eventual solução;
- Escalabilidade de uma eventual solução;
- Tempo que um dispositivo permanece com o problema (Latência de gerenciamento);

4.5 Conclusões

As discussões sobre desenvolvimento de abordagens para gerenciamento menos centralizadas onde entidades mais autônomas são responsáveis pela gerência de um dado recurso, devem ser transladadas para um novo contexto onde uma rede é um desses recursos. Nesse novo contexto o administrador passa a necessitar apenas de uma ferramenta que lhe passe informações sobre o estado global de cada rede de computadores – já que a rede é o recurso – e não mais de um recurso em particular. O objetivo é se ter um sistema de gerenciamento distribuído a ponto de não existir uma estação central e onde a inteligência simplesmente emane de um conjunto de agentes inteligentes que se relacionam para resolver o problema de um determinado RR.

Para que isso seja viável, algumas questões têm de ser resolvidas. Particularmente, as relações entre mobilidade, autonomia e distribuição de agentes dentro de uma solução. Como foi observado no capítulo 3, estas propriedades ainda são utilizadas de maneira não muito consensual e o objetivo deste trabalho é auxiliar para a confecção de soluções para o GCS, provendo guias (dicas) de como essas propriedades podem ser utilizadas coerentemente.

Para isso, foi criada uma metodologia de trabalho de mestrado que consiste das seguintes etapas:

- Tipos de agentes são definidos baseados nas propriedades *mobilidade* e *autonomia:* agentes móveis com total autonomia ou com alguma

funcionalidade específica (especialização); agentes estacionários que agem localmente, agentes estacionários que agem remotamente, etc;

- Arquiteturas compostas por combinações desses tipos de agentes são elaboradas. Por exemplo, uma arquitetura pode conter um tipo de agente móvel especializado em monitorar estado de dispositivos e um tipo para resolução e execução de ações cabíveis. Outra arquitetura pode ser formada com apenas um tipo de agente estacionário local com total autonomia;

- Métricas para comparação de performance entre arquiteturas são definidas (tempo de processamento, consumo de CPU, etc.);

- Um estudo de caso é utilizado para adequar os tipos de agentes e as arquiteturas definidas;

- Um simulador é desenvolvido para se poder realizar experimentos com o estudo de caso escolhido e seguindo as métricas estabelecidas.

Os resultados dos experimentos podem servir de guias para o desenvolvimento de uma eventual solução GCS. Futuramente, essas guias podem ser organizadas para criação de uma metodologia de construção de sistemas de agentes inteligentes para o GCS.

5

Aplicação na Administração de Espaço em Disco

A administração do espaço em disco é hoje uma atividade de extrema importância para o perfeito funcionamento de uma rede corporativa, visto que vários aplicativos e serviços de rede, como clientes e servidores de e-mail possuem alto grau de dependência do espaço de armazenamento livre. Para que haja um funcionamento adequado desses sistemas, é preciso que uma certa porcentagem de ocupação dos recursos de armazenamento não seja ultrapassada. A experiência descrita na seção 4.2, mostrou que um sistema de automação para este fim baseado em agentes consiste, de uma maneira geral, de uma entidade monitora que checa periodicamente a porcentagem de utilização de uma partição de disco UNIX e determina se esta está super-utilizada (*estouro de partição*) ou não e uma entidade de conhecimento especialista do domínio responsável por determinar as medidas de correção necessárias, e uma entidade de execução para aplicar essas medidas.

Este capítulo descreve a proposta para automatização do processo de administração do espaço em disco e a modelagem dos agentes e arquiteturas definidas no capítulo anterior para o domínio em questão.

A próxima seção fará uma pequena revisão do tratamento de arquivos do UNIX e um exame do funcionamento do *Network File System* (NFS), já que toda modelagem apresentada neste capítulo se baseia no sistema de arquivos do UNIX.

5.1 Sistema de Arquivos do UNIX/NFS

Os recursos de arquivamento de uma máquina UNIX consistem em uma coleção de dispositivos (normalmente, discos rígidos), cada qual contendo um *sistema de arquivos* nativo do UNIX, ou compatível com este. Através de procedimentos denominados montagem e desmontagem, estes sistemas de arquivos separados são agrupados de modo a compor uma única árvore de diretórios virtual, a qual forma a imagem que os usuários têm de todo o sistema de arquivamento. Diferentemente de outros sistemas operacionais (como DOS, por exemplo), em que o conceito de dispositivo é visível ao usuário, em UNIX busca-se esconder a existência de vários dispositivos físicos, deixando sua manipulação ao encargo apenas do administrador do sistema.

Na construção da árvore de diretórios virtual, um dos dispositivos existentes é considerado "dispositivo raiz", e contém o diretório base do sistema de arquivos e outros diretórios. Quando da inicialização da máquina, outros dispositivos contendo árvores adicionais de diretórios são agregados à árvore pré-existente no dispositivo raiz através de comandos de montagem *mount.* Embora as montagens sejam feitas automaticamente na inicialização das máquinas, nada impede que outras chamadas *mount* sejam feitas para montar outros dispositivos não usuais (como uma unidade de fita) quando necessário. O procedimento inverso à montagem denomina-se *desmontagem* (comando *umount*), e é acionado automaticamente no desligamento das máquinas para desconectar dispositivos.

Os recursos de montagem e desmontagem são uma característica importante do UNIX. Além de oferecerem uma visão conveniente do sistema de arquivos como uma única árvore lógica, oferecem também a vantagem de permitir que dispositivos heterogêneos ou remotos sejam agregados ao dispositivo raiz.

O NFS é um ambiente para compartilhamento e distribuição transparente de arquivos em redes de máquinas UNIX. Ele implementa um ambiente cliente-servidor, que estende as funcionalidades do acesso normal a arquivos no UNIX, de modo a permitir o compartilhamento de porções de qualquer sistema de arquivos local a uma máquina por quaisquer outras máquinas situadas na mesma rede, resultando em uma distribuição dos recursos de armazenamento e permitindo o compartilhamento remoto de arquivos.

Os serviços de NFS são conectados ao sistema através do recurso de montagem, pela importação de sub-árvores de diretórios fornecidos por outra máquina. A Figura 5.1 mostra parte de um sistema de arquivos de uma máquina UNIX composto de três dispositivos: um disco raiz, um repositório de ferramentas acessado por NFS, e uma unidade de CD-ROM.

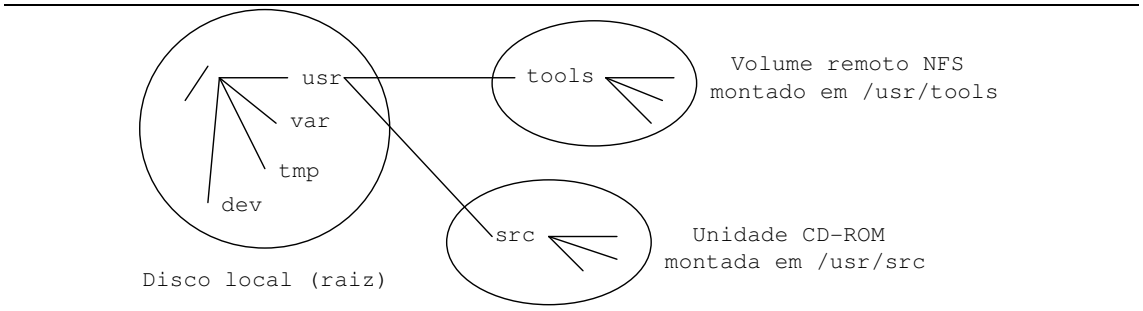


Figura 5.1 - Sistema de arquivos UNIX composto de três dispositivos.

O NFS implementa um ambiente de distribuição seletiva de arquivos, configurável com granulidade de volumes de montagem. Em uma distribuição de arquivos através de NFS, determinadas máquinas em uma rede são escolhidas como servidoras, fornecendo parte de seu espaço de arquivos a quantas máquinas clientes se desejar. As máquinas podem funcionar como clientes e servidoras ao mesmo tempo, formando contextos de

distribuição bem complexos, e melhorando bastante o aproveitamento do espaço total em disco existente. Na Figura 5.2, o arquivo montado em */usr/students* no cliente é na verdade a sub-árvore localizada em */export/people* no servidor A

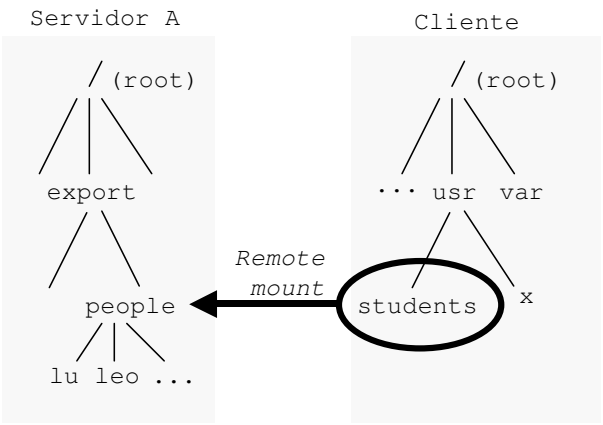


Figura 5.2 - Árvore de diretórios virtual.

A configuração é responsabilidade do administrador do sistema, que é encarregado de compor arquivos de configuração nos servidores e clientes que definem como se dá o compartilhamento. Nos servidores, um arquivo denominado “exports”, (normalmente, etc/exports), descreve quais diretórios (e suas sub-árvores, conseqüentemente), podem ser exportadas para máquinas clientes, interessadas no acesso a esses arquivos. Nas máquinas clientes, um arquivo denominado “fstab” lido durante a inicialização da máquina, dispara comandos de montagem de alguns dos diretórios exportados (*remote mount*) e dos diretórios provenientes de dispositivos físicos locais de modo a compor a árvore de diretórios virtual desta máquina (Figura 5.2). No caso do NFS, os acessos aos arquivos são transferidos através de RPCs para serem realizados na máquina que efetivamente contém os arquivos, sendo os resultados transferidos de volta.

Cada dispositivo de armazenamento (coluna “Filesystem” da Figura 5.3) usado para compor a árvore de diretórios virtual de uma máquina UNIX será chamado de *partição*. Para um melhor entendimento de como funciona toda essa organização de partições em discos de máquinas UNIX em rede, segue o exemplo representativo de duas máquinas

UNIX da rede do CIn/UFPE. A máquina "Caruaru" funciona como cliente da máquina "Olinda". A Figura 5.3 mostra as partições da máquina "Caruaru". A linha em destaque mostra o exemplo de um *remote mount* onde o diretório */export/home6* da máquina "Olinda" foi montado na máquina "Caruaru" juntamente com sua sub-árvore de diretórios. A Figura 5.4 ilustra a árvore montada na máquina "Caruaru".

Filesystem	kbytes	used	avail	capacity	Mounted on
/proc	0	0	0	0%	/proc
/dev/dsk/c0t3d0s0	61735	52768	2794	95%	/
/dev/dsk/c0t3d0s6	616551	480231	80831	86%	/usr
fd	0	0	0	0%	/dev/fd
/dev/dsk/c0t3d0s4	123455	37746	73364	34%	/var
/dev/dsk/c0t1d0s7	2012390	1747656	204363	90%	/export/local
/dev/dsk/c0t3d0s3	61735	42748	12814	77%	/tmp
/dev/dsk/c0t0d0s7	966127	492077	416083	55%	/export/install
/dev/dsk/c0t4d0s7	972197	802796	72182	92%	/export/local2
olinda:/export/home2	3276800	2706740	570060	83%	/a/olinda/export/home2
olinda:/export/home3	3276800	2630212	646588	81%	/a/olinda/export/home3
olinda:/export/home1	4259840	3835776	424064	91%	/a/olinda/export/home1
olinda:/export/home7	3276800	1642156	1634644	51%	/a/olinda/export/home7
olinda:/export/home6	3276800	2033724	1243076	63%	/a/olinda/export/home6
olinda:/export/home5	3276800	2173708	1103092	67%	/a/olinda/export/home5

Figura 5.3 - Listagem das partições montadas na máquina Caruaru[1].

[1] As partições que se iniciam com "olinda:" estão presentes na máquina "Olinda" e foram montadas remotamente na máquina "Caruaru".

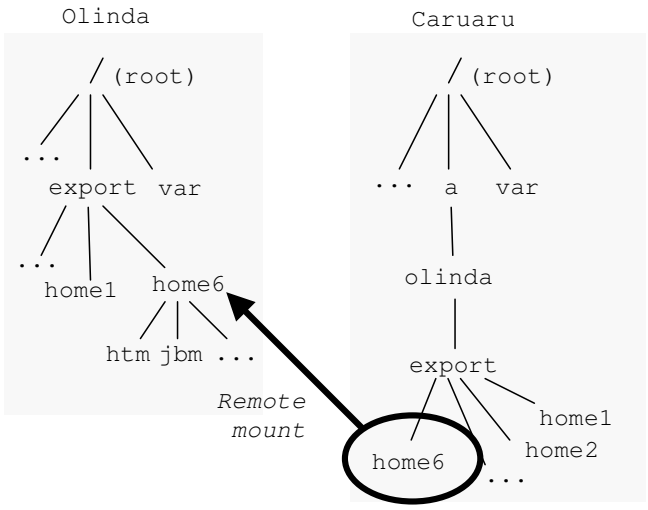


Figura 5.4 - Árvore de diretórios montada na máquina "Caruaru"[2].

5.2 A Administração do Sistema

O grande problema que pode ocorrer em ambientes como o descrito anteriormente é a super utilização de uma determinada partição. Isto é, o número de arquivos existentes em uma partição pode crescer a ponto de ocupar todo o espaço reservado em disco para aquela partição. Têm-se configurado então, um *estouro de partição*.

A gravidade de um estouro de partição depende do tipo da partição. As partições podem ser classificadas de duas formas diferentes: quanto à freqüência e volume de atualizações elas podem ser *estáticas* ou *dinâmicas* e quanto à importância para o sistema elas podem ser *críticas* ou *normais*.

Partições do tipo "/", "/opt" ou "/usr" são estáticas, pois sofrem pouca alteração se comparadas a partições dinâmicas como "/var" ou "/tmp", por exemplo. Partições do tipo "/", "/var/mail" são partições de fundamental importância para o funcionamento do sistema (críticas) enquanto que partições como "/home" ou "/tmp" são importantes para

usuários dessas áreas, mas um eventual estouro não chega a prejudicar o sistema como um todo (partições normais).

Em ambientes que consistem de um sistema operacional em rede com um sistema distribuído de arquivos (tipo NFS), é comum que cada usuário tenha um espaço limitado para uso pessoal: sua *cota*. Isto é necessário para se controlar o espaço em disco utilizado por cada usuário, evitando que ocupem uma área de armazenamento da máquina e cause problemas de espaço. Cada conta de usuário se localiza em uma partição de disco compartilhado, onde o espaço localmente disponível é dividido com outras contas de usuários. A linha em destaque da Figura 5.5 mostra que a partição */export/home6* da máquina "Olinda" está com 63% de utilização. Esta partição contém contas de usuários da rede do CIn/UFPE.

Filesystem	512-blocks	Free	%Used	Mounted on
/dev/hd4	131072	98920	25%	/
/dev/hd2	4063232	84040	98%	/usr
/dev/hd9var	655360	601912	9%	/var
. . .				
/dev/hd15	6553600	2200632	67%	/export/home5
/dev/hd16	6553600	2485168	63%	/export/home6
. .				

Figura 5.5 - Listagem das partições montadas na máquina "Olinda".

A maioria dos usuários, normalmente, não utiliza todo o espaço permitido, assim sendo, é política corrente dos administradores de sistema de se disponibilizar menos espaço em cada partição, do que a soma do limite das contas ali existentes. Conseqüentemente, pode acontecer de crescer a utilização do espaço da partição e impedir que usuários usem suas contas mesmo não tendo ultrapassado seu limite particular. Tem-se assim, configurado um problema de espaço em partição de disco.

[2] O arquivo /a/olinda/export/home6 representa a sub-árvore home6 na máquina "Olinda".

O problema de estouro de partição de contas de usuário descritas acima constitui apenas um exemplo de caso em que pode ocorrer um problema de espaço em partições e, como já foi dito, é um problema particular dos usuários dessa partição que não chega a causar maiores danos ao resto da rede. Entretanto, o problema pode ocorrer em outros tipos de partições e com conseqüências bem mais nefastas. A falta de controle em uma partição relativa à área de e-mail ("/var/mail"), por exemplo, pode levar à indisponibilidade desse serviço, fazendo com que mensagens de correio eletrônico advindas da Internet sejam rejeitadas, podendo ser devolvidas ou descartadas.

Para administrar a utilização do espaço em disco em sistemas desse tipo, o administrador deve checar freqüentemente a porcentagem de utilização de cada partição em uma máquina e tomar alguma medida de correção caso esse valor ultrapasse algum limite preestabelecido (*threshold*). Só a vivência e o pleno conhecimento da dinâmica de utilização de disco pode dizer quais são esses limites, que variam entre os tipos de partições. Por exemplo, para uma partição crítica como a "/" é interessante que a utilização não ultrapasse 90% do espaço disponível.

Alguns administradores de sistema sugerem maneiras de resolver o problema de super utilização de espaço em disco. Basicamente, as soluções apontadas consistem em vários tipos de *ações sobre arquivos*. Pode-se procurar por arquivos muito grandes ou inúteis e removê-los, pode-se compactar arquivos, pode-se mover arquivos para partições de outras máquinas, etc. A escolha de que ação deve ser tomada deve seguir alguns critérios: tempo de vida ociosa do arquivo, arquivos replicados, arquivos temporários, arquivos de cachê, arquivos de tipo desconhecido, arquivos de mail, etc.

Na próxima seção será explicitada a automatização do processo de administrar a utilização do espaço em partições de disco. As etapas descritas são resultantes de todo um trabalho de aquisição do conhecimento sobre o assunto com especialistas da área.

5.3 Automatização do Processo

Para o nosso estudo de caso, foram selecionados oito tipos diferentes de partições (Tabela 5.1). Estes tipos de partições foram escolhidos por serem partições de disco, mais ou menos padrão de sistemas UNIX, particularmente do nosso ambiente de trabalho (rede do Cin/UFPE). O limite de utilização s*uperior* representa a porcentagem de utilização máxima que aquele tipo partição deve possuir: valores maiores que esse representam problemas de super utilização. O limite *inferior* é a porcentagem de utilização máxima que a partição deve chegar após a atividade de gerenciamento. A importância desse limite inferior é evitar que a atividade de gerenciamento seja repetida com grande freqüência, principalmente para o caso de partições mais dinâmicas (requerem um limite inferior mais baixo). Os valores dos limites para cada uma das partições foram determinados com a ajuda de administradores.

Tipo de Partição	Limite Superior	Limite Inferior
/	90%	80%
/usr	90%	80%
/proc	95%	85%
/tmp	80%	50%
/var	90%	70%
/var/mail	80%	70%
/opt	95%	85%
/export/<nome>	90%	70%

Tabela 5.1 - Lista de partições

Consultando especialistas da área e baseado numa certa experiência de trabalhos anteriores de alunos do CIn/UFPE [3] e [67] definimos que uma ferramenta de automação para administração de espaço em disco de máquinas, como as exemplificadas anteriormente, deveria realizar pelo menos cinco operações básicas:

- Classificação das partições do disco;
- Checagem da porcentagem de utilização das partições;

- Classificação de todos os arquivos de partições com problema de super-utilização;
- Determinação das melhores ações de correção a serem tomadas;
- Execução das ações decididas.

As operações serão todas detalhadas em pseudocódigo a fim de facilitar a compreensão, porém todas as implementações do projeto de mestrado foram feitas na linguagem de programação Java. Java foi utilizada por ser orientada a objetos e possuir grande documentação de suporte, APIs, e estar disponível gratuitamente na Web.

5.3.1 Classificação de Partições

A operação de *classificação de partições* tem a função e *identificar* partições existentes no disco de uma máquina e em seguida *classificá-las* segundo às pré-definidas na tabela anterior. Qualquer outra partição identificada que não faça parte desse conjunto é classificada como "partição não usual". Os pseudocódigos a seguir ilustram esses dois processos: identificação (Figura 5.6) e classificação (Figura 5.7).

```
getFilesystems() {
   Liste partições existentes na máquina com "/bin/df -kl"
   Guarde listagem na memória
   Para ∀ linha da listagem faça:
      Identifique o tamanho, o total utilizado e o nome da partição
      classify(nome da partição)
}
```

Figura 5.6 – Identificação de partições duma máquina

```
classify(Filesystem nome) {
    Se nome = "/": partição é do tipo ROOT
    Se nome = "/usr": partição é do tipo USR
    ...
}
```

Figura 5.7 – Classificação das partições

5.3.2 Checagem de Utilização de Partições

A operação de *checagem de utilização de partições* tem a função de identificar se, segundo os limites de utilização respectivos, a partição está super utilizada ou não. Existe um arquivo texto com valores de limites correspondentes a cada tipo de partição (semelhante à Tabela 5.1).

