Aplicação de Agentes Móveis em Ambientes de Computação Ubíqua

Roberto Speicys Cardoso   Fabio Kon

{speicys,kon}@ime.usp.br
http://gsd.ime.usp.br

Relatório Técnico MAC-IME-USP 2002-02

Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

Fevereiro 2002

Resumo

Ambientes de Computação Ubíqua são espaços permeados por equipamentos com capacidade computacional com o objetivo de aumentar e melhorar a interatividade do usuário com o ambiente. Estes sistemas têm de lidar com diversas questões ainda em aberto da computação, como por exemplo o desenvolvimento de um sistema de software robusto e confiável para equipamentos com restrições de processamento, memória e energia. Estas características tornam esses sistemas bastante peculiares, com arquiteturas muito diferentes das arquiteturas de sistemas distribuídos tradicionais. Os Agentes Móveis, por sua flexibilidade e mobilidade, se apresentam como uma tecnologia interessante para ser utilizada nesse tipo de ambiente, pois propiciam redução no tráfego da rede, maior capacidade de tolerância a falha e maior robustez. Neste documento, descrevemos alguns dos principais sistemas de Computação Ubíqua e de Agentes Móveis, e propomos algumas aplicações de Agentes Móveis em Ambientes de Computação Ubíqua.

1 Introdução

Nos últimos anos, o aumento da capacidade de processamento dos dispositivos fabricados pela indústria eletrônica, aliado à redução das dimensões físicas destes dispositivos, vêm criando um novo ambiente computacional com computadores espalhados por todo o meio, e no qual os paradigmas tradicionais de desenvolvimento de aplicações distribuídas (como cliente-servidor ou *peer-to-peer*) se mostram ineficientes e inapropriados.

Para lidar com tais ambientes, é necessário um novo paradigma que vem sendo chamado de **Espaço Ativo** ou **Computação Ubíqua**. Dentre as questões que devem ser consideradas nesse tipo de ambiente está a da reconfiguração dinâmica, graças à heterogeneidade de equipamentos que podem existir. Outras questões importantes são a segurança no acesso aos recursos do sistema, o armazenamento de dados persistente e sensível à localização do usuário e a descoberta de recursos existentes no espaço ativo.

Neste relatório procuramos fazer uma análise de vários dos ambientes de computação ubíqua existentes atualmente, de alguns dos sistemas de desenvolvimento de agentes móveis, e propor aplicações de agentes móveis em ambientes de computação ubíqua.

2 Ambientes de Computação Ubíqua

2.1 Gaia

O Gaia[1] é um sistema desenvolvido pela Universidade de Illinois em Urbana-Champaign que procura criar um ambiente computacional em que não existam diferenças entre entidades reais (pessoas, objetos) e entidades virtuais (dados, estruturas).

O principal objetivo deste projeto é o de criar um sistema operacional intermediário capaz de gerenciar os recursos de um espaço ativo, assim como os sistemas operacionais tradicionais gerenciam os recursos de um computador [RHR+01].

O nome Gaia foi dado ao projeto baseado no conceito popularizado por James Lovelock nos anos 70, de que o planeta Terra é um único superorganismo, formado pela atmosfera, oceanos, litosfera, etc. A esse organismo ele deu o nome de *Gaia*, que era a Deusa grega que representava o planeta [RC00]. O projeto Gaia quer aplicar essa visão à computação.

Para atingir este objetivo, foi desenvolvido um sistema operacional, chamado de GaiaOS, capaz de integrar e gerenciar todos os dispositivos existentes em um espaço ativo.

2.1.1 Arquitetura

O GaiaOS é um sistema operacional intermediário, que roda como *middleware* sobre os sistemas operacionais existentes, como Windows 2000, Windows CE e Solaris. Ele é formado por quatro componentes principais:

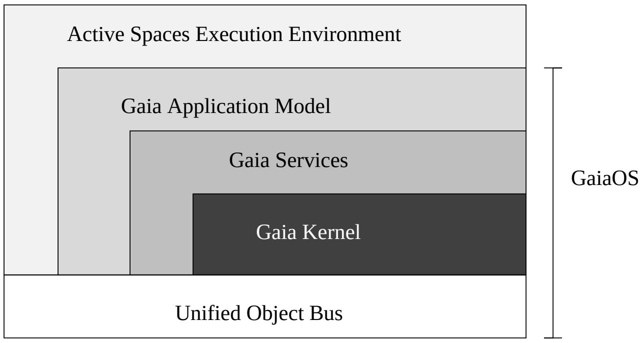

Figura 1: Estrutura do Gaia

Unified Object Bus: fornece os meios para manusear de forma homogênea os diversos componentes que estiverem em execução no sistema.

Gaia Kernel: implementa os serviços essenciais para o funcionamento do sistema, como os serviços de Descoberta, Repositório de Componentes, Eventos, Nomes, Segurança e Armazenagem e Manipulação de Dados.

Gaia Services: implementa serviços adicionais como Qualidade de Serviço (QOS).

Gaia Application Model: define um *framework* para criação de aplicações ubíquas.

A seguir descrevemos as duas partes principais do sistema: o *Unified Object Bus* e o *Kernel* do Gaia.

[1] http://choices.cs.uiuc.edu/gaia

2.1.2 Unified Object Bus

Um ambiente de computação ubíqua é intrinsecamente heterogêneo. Diversos equipamentos diferentes, com hardwares distintos executando softwares variados podem estar operando em um mesmo ambiente simultaneamente.

Para que a tarefa de desenvolver aplicações para um ambiente com este nível de complexidade seja um pouco menos trabalhosa, é necessário algum software intermediário capaz de exportar uma interface comum de manipulação dos objetos do sistema, independente das suas características de implementação.

O *Unified Object Bus* é responsável por instanciar, inspecionar, remover e nomear os diversos componentes do sistema (escritos em CORBA, Scripts ou Java Beans). Existem quatro abstrações fornecidas pelo *UOB*:

Unified Component: são componentes básicos do sistema que seguem um padrão de nomenclatura, podem ser manipulados dinamicamente e enviados pela rede. No entanto não têm uma implementação definida: qualquer objeto já existente pode se tornar um *Unified Component* desde que siga o ciclo de vida definido para tal.

UOB Host: é qualquer equipamento do sistema capaz de hospedar componentes. Possui interfaces para a criação e destruição de *Component Containers*, e também para criação de componentes em *Component Containers* específicos.

Component Container: fornece o ambiente de execução para componentes, gerencia as dependências e permite a manipulação dos componentes que ele contém.

Component Manager: possui a interface para manipulação do ciclo de vida do componente: criação, ativação, nomeação e destruição.

2.1.3 *Kernel* do Gaia

É o núcleo central do sistema, responsável por carregar os serviços essenciais para funcionamento do *GaiaOS*. Os principais serviços são descritos abaixo.

Serviço de Nomes: como o *GaiaOS* é um sistema baseado em componentes, é preciso algum mecanismo que atribua nomes a eles e permita que qualquer aplicação seja capaz de localizá-los pelo sistema. Através do Serviço de Nomes, os componentes podem se registrar com pares `<nome, valor>`, facilitando sua descoberta pelos demais componentes do ambiente.

Gerenciador de Eventos: é utilizado para distribuir informações entre os componentes do sistema. O Gerenciador de Eventos disponibiliza canais de eventos, que distribuem informações específicas. Componentes interessados em um determinado tipo de informação assinam o canal, e passam a receber as informações que por lá passarem. Alguns exemplos de canais são o *Discovery*, responsável por notificar a entrada e saída de entidades no espaço, e o canal de *Erros* usado pelos componentes para enviar eventos com informações de erros específicos.

Serviço de Descoberta: responsável por manter controle sobre os componentes, pessoas e entidades presentes em um Espaço Ativo. Ele é composto por dois subserviços: o **Serviço de Presença**, responsável por controlar os componentes de software ativos no espaço, e o **Serviço de Informação**, responsável por detectar usuários e entidades. Os componentes de software devem enviar pulsos em intervalos constantes de tempo para indicar que ainda estão ativos. Já as pessoas e entidades são consideradas ativas ou inativas no espaço a partir das informações coletadas pelos sensores do ambiente.

Repositório do Espaço: armazena quais equipamentos, serviços, aplicações e usuários estão ativos no sistema em um dado instante. Toda entidade armazenada possui uma descrição em XML associada. O Repositório recebe os eventos do Serviço de Descoberta para manter as suas informações atualizadas, e permite que se procure por uma entidade em particular.

Serviço de Segurança: fornece mecanismos para possibilitar autenticação de usuário, controle de acesso, carga dinâmica segura de componentes, rastreamento seguro e privacidade de localização. A autenticação pode ser feita de várias formas, como reconhecimento de voz, escaneamento de retina e impressão digital, e não só apenas por usuário e senha.