O pseudocódigo da Figura 5.8 mostra o funcionamento dessa operação.

```
checkUtil() {
    Para ∀ partição p já classificada faça:
       Identifique o limite superior de utilização no arquivo para
       partições de tipo igual ao de p (lsp)
       Se % utilização de p ≥ lsp:
          p está super utilizada
}
```

Figura 5.8 – Checagem de utilização das partições de uma máquina

5.3.3 Classificação dos Arquivos das Partições

A operação de *classificação dos arquivos das partições* é uma das mais complicadas. Resumidamente, consiste de um algoritmo recursivo de *busca em profundidade* [57] que percorre toda a árvore de sub-diretórios de uma partição e vai identificando os arquivos (o nome do arquivo constitui de todo o caminho, *path* completo).

Durante a fase de aquisição do conhecimento, assim como no caso das partições, também foram definidos alguns tipos de arquivos (Tabela 5.2).

Tipos de arquivos		
imagem	texto	Executável
CORE	hiper-texto	de cachê
temporário	de e-mail	de log
de backup	de impressão (mqueue ou print)	

Tabela 5.2 - Tipos de arquivos.

A análise do algoritmo de classificação de arquivos desenvolvido em [43] mostrou que o tempo de processamento da operação ficou bastante alto em comparação com o tempo de processamento das demais operações (Figura 5.9).

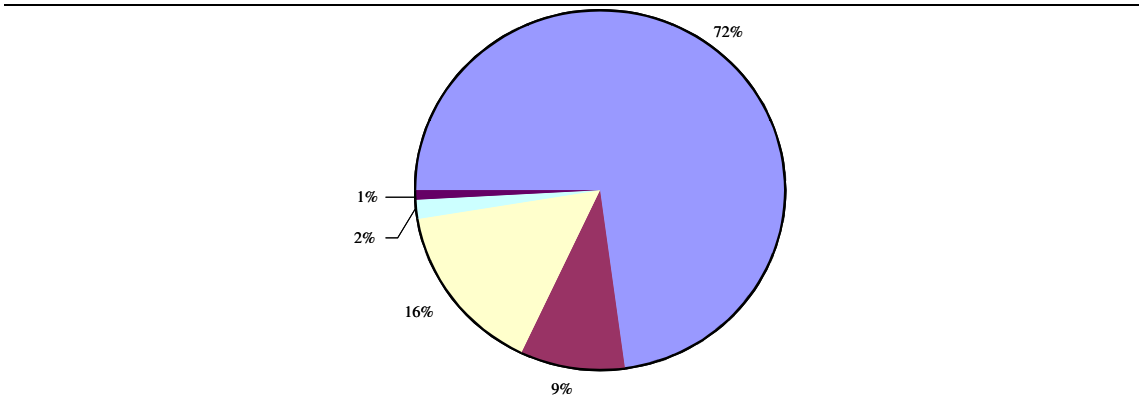


Figura 5.9 - Tempo de processamento proporcional do classificador de arquivos

O algoritmo foi reimplementado e ao invés de fazer a classificação de todos os arquivos existentes em uma partição super utilizada, o "novo" algoritmo processa um certo número de arquivos e passa para a operação de inferência. Se ainda não resolver o problema da partição, então mais um conjunto de arquivos será classificado e assim por diante.

Fundamentalmente, essa mudança implicou em um tratamento explícito para recursividade. O pseudocódigo da Figura 5.10 ilustra esse novo algoritmo para classificação dos arquivos.

```
classifyFiles() {
   Para cada partição p super utilizada faça:
      acumulate()
      Enquanto o vetor não estiver vazio:
          Leia a linha em questão da última posição do vetor
          Se linha = fim da listagem:
             Remova última posição do vetor
             Se vetor ficar vazio:
                Saia do Enquanto
          Se começar por "d":
             É um diretório
             acumulate()
          Se começar por "-":
             É um arquivo
             Obtenha informações sobre o arquivo com "/bin/file"
             Classifique arquivo como mostra exemplo da figura XX
             Se um certo número x de arquivos já tiver sido identificado:
                reasoning()  (ver próxima operação)
                Saia do Enquanto
}

acumulate() {
   Liste os arquivos|diretórios da partição com "/bin/ls -lAc"
   Acumule a listagem em um vetor na memória
}
```

Figura 5.10 – Classificação de arquivos

Para que seja mais bem entendido o funcionamento do algoritmo, seja o exemplo de uma máquina com duas partições já classificadas, cujos nomes são respectivamente "/a" e "/b". A Figura 5.11 ilustra o conteúdo das partições numa máquina imaginária.

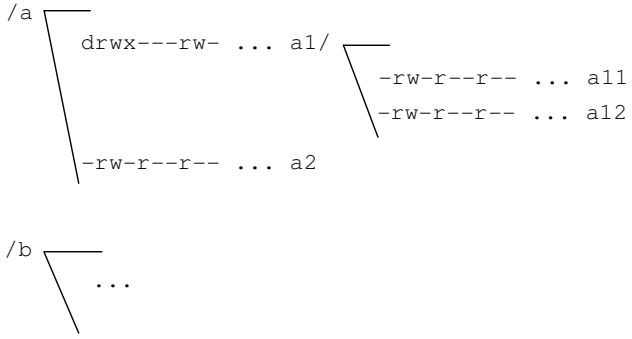


Figura 5.11 - Conteúdo de partições de uma máquina imaginária.

A Figura 5.12 ilustra o *trace* do algoritmo até que os arquivos “a11”, “a12” e “a2” tenham sido classificados.

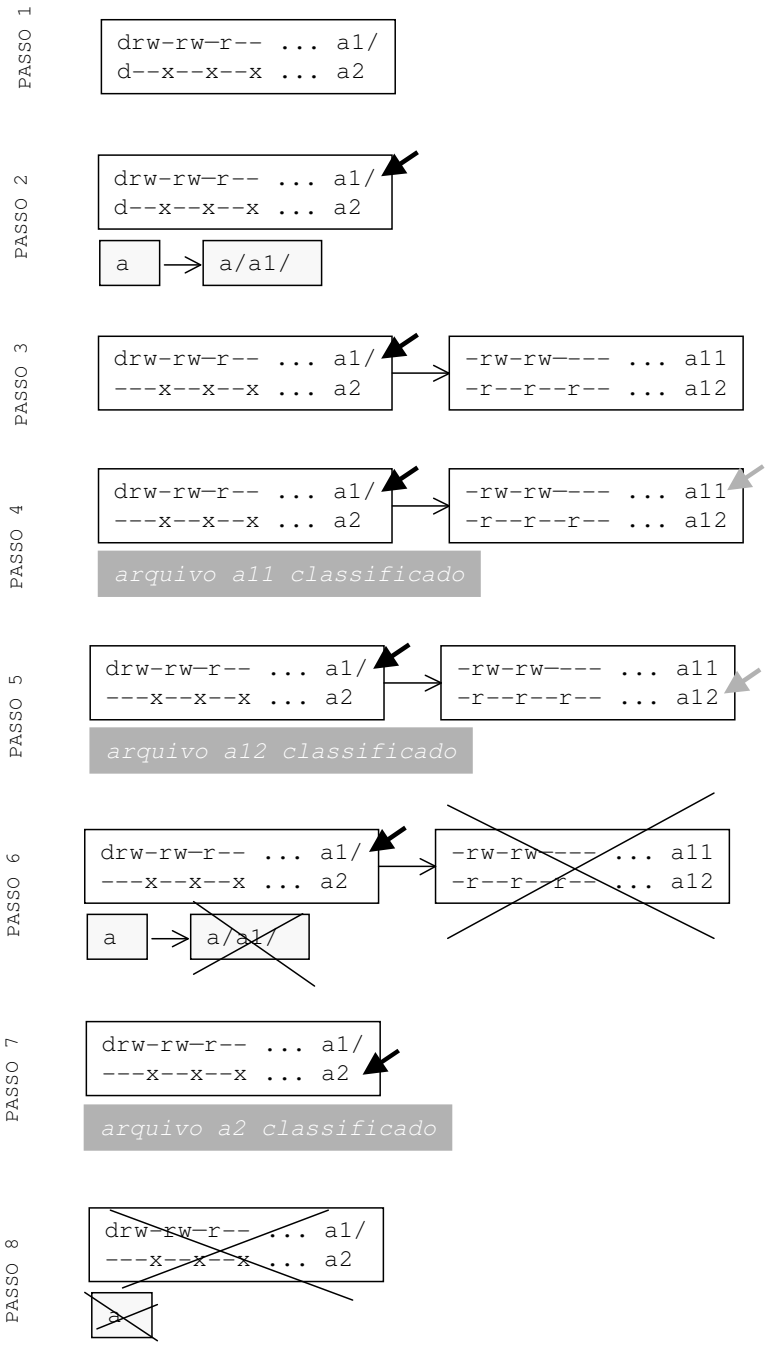


Figura 5.12 - Passos para classificação dos arquivos da Figura 5.13

No passo 1 uma listagem com os arquivos da partição "/a" é armazenada na primeira posição de um vetor na memória. No passo 2, a primeira linha dessa listagem é lida e como o primeiro caractere é a letra "d", significa dizer que "a1/" é um diretório. Sendo um diretório o algoritmo adiciona o nome da partição em questão num outro vetor na memória, concatena o nome da partição com o do diretório ("/a/a1/") e adiciona a este mesmo vetor. Como "/a/a1" ainda se trata de um diretório, então o procedimento do passo 1 é repetido no passo 3. No passo 4, a primeira linha da nova listagem gerada é lida e como o primeiro caractere é "-", significa dizer que "a11" é um arquivo e que tem de ser classificado. No passo 5, a segunda linha da nova listagem é lida e "a12" também é um arquivo a ser classificado. No passo 6, como todas as linhas da listagem na última posição do vetor foram lidas, então remove-se este último elemento do vetor, bem como do outro vetor que acumula os diretórios. No passo 7, a linha seguinte da listagem anterior é lida e como "a2" foi identificado como um arquivo, tem de ser classificado. Assim como no passo 6, no passo 8 remove-se o último elemento do vetor.

A classificação de um arquivo é realizada de acordo com seu próprio nome e com as informações obtidas com a chamada de sistema "/bin/file".

A Figura 5.13 ilustra um exemplo de classificação de um arquivo de e-mail.

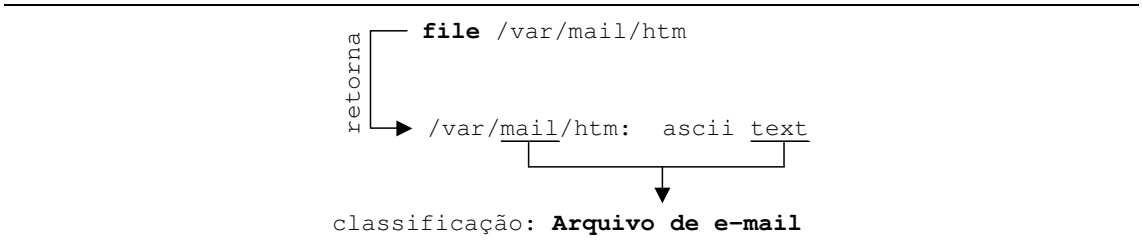


Figura 5.13 - Exemplo de classificação de um arquivo.

No exemplo, o arquivo a ser classificado se chama "/var/mail/htm". A chamada de sistema UNIX indica que o arquivo é um arquivo de texto simples ("ascii text"), entretanto, como no seu nome (do arquivo) existe a string "/mail/", ele é classificado como *arquivo de e-mail*.

A Tabela 5.3 mostra as regras desenvolvidas para classificação dos arquivos de acordo com *strings* identificadas na linha de retorno (SLR) do comando `file` aplicado, e *strings* no nome do arquivo em si (SN). Cada linha da tabela rerepesenta uma regra no formato

```
Se (SLR = X) e (SN = Y) então (Classificação = C).
```

Essas regras (conhecimento) foram obtidas com o auxílio de especialistas na área de administração.

SLR	SN	Classificação
-	"/tmp/"	temporário
-	"/cache/"	cache
-	"/var/log/"	log
-	"/var/adm/"	log
-	"/var/spool/mqueue/"	impressão
-	"/var/spool/queue/"	impressão
".bak"	-	backup
".bkp"	-	backup
"~"	-	backup
"core"	-	core
"executable"	-	executável
"image"	-	imagem
"JPEG"	-	imagem
"GIF"	-	imagem
“text”	".htm"	hipertexto
“text”	"/mail/"	e-mail
“text”	"/Mail/"	e-mail
“text”	"Inbox"	e-mail
“text”	"mbox"	e-mail
“text”	"/nsmail/"	e-mail
“text”	-	texto
-	“/a/”	remoto[3]

Tabela 5.3 - Regras para classificação dos arquivos.

[3] Como pode ser observado nas figuras 5.3 e 5.4, partições de máquinas remotas são montadas num diretório “/a”. Um arquivo deste diretório é classificado como “Remoto” por estar presente em uma outra máquina, na verdade.

5.3.4 Decisão sobre Ações de Correção

A operação de *decisão sobre ações de correção* é responsável por determinar que ações devem ser tomadas para cada um dos arquivos identificados numa partição que apresente problema de super utilização.

Essas ações foram determinadas durante a fase de aquisição de conhecimento junto a especialistas. Uma entre cinco diferentes ações pode ser aplicada a um determinado arquivo. Arquivos podem ser:

- Removidos;
- Compactados;
- Compactados e recriados vazios (para o caso de arquivos de *log*);
- Migrados (para uma outra partição com mais espaço livre);
- Mantidos.

Por exemplo, pode-se determinar que um arquivo temporário cujo último acesso se deu a mais de uma semana deva ser removido, ou um arquivo de imagem muito grande deva ser compactado.

O pseudocódigo da Figura 5.14 ilustra a operação de raciocínio para decisão sobre as ações de correção para cada um dos arquivos classificados.

```
reasoning() {
   Insira na base de conhecimento KB a partição super-utilizada
   Insira ∀ arquivos já classificados na KB
   Execute a KB
}
```

Figura 5.14 – Decisão sobre ações de correção

As regras para ações de correção foram descritas num formato semelhante à *Lógica de Primeira Ordem* (FOL)[57]. A Figura 5.15 a seguir, ilustra algumas dessas regras de produção.

```
∀p,a, Partição(p) ∧ Arquivo(a) ∧ (Tipo(a)=CORE) ⇒ Remova(a)
∀p,a, Partição(p) ∧ Arquivo(a) ∧ SuperUtil(p) ∧ (Tipo(a)=MAIL) ∧
      (Tamanho(a)>2000000) ∧ (Tipo(p)=EXPORT) ⇒ Compacte(a)
∀p,a, Partição(p) ∧ Arquivo(a) ∧ SuperUtil(p) ∧ (TempoDeVida(a)>5) ∧
      (Tipo(a)=TMP) ⇒ Remova(a)
```

Figura 5.15 – Exemplos de regras de produção para decisão sobre ações de correção

Todas as regras elaboradas foram codificadas para o formato do motor de inferência JEOPS [16] (ver Anexo A). O JEOPS é uma proposta de extensão para a linguagem JAVA com um mecanismo para embutir regras de produção em aplicações JAVA. O JEOPS funciona como um pré-compilador que traduz um arquivo contendo essas regras para um arquivo JAVA que implementa o motor de inferência, de acordo com as regras do arquivo original.

5.3.5 Execução das Ações

A operação que cuida da execução das ações inferidas na operação anterior, realiza chamadas UNIX para executar de fato essas ações (ver Figura 5.16).

```
executeActions() {
   Remover: "rm "+[nome do arquivo]
   Compactar: "gzip "+[nome do arquivo]
   Compactar e Recriar Vazio: "gzip "+[nome do arquivo]
                              e Criar novo arquivo de mesmo nome
   Mover: Exportar partição com espaço na máquina destino ("exportfs")
          Montar partição remota na máquina origem ("mount")
          Mover arquivo da máquina de origem para a de destino
}
```

Figura 5.16 – Execução das ações de correção

5.4 Adaptação das Arquiteturas definidas

No capítulo anterior foram definidas dez arquiteturas diferentes de agentes (Figura 4.10). Dessas arquiteturas, cinco foram escolhidas para o nosso estudo de caso:

(a) MobMonoType

(b) LocalMonoType

(c) ConectMonoType

(d) MobLocalMixed

(e) MobTwoType

Com estas 5 arquiteturas podemos comparar um bom número de tipos diferentes de agentes. Nesta seção será mostrado como os tipos de agente integrantes dessas arquiteturas foram adaptados para o domínio da administração do espaço em disco. De acordo com a Figura 4.10, estes tipos são:

- MobEntire;
- Entire;
- FarEntire;
- Sentinel;
- MobSentinel;
- MobDecideMobDoit.

Nas subseções seguintes será ilustrada a arquitetura de cada um dos tipos de agentes adequados à administração do espaço em disco. Os agentes executam as operações descritas na seção anterior: classificação de partições (CP), checagem de utilização (CU), classificação de arquivos (CA), inferência (I) e execução (EX).

5.4.1 MobEntire

Um agente do tipo *MobEntire* executa todas as operações. Os sensores do agente percebem além do disco da máquina onde se encontra, todas as máquinas da rede em gerenciamento (Figura 5.17). Após terminar de agir na máquina, migra para uma outra máquina.

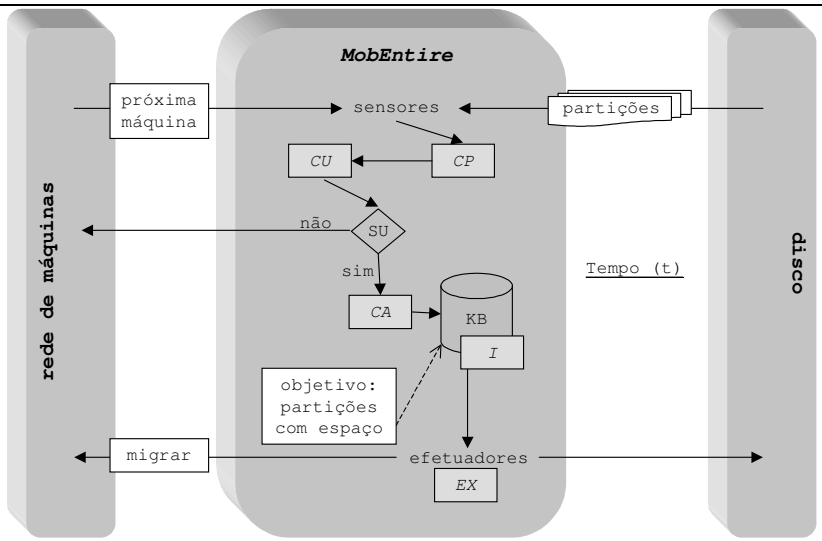


Figura 5.17 - Arquitetura do agente tipo *MobEntire*.

5.4.2 Entire

Um agente do tipo *Entire* se localiza em cada uma das máquinas a ser gerenciada e também acumula as cinco operações. A Figura 5.18 ilustra a arquitetura do agente.

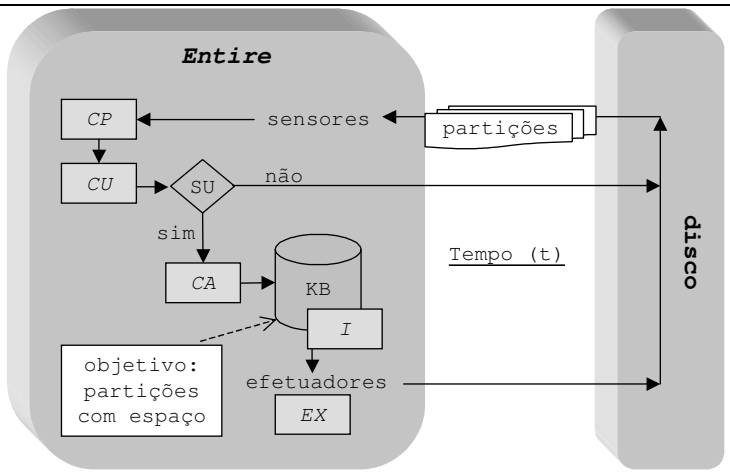


Figura 5.18 - Arquitetura do agente tipo *Entire*.

Os sensores do agente percebem as partições do disco da máquina onde está sendo executado, realiza as operações devidas e repete todo o processo de tempos em tempos.

5.4.3 FarEntire

Um agente desse tipo tem a visão de toda a rede de máquinas. O acesso a partições dos discos das máquinas da rede é feito remotamente. Eles executam, também remotamente, todas as operações e a operação de execução das ações de correção é feita também remotamente (Figura 5.19).