Serviço de Dados: devido à heterogeneidade típica de um espaço ativo, o sistema não pode supor nada a respeito dos equipamentos existentes nesse espaço e suas preferências, mas deve se adaptar às condições impostas por eles. Através do Serviço de Dados, a aplicação deve apenas especificar o tipo dos dados que ela quer acessar, que eles são convertidos automaticamente do seu tipo de origem para o tipo solicitado. Além disso, o Serviço de Dados também oferece suporte para localização, de forma que o usuário seja capaz de definir uma área de armazenagem pessoal, e os dados dessa área acompanhem-no por todos os espaços ativos em que ele estiver.

2.1.4 Trabalho em Andamento

Atualmente os integrantes do projeto Gaia estão ainda trabalhando na implementação do núcleo do sistema. Já foram implementados o núcleo do gerenciamento de componentes, o repositório do espaço, o gerenciador de eventos, o sistema de arquivos do contexto, o serviço de presença e o repositório de componentes. O *framework* para criação de aplicações também já funciona, e existem algumas aplicações de teste desenvolvidas e utilizáveis.

A partir de 2002 os pesquisadores começarão a concentrar seus esforços na criação de aplicações concretas que utilizem a infraestrutura do Gaia.

2.2 Ninja

O Ninja[2] é um projeto desenvolvido pela Universidade de Berkeley na Califórnia que tem como objetivo criar um sistema capaz de oferecer serviços em rede de forma escalável e acessíveis a partir de equipamentos de capacidade de processamento reduzida [GWvB$^+$00].

Ele aborda alguns dos problemas comuns aos espaços ativos, dentre eles como se adaptar a um ambiente com equipamentos de características diversas, como oferecer serviços escaláveis, tolerantes a falhas e acessíveis por milhares de clientes simultaneamente, como localizar um serviço no ambiente, e como permitir acesso seguro a estes serviços.

2.2.1 Arquitetura

A arquitetura do Ninja é composta por quatro componentes: *bases, proxies, units* e *paths.*

Bases: são *clusters* de diversas máquinas de uma arquitetura qualquer executando algum tipo de serviço que será disponibilizado às *units.*

Proxies: são os componentes intermediários responsáveis pela adaptação dos serviços às *units* que os utilizam. Os *proxies* podem converter tanto protocolos (como transformar uma conexão HTTP em uma conexão SSL, em casos onde seja necessária maior segurança ao acesso do serviço) como tipo de dados (*little endian* para *big endian* para computadores de arquiteturas diferentes, ou HTML para WML, para mostrar uma página da Web em um dispositivo com capacidade gráfica reduzida).

Units: são os aparelhos que utilizam os serviços do sistema. Estes aparelhos podem ser de qualquer arquitetura existente e usar qualquer tipo de protocolo, independentemente do serviço que eles queiram utilizar.

Paths: são caminhos entre uma *unit* e uma *base*, passando por um ou mais *proxies*, que viabilizam a utilização do serviço pela *unit.*

[2] http://ninja.cs.berkeley.edu

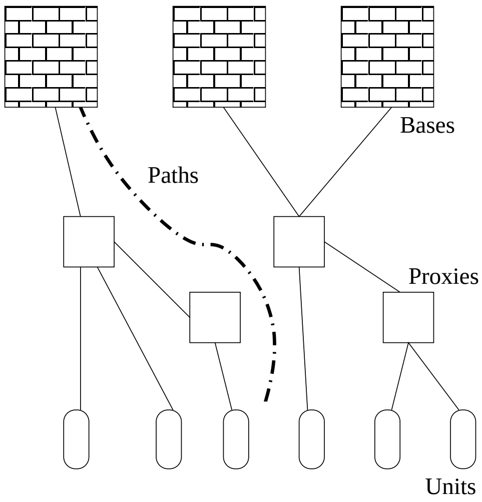

Figura 2: Estrutura do Ninja

Através desta arquitetura o Ninja consegue distribuir um serviço para máquinas de qualquer tipo, desde que se desenvolvam *proxies* para adaptar a arquitetura dessas máquinas à arquitetura do *cluster* em que o serviço está sendo executado. Ela também permite que a recuperação do sistema em caso de falhas de algum componente seja feita de forma simples, já que basta a alteração do *path* que estiver sendo utilizado para que a *unit* continue tendo acesso à *base*.

2.2.2 Serviços

Os serviços executados nas *bases* são escritos em Java. Desta forma, máquinas de qualquer arquitetura podem fazer parte do cluster rodando o mesmo serviço. As *units* que forem utilizar os serviços, entretanto, não precisam necessariamente de suporte a Java, pois muitas vezes as *units* não possuem capacidade computacional para rodar uma JVM (*Java Virtual Machine*). A interface entre as *units* sem suporte a Java e as *bases* nesse caso é feita através dos *proxies*.

As *units* podem se autenticar com os *proxies* quando utilizarem serviços que necessitem de um nível maior de segurança. Além disso, a comunicação entre o *proxy* e a *unit* pode ser criptografada para evitar que intrusos tenham acesso aos dados que estão sendo enviados à *unit*. Infelizmente alguns equipamentos, por terem restrições de processamento e memória, são incapazes de realizar a fase inicial do SSL, por exemplo, em um tempo razoável. Para estes tipos de equipamentos os *proxies* poderiam implementar algoritmos de criptografia mais simples em prol de uma maior velocidade no acesso aos serviços.

Para possibilitar a Descoberta de Serviços no Ninja, foi criada uma estrutura de servidores responsáveis por receber os anúncios de serviços existentes no sistema e responder às solicitações dos clientes que procuram por serviços. Os servidores anunciam em um canal multicast conhecido, qual o seu endereço e qual a taxa de repetição em que devem ser feitos os anúncios. Os provedores de serviço ouvem este canal e periodicamente anunciam em outro canal seus serviços, utilizando o endereço enviado pelos servidores. Os clientes enviam suas consultas em formato XML aos servidores, que respondem com uma descrição em XML do serviço que melhor se encaixa na consulta realizada. A segurança do serviço é feita através de Autoridades Certificadoras, que garantem a associação entre um serviço anunciado e uma entidade do mundo real, e Gerenciadores de Capacidades, que impedem o acesso de clientes a serviços não autorizados.

2.2.3 Trabalho em Andamento

Depois de três anos de financiamento governamental, o projeto Ninja foi encerrado em 2001. No último ano foram realizadas pesquisas relacionadas à estrutura de dados clusterizada, ao gerenciamento de eventos em aplicações com alta carga e concorrência, e à troca de mensagens entre os componentes do sistema, com a troca do RMI por um protocolo próprio assíncrono.

2.3 Oxygen

O Oxygen[3] é um grande projeto sendo desenvolvido pelo Massachusetts Institute of Technology (MIT). Ele pretende investigar diversos assuntos relacionados à computação em um Espaço Ativo, que vão desde aspectos de Interface com o Usuário até Redes, passando por Engenharia de Software e Sistemas Operacionais.

As principais áreas de pesquisa do Oxygen estão melhor descritas abaixo:

Tecnologia ao Usuário

Acesso ao Conhecimento: pesquisa melhores formas de administrar e lidar com o conhecimento existente. Aborda aspectos como Representação de Dados, Aquisição de Dados, Métodos de Acesso Automáticos, Métodos de Acesso para Usuários e Observadores Automatizados, que são subsistemas que observam o comportamento do usuário em busca de padrões de utilização que possam ser usados para refinar os dados apresentados.

Automação de Tarefas: procura formas de automatizar a execução de tarefas rotineiras baseadas em informações pré-existentes. Uma reunião poderia ser marcada automaticamente, baseando-se nas informações das agendas de cada um dos participantes, por exemplo. Pesquisa Objetos Básicos de Automação, que são 'caixas-pretas' reais ou virtuais responsáveis por executar tarefas específicas de baixo nível, e Controle sobre Objetos, que procura formas de combinar e controlar os objetos para que eles realizem tarefas automaticamente.

Colaboração: utiliza as tecnologias de Acesso ao Conhecimento e Automação de Tarefas para facilitar a interação entre os participantes do ambiente. Pesquisa como manter contextos de colaboração e como automatizar tarefas de colaboração.

Tecnologias de Interação

Voz: pesquisa formas de interação através da fala dos seres humanos. Abrange temas como Reconhecimento de Voz, Compreensão de Linguagens, Geração de Linguagens e Sintetização de Voz.

Visão: pesquisa formas de interação com o computador através da visão. Abrange temas como Reconhecimento de Objetos e Monitoração e Classificação de Atividades.