Essas operações remotas foram implementadas com duas tecnologias distintas: r*emote shell*, o*bjetos remotos*. No capítulo 6 os detalhes de implementação dessas tecnologias são evidenciados.

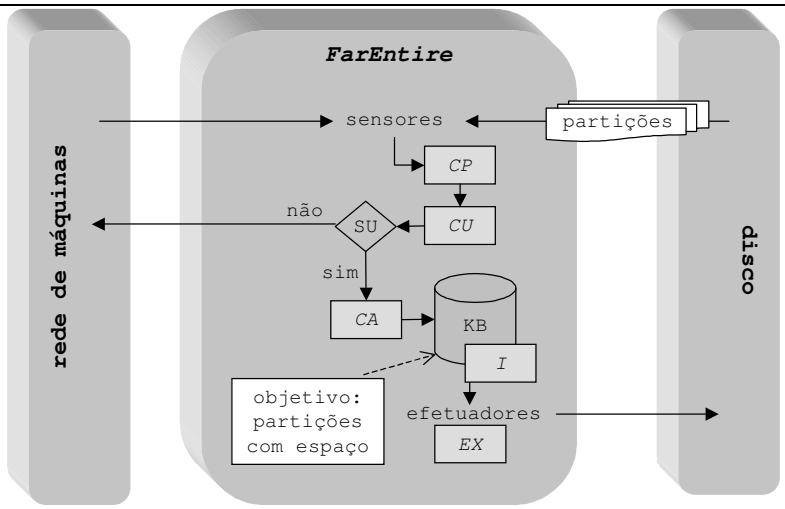


Figura 5.19 - Arquitetura do agente tipo *FarEntire*.

5.4.4 Sentinel

A função de um agente tipo *Sentinel* é monitorar o estado de utilização das partições dos discos das máquinas onde atua. Quando detecta algum nível crítico de utilização, avisa algum agente de decisão disponível na rede que possa resolver o problema. Um

Sentinel implementa as operações de classificação de partições, checagem de utilização e classificação de arquivos. A Figura 5.20 ilustra sua arquitetura.

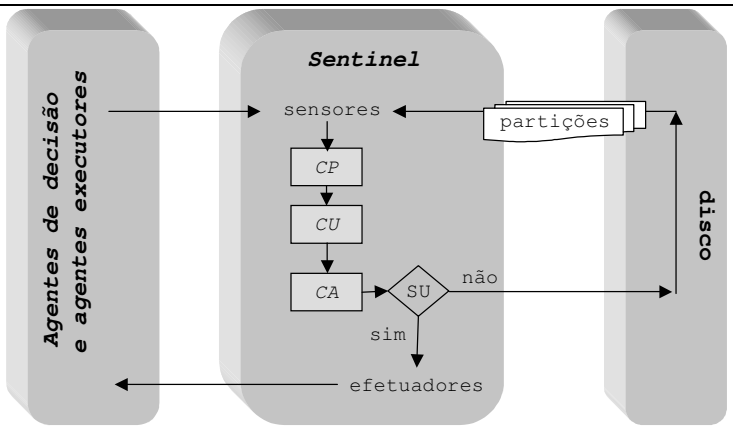


Figura 5.20 - Arquitetura do agente tipo *Sentinel*

5.4.5 MobSentinel

O MobSentinel é um tipo de agente que possui a mesma função do Sentinel, com a diferença de que não é estacionário. Assim que termina de executar em uma máquina, ele migra para outra máquina da rede. A Figura 5.21 ilustra sua arquitetura

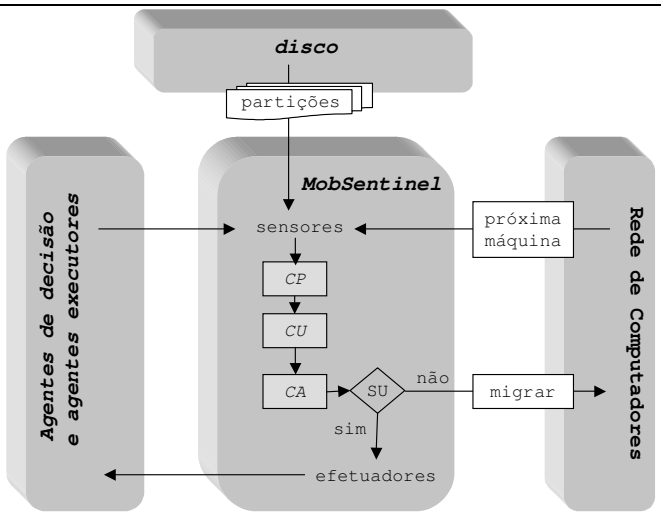


Figura 5.21 - Arquitetura do agente tipo *MobSentinel*.

5.4.6 MobDecideMobDoit

Um agente do tipo *MobDecideMobDoit* executa as operações de inferência e execução. A diferença dele para um do tipo *MobEntire* é que este migra incondicionalmente de máquina em máquina, enquanto que aquele só migra quando é chamado por um agente de percepção. A Figura 5.22 ilustra sua arquitetura.

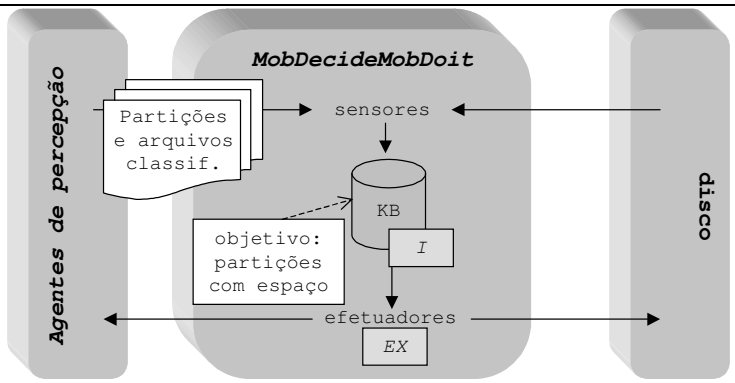


Figura 5.22 - Arquitetura do agente tipo *MobDecideMobDoit.*

5.5 Conclusões

Para que se pudessem validar as arquiteturas e os tipos de agentes definidos, fez-se necessário um estudo de caso. A administração do espaço em disco de redes UNIX/NFS mostrou-se um domínio bastante interessante para aplicação das arquiteturas.

A atividade consiste em checar periodicamente o nível de utilização de cada uma das partições de disco e determinar, segundo conhecimento especialista, se estão super utilizadas ou não. As ações de correção para este tipo de problema consistem de ações sobre arquivos dos discos. Assim, arquivos podem ser removidos, compactados, ou movidos para outras partições, dependendo apenas do tipo do arquivo (texto, imagem, log, etc.) e do tipo de partição onde se encontra.

A automação da atividade de gerenciamento foi dividida em cinco operações principais: classificação das partições de um disco, checagem do grau de utilização das partições, classificação dos arquivos presentes nas partições, decisão sobre ações de correção e execução dessas ações.

Dependendo do tipo, os agentes podem implementar todas ou apenas algumas dessas operações.

O próximo capítulo descreve o simulador desenvolvido para viabilização dos experimentos.

6

Simulador de Arquiteturas

"Vejo, / Como nos contos da mitologia, / Que a água verde do mar simula um prado imenso". (Martins Fontes, *Verão*, p. 48).

Como mostrado no capítulo 4, a construção de um simulador de gerenciamento faz parte da metodologia empregada no trabalho.

A intenção com a criação do simulador foi a de possibilitar que um maior número de experimentos pudessem ser realizados de maneira mais controlada. A dificuldade em se obter exclusividade de uma rede para testes reais aliada as variações de desempenho que uma rede pode ter, iria dificultar ou até impossibilitar que testes significativos fossem realizados. Com o simulador é possível variar desde o número de dispositivos numa rede até cenários diferentes para estado de funcionamento dos mesmos.

O simulador serve para verificar o desempenho das arquiteturas propostas levando-se em consideração o custo associado a cada tipo de operação em rede: mensagens remotas e locais, migração de código, multicast de mensagens, consumo de CPU e consumo de espaço em disco para todas estas operações, etc.

Neste capítulo será mostrado como está organizado o simulador, as classes desenvolvidas, a modelagem das classes para o estudo de caso, o processo de funcionamento, além de detalhes da implementação das operações citadas anteriormente.

6.1 Organização do Simulador

A Figura 6.1 ilustra a interface do *Agents Simulator for Network Management*.

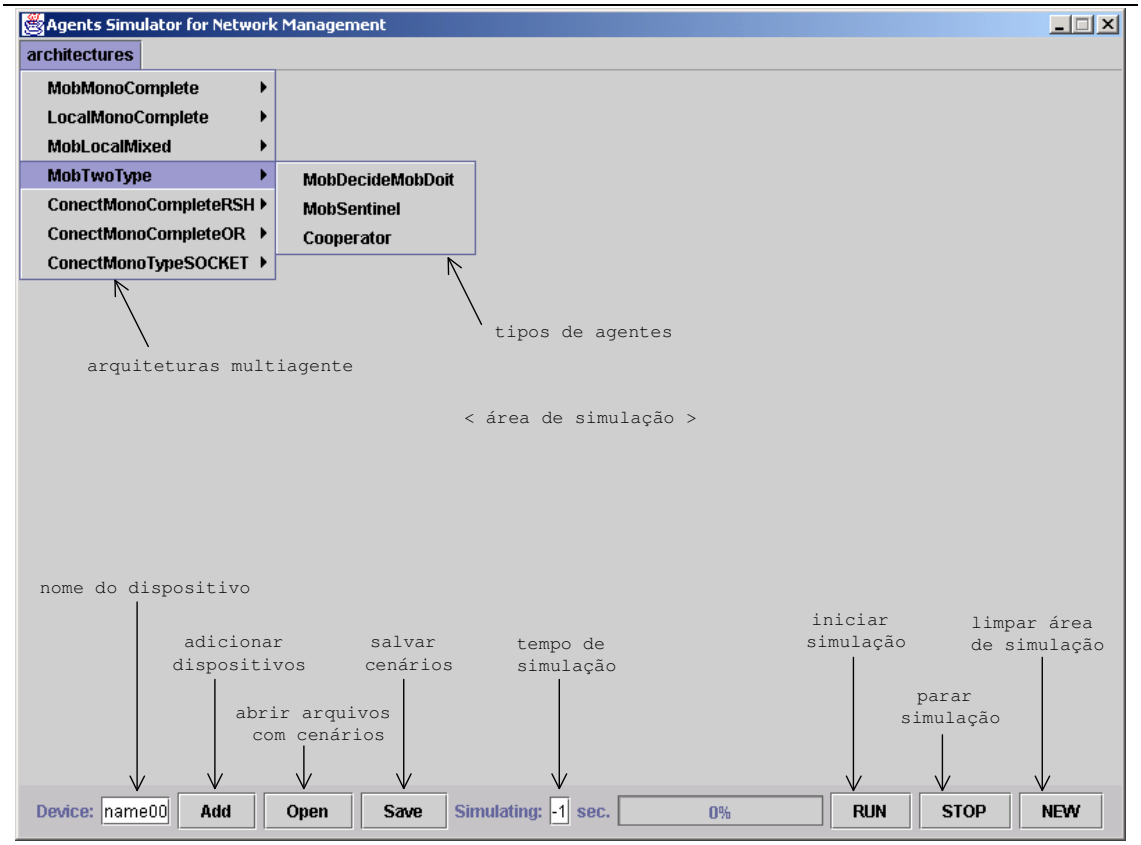


Figura 6.1 - Interface do Simulador

O simulador foi construído de forma bem modular para que funcionasse independentemente da aplicação de gerenciamento. As classes iniciadas com "S" constituem a API para criação de todo um ambiente de simulação de uma atividade de gerenciamento qualquer.

O desenvolvedor pode definir arquiteturas de agentes, tipos de agentes, e definir o tipo de dispositivo desejado.

A Figura 6.2 ilustra o diagrama de classes[1] da API do simulador.

[1] Foi utilizado o padrão UML [55] para modelagem do sistema.

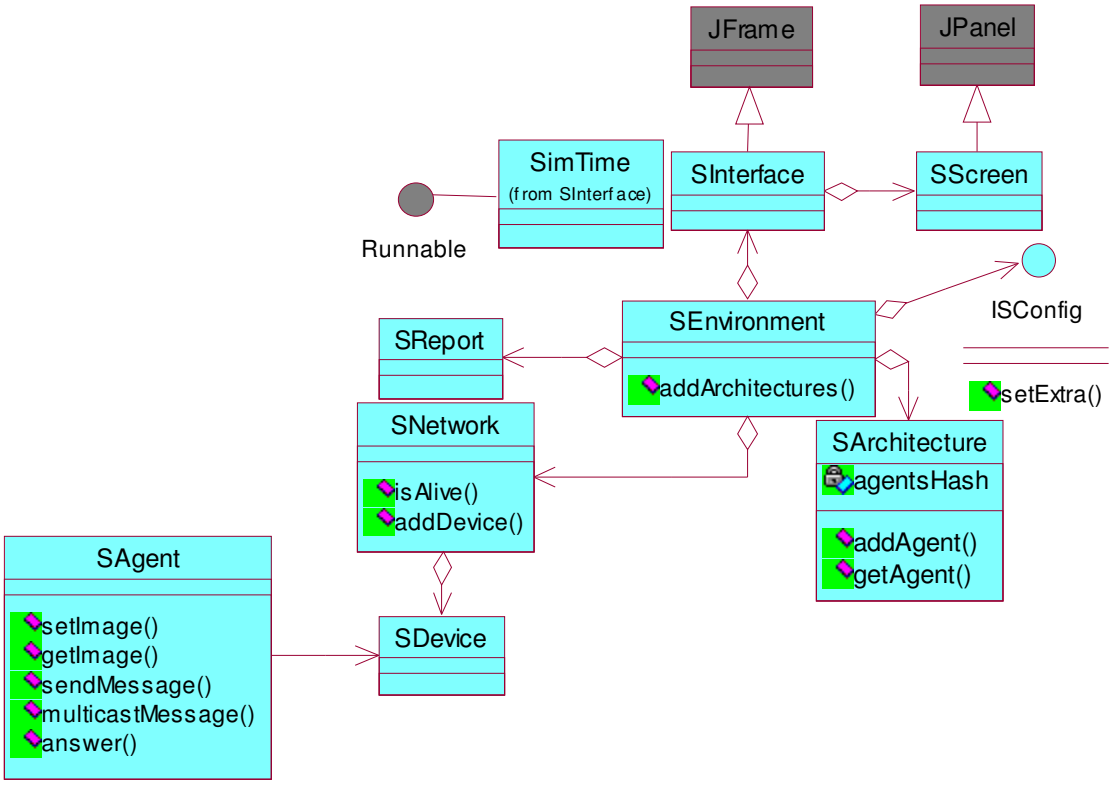


Figura 6.2 - Diagrama de classes da interface do simulador.

A classe `SInterface` é responsável por construir a janela principal do simulador. É nela que estão implementados métodos para construção de menus e botões e os eventos de cada um. Ela possui uma classe interna responsável por controlar o tempo de simulação (`SimTime`). A classe `SScreen`, parte da `SInterface`, é responsável pelo controle das imagens na tela do simulador. Ela implementa métodos para adição de imagens (de dispositivos ou de agentes), remoção das imagens, *refresh* da tela de simulação, enfim, é a classe responsável por controlar o mecanismo de animação das imagens na tela.

A classe `SEnvironment` é a classe que controla o ambiente de simulação. Possui estruturas de armazenamento para as arquiteturas definidas, a rede e o conjunto de dispositivos.

`SReport` é uma classe responsável pela geração do relatório de execução de uma simulação de gerenciamento. O relatório consiste de informações sobre o tempo total de simulação, número de dispositivos em rede, número de agentes, arquitetura utilizada, e outras informações específicas do tipo de gerenciamento em questão.

Alguns tipos de gerenciamento podem necessitar de informações adicionais. A API provê uma interface `ISConfig` que define um método (`setExtra`) para leitura de uma base de dados contendo essas informações adicionais. Por exemplo, para o gerenciamento do espaço em disco, essa base de dados contém as definições do administrador sobre limites de utilização de partições (ver seção 5.3.2).

`SArchitecture` é a super classe de todas as arquiteturas definidas. Uma classe de arquitetura herda métodos para adição, remoção e manipulação dos agentes pertencentes. Ela implementa uma tabela *hash* (`agentsHash`) que mantém todos os agentes presentes na arquitetura e em que dispositivos estão localizados.

A classe `SAgent` define métodos para comunicação entre agentes que devem ser implementados de acordo com as características dos tipos de agentes definidos e o tipo de atividade de gerenciamento em questão.

A Figura 6.3 ilustra o diagrama de classes das arquiteturas, tipos de agentes e tipo de dispositivo definidos para o problema do gerenciamento do espaço em disco.

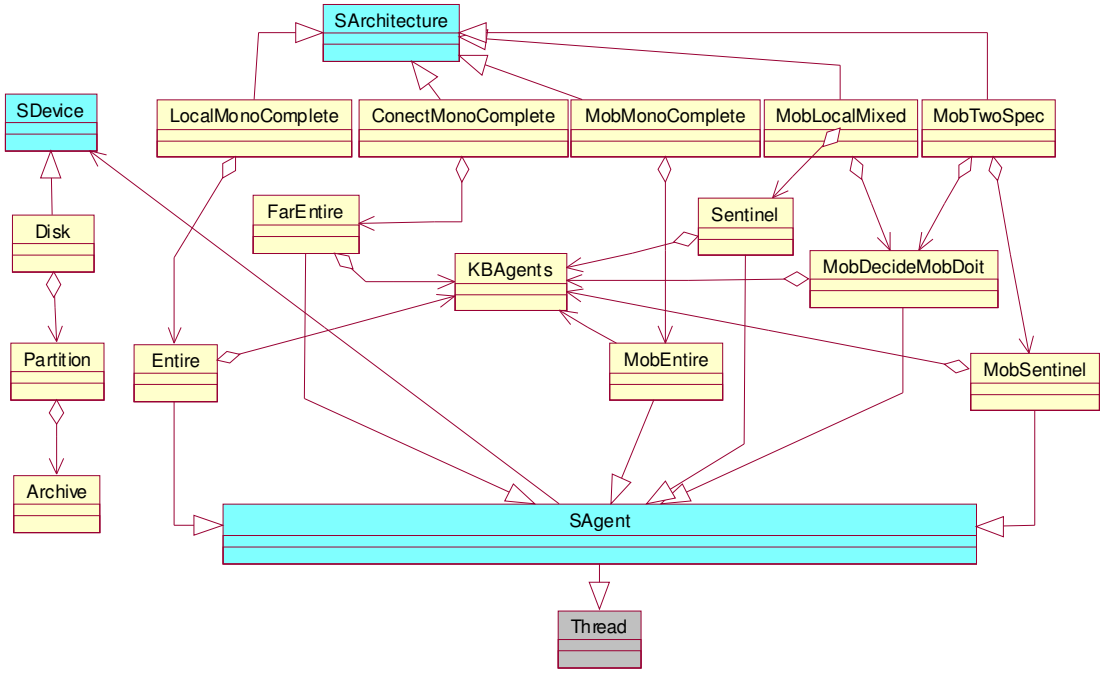


Figura 6.3 - Classes de arquiteturas e de agentes.

Pode-se observar que as arquiteturas e tipos de agentes ilustrados aqui são os mesmos escolhidos na seção 5.4 para o estudo de caso.

Os tipos de agentes existentes variam com o tipo da arquitetura. Por exemplo, pode-se observar que a arquitetura `MobLocalMixed` agrega dois tipos de agentes, o `Sentinel` e o `MobDecideMobDoit` (a composição das arquiteturas utilizadas pode ser vista na seção 5.4.)

Existe uma associação entre a classe `SAgent` e a classe `SDevice` porque o agente tem de enxergar o dispositivo onde é executado.

A classe `KBAgents` representa a base de conhecimento dos tipos de agentes. Como foi observado na seção 5.4, agentes como o `Sentinel` ou o `MobSentinel` não possuem uma base de conhecimento e, por esse motivo, não estão agregados a uma classe desse tipo.

As classes `Disk`, `Partition` e `Archive` representam o disco de uma máquina, a partição de um disco e os arquivos de uma partição. Essas classes são específicas para o estudo de caso que utilizamos. A classe `Partition` dispõe de métodos para criar arquivos de diferentes tamanhos, datas e tipos. O processo de preenchimento de disco com partições e de partições com arquivos, será explicado nas seções seguintes.