Tecnologia de Sistemas

Software Preparado para Mudanças: em espaços ativos, as freqüentes mudanças no ambiente exigem que o software esteja preparado para se adaptar a estas mudanças. Este grupo pesquisa técnicas de Abstração, de Especificação e Armazenamento Persistente de Objetos com Semântica Transacional, que tornem mais fácil a adaptação do software a essas mudanças.

Tecnologias de Rede: para permitir a integração de diversos equipamentos distribuídos por todo ambiente, várias tecnologias de rede devem ser desenvolvidas. Entre elas como estabelecer Regiões de Colaboração, Descoberta de Recursos e Locais, Segurança e Adaptação.

[3] http://oxygen.lcs.mit.edu

Pontos de Interação

Espaços Inteligentes: pesquisa características dos equipamentos que possam permear o ambiente e constituir um espaço inteligente. Estuda aspectos como Conexões com o Mundo Físico, Poder Computacional e Políticas de Software.

Equipamentos Móveis: pesquisa características dos equipamentos móveis que podem integrar um espaço inteligente. Estuda aspectos como Conexões com o Mundo Físico, Poder Computacional e Políticas de Software.

Infelizmente ainda não existem resultados práticos na área de pesquisa em Sistemas, que é o nosso principal interesse.

2.4 Context Toolkit

O Context Toolkit[4] é um projeto desenvolvido pelo grupo de *Future Computing Environments* do Georgia Institute of Technology (GeorgiaTech). A principal idéia do Context Toolkit é a de criar um conjunto de ferramentas que facilitem o desenvolvimento de aplicações sensíveis a contexto.

Como um dos pontos mais importantes de um espaço ativo é a coleta de informações contextuais e a utilização dessas informações pelas aplicações como forma de facilitar a interação entre o usuário e o sistema, o Context Toolkit acaba reunindo algumas características em comum com ambientes de computação ubíqua.

2.4.1 Arquitetura

A arquitetura do Context Toolkit pode ser melhor descrita através da figura abaixo:

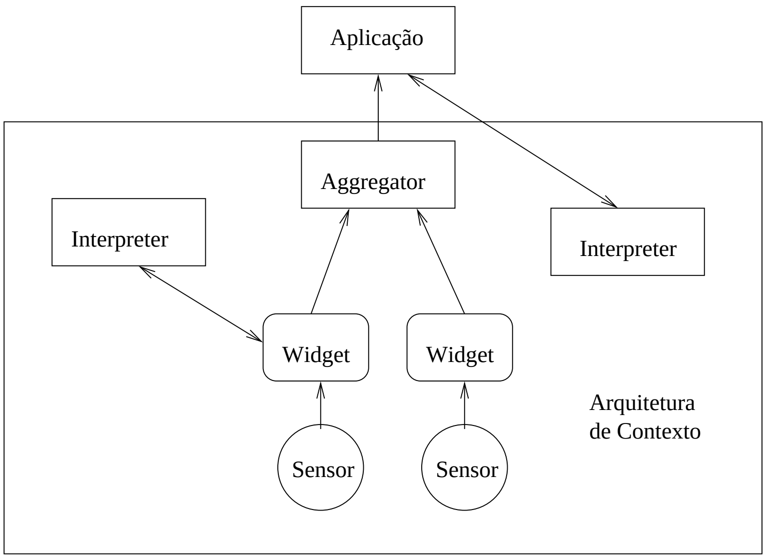

Figura 3: Estrutura do Context Toolkit

O Toolkit é composto por três abstrações principais [DA00a]:

[4]http://www.cc.gatech.edu/fce/contexttoolkit

Widget: utilizando o mesmo conceito dos *Widgets* de interfaces gráficas, foram criados os *Widgets* do Context Toolkit. Numa GUI, um *Widget* faz a mediação entre o usuário e a aplicação. No Context Toolkit, o *Widget* faz a mediação entre o usuário e o ambiente. Eles são responsáveis por encapsular informações de uma forma de contexto específica. Além disso os *Widgets* fornecem uma interface comum para as aplicações que utilizem contexto, independentemente dos sensores utilizados para captar a informação.

Aggregator: os *Aggregators*, também chamados de Servidores de Contexto, são *Widgets* um pouco mais poderosos. Eles são responsáveis por coletar as informações de contexto que se referem a uma entidade do mundo real, como uma pessoa ou um lugar. O *Aggregator* recebe dados de todos os *Widgets* que forneçam informações relevantes a uma dada pessoa por exemplo, e concentra essas informações num mesmo lugar.

Interpreters: os Interpretadores de contexto são responsáveis apenas por transformar uma forma de contexto em outra. Pode ser uma conversão simples, como retornar o e-mail dado o nome de um usuário, ou mais complexa, como determinar se uma reunião está ocorrendo a partir da quantidade de pessoas numa sala, do nível de ruído e da direção em que cada uma estiver olhando.

O Context Toolkit foi escrito inicialmente em Java, e depois foram desenvolvidas versões em outras linguagens (Perl, Frontier e Squeak). A maior parte das aplicações no entanto utiliza a versão em Java, já que esta linguagem propicia uma certa transparência de arquitetura graças à existência de JVM's para diversas arquiteturas de computadores diferentes. Isso é importante nesse tipo de aplicação, que além de lidar com as diversas arquiteturas de computadores existentes num ambiente distribuído tradicional, ainda tem que lidar com as diversas arquiteturas de sensores.

2.4.2 Serviços

Alguns dos serviços oferecidos pelo Context Toolkit seguem naturalmente a partir da arquitetura empregada. Os sensores são encapsulados em um conjunto de métodos comuns para acessá-los, independente do tipo de sensor. Os dados de um componente podem ser acessados via rede, de forma transparente pelas aplicações. Os dados de contexto podem ser compartilhados, já que cada *Widget* ou *Aggregator* pode ser utilizado por várias aplicações diferentes simultaneamente.

Todos os componentes são capazes de armazenar os dados de contexto captados. Esse armazenamento pode ser desabilitado no momento da criação do componente. No caso da implementação em Java, os dados são armazenados em um banco de dados JDBC.

A Segurança do sistema é feita de forma bastante simplória, através apenas de controle de acesso por origem/destino. Quando uma aplicação requisita uma informação de contexto, ela deve enviar juntamente com o pedido, o seu id. Se este id estiver habilitado para receber as informações de contexto solicitadas, ela consegue obter os dados. Caso contrário os dados não são enviados.

Descoberta de Recursos Nas primeiras versões do Context Toolkit as aplicações tinham de saber o *hostname*, a porta de comunicação e o id dos *Widgets/Aggregators* com os quais precisavam se comunicar. Nas versões seguintes foi desenvolvido um Serviço de Descoberta para tornar mais simples esta tarefa.

Uma aplicação que se utilize do Serviço de Descoberta deve especificar ao Servidor as informações de contexto que ela deseja receber (como local, nome do usuário, hora, etc.) e o atributo fixo (como "local = Bloco A"). O Serviço de Descoberta então devolve uma lista, contendo *hostname*, porta e id de todos os *Widgets/Aggregators* que forneçam as informações especificadas.

A aplicação também pode se inscrever no Serviço de Descobertas para receber informações de mudanças no ambiente. A aplicação seria informada se ocorresse algum problema com o *Widget* que ela está utilizando, ou do surgimento de outro componente que possa devolver as mesmas informações já especificadas. Além disso os programadores podem utilizar o Serviço de Descoberta para identificar quais informações de contexto podem ser identificadas no ambiente.

Quando qualquer componente é iniciado, ele se registra em um Servidor conhecido, informando seu id, *hostname*, porta, atributos que pode interpretar e atributos que pode gerar como saída. O Servidor periodicamente testa o componente registrado para verificar se ele ainda está ativo. Quando o componente é desativado manualmente, ele notifica o Servidor através da chamada a um método específico. Em caso de falha do componente no entanto, o Servidor só descobrirá que ele está inativo depois do componente não responder a um certo número de testes.

2.5 one.world

O one.world[5] é um sistema desenvolvido pela Universidade de Washington como parte do projeto Portolano, que procura resolver três dos principais problemas de um espaço ativo [GDH$^+$01]. O primeiro problema são os objetos, que não escalam facilmente em sistemas grandes e distribuídos, pois amarram muito fortemente código e dados numa única abstração. O segundo problema é a disponibilidade intermitente e limitada dos serviços num ambiente com as características de um espaço ativo. E finalmente, a heterogeneidade de equipamentos e plataformas de software de um tal ambiente.

Para evitar o primeiro problema, no one.world dados e código são mantidos isoladamente, e uma nova abstração é criada para agrupar os dois, mas ainda mantendo seu isolamento.

A responsabilidade por evitar o segundo tipo de problemas é passada quase que totalmente para o desenvolvedor das aplicações