A API do simulador possui ainda duas outras interfaces muito importantes (Figura 6.4). A interface `ISParameter` deve ser implementada por uma classe que contenha constantes com valores calculados de eventos específicos da atividade de gerência em questão. Por exemplo, no caso da administração do espaço em disco, existem constantes com os valores de tempo de processamento e consumo de CPU para cada uma das operações descritas na seção 5.3, além de outras para operações que serão explicitadas mais adiante. A interface `ISEvent` define um método para adição de eventos ocorridos na simulação e uma lista de eventos para armazenar os eventos ocorridos. A próxima seção mostrará como funciona todo esse processo.

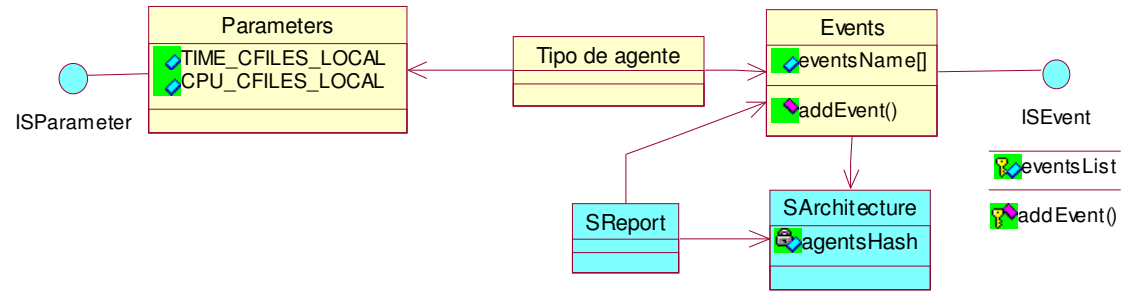


Figura 6.4 - Agentes, parâmetros e eventos.

Com informações sobre os eventos ocorridos e os dispositivos onde ocorreram, pode-se gerar um relatório da simulação.

No total, o *Agents Simulator for Network Management* consiste de ~ 6200 linhas de código Java que implementam a interface gráfica do sistema, as classes da API, e as classes do estudo de caso utilizado, e mais ~ 4000 linhas de código de implementação das operações para coleta de valores reais na rede.

6.2 Simulação do Gerenciamento do Espaço em Disco

Esta seção irá descrever o processo de operação e funcionamento do simulador a partir de um exemplo de simulação para a atividade de gerenciamento de espaço em disco. Vai-se considerar para o exemplo, a simulação da arquitetura `MobLocalMixed` com agentes do tipo `Sentinel e MobDecideMobDoit`. A rede será composta de 10 máquinas.

6.2.1 Adicionando máquinas

O primeiro passo para a construção do cenário de simulação é adicionar as máquinas da rede. Para cada máquina adicionada, aparece uma caixa de diálogo com opção para especificar o *tipo* da máquina. Foram concebidos cinco tipos diferentes de máquinas segundo o nível de utilização das partições: `Normal`, `NormalLimit`, `Problem`, `LargerProblem`, e `ReallyBigProblem`. Também é permitida a seleção dos tipos de partições desejados em cada uma das máquinas (Figura 6.5).

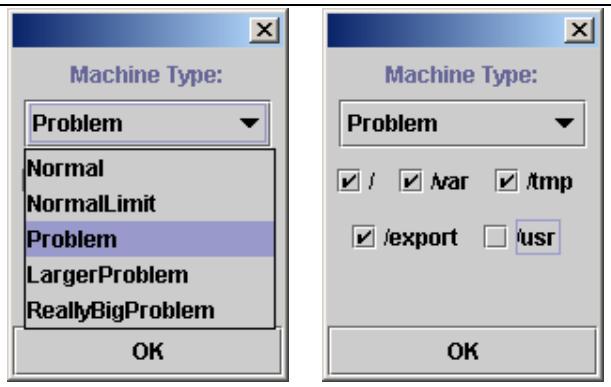


Figura 6.5 - Caixa para seleção do tipo de máquina e das partições presentes.

O método para geração de arquivos para partições se baseia no tipo da máquina adicionada e numa tabela contendo os limites de utilização (inferior e superior) de cada uma das partições (Figura 6.6). Essa tabela é lida, utilizando-se o método `setExtra` definido na interface `ISConfig`.

Esse método é apresentado em pseudocódigo na Figura 6.7.

Partições	Size	I	S
/	100000	88	91
/var	100000	50	65
/tmp	100000	50	80
/export	100000	70	80
/usr	100000	70	81

Figura 6.6 - Arquivo com valores de limites para partições

```
fillPartition(MachineType mt, LowerLimit ll, UpperLimit ul) {
   Se mt é do tipo Normal então:
      Enquanto a % de utilização da particao for ≤ (ll-5) então:
         generateArchive();
   Se mt é do tipo Problem então:
      Enquanto a % de utilização da particao for ≤ ul então:
         generateArchive();
   ...
}

generateArchive() {
      gere um nome aleatório para o arquivo
      atribua um tipo (“txt”,”img”,”core”,”mail”,”log”) aleatoriamente
      gere um tamanho aleatorio de 1 a 5000
      gere um tempo de criação aleatório (entre 1 e 100 dias)
}
```

Figura 6.7 - Prenchimento de partições com arquivos

A Figura 6.8 mostra a tela do simulador após a adição de todas as 10 máquinas. As máquinas que necessitam de gerenciamento (tipos `Problem`, `LargerProblem` e `ReallyBigProblem`) apresentam a tela na cor preta. É importante deixar claro que uma vez definidas, as máquinas não mudam, a não ser pela intervenção dos agentes. Não era objetivo simular eventuais ações (criação e remoção de arquivos, por exemplo) de usuários de uma rede concomitantemente a atividade de gerenciamento. O objetivo era simplesmente verificar o comportamento dos diversos tipos de agentes em um dado cenário.

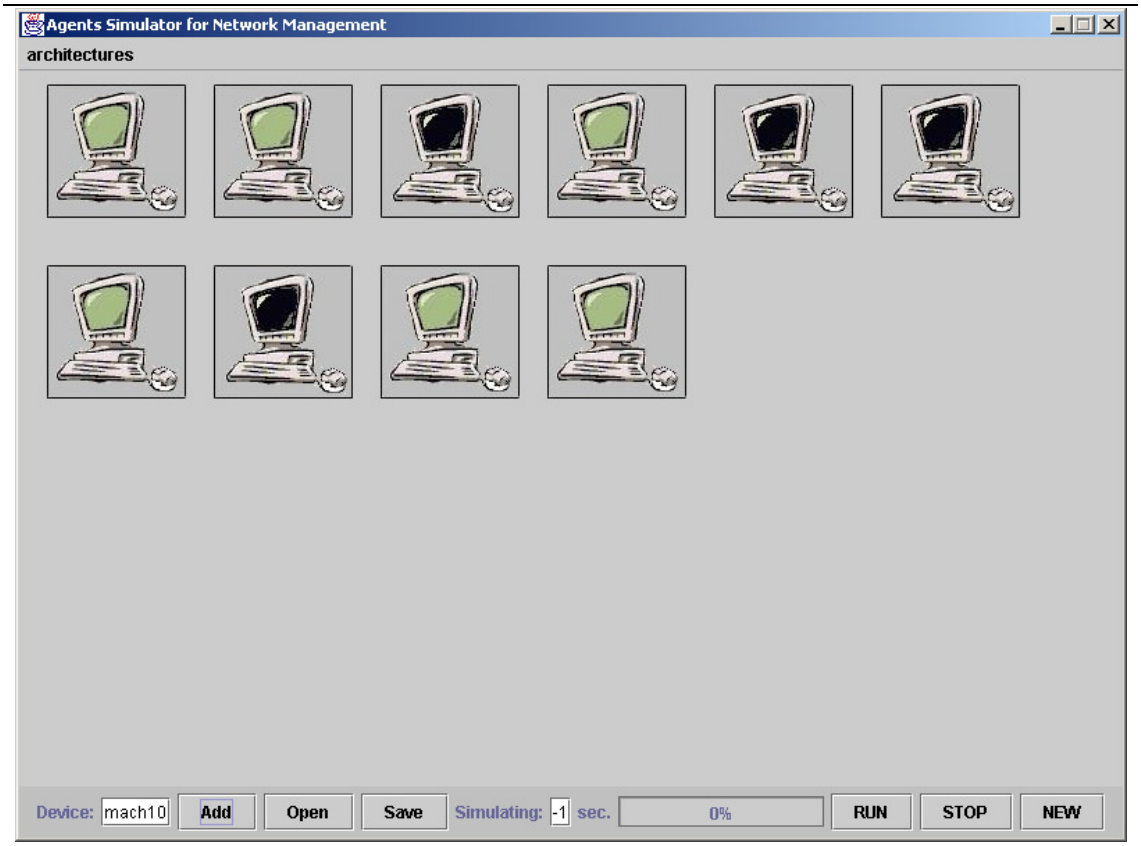


Figura 6.8 - Tela do simulador com 10 máquinas

6.2.2 Populando o ambiente

Em seguida, é necessário escolher a arquitetura para a simulação. A Figura 6.9 ilustra o processo de escolha da arquitetura `MobLocalMixed` e de cada um dos tipos de agentes disponíveis para a arquitetura: `Sentinel` e `MobDecideMobDoit`. Para se adicionar um agente numa determinada máquina, basta selecioná-lo e clicar com o mouse sobre a máquina desejada. É importante dizer, que o simulador realiza o controle de eventuais inconsistências na simulação. Por exemplo, não permite que mais de um agente de um mesmo tipo seja adicionado a uma mesma máquina.

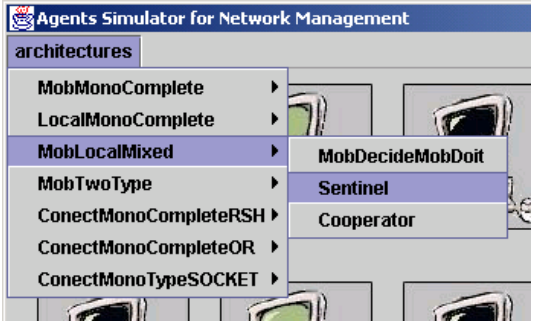


Figura 6.9 - Escolha dos agentes para a arquitetura `MobileTwoType`

Após ter se populado a rede com os agentes, a tela de simulação ficará como mostra a Figura 6.10.

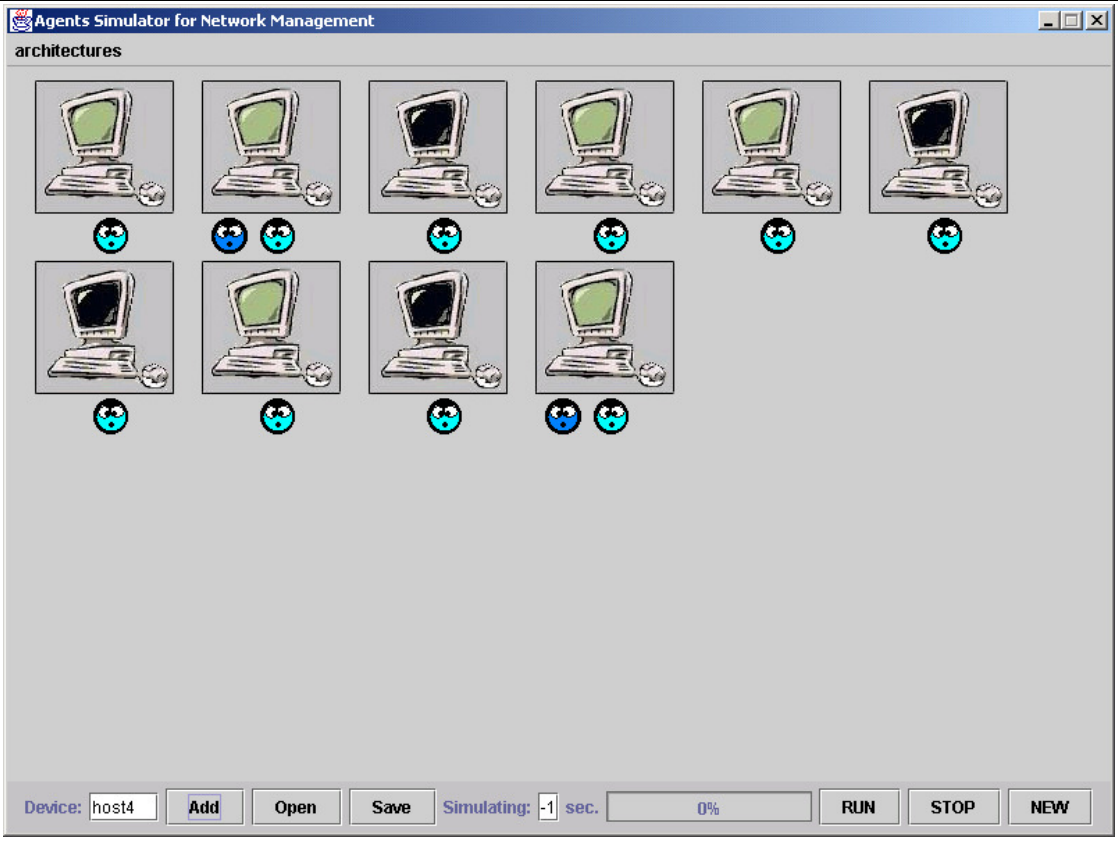


Figura 6.10 - Dois agentes `MobDecideMobDoit` e vários `Sentinels`

Quando um agente é inserido na rede, ele fica armazenado na tabela hash `agentsHash` mantida pela classe `MobLocalMixed`. A tabela guarda uma referência para o agente juntamente com uma referência para a máquina onde ele se encontra no momento. O esquema de nomeação para os agentes é implementado da seguinte maneira: o primeiro agente de um certo tipo é identificado pelo seu tipo mais o número 1, para o segundo adiciona-se o número 2 e assim sucessivamente. Nesse exemplo, o primeiro agente `MobDecideMobDoit` adicionado será identificado por `MobDecideMobDoit1` e o segundo por `MobDecideMobDoit2`. Da mesma forma, os agentes `Sentinel` serão identificados por `Sentinel1`, `Sentinel2`, `Sentinel3`, `...`, `Sentinel10`.

6.2.3 Um, dois, três, *fire*!

Quando se inicia a simulação, são disparados vários *Threads* que executarão concorrentemente, um para cada agente presente. Os agentes `Sentinel` monitoram o estado do disco da máquina onde se encontram. Se a máquina possui alguma partição com super-utilização, o agente requisita a ação de um agente `MobDecideMobDoit`, fazendo um multicast da mensagem para todo o grupo de agentes presentes na arquitetura. O pseudocódigo da Figura 6.11 mostra como funciona todo esse processo.

```
act() {
   Enquanto não terminar simulação:
      Verifique se a máquina está ok
      Se a máquina não estiver ok:
         Procure por um agente MobDecideMobDoit na tabela agentsHash
         que não esteja em execução no momento (multicast da mensagem)
         Para ∀ partição super-utilizada:
            Espere MobDecideMobDoit terminar de inferir
}
```

Figura 6.11 - Ação do agente `Sentinel`

Quando o agente `MobDecideMobDoit` atende ao chamado e migra para uma máquina para tentar resolver o problema, ele insere em sua base de conhecimento (`KBAgents`) os

arquivos das partições com problema e executa as ações decididas. A Figura 6.12 ilustra esse processo em pseudocódigo.

```
act() {
   Insere na base de conhecimento os arquivos da partição
   Para cada arquivo:
      Executa a ação inferida pela base (remoção, compactação, etc.)
}
```

Figura 6.12 - Ação do agente `MobDecideMobDoit`

Ao final da simulação, pode ser que todos os problemas de todas as máquinas sejam resolvidos ou de apenas algumas. Isso depende dos tipos das máquinas e do tempo de simulação.

6.3 Calibrando o Simulador

Esta seção irá descrever como funciona o processo de calibração do simulador. Na verdade, esse processo está relacionado com a definição dos parâmetros da classe `Parameters` (ver Figura 6.4).

A adaptação das arquiteturas definidas ilustrada na seção 5.4 mostrou que, além de realizarem as operações básicas descritas na seção 5.3, alguns tipos de agentes podem migrar, se comunicar com outro(s) agente(s) ou mesmo agir remotamente em outras máquinas.

Calibrar o simulador significa calcular o impacto (em termos de tempo de processamento, consumo de CPU, etc.) que cada uma dessas operações causa realmente numa rede e embutir esses valores no código do simulador, e assim poder efetivamente avaliar as arquiteturas.

Um agente do tipo FarEntire gerencia máquinas remotamente, porém existem várias maneiras de se realizar esse gerenciamento e cada uma terá um impacto diferente na performance. Decidimos por utilizar duas formas distintas de implementação para esse

gerenciamento remoto: via chamada de sistema *Remote Shell* (RSH) do UNIX e via *objetos remotos*.

Dessa forma, as operações básicas executadas pelos agentes foram implementadas de três formas distintas: uma para execução local, uma para execução remota via RSH e outra para execução remota via objetos remotos.

Também foram implementados códigos para possibilitar medição do custo de migração de agentes e de comunicação entre os mesmos.

Assim, foram desenvolvidos códigos de implementação de cada uma das operações básicas, código de implementação de migração de agentes, e código de implementação e de comunicação entre agentes. A tabela com os valores das medições aparece na seção 6.3.2.

6.3.1 Códigos de Teste

Pelo menos duas ferramentas de suporte fizeram-se necessárias além da linguagem de programação utilizada: uma *plataforma de distribuição* para se implementar os objetos remotos; e uma *plataforma de mobilidade* para implementar migração de agentes.

A plataforma de distribuição utilizada foi o Visibroker (CORBA [51]) da Borland por estar disponível nos laboratórios do CIn com compilador IDL para Java. A plataforma de mobilidade utilizada foi o Voyager [52] da ObjectSpace por ser 100% Java compatível, possuir um vasto acervo de documentação disponível e por estar disponível nos laboratórios do CIn.

Detalhes sobre o tratamento da mobilidade de agentes e da comunicação entre os agentes feito por Voyager são mostrados no Anexo B.

As operações básicas descritas no capítulo anterior foram implementadas com a tecnologia de objetos distribuídos CORBA. O objeto que implementa as operações localiza-se na máquina a ser gerenciada, e seus métodos são acessados remotamente

pelo objeto que representa o agente (`FarEntire`) e que se localiza numa outra máquina (Figura 6.13). O Anexo C traz mais informações sobre objetos remotos e CORBA.

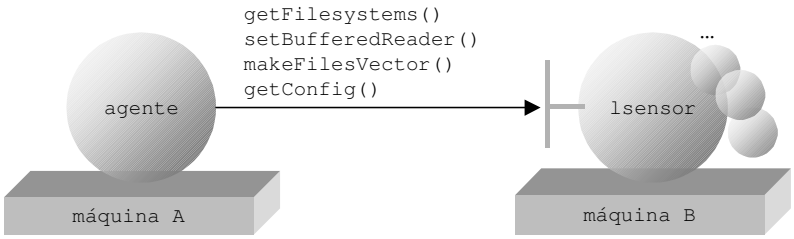


Figura 6.13 - O agente acessa a interface do objeto `lsensor`.

A definição completa da interface IDL CORBA pode ser vista no Anexo D.

As operações também foram implementadas com remote shell do UNIX. No UNIX os destinos das mensagens são especificados por endereços de *socket* – um endereço de *socket* é um identificador de comunicação que consiste do número de uma porta local e um endereço *Internet*[2]. O mecanismo de RSH é uma chamada de sistema UNIX que utiliza o modelo de comunicação com sockets do tipo *stream*. Ele é utilizado para executar um comando em um computador remoto com o input e output padrão conectado, via canais TCP/IP, a processos locais. Qualquer computador que aceite estas chamadas remotas age como servidores para cada uma destas requisições – escutando pedidos de conecções em sockets ligados a endereços contendo números de porta TCP/IP reservadas.

A Figura 6.14 ilustra códigos Java que representam trechos de chamadas com "rsh" relativos à listagem de partições de um *host*, listagem de arquivos presentes em um diretório, informações sobre um determinado arquivo, e remoção de um arquivo, respectivamente.

[2] Mais informações sobre Redes de Computadores em [64]

```
Process p = Runtime.getRuntime().exec("ssh "+host+" /bin/df -kl");
Process p = Runtime.getRuntime().exec("ssh "+host+" /bin/ls -lAc " + str);
Process p = Runtime.getRuntime().exec("ssh "+host+" /bin/file " + str);
Process p = Runtime.getRuntime().exec("ssh "+host+" rm "+name);
```

Figura 6.14 - Códigos Java para chamadas remotas (rsh) UNIX

6.3.2 Testes em Rede

Para calibrar o simulador, os códigos de implementação das operações básicas para processamento local, processamento via rsh e processamento via objetos remotos, os códigos de implementação para migração de agentes com base de conhecimento (`MobEntire`) e sem base de conhecimento (`MobSentinel`), os códigos para implementação de comunicação entre agentes e os códigos para inicialização de plataformas (CORBA e Voyager) foram executados em uma rede real e os tempos de processamento e consumo de CPU foram observados. Cada uma das operações foram executadas 100 vezes.

As tabelas seguintes mostram os valores médios obtidos para o tempo de processamento (em milisegundos) e consumo de CPU (% de utilização) para cada uma dessas operações[3]: a Tabela 6.1 mostra os valores médios obtidos para cada uma das operações básicas implementadas de três formas distintas; a Tabela 6.2 mostra os valores médios para operações de comunicação entre agentes e de migração de agentes; e a Tabela 6.3 mostra os valores médios para operações de inicialização de plataformas.

[3] Na verdade, os valores para tempo de processamento são para cada iteração de uma operação. Por exemplo, para o caso da operação de classificação de partições, o valor é do tempo necessário para classificação de apenas uma delas.

Operação básica	Execução Local		Objetos Remotos		Remote Shell	
	tp	cc	tp	cc	tp	cc
Classificação de Partições	9.0	1%	90.0	1%	399.0	1%
Checagem de Utilização	1.0	1%	110.0	1%	698.0	1%
Classificação de Arquivos	111.0	12%	325.0	15%	1896.0	14%
Decisão sobre Ações de Correção	0.36	2%	13.3	9%	0.36	2%*
Execução das Ações Inferidas	125.0	3%	259.0	3%	160.0	3%

Tabela 6.1 - Tempo de processamento (tp) e consumo de CPU (cc) para operações básicas.

Migração				Comunicação			
KB		sem KB		Unicast		Mulicast	
tp	cc	tp	cc	tp	cc	tp	cc
430	10%	120	6%	306.0	1%	145.0	4%

Tabela 6.2 - Medidas para Comunicação entre agentes e Migração de agentes.

Operações	Voyager		CORBA	
	tp	cc	tp	cc
Inicialização de Plataforma	5399.0	10%	2977.0	8%
Bind (ligação ao objeto)	-	-	2855.0	8%

Tabela 6.3 - Medidas para Inicialização de Plataforma e Bind de Objetos.

6.3.3 Ajuste dos Parâmetros

Estes valores obtidos são representados por constantes estáticas presentes na classe `Parameters`. A Figura 6.15 mostra a definição de algumas dessas constantes:

* Na fase de raciocínio não há chamada remota. O processo é feito localmente. Daí a justificativa por valores iguais aos das duas primeiras colunas da tabela.

```
public class Parameters implements ISParameters {
   public static final int TIME_INIT_VOYAGER = 5399;
   public static final int CPU_INIT_VOYAGER  = 10;
   public static final int TIME_INIT_CORBA   = 2977;
   public static final int CPU_INIT_CORBA    = 8;
   ...
   public static final int TIME_CPART_OR     = 90;
   public static final int CPU_CPART_OR      = 1;
   ...
   public static final int TIME_CFILES_LOCAL = 111;
   public static final int CPU_CFILES_LOCAL  = 12;
   ...
   public static final int TIME_MULTICAST    = 145;
   public static final int CPU_MULTICAST     = 4;
   ...
}
```

Figura 6.15 - Classe `Parameters`

As duas primeiras constantes referem-se, respectivamente, ao tempo médio de processamento da inicialização da plataforma Voyager e o consumo de CPU para a mesma (conferir valores dessas operações na Tabela 6.3). As outras medidas são representadas da mesma forma.

Para que haja um melhor entendimento sobre o mecanismo de acumulação desses valores durante uma simulação, seja o exemplo concreto de uma simulação da arquitetura MobMonoComplete com um único agente do tipo MobEntire (Figura 6.16).

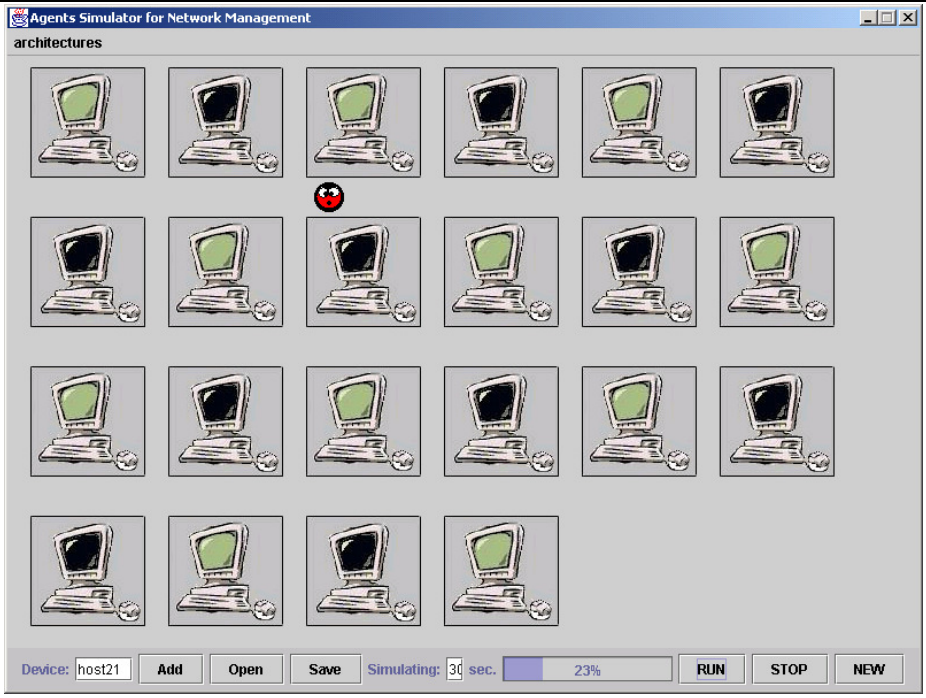


Figura 6.16 – Simulação da arquitetura MobMonoComplete com um agente MobEntire

Esse exemplo envolve as seguintes operações:

- Inicialização da plataforma Voyager;
- Classificação das partições;
- Checagem de Utilização;
- Classificação dos arquivos;
- Decisão sobre ações de correção;
- Execução das ações.
- Migração com base de conhecimento.

Para cada uma delas, está reservada uma variável que acumulará o tempo total consumido. O pseudocódigo da Figura 6.17 ilustra como funciona o processo de acumulação desses tempos.

```
initSimulation() {
   tot_init_voyager ← Parameters.TIME_INIT_VOYAGER
   Inicia thread de execução do agente MobEntire
   Enquanto não terminar simulação:
      tot_cpart_local ← Parameters.TIME_CPART_LOCAL x NP
      tot_checkUtil_local ← Parameters.TIME_CHECKUTIL_LOCAL x NP
      Para cada partição super utilizada:
          Insere na KB a patição
          Para cada arquivo presente na partição:
              tot_cfiles_local ← Parameters.TIME_CFILES_LOCAL
              Insere o arquivo na KB
          Insere na KB um contador de chamadas C
          Insere na KB um contador de decisões tomadas D
          Executa KB
          tot_decision_local ← Parameters.TIME_DECISION_LOCAL x C
          tot_exec_local ← Parameters.TIME_EXEC_LOCAL x D
      migração()
}

migração() {
   tot_KBmigration ← Parameters.TIME_KBMIGRATION
}

/* onde,
   NP = número de partições existentes na máquina onde está executando
   tot_... = variável que acumula os tempos das operações definidos na
             classe Parameters
   KB = base de conhecimento do agente
*/
```

Figura 6.17 – Processo de acumulação dos tempos definidos na classe `Parameters`

A classe `Events` contém uma lista `eventsName[]` (ver Figura 6.4) com os *nomes de todos os eventos* possíveis para o caso da administração de espaço em disco. Entre eles estão os seguintes nomes que se aplicam para o exemplo em questão: “Inicialização do Voyager”, “Classificação das Partições”, “Checagem de Utilização”, “Classificação dos Arquivos”, “Decisão sobre ações de correção”, “Execução das Ações” e “Migração com base de conhecimento”.

O método `addEvent(..)`, implementado na classe `Events`, é responsável por criar um evento e adicioná-lo na lista de eventos `eventsList[]`. Esta lista contém informação sobre o nome do evento, o tempo total consumido pelo mesmo (variáveis `tot_...` da

Figura 6.17), o consumo de CPU (obtido diretamente das constantes da classe `Parameters`), a máquina onde ocorreu e o agente que o executou (obtidos da classe `SArchitecture`) (ver relações entre classes na Figura 6.4). A Figura 6.18 mostra o exemplo de dois eventos possíveis para o exemplo.

```
Classificação dos Arquivos → TIME: 14315ms, CPU: 12% in host01 by MobEntire1
Migração                   → TIME:   430ms, CPU: 10% in host13 by MobEntire1
```

Figura 6.18 – Exemplo de dois eventos gerados durante a simulação

6.4 Conclusões

Este capítulo descreveu alguns detalhes da implementação do simulador e dos códigos de testes desenvolvidos para a calibração do mesmo. O simulador é geral e serve para aplicação de um outro estudo de caso de administração de sistemas, que não a administração do espaço em disco.

O próximo capítulo apresenta os resultados e discussões sobre os experimentos realizados.

7

Experimentos e Resultados

Este capítulo apresenta os resultados de diversos experimentos realizados com as arquiteturas definidas anteriormente.

As arquiteturas foram submetidas a alguns cenários diferentes de gerenciamento (situações distintas de gerenciamento) e a performance global foi observada.

7.1 Cenários

Os cenários de gerenciamento podem variar segundo muitas propriedades. Nos limitamos a algumas que consideramos ser suficiente para resultados conclusivos:

- Dimensão da rede;
- Tipos de máquinas;
- Estabilidade da rede;

- Dinamicidade da rede.

A dimensão de uma rede se refere ao número de máquinas gerenciáveis presentes nela. Realizamos experimentos com diferentes números de máquinas.

O tipo de máquinas utilizadas se refere ao grau de utilização de partições das mesmas. Foram feitas simulações com em redes com máquinas com grau de super-utilização pouco acima do limite, muito acima do limite e máquinas sem problemas de super utilização.

A estabilidade de uma rede refere-se ao fato da rede estar susceptível à perda de conexão repentina ou não.

Uma rede dinâmica é uma rede onde máquinas podem ser adicionadas ou retiradas a qualquer momento.

Pretendeu-se testar com estas propriedades, a escalabilidade das arquiteturas, a tolerância a falhas, o tempo de processamento, consumo de CPU, consumo de espaço em disco e latência de gerenciamento.

Variando-se as arquiteturas em termos de número de agentes presentes, número de agentes móveis e a autonomia dos agentes, foi possível avaliar a influência da *mobilidade*, *autonomia* e *distribuição* numa solução para o gerenciamento corporativo de sistemas.

7.2 Hipóteses gerais

H1 A mobilidade contribui para redução do consumo de CPU;

H2 Com mobilidade, há uma redução no consumo de espaço em disco total na rede;

H3 Mobilidade contribui para um gerenciamento mais escalável e robusto;

H4 Quanto menor o grau de autonomia em agentes móveis, menor o *overhead* gerado pela migração;

H5 Agentes especializados otimizam o gerenciamento;

H6 Um número maior de agentes móveis otimiza o gerenciamento;

H7 Quanto maior o número de agentes, maior o *overhead* de comunicação.

7.3 Resultados dos Experimentos

7.3.1 Primeira série de experimentos

O primeiro experimento foi conduzido em uma rede estática e estável, com 5 máquinas livres de problemas de super utilização em partições. Apenas as operações de classificação de partições e checagem de utilização foram realizadas nesse caso já que não havia problemas de super utilização. O tempo de simulação foi limitado em 20 segs. Dentro desse tempo, a arquitetura ConectMonoComplete com RSH e com Objetos Remotos (que denominaremos de ConectMonoCompleteRSH e ConectMonoCompleteOR, respectivamente) conseguiu avaliar apenas 4 máquinas.

O gráfico da Figura 7.1 mostra o número de vezes que cada máquina foi avaliada dentro desse tempo para cada uma das demais arquiteturas, que conseguiram avaliar todas as máquinas.

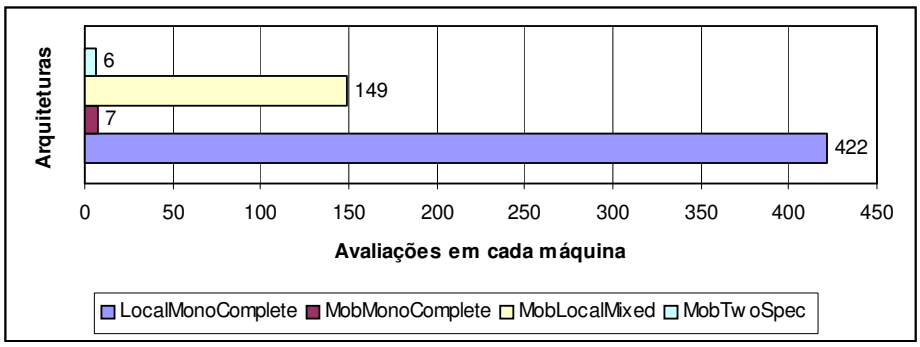


Figura 7.1 - Avaliações em máquinas numa rede com 5 máquinas sem problemas de super utilização

Com base no gráfico acima podemos concluir que as arquiteturas LocalMonoComplete e MobLocalMixed fizeram muitas avaliações em cada uma das máquinas. Para o caso específico da administração de espaço em disco, como o consumo de CPU para as operações de classificação de partições e checagem de utilização local é muito baixo, não há grandes problemas nesse sentido. Como não ocorrem migrações, não há problema de tráfego de rede também. As outras duas arquiteturas apresentaram valores de avaliações bem mais baixos e semelhantes entre si. Entretanto, na MobTwoSpec a atividade de monitoramento é realizada pelo agente especializado MobSentinel que possui apenas 15,9kb de tamanho contra os 42,2kb do agente MobEntire da outra arquitetura. Isso significa que o tráfego de rede gerado é maior nesta do que naquela.

7.3.2 Segunda série de experimentos

Aumentando o número de máquinas para 23 e o tempo de simulação para 50, observamos que as duas arquiteturas ConectMonoCompleteOR e ConectMonoCompleteRSH conseguiram avaliar apenas 12 e 10 máquinas, respectivamente. O gráfico da Figura 7.2 mostra o número de vezes que cada máquina foi avaliada nesse novo experimento.

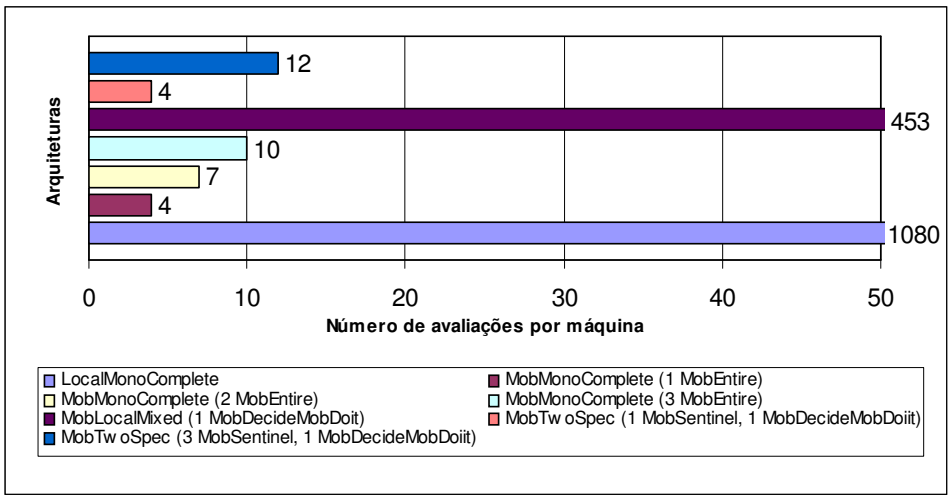


Figura 7.2 - Avaliações em máquinas numa rede com 23 máquinas sem problemas de super utilização

Como era de se esperar, as arquiteturas LocalMonoComplete e MobLocalMixed fizeram demasiadas avaliações em cada máquina. Na verdade, particularmente, para estas duas arquiteturas, o número de avaliações realizadas crescerá com o tempo de simulação, independentemente do número de máquinas que a rede contenha, já que cada máquina possui um agente responsável. Ao contrário, pode-se perceber comparando os dois gráficos, que para a arquitetura MobMonoComplete, apesar do aumento no tempo de processamento o número de avaliações por máquina baixou de 7 para 4 (para apenas um agente MobEntire nos dois casos). A arquitetura MobTwoSpec com apenas um agente MobSentinel mostrou-se equivalente à MobMonoComplete. Mas com três agentes MobSentinel, mais avaliações foram feitas do que três MobEntires realizaram. Isto se deve ao fato do tempo de migração do agente MobSentinel (que é mais leve) ser maior que o do MobEntire.

Em cenários com esses, onde a rede se constitui de máquinas com baixa taxa problemas de super utilização, se por um lado, arquiteturas que primam pela estaticidade de seus agentes pagam por processarem em demasiado numa máquina, arquiteturas com mobilidade pecam pelo alto número de migrações desnecessárias em rede.

A fim de evitar um número exagerado de avaliações desnecessárias, seria interessante prover esses agentes de alguma heurística para determinar quando devem dar uma pausa no processamento e quando devem reiniciar.

7.3.3 Terceira série de experimentos

O próximo experimento foi conduzido numa rede com 15 máquinas, das quais, 4 apresentavam problemas de super utilização. Em máquinas com problema, todas as cinco operações (seção 5.3) são realizadas. A Figura 7.3 apresenta os resultados desse experimento.

Arquitetura	tempo	latência	êxito (%)	migKB	mig	mgs
LocalMonoComplete	34	0,38	100,00			
ConectMonoCompleteOR	300		25,00			
ConectMonoCompleteRSH	300		25,00			
MobMonoComplete (1 MobEntire)	132	52,35	100,00	14		
MobMonoComplete (2 MobEntire)	75	23,55	100,00	38		
MobLocalMixed (1 MobDecideMobDoit)	89	47,05	100,00	4		915
MobLocalMixed (2 MobDecideMobDoit)	72	22,08	100,00	4		430
MobTwoSpec (1 MobDecideMobDoit, 1 MobSentinel)	89	47,70	100,00	4	40	18
MobTwoSpec (2 MobDecideMobDoit, 2 MobSentinel)	56	32,10	100,00	4	64	72
MobTwoSpec (2 MobDecideMobDoit, 4 MobSentinel)	58	32,10	100,00	4	207	732
MobTwoSpec (3 MobDecideMobDoit, 2 MobSentinel)	57	29,98	100,00	4	76	6
MobTwoSpec (4 MobDecideMobDoit, 2 MobSentinel)	60	30,98	100,00	4	87	8

Figura 7.3 - Resultados para uma rede com 4 máquinas com super utilização[1]

Nesse experimento, as arquiteturas ConectMonoComplete só conseguiram resolver o problema de 25% das máquinas dentro do tempo limite estipulado para simulação (300segs). A arquitetura LocalMonoComplete apresentou níveis comparativamente muito baixos de tempo de processamento e, particularmente, de latência de gerenciamento em relação às outras arquiteturas. Aumentando-se o número de agentes MobEntire presentes na arquitetura MobMonoComplete, percebe-se que o tempo de processamento total cai e a média de latência de gerenciamento também, o que era

[1] Onde migKB representa o número de migrações realizadas por agentes com conhecimento completo; mig representa o número de migrações realizadas por agentes MobSentinel; e msg significa o número de mensagens enviadas para outros agentes na rede.

esperado. Entretanto o número de migrações na rede aumenta numa proporção maior do que as quedas. Isto significa que houve um grande aumento no número de avaliações desnecessárias (máquinas sem problema). A arquitetura MobLocalMixed com 1 agente MobDecideMobDoit apresentou um ganho de performance razoável em termos de tempo de processamento e latência de gerenciamento em relação ao equivalente da arquitetura MobMonoComplete. A principal razão é que em execuções locais (Figura 7.9), a operação de classificação de arquivos (CA) é a operação que consome mais tempo numa máquina. Na arquitetura MobLocalMixed, os agentes Sentinels localizados em cada uma das máquinas ficam responsável por esta etapa. Ou seja, preparam a máquina para a chegada do MobDecideMobDoit, o que termina por otimizar o gerenciamento. Na arquitetura MobMonoComplete, ao chegar numa máquina, o agente MobEntire tem de executar todas as operações, não há preparação prévia. Em contrapartida, o *overhead* de comunicação é muito maior na arquitetura MobLocalMixed, pois os Sentinels assim que realizam a classificação dos arquivos, passam a realizar multicast de mensagens para agentes MobDecideMobDoit até que sejam atendidos. Aumentando-se o número de agentes MobDecideMobDoit na arquitetura fica clara a redução no número de mensagens, devido a maior disponibilidade desses agentes.

Comparando-se a primeira composição na tabela da arquitetura MobTwoSpec com a primeira da MobLocalMixed, pode-se perceber que os resultados para tempo de processamento e latência foram praticamente os mesmos. O número de migrações realizadas na arquitetura MobTwoSpec pelo agente MobSentinel, e que não existe na outra arquitetura já que os Sentinels são estáticos, é compensado pelo alto número de mensagens realizadas na arquitetura MobLocalMixed por esses mesmos Sentinels. Como a freqüência de multicast de mensagens realizadas por agentes Sentinel é bem maior que a de migrações dos MobSentinels, o overhead de comunicação no primeiro caso deve ser bem mais custoso para a rede do que neste último caso.

As outras 4 combinações da arquitetura MobTwoSpec representaram, como era esperado, um ganho de performance em termos de tempo de processamento e latência de gerenciamento, em relação à primeira composição, mas tiveram performance semelhante entre si. Pode-se perceber que um número maior de agentes MobSentinel em relação ao de MobDecideMobDoit na arquitetura causa um exacerbado aumento no número de mensagens, já que não existem agentes deste tipo suficientes para atender os chamados em tempo hábil. Fica claro também, que se o número de agentes MobSentinel for muito grande em relação ao número provável de máquinas com super utilização, ter-se-á um número alto de migrações desnecessárias.

7.3.4 Quarta série de experimentos

Outro experimento foi realizado com a mesma configuração do anterior, mas dessa vez, as máquinas possuem um grau de super utilização de partições maior. A Figura 7.4 mostra os resultados desse experimento.

Arquitetura	tempo	latência	êxito (%)	migKB	mig	mgs
LocalMonoComplete	45	0,38	100,00			
ConectMonoCompleteOR	300		25,00			
ConectMonoCompleteRSH	300		0,00			
MobMonoComplete (1 MobEntire)	178	67,31	100,00	14		
MobMonoComplete (2 MobEntire)	98	27,02	100,00	33		
MobLocalMixed (1 MobDecideMobDoit)	180	96,52	100,00	4		3514
MobLocalMixed (2 MobDecideMobDoit)	107	43,98	100,00	4		1590
MobTwoSpec (1 MobDecideMobDoit, 1 MobSentinel)	174	75,35	100,00	4	65	932
MobTwoSpec (2 MobDecideMobDoit, 2 MobSentinel)	118	53,91	100,00	4	132	1773
MobTwoSpec (3 MobDecideMobDoit, 2 MobSentinel)	67	33,68	100,00	4	97	143

Figura 7.4 - Resultados para uma rede com 4 máquinas com grau mais alto de super utilização

Comparando-se a tabela de resultados desses dois últimos experimentos, percebemos que o tempo de processamento e latência de gerenciamento aumentou bastante quando o nível de utilização das partições das máquinas em rede aumentou, o que era esperado.

Com este novo experimento, o impacto de se prover menos agentes MobDecideMobDoit nas arquiteturas foi bem maior, com relação ao *overhead* gerado pelos multicasts de mensagens, se comparado aos do resultado do experimento anterior.

Significa dizer, que o número de mensagens geradas por agentes Sentinel (ou MobSentinel) cresce exponencialmente com o aumento da latência de gerenciamento. De fato, estes agentes permanecem realizando multicast de mensagens até que sejam atendidos por algum MobDecideMobDoit disponível na rede.

7.3.5 Quinta série de experimentos

A tabela da Figura 7.5 apresenta os resultados de mais um experimento realizado. A rede consistiu de 23 máquinas, todas com partições super utilizadas. A estabilidade de estaticidade da rede foi mantida. O tempo limite para as simulações foi de 500 segs.

Arquitetura	tempo	latência	êxito (%)	migKB	mig	mgs
LocalMonoComplete	82	0,41	100,00			
ConectMonoCompleteOR	500		4,35			
ConectMonoCompleteRSH	500		4,35			
MobMonoComplete (1 MobEntire)	500	237,60	47,80	11		
MobMonoComplete (2 MobEntire)	500	229,58	82,60	28		
MobMonoComplete (4 MobEntire)	328	132,50	100,00	119		
MobLocalMixed (2 MobDecideMobDoit)	488	251,08	100,00	23		102764
MobLocalMixed (4 MobDecideMobDoit)	301	163,37	100,00	23		98501
MobLocalMixed (8 MobDecideMobDoit)	234	144,96	100,00	23		54677
MobTwoSpec (1 MobDecideMobDoit, 1 MobSentinel)	500	215,61	73,90	18	18	2792
MobTwoSpec (2 MobDecideMobDoit, 2 MobSentinel)	342	150,00	100,00	23	138	5044
MobTwoSpec (2 MobDecideMobDoit, 5 MobSentinel)	280	132,96	100,00	23	464	15362
MobTwoSpec (5 MobDecideMobDoit, 5 MobSentinel)	146	70,70	100,00	23	244	7416
MobTwoSpec (5 MobDecideMobDoit, 2 MobSentinel)	266	166,88	100,00	23	85	1720

Figura 7.5 - Resultados para uma rede com 23 máquinas com partições super utilizadas

Mais uma vez as arquiteturas ConectMonoComplete não obtiveram bom desempenho, conseguindo resolver os problemas de apenas 4,35% das máquinas na rede.

O melhor tempo de processamento e latência foi obtido pela arquitetura LocalMonoComplete, o que já era esperado.

A arquitetura MobMonoComplete com 1 e com 2 agentes MobEntire resolveu o problema de 47,8% e 82,6% das máquinas em rede dentro do tempo limite, respectivamente. Dobrando-se o número de agentes MobEntire para quatro, o tempo de processamento total para resolução do problema de todas as 23 máquinas foi de 328 segs. Pode-se notar que o número de migrações cresceu bastante em relação às duas

situações anteriores. Isso é resultado do *efeito de saturação* no gerenciamento. Isto é, quanto maior o número de agentes, mais rapidamente é feito o gerenciamento total na rede, mas a partir do momento que não houver mais máquinas a serem gerenciadas, os agentes desocupados irão migrar bem mais rapidamente entre as máquinas.

Comparando os resultados de performance das três composições da arquitetura MobLocalMixed (com 2, 4 e 8 agentes MobDecideMobDoit), pode-se perceber que houve uma melhora no tempo de processamento e na latência de gerenciamento com o aumento do número de agentes MobDecideMobDoit, entretanto a taxa de melhora parece tender para uma estabilização com o aumento do número de agentes (observe os gráficos da Figura 7.6).

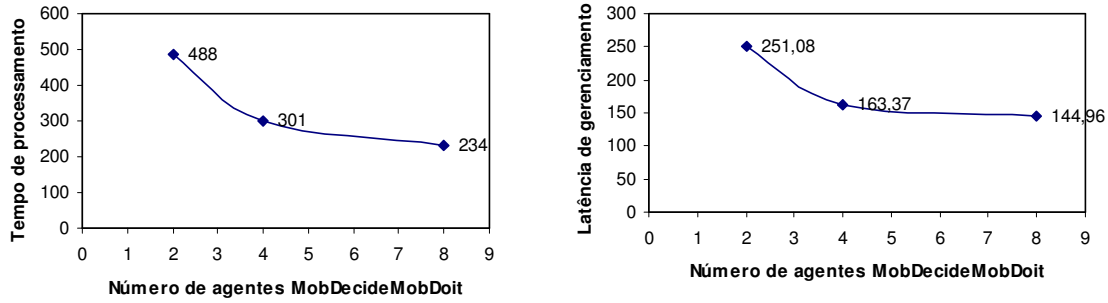


Figura 7.6 - Tendência de estabilização com o aumento de agentes

Já em relação ao número de multicast de mensagens realizados, pode-se perceber que um maior número de agentes disponíveis influenciou positivamente (Figura 7.7).

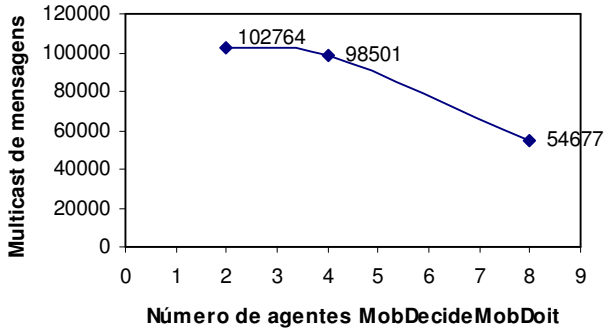


Figura 7.7 - Redução acentuada no número de multicast de mensagens

A arquitetura MobTwoSpec com apenas um agente MobDecideMobDoit e um agente MobSentinel só conseguiu resolver o problema de 73,9% das máquinas dentro do tempo de 500 segs. Aumentando o número de agentes para 2 de cada tipo, os problemas foram resolvidos em 342 segs com uma latência de gerenciamento de 150 segs. A diferença no número de migrações de agentes MobSentinel entre as duas composições da arquitetura foi imenso. É importante notar que a quarta composição para esta arquitetura apesar de ser composta por um número bem maior de agentes móveis (5 MobDecideMobDoit e 5 MobSentinel) não apresentou valores de migração e de multicast de mensagens tão maiores, ao mesmo tempo em que foi bem mais eficiente, com redução de mais de 50% no tempo de processamento e de latência de gerenciamento.

Somando esta reflexão à análise das duas outras composições, chegamos a uma importante conclusão: o aumento do número de agentes do tipo MobSentinel só causa um grande aumento no número de migrações pela rede e no número de multicast realizado caso não seja acompanhado de um aumento do número de agentes do tipo MobDecideMobDoit também.

Comparando a composição com 2 MobDecideMobDoit e 5 MobSentinel e a composição com 5 MobDecideMobDoit e 2 MobSentinel podemos chegar a uma outra conclusão bastante interessante: o aumento no número de agentes do tipo

MobDecideMobDoit otimiza o tempo de processamento da atividade, enquanto que um número maior de agentes do tipo MobSentinel diminui a latência do gerenciamento. Entretanto, aquela composição representa um custo bastante acentuado em termos de *overhead* de comunicação.

Também foram realizados experimentos que simularam uma queda de rede repentina e que simularam uma rede dinâmica, com inclusão de máquinas durante o processamento.

Para o caso de queda repentina de rede, foi observado que as arquiteturas ConectMonoComplete pararam a execução imediatamente. Não houve influência na execução da arquitetura LocalMonoComplete. Após a queda de conexão foi observado ainda uma certa atividade na rede relativa às ações locais dos agentes MobEntire e MobDecideMobDoit nas arquiteturas MobMonoComplete, MobLocalMixed e MobTwoSpec. Entretanto, nenhuma migração foi mais detectada.

Para o caso de adição de mais máquinas em tempo de execução, notou-se que as arquiteturas ConectMonoComplete, MobMonoComplete e MobTwoSpec conseguiram realizar o gerenciamento nas novas máquinas. Já as arquiteturas LocalMonoComplete e MobLocalMixed ignoraram a presença delas.

7.4 Discussão

A baixa performance apresentada pelas arquiteturas ConectMonoComplete deve-se ao fato de que o seus tempos de processamento para as operações ser muito alto. Pode-se observar no gráfico das Figura 7.8 e Figura 7.9 que para a arquitetura ConectMonoCompleteRSH, o grande gargalo é a classificação de arquivos (CA), enquanto que para a arquitetura ConectMonoCompleteOR, é a operação de tomada de decisão (DEC) sobre ações de correção.

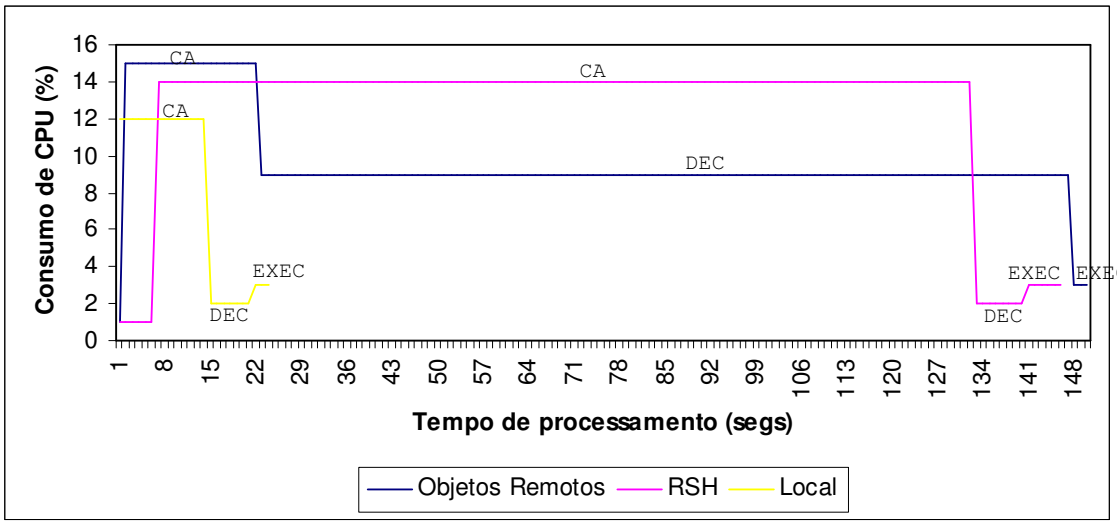


Figura 7.8 - Tempo de processamento e consumo de CPU para as operações

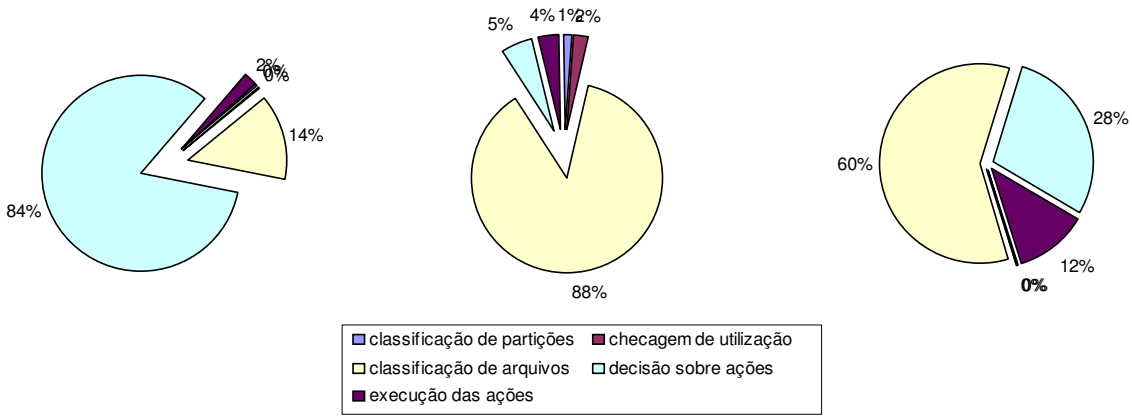


Figura 7.9 - Contribuição das operações no tempo de processamento para processamento via OR, RSH e local, respectivamente

Particularmente, na arquitetura ConectMonoCompleteOR, ficou claro que o tempo de *bind* aos objetos remotos representa um alto overhead de comunicação, custo acentuado de tempo de processamento e consumo de CPU da máquina onde o agente está executando. Sabendo-se que o tempo de processamento da operação é de ~ 3 segundos, cada atividade de gerenciamento numa determinada máquina irá consumir no mínimo 3

segundos. Além dessas questões, há um consumo de espaço em disco de ~ 97kbytes em cada uma das máquinas gerenciáveis. Esse valor chega a ser maior do que os 40,6kbytes requerido pela arquitetura LocalMonoComplete. A necessidade de um *middleware* para distribuição instalado em cada máquina é um outro ponto negativo da arquitetura. Nesse sentido, a arquitetura ConectMonoCompleteRSH apresenta vantagens consideráveis pois não necessita de plataformas extra a JVM alguma nas máquinas gerenciáveis da rede. O consumo de espaço em disco nessas máquinas também é nulo.

Ainda em termos de consumo de espaço em disco, o mecanismo de provisão de migração do Voyager (seção 6.3.1) evitou que fosse necessário guardar as classes dos agentes móveis das arquiteturas em todas as máquinas gerenciáveis. Sendo assim, o consumo do espaço em disco das arquiteturas baseadas em agentes móveis foi relativamente baixo. O agente MobDecideMobDoit possui o tamanho de 42,2kb e o MobSentinel possui 15,9kb de tamanho.

As arquiteturas baseadas em mobilidade de agentes apresentaram um ganho médio de performance em termos de tempo de processamento e latência de gerenciamento quando se aumenta o número de agentes com poder de decisão e execução. Entretanto, como esses agentes necessitam de agentes de monitoramento (móveis ou não), não se pode ser tão categórico nessa afirmação, sem antes atentar para pelo menos duas questões: os tipos de máquinas da rede e o número delas. Se as máquinas possuem freqüência baixa de ocorrência de super utilização em partições, então agentes de monitoramento estáticos em cada máquina da rede podem gerar um alto overhead de consumo de CPU. Nesse caso, seria mais adequado, o uso de agentes de monitoramento móveis. Se o número de máquinas é grande e a freqüência de problemas é maior, então, talvez o uso de agentes de monitoramento estáticos sejam mais adequados por propiciarem uma latência média de gerenciamento menor. Uma opção para o uso de agentes de monitoramento móveis nesse caso, seria aumentar o número destes na rede,

entretanto como foi observado, isto gera um número de mensagens exagerado que pode ocasionar um alto tráfego de rede.

Os resultados da arquitetura MobMonoComplete mostrou que se dotando um agente móvel de todo o conhecimento, evita-se um eventual overhead de mensagens na rede e diminui-se a dependência da estabilidade da mesma. Em contrapartida, a provisão de agentes mais especializados reduziu o tempo de processamento. Para o caso específico da administração do espaço em disco, a razão para esse ganho é o fato de que a operação mais custosa é a de classificação de arquivos e que passa a ser realizada pelos agentes Sentinel (móvel ou estático). As máquinas dessa forma, já ficam pré-gerenciadas restando apenas as operações de tomada de decisão e execução das ações, que possuem um custo menor. A distribuição de agentes presentes numa arquitetura é completamente dependente do número de dispositivos em rede.

7.5 Resumo

Os resultados dos experimentos suscitaram considerações acerca das hipóteses relacionadas na seção 7.2.

H1 A mobilidade contribui para redução do consumo de CPU

Foi observado que arquiteturas baseadas em agentes estáticos locais possuem uma freqüência de consumo de CPU mais alta. Agentes “dedicados” a uma única máquina processam indefinidamente, mesmo quando não são detectados problemas de super-utilização. O agente estático conectado provoca um alto consumo de CPU na máquina onde reside.

H2 Com mobilidade, há uma redução no consumo de espaço em disco total na rede

De fato, o mecanismo de provisão de migração do Voyager evita que se faça necessário guardar as classes dos agentes móveis em todas as máquinas gerenciáveis. Nesse sentido, a arquitetura com agentes estáticos conectados implementada com remote shell

também obteve um gasto médio baixo de espaço em disco já que apenas a máquina onde o agente executa precisa conter as classes.

H3 Mobilidade contribui para um gerenciamento mais escalável e robusto

Os experimentos mostraram que ao se adicionar máquinas para gerenciamento, as arquiteturas que dispõe de um alto grau de mobilidade atenderam as novas máquinas. Agentes estáticos locais precisam ser manualmente instalados nessas máquinas. As arquiteturas baseadas em agentes estáticos conectados também possuem uma boa escalabilidade de gerenciamento, entretanto, por dependerem em demasiado de comunicação em rede (processamento remoto), possuem faca robustez, o que ficou comprovado com experimentos que simularam queda de rede. Ainda em relação à robustez, a arquitetura com agentes totalmente autônomos e locais não teve sua atividade influenciada pela queda de rede e a mobilidade garantiu, pelo menos, o resumo da atividade em uma determinada máquina.

H4 Quanto menor o grau de autonomia em agentes móveis, menor o *overhead* gerado pela migração

O overhead de migração não chegou a ser um grande problema. A diferença de tamanho de agentes móveis com base de conhecimento e sem base de conhecimento não foi suficiente para influenciar no tempo total de processamento ou latência de gerenciamento.

H5 Agentes especializados otimizam o gerenciamento

Os experimentos mostraram que o grau de autonomia influencia diretamente o tempo total de processamento também. Na administração de espaço em disco, a operação mais custosa é a classificação de arquivos. Os arquivos de partições super utilizadas já vão sendo classificados por agentes especializados, restando apenas as operações de tomada de decisão e execução das ações (menos custosas) que são realizadas por outros agentes disponíveis.

H6 Um número maior de agentes móveis otimiza o gerenciamento

Os experimentos mostraram que aumentando o número de agentes, o tempo de processamento e latência média de gerenciamento reduziram. Entretanto, é importante atentar para o efeito de saturação que foi observado e que pode gerar um *overhead* de migrações: a partir do momento em que não houver mais máquinas a serem gerenciadas, os agentes desocupados irão migrar indefinidamente.

H7 Quanto maior o número de agentes, maior o *overhead* de comunicação

A hipótese é válida para o caso de organizações com diferentes modelos de agentes. De fato, agentes de monitoramento (Sentinel), por exemplo, permanecem realizando multicast de mensagens para agentes que decidem e executam até que sejam atendidos por algum deles.

8

Conclusões

A administração de sistemas em rede é uma atividade de grande importância hoje em dia, já que o volume de negócios que dependem do pleno funcionamento e estabilidade de uma rede de comunicação é cada vez mais alto. Ao mesmo tempo em que a necessidade de gerenciamento aumenta, a complexidade da atividade também. Essa complexidade aliada à dimensão das redes atuais, leva à necessidade de automação do processo.

As ferramentas de administração de redes existentes pecam pela alta centralização da atividade de gerenciamento e ainda dependência do monitoramento humano. O uso de agentes inteligentes que implementem propriedades como *mobilidade*, *autonomia* nessas ferramentas, pode prover um grau maior de *distribuição* de gerenciamento e de independência da intervenção humana.

Uma das contribuições desta dissertação foi a de realizar uma análise criteriosa e original sobre as propriedades mobilidade, autonomia e distribuição em agentes para o

gerenciamento corporativo de sistemas com o intuito de provê guias de utilização dessas propriedades em eventuais soluções multiagentes para o domínio.

Outra contribuição do trabalho foi definir alguns tipos de agentes distintos que variam com essas propriedades e arquiteturas multiagentes baseadas na combinação desses diferentes tipos.

Para que pudéssemos avaliar a influência das propriedades no gerenciamento de sistemas em rede, adequamos essas arquiteturas a um estudo de caso particular, *a Administração do Espaço em Disco em Redes UNIX/NFS* e realizamos diversos experimentos com o auxílio de um simulador que também desenvolvemos.

A construção do simulador constituiu num trabalho de programação extremamente árduo, onde foram escritas ~ 10200 linhas de código Java, incluindo programação com a API do Voyager e CORBA. O resultado foi um simulador para gerenciamento genérico de sistemas e que mostra animações dos agentes em ação.

Os resultados dos experimentos realizados, onde várias composições das arquiteturas definidas foram testadas, mostraram que o uso de mobilidade em agentes deve ser encorajado, dependendo do grau de autonomia que se vai prover. Agentes móveis com autonomia total mostraram um bom tempo de processamento e bom nível de independência de estabilidade de rede, mas apresentaram uma latência de gerenciamento em média maior do que uma arquitetura com agentes estáticos locais (ou mesmo móveis) especializados em monitoramento e móveis de decisão e execução.

Os resultados ainda mostraram ser possível a concepção de uma solução de gerenciamento, baseada numa arquitetura multiagente, que seja eminentemente distribuída e completamente independente de intervenção humana.

Como trabalho futuro pode-se aplicar as arquiteturas e realizar experimentos no simulador para outros domínios de gerenciamento.

Com os resultados obtidos desse conjunto de domínios, pode-se propor uma metodologia generalizada para construção de sistemas multiagentes para o gerenciamento corporativo de sistemas.

E por fim, a construção de um sistema real para administração de espaço em disco todo baseado na filosofia de agentes móveis e autônomos e no conceito de distribuição total de gerenciamento.

Referências Bibliográficas

[1] Abeck, S., Köppel, A. & Seitz, J. (1998). A Management Architecture for Multi-Agent Systems. *Proceedings of the IEEE Third International Workshop on System Management,* pp. 133-138.

[2] Baldi, M., Gai, S. & Picco, G. P. (1997). Exploiting Code Mobility in Decentralized and Flexible Network Management. In *the proceedings of Mobile Agents 97 (MA97).*

[3] Bezerra, H., Goulart Júnior, F. S. & Pereira, P. N. (1998). Controle de Discos do DI. *Relatório técnico das disciplinas Sistemas Distribuídos e Tópicos Avançados de Computação Inteligente I.* University of Pernambuco.

[4] Bieszczad, A. & Pagurek, B. (1998). Network Management Application-Oriented Taxonomy of Mobile Code. In *IEEE/IFIP Network Operations and Management Symposium (NOMS'98).* New Orleans, Louisiana.

[5] Bieszczad, A., Pagurek, B. & White, T. (1998). Mobile agents for network management. *IEEE Communications Surveys.*

[6] Carzaniga, A., Picco, G. P. & Vigna, G. (1997). Designing distributed applications with mobile code paradigms. In R.Taylor, editor, *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, (pp. 22-32). ACM Press.

[7] Case, J., Fedor, M., Schoffstall, M. L. & Davin, C. Simple Network Management Protocol (SNMP). RFC 1157, 1990.

[8] Case, J., McCloghrie, K., Rose, M. & Waldbusser, S. Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2). RFC 1442, 1993.

[9] Cheiknrouhou, M. M., Conti, P. & Labetoulle, J. (1998). Intelligent Agents in Network Management a State-of-the-Art. *Networking and Information Systems Journal*. Volume 1 – n^{o} 1/1998, pp 9-38.

[10] Ciancarini, P., Omicini, A. & Zambonelli, F. (2000). Multiagent systems engineering: the coordination viewpoint. In Intelligents Agents VI (ATAL99), LNAI. Springer-Verlag.

[11] Cockayne, W. R. & Zyda, M. (1997). Mobile Agents. Manning Publ. Co., Greenwich.

[12] Computer Associates. (1999). Unicenter TNG: Product Information. Computer Associates International.

[13] Coulouris, G., Dollimore, J. & Kindberg, T. (1994). Distributed Systems: Concepts and Design. Addison-Wesley, second edition.

[14] Covaci, S., Zhang, T. & Busse, I. (1997). Java-based Intelligent Mobile Agents for Open System Management. IEEE.

[15] Cugola, G., Ghezzi, C., Picco, G. & Vigna, G. (1997). Analyzing Mobile Code Languages. Mobile Object Systems, Lecture Notes in Computer Science, No. 1222, Springer-Verlag (D), (pp. 94-109).

[16] Figueira Filho, C. & Ramalho, G. (2000). Jeops – the java Embedded Object Production System. In M. Monard e J. Sichman (eds). Advances in Artificial Intelligence. Lecture Notes on Artificial Intelligence Series, vol. 1952, pp 52-61. London: Springer-Verlag.

[17] Finin, T. et.al. (1994). KQML as an Agent Communication Language. In 3rd International Conference on Information and Knowledge Management (CIKM94), ACM Press.

[18] FLASH (Formalizações da Administração de Sistemas Heterogêneos) http://www.cin.ufpe.br/~flash

[19] Foundation for Intelligent Physical Agents, http://www.**fipa**.org

[20] Frisch, A. (1995). Essential System Administration, 2nd Edition, O'Reilly & Associates.

[21] Ghezzi, C. & Vigna, G. (1997). Mobile Code Paradigms and Technologies: A Case Study. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the 1 st International Workshop on Mobile Agents (MA '97)*, vol. 1219 of Lecture Notes on Computer Science.

[22] Gray, R. (1996). Agent Tcl: A flexible and secure mobile-agent system. In *The Fourth Annual Tcl/Tk Workshop Proceedings*. The USENIX Association.

[23] Harrison, C. G., Chess, D. M. & Kershenbaum, A. (1996). Mobile agents: Are they a good idea? Technical report, IBM Research Division.

[24] Harrison, H. E., Mitchell, M. C. & Shaddock, M. E.. (1994). Pong: a Flexible Network Monitoring System. *Proceedings of USENIX/LISA Conference,* (pp. 167-174).

[25] Hattori, F., Ohguro, T., Yokoo, M., Matsubara, S. & Yoshida, S. (1999). Socialware: Multiagent systems for supporting network communities. *Communications of the ACM*, 42(3):55--61. Special Section on Multiagent Systems on the Net.

[26] Haykin, S. (1994). Neural Networks: A Comprehensive Foundation. IEEE Press/Macmillan College Publishing Company, New York.

[27] Hewlard Packard. Hp openview. http://www.hp.com/openview/index.html

[28] IBM Tokyo Research Labs. (1996). Aglets Workbench : Programming Mobile Agents in Java. http://www.trl.ibm.co.jp/aglets

[29] Iglesias, C. A., Garijo, M. & Gonzalez, J. C. (1998). A Survey of Agent-Oriented Methodologies. In Intelligent Agents V -- Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98), Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg.

[30] ISO -International Organization for Standardization. Information Technology -- Hypermedia/Time-based Structuring Language (HyTime), ISO/IEC 10744, 1992.

[31] JATLite. http://java.stanford.edu/java_agent/html.

[32] Java Management Extensions
http://java.sun.com/products/JavaManagement/index.html

[33] Jeon, H., Petrie, C. & Cutkosky, M. R. JATLite: A Java Agent Infrastructure with Message Routing. *IEEE Internet Computing.*

[34] Johansen, D., Renesse, R. & Schneider, F. B. (1995). An introduction to the TACOMA distributed system. Technical report, University of Tromso.

[35] Karnik, N. M. & Tripathi, A. R. (1998). Design Issues in Mobile-Agent Programming Systems. *IEEE Concurrency*, Vol. 6, No. 3, pp. 52-61.

[36] Kiniry, J., Zimmermann, D. (1997). A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, (pp 21-30).

[37] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I. & Osawa, E. (1995). RoboCup: The Robot World Cup Initiative. In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/Alife.*

[38] Klir, G. & Yuan, B. (1995). Fuzzy sets and fuzzy logic: theory and applications. Prentice Hall, Upper Saddle River, NJ.

[39] Ku, H., Luderer, G. W., Subbiah, B. (1997). An Intelligent Mobile Agent Framework for Distributed Network Management. *Globecom`97*, Arizona.

[40] Labrou, Y., Finin, T. & Peng, Y. (1999). The Interoperability Problem: Bringing together Mobile Agents and Agent Communication Languages. *Proceedings of the 32nd Hawaii International Conference on System Sciences*.

[41] Lesser, V. R. (1999). Cooperative Multiagent Systems: A Personal View of the State of the Art. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 1.

[42] Lindholm, T. & Yellin, F. (1996). The Java Virtual Machine Specification. AddisonWesley. http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

[43] Macedo, H. T. & Ramalho, G. L. (1999). Arquiteturas de Agentes Inteligentes para o Gerenciamento do Espaço em Disco. *Relatório técnico da disciplina TACI 3*. University of Pernambuco.

[44] Macedo, H. T. & Ramalho, G. L. Mobilet: Um Agente Móvel para Gerenciamento de Espaço em Disco. *Relatório técnico da disciplina MCI*. Unversity of Pernambuco.

[45] MacGuire, S. Big Brother: Monitering & notification for systems and networks, http://maclawran.ca/bb/help/bb-info.html

[46] McCloghrie, K. & Rose, M. T. Management Information Base for network management of TCP/IPbased internets, MIB-II. Internet Request for Comments Series RFC 1213, March 1991

[47] Michalski, R. S. (1983). Theory and methodology of inductive learning. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), Machine learning: An artificial intelligence approach (pp. 323--348). Tioga Publishing Co.

[48] Microsoft Inc. (1999). Distributed Component Object Model (DCOM). General Microsoft web site containing links to information about the DCOM Technology, http://www.microsoft.com/com/dcom.asp

[49] Miller, T., Stirline, C. & Nemeth, E. (1993). Satool – A System Administrator's Cockpit, An implementation. *Proceedings of the 7th System Administration Conference (LISA VII)*, UDENIX, Monterey, pp. 119-129.

[50] Minar, N., Kramer, K. H. & Maes, P. (1999). Cooperating Mobile Agents for Mapping Networks. In *Proceedings of the First Hungarian National Conference on Agent Based Computation*.

[51] Object Management Group. (1998). The Common Object Request Broker: Architecture and Specification (CORBA), Framingham, MA.

[52] Object Space Voyager. http://www.objectspace.com/products/voyager/

[53] Pagurek, B., Wang, Y. & White, T. (2000). Integration of mobile agents with SNMP: Why and how. *Submitted to NOMS'2000.*

[54] Puliafito, A. & Tomarchio, O. (1999). Advanced Network Management Functionalities through the use of Mobile Software Agents. In *3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA'99).* Stockholm, Sweden.

[55] Rational Software Corporation. Unified Modeling Language (UML). http://www.rational.com

[56] Riggs, R., Waldo, J., Wolrath, A. & Bharat, K. Pickling State in the Java System. *COOTS-2, Conference Proceedings*, pp. 241-250

[57] Russell, S. & Norvig, P. (1995). Artificial Intelligence: A Modern Approach, Prentice Hall Series in Artificial Intelligence. Englewood Cliffs, New Jersey.

[58] Schneier, B. (1996). Applied Cryptography. John Wiley and Sons, Inc., New York.

[59] Shoham, Y. & Tennenholtz, M. (1995). Social Laws for Artificial Agent Societies: Off Line Design. Artificial Intelligence, 73.

[60] Singh, M. P. (1998). Agent communication languages: Rethinking the principles. *IEEE Computer.* 31(12): 40--47.

[61] Stallings, W. SNMP, SNMPv2 and CMIP: The practical guide to network management standards. Addison-Wesley publication, 1994

[62] Stamos, J. & Gifford, D. K. (1990). Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537—565.

[63] Sun Microsystems. (1999). Solstice SunNet Manager. http://www.sun.com/software/solstice/em-products/network/sunnetmgr.html

[64] Tanembaum, A. S. (1996). Computer Networks.

[65] Tarouco, L. M. R., Weissheimer, C. G., Schmitt, M. & Krahe, F. (1996). Um ambiente para Gerenciamento Integrado e Cooperativo. *Anais do II Workshop sobre Administração e Integração de Sistemas (em conjunto com SBRC'96) WAIS'96.* Fortaleza-CE, pp. 149-160.

[66] Tivoli. http://www.tivoli.com/products/index/

[67] Urbano, P. G. (1998). Agentes Móveis Inteligentes para Sistemas Distribuídos Heterogêneos: Uma Aplicação no Gerenciamento do Espaço em Disco. *Relatório de trabalho de iniciação científica.* University of Pernambuco.

[68] Vinoski, S. (1997). CORBA overview: CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications*. Mag., vol. 14, no. 2.

[69] Volpano, D. & Smith, G. (1998). Language issues in mobile program security. In Giovanni Vigna, editor, Mobile Agents and Security, volume 1419 of Lecture Notes in Computer Science, pages 25-43. Springer Verlag.

[70] Waldbusser, S. Remote Network Monitoring Management Information Base, RFC 1757 02/10/1995.

[71] White, T. & Pagurek, B. (1998). Towards Multi-Swarm Problem Solving in Networks. *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS '98).*

[72] White, T., Pagurek, B., Bieszczad, A., Sugar, G. & Tran, X. (1998). Intelligent Network Modeling Using Mobile Agents. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM'98)*, p. 1082--1087, Sydney, Australia.

[73] Wiederhold, G. (1992). Mediators in the architecture of future information systems. *IEEE Computer* 25, 3 (March), 38—49.

[74] Wong, D., Paciorеck, N. & Moore, D. (1999). Java-based Mobile Agents. *Communications of the ACM*. Vol.42, No.3, pp.92-102.

[75] Wooldridge, M. & Jennings, N. R. (1998). Pitfalls of agent-oriented development. In *Proceedings of the Second International Conference on Autonomous Agents (Agents 98).* pp 385--391, Minneapolis/St Paul, MN.

[76] Wooldridge, M. J. & Jennings, N. R. (1994). Agent Theories, Architectures, and Languages: a Survey.

[77] Wooldridge, M. J. & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review* 10(2).

[78] Wooldridge, M., Jennings, N. R. & Kinny, D. (1999). A Methodology for Agent-Oriented Analysis and Design. *Proc. 3rd Int Conference on Autonomous Agents (Agents-99)*. Seattle, WA.

[79] www.robocup.org

[80] Yemini, Y. (1993). The OSI Network Management Model. *IEEE Communication Magazine*, pages 20-29.

Glossário

Arquitetura multiagente	Organização de um ou mais agentes. Neste último caso, os agentes podem ser do mesmo tipo ou de tipos distintos.
ConectMonoComplete	Arquitetura multiagente constituída por um ou mais agentes do tipo FarEntire.
ConectMonoCompleteOR	Pra o caso da administração de espaço em disco escolhemos duas formas de implementar o gerenciamento remoto: através de chamadas RSH do UNIX e através de objetos remotos distribuídos. Essa arquitetura é a arquitetura ConectMonoComplete implementada através de objetos remotos.
ConectMonoCompleteRSH	Pra o caso da administração de espaço em disco escolhemos duas formas de implementar o gerenciamento remoto: através de chamadas RSH do UNIX e através de objetos remotos distribuídos. Essa arquitetura é a arquitetura ConectMonoComplete implementada através de chamadas RSH.
ConectThreeSpec	Arquitetura multiagente constituída por agentes de três especializações distintas: FarSentinel, FarDecide, e FarDoit.
Decide	Tipo de agente que representa agentes estáticos e que agem localmente, especializados em tomada de decisão.
Doit	Tipo de agente que representa agentes estáticos e que agem localmente, especializados em executar as ações decididas por um agente de decisão..
Entire	Tipo de agente que representa agentes estáticos e que agem localmente, com autonomia total para monitorar estado de dispositivos, decidir sobre ações de correção e executar as ações decididas. No caso específico da administração do espaço em disco, executam as cinco operações básicas.
FarDecide	Semelhante ao tipo Decide, mas os agentes desse tipo agem remotamente.
FarDecideFarDoit	Tipo associado de agente. Representa agentes que possuem as funções de um FarDecide e de um FarDoit. No caso da administração de espaço em disco, um agente desse tipo fica responsável tanto pela operação de decisão sobre ações de correção quanto pela execução dessas ações decididas.

FarDoit	Semelhante ao tipo Doit, mas os agentes desse tipo agem remotamente.
FarEntire	Semelhante ao tipo Entire, mas os agentes desse tipo agem remotamente (estão conectados)
FarSentinel	Semelhante ao tipo Sentinel, mas com a capacidade de migrar de um dispositivo para outro.
LocalMonoComplete	Arquitetura multiagente constituída por um ou mais agentes do tipo Entire.
LocalThreeSpec	Arquitetura multiagente constituída por agentes de três especializações distintas: Sentinel, Decide, e Doit.
MobConectMixed1	Arquitetura multiagente constituída por agentes do tipo FarSentinel e agentes do tipo associado MobDecideMobDoit.
MobConectMixed2	Arquitetura multiagente constituída por agentes do tipo MobSentinel e agentes do tipo associado FarDecideFarDoit.
MobDecide	Semelhante ao tipo Decide, mas com a capacidade de migrar de um dispositivo para outro.
MobDecideMobDoit	Tipo associado de agente. Representa agentes que possuem as funções de um MobDecide e de um MobDoit. No caso da administração de espaço em disco, um agente desse tipo fica responsável tanto pela operação de decisão sobre ações de correção quanto pela execução dessas ações decididas.
MobDoit	Semelhante ao tipo Doit, mas com a capacidade de migração.
MobEntire	Semelhante ao tipo Entire, mas com a capacidade de migrar de um dispositivo para outro.
MobLocalMixed	Arquitetura multiagente constituída por agentes do tipo Sentinel e agentes do tipo associado MobDecideMobDoit.
MobMonoComplete	Arquitetura multiagente constituída por um ou mais agentes do tipo MobEntire.
MobSentinel	Semelhante ao tipo Sentinel, mas com a capacidade de migrar de um dispositivo para outro.
MobThreeSpec	Arquitetura multiagente constituída por agentes de três especializações distintas: MobSentinel, MobDecide, MobDoit.
MobTwoType	Arquitetura multiagente constituída por agentes do tipo MobSentinel e agentes do tipo associado MobDecideMobDoit.
Sentinel	Tipo de agente que representa agentes estáticos e que agem localmente, especializados em checar o estado de dispositivos. No caso específico da administração do espaço em disco, executam as operações de classificação de partições, checagem de utilização e classificação de arquivos.

Tipo associado	Em alguns domínios de gerenciamento, é mais interessante (e talvez, necessário) que existam agentes com mais de um tipo. O tipo deste agente é representado pela concatenação dos nomes dos tipos que o formam.
Tipo de um agente	Os tipos de agentes definidos variam segundo o grau de mobilidade e autonomia. Existem tipos que representam agentes estáticos e que agem localmente em um dispositivo, tipos que representam agentes estáticos mas que gerenciam os dispositivos remotamente, e tipos que representam agentes que migram para os dispositivos e agem localmente.

[81]

Anexo A - Regras Jeops

```
public ruleBase AgentBase {

  rule stop {
     declarations
       Filesystem f;
     preconditions
       !f.isSuperUtil();
     actions
       retract(f);
  }

  rule core {
     declarations
       Filesystem f;
       Archive a;
     preconditions
       f.isSuperUtil();
       a.getType() == Type.CORE;
     actions
       a.setSituation(Situation.REMOVE);
       retract(a);
       modified(f);
  }

  rule unknown {
     declarations
       Filesystem f;
       Archive a;
     preconditions
       f.isSuperUtil();
       a.getType() == Type.UNKNOWN;
     actions
       a.setSituation(Situation.ALERT);
       retract(a);
       modified(f);
  }
```

```
rule mail_export {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getType() == Type.MAIL;
     a.getSize() > 2000000;
     f.getType() == Type.EXPORT;
   actions
     a.setSituation(Situation.COMPACT);
     retract(a);
     modified(f);
}

rule cache_export {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getType() == Type.CACHE;
     f.getType() == Type.EXPORT;
   actions
     a.setSituation(Situation.REMOVE);
     retract(a);
     modified(f);
}

rule backup_export {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getLifeTime() > 5;
     a.getType() == Type.BACKUP;
     f.getType() == Type.EXPORT;
   actions
     a.setSituation(Situation.REMOVE);
     retract(a);
     modified(f);
}
```

```
rule image_export {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getType() == Type.IMAGE;
     f.getType() == Type.EXPORT;
   actions
     a.setSituation(Situation.COMPACT);
     retract(a);
     modified(f);
}

rule text_export {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getType() == Type.TEXT;
     f.getType() == Type.Export;
   actions
     a.setSituation(Situation.COMPACT);
     retract(a);
     modified(f);
}

rule tmp {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getLifeTime() > 15;
     f.getType() == Type.TMP;
   actions
     a.setSituation(Situation.REMOVE);
     retract(a);
     modified(f);
}
```

```
rule var_mail {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     f.getType() == Type.VAR_MAIL;
     a.getSize() > 5000000;
   actions
     a.setSituation(Situation.MOVE);
     retract(a);
     modified(f);
}

rule mail_var {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getType() == Type.MAIL;
     a.getSize() > 5000000;
     f.getType() == Type.VAR;
   actions
     a.setSituation(Situation.MOVE);
     retract(a);
     modified(f);
}

rule backup_var {
   declarations
     Filesystem f;
     Archive a;
   preconditions
     f.isSuperUtil();
     a.getLifeTime() > 5;
     a.getType() == Type.BACKUP;
     f.getType() == Type.VAR;
   actions
     a.setSituation(Situation.REMOVE);
     retract(a);
     modified(f);
}
```

```
  rule log_var {
     declarations
       Filesystem f;
       Archive a;
     preconditions
       f.isSuperUtil();
       a.getType() == Type.LOG;
       f.getType() == Type.VAR;
     actions
       a.setSituation(Situation.COMPACT_CREATE);
       retract(a);
       modified(f);
  }

  rule image_var {
     declarations
       Filesystem f;
       Archive a;
     preconditions
       f.isSuperUtil();
       a.getType() == Type.IMAGE;
       f.getType() == Type.VAR;
     actions
       a.setSituation(Situation.COMPACT);
       retract(a);
       modified(f);
  }

  rule text_var {
     declarations
       Filesystem f;
       Archive a;
     preconditions
       f.isSuperUtil();
       a.getType() == Type.TEXT;
       f.getType() == Type.VAR;
     actions
       a.setSituation(Situation.COMPACT);
       retract(a);
       modified(f);
  }

}
```

Anexo B - Voyager

O Voyager é uma plataforma para desenvolvimento de sistemas distribuídos que provê meios para implementação de agentes móveis. Por ser construída sobre Java, ela implementa um mecanismo de migração fraco, isto é, apenas o código e os dados do agente são transferidos de uma máquina para outra. O estado (thread) em execução não é transferido.

No Voyager, um agente executa em uma máquina virtual que funciona como servidor de agentes. Esse servidor provê um ponto de acesso que permite que os agentes sejam recebidos, e uma interface que permite que os agentes migrem para um outro servidor de agentes.

Para fazer um objeto funcionar como um agente móvel é preciso utilizar a chamada `Agent.of()` para obter a *faceta* de agente do objeto. O mecanismo de *agregação dinâmica,* que o Voyager provê, permite que objetos secundários (ou *facetas*) sejam agregados ao objeto primário em tempo de execução (Figura B.1).

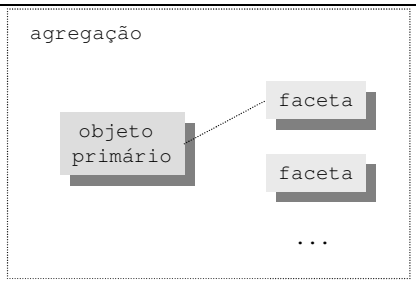


Figura B.1 - Agregação Dinâmica de um objeto com facetas.

Após obter a faceta de agente, podem ser utilizados métodos definidos na interface `IAgent`. O método `moveTo(...)` é o mais importante deles e move de fato o objeto

para a máquina especificada e reinicia executando a função do objeto passada como parâmetro. A sintaxe do método é

```
moveTo(String url, String callback[,Object[] args]),
```

onde *url* é o endereço da máquina destino e *callback* é a função a ser executada. Uma chamada a esse método causa a seguinte seqüência de eventos:

1. Espera-se que todas as mensagens que o objeto esteja processando se completem e qualquer mensagem nova que chegue ao objeto é suspendida;
2. O objeto e suas partes não transientes são copiadas para a nova localidade utilizando-se do mecanismo de serialização [56] de Java.;
3. O novo endereço do objeto e de todas as partes não transientes é guardado no local anterior;
4. O objeto antigo é destruído;
5. Quando uma mensagem enviada via um *proxy* chega no endereço antigo, uma exceção especial contendo o novo endereço do objeto é enviada de volta para o *proxy*. O *proxy* se religa ao objeto e reenvia a mensagem para o novo endereço;
6. O método retorna após o objeto ter sido movido com sucesso.

Existem pelo menos três maneiras de se disponibilizar os arquivos de classe Java para uma máquina para onde o agente irá migrar. O mecanismo utilizado em nossa implementação foi o registro de um visualizador na máquina destino que permite que as classes necessárias para a instanciação do agente, e que estão presentes na máquina de origem, sejam visualizadas. Ao mesmo tempo, um programa na máquina origem deve explicitamente disponibilizar essas classes. Os códigos da Figura B.2 são trechos do programa desenvolvido e ilustram esse mecanismo.

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.loader.*;

public class VoyagerServer
{
   public static void main(String args[]) throws IOException {
        ...
        ClassManager.enableResourceServer();                          (1)
        Voyager.startup("9000");                                      (2)
        ...
   }
}

public class VoyagerClient
{
  public static void main( String[] args )
  {
      ...
      Voyager.startup("8000");                                        (3)
      VoyagerClassLoader.addURLResource("http://paulista:9000/");     (4)
      ...
  }
}
```

Figura B.2 - Mecanismo visualizador de classes

No programa da máquina de origem, a chamada em (1) torna o programa Voyager servidor *HTTP-ativo* e permite que as classes fiquem disponíveis para visualização via HTTP. (2) e (3) são, respectivamente, as chamadas para inicialização do Voyager na porta "9000" da máquina servidora e na porta "8000" da máquina cliente. A chamada em (4) é o registro do visualizador. Deve ser passado como parâmetro para o visualizador a URL da máquina de onde o agente está partindo.

A Figura B.3 é a classe que representa o agente móvel. Ele tem de implementar a interface *Serializable* de Java para possibilitar a transferência de seu código e dos dados que processa (1). Um arquivo texto "hosts.cfg" contendo o nome da próxima máquina é lido (2) e o nome é guardado na variável `hostname`. Em seguida, é feita a chamada Agent.`of`(this) para se obter a faceta de agente do próprio objeto (3). O processo de migração se dá com a chamada do método `moveTo`() dessa faceta (3). Como

parâmetros, são passados o nome da máquina destino previamente lido e a função que deve ser executada ao se chegar na máquina (4).

```
public class Executor implements IExecutor, Serializable            (1)
{
  ...
  public void work()  {
      ...
      FileInputStream is = new FileInputStream("hosts.cfg");         (2)
      Reader reader = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(reader);
      String line = br.readLine();
      hostname = line;
      Agent.of(this).moveTo("//"+hostname,"aoChegarNoServidor");     (3)
  }
  public void aoChegarNoServidor()  {                                (4)
     ...
  }
  ...
}
```

Figura B.3 - Agente móvel implementa *Serializable*

Os mecanismos de comunicação entre os agentes também foram implementados com Voyager. Duas formas de comunicação foram implementadas: *unicast* e *multicast*.

A comunicação unicast é implementada com chamada de método remoto tradicional. No Voyager, um objeto remoto é representado por um objeto especial *proxy* que implementa as mesmas interfaces do objeto remoto. No código da Figura B.4, desenvolvido para medição de performance de uma chamada remota, a classe `Objeto` implementa o método `getName()`.

```
public class Objeto implements IObjeto
{
  String name;
  public Objeto(String name) {
     this.name = name;
  }
  public String getName(){
     return name;
  }
}
```

Figura B.4 - Classe Objeto implementa o método getName()

Realizando a seguinte chamada, uma instância da classe `Objeto` é criada na máquina "paulista" na porta 9000:

```
IObjeto objeto    = (IObjeto) Factory.create("Objeto",new Object[]{"Aline"},"//paulista:9000");
```

A Figura B.5 ilustra como a mensagem remota é processada, admitindo-se que um programa localizado na máquina "carioca" na porta 8000 fez uma chamada ao método `getName()` do objeto.

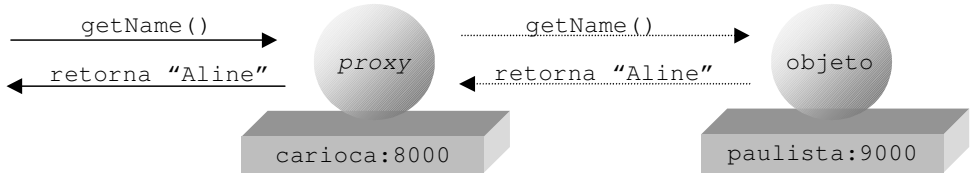


Figura B.5 - Chamada unicast a método de um objeto remoto.

A comunicação multicast é utilizada para comunicação com grupos de agentes. Voyager utiliza o conceito de *Space* que representa uma arquitetura escalável para replicação de mensagens. Um *Subspace* é um grupo de objetos que se localizam numa determinada máquina e que estão interessados na mensagem. Um Space é criado ligando-se um ou mais Subspaces. Uma mensagem enviada via um *proxy de multicast* a um Subspace é clonada para cada um dos Subspaces pertencentes ao Space antes de ser enviada a cada um dos objetos no Subspace local (Figura B.6).

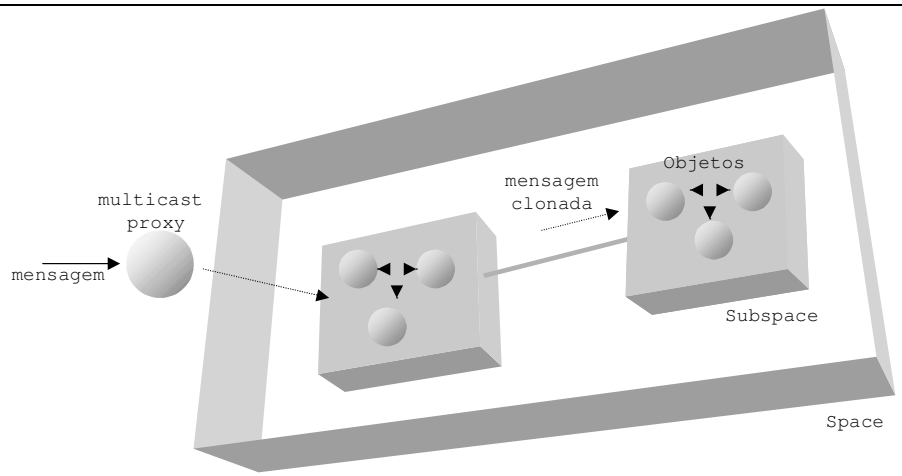


Figura B.6 - Multicast de mensagem para *Subspaces* de um *Space.*

O programa mostrado na Figura B.7 cria dois Subspaces (1) e (4) nas máquinas carioca:8000 e paulista:9000 respectivamente, cria objetos remotamente nessas máquinas (2) e (5), e os adiciona a seus respectivos Subspaces (3) e (6). Em seguida os dois Subspaces são conectados (7). Enfim, é criado um proxy de multicast para o `Subspace1` (8) e é enviada uma mensagem para o Subspace1 via o proxy (9).

```
import com.objectspace.voyager.space.*;
public class MessageMulticast
{
  public static void main( String[] args )
  {
     Voyager.startup();
     ISubspace subspace1 =
      (ISubspace) Factory.create("com.objectspace.voyager.space.Subspace","//carioca:8000/Subspace1");   (1)
     IObjeto objeto1    = (IObjeto) Factory.create("Objeto",new Object[]{"hendrik"},
                                                   "//carioca:8000/Hendrik");
     IObjeto objeto2    = (IObjeto) Factory.create("Objeto",new Object[]{"aline"},                        (2)
                                                   "//carioca:8000/Susanne");
     subspace1.add(objeto1);
     subspace1.add(objeto2);                                                                             (3)

     ISubspace subspace2 =
      (ISubspace) Factory.create("com.objectspace.voyager.space.Subspace","//paulista:9000/Subspace2");   (4)
     IObjeto objeto3    = (IObjeto) Factory.create("Objeto",new Object[]{"hendrik"},
                                                   "//paulista:9000/Claudio");
     IObjeto objeto4    = (IObjeto) Factory.create("Objeto",new Object[]{"aline"},                        (5)
                                                   "//paulista:9000/Arlene");
     subspace2.add(objeto3);
     subspace2.add(objeto4);                                                                             (6)

     subspace1.connect(subspace2);                                                                       (7)

     IObjeto objeto = (IObjeto) subspace1.getMulticastProxy("IObjeto");                                  (8)
     objeto.getName();                                                                                   (9)
   }
}
```

Figura B.7 - Criação de dois *Subspaces* na máquina "Carioca"

Anexo C - Objetos Remotos e CORBA

A arquitetura CORBA (*Common Object Request Broker Architecture*) é uma espécie de software conceitual que vem sendo proposto como um padrão pelo OMG[1] (*Object Management Group*) para desenvolvimento de aplicações que desejam possuir total interoperabilidade. Ou seja, desenvolvimento de aplicações que possam se comunicar, independentemente de quem as desenvolveu, a plataforma onde executam, a linguagem em que foram implementadas, e onde estão executando.

Um Objeto Remoto modela um objeto do mundo real e consiste de dados e métodos que devem ser invocados tal qual o conceito tradicional de objetos. A interface de um objeto remoto (descrição dos métodos e atributos que pertencem ao mesmo) é definida em CORBA através da IDL (*Interface Definition Language*). A IDL é uma linguagem de definição que esconde os detalhes de implementação do objeto das invocações. O *Object Request Broker* (ORB) é um *middleware* que funciona como um componente intermediário entre o objeto cliente e o objeto servidor, e gerencia as requisições dos clientes, escondendo a localização e detalhes de implementação do objeto servidor. A Figura C.1 mostra como estão relacionados os objetos clientes e servidores, a IDL e o ORB na arquitetura CORBA.

---

[1] A OMG é um consórcio composto por mais de 700 empresas membro como a Netscape, IBM, Sun, Motorola, Nokia, Boeing, etc. A Microsoft não faz parte do grupo pois possui sua própria solução para implementação de aplicações distribuídas, o DCOM (*Distributed Component Object Model*) [48].

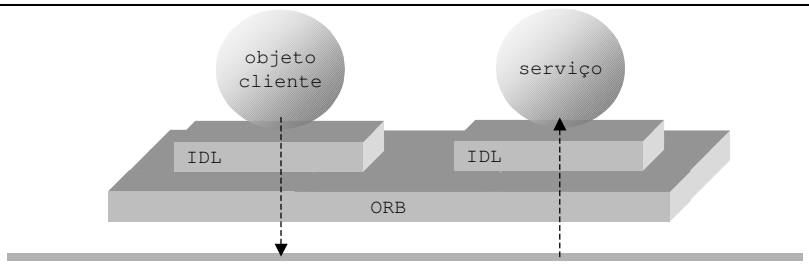


Figura C.1 - Arquitetura geral de comunicação entre objetos remotos CORBA.

O trecho de código na Figura C.2 representa a implementação da classe `LSensor`. As linhas numeradas representam os passos para inicialização do ORB e disponibilização do objeto `lsensor` (serviço). Em (1) o ORB é inicalizado, (2) o BOA é inicializado, (3) instancia-se um objeto LSensor, (4) o objeto é exportado para o ORB, e (5) espera-se por requisições do cliente (agente).

```
public class LSensor extends Default._sk_ILSensor {
  public LSensor(java.lang.String name) {
    super(name);
  }
  public static void main(String args[])
  {
     org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);  (1)
     org.omg.CORBA.BOA boa = orb.BOA_init();                     (2)
     LSensor lsensor = new LSensor("Local Sensor");              (3)
     boa.obj_is_ready(lsensor);                                  (4)
     boa.impl_is_ready();                                        (5)
  }
  public Ifilesystem[] getFilesystems() {…}
  ...
}
```

Figura C.2 - Inicialização do ORB e disponibilização do serviço

O código ilustrado na Figura C.3 é referente à implementação do agente. Em (1) o ORB é inicializado, e em (2) é feita a ligação ao objeto `LSensor`. Assim como no caso de um objeto remoto Voyager, um proxy do objeto LSensor é criado na máquina onde o agente está executando e as chamadas de métodos são feitas através desse proxy.

```
public class agente
{
   public static void main(String args[]) {
      ILSensor lsensor;
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);                    (1)
      IDefault.LSensor lsensor = IDefault.LSensorHelper.bind(orb,"Local Sensor");   (2)
      ...
  }
  ...
}
```

Figura C.3: Inicalização do CORBA no programa "cliente"

Anexo D - Interface IDL

```
module DDefault
{
     interface IType
     {
         const long ROOT     =    0;
         const long VAR      =    1;
         const long TMP      =    2;
         const long VAR_MAIL =    3;
         const long EXPORT   =   50;
           const long USR      =  100;
           const long OPT      =  101;
           const long PROC     =  500;
           const long DEV_FD   =  501;
           const long CDROM    =  502;
           const long _DEFAULT =  999;
       const long IMAGE    = 1000;
           const long TEXT     = 1001;
           const long EXEC     = 1002;
           const long CORE     = 1003;
           const long HTML     = 1004;
           const long CACHE    = 1005;
           const long TEMP     = 1006;
           const long MAIL     = 1007;
           const long LOG      = 1008;
           const long BACKUP   = 1009;
           const long MQUEUE   = 1010;
           const long PRINT    = 1011;
           const long UNKNOWN  = 9999;
     };

     interface ISituation
     {
         const long REMOVE         = 0;
         const long COMPACT        = 1;
         const long COMPACT_CREATE = 2;
         const long MOVE           = 3;
         const long MAINTAIN       = 4;
```

```
    const long ALERT          = 5;
};

interface IArchive
{
    string getName();
    long getType();
    long getLifeTime();
    long getChecksum();
    long getSize();
    string getOwner();
    long getSituation();
    void setSituation(in long situation);
    long getCompactDegree();
    void setCompactDegree();
    void setName(in string name);
    void setType(in long type);
    void setLifeTime(in long life_time);
    void setChecksum(in long checksum);
    void setSize(in long size);
    void setOwner(in string owner);
    void remove();
    void compact();
    void compact_create();
    void move();
    void maintain();
    void inform();
};

interface IConfig
{
    string getPartition();
    long getLowerBound();
    long getUpperBound();
    long getType();
};

typedef sequence< IArquivo > VArquivos;
typedef sequence< IConfig  > VConfig;

interface IFilesystem
{
    float computePerc();
    void decrease(in long value, in long fator);
```

```
            boolean isUsedOk();
            string getName();
            string getPartition();
            long getSize();
            long getUsed();
            long getType();
            long getNFiles();
            VArquivos getFiles();
            IArquivo getFile(in long pos);
            void setUsed(in long used);
            void setFiles();
            void addFile(in IArquivo file);
            void setNFiles(in long nfiles);
            void setIdeal(in VConfig config);
        };

        typedef sequence< IFilesystem > VFilesystems;

        interface ILSensor
        {
            VFilesystems getFilesystems();
            void setBufferedReader(in string str);
            any makeFilesVector(in string str, in
VFilesystems table);
            VConfig getConfig(in string filename);
        };

};
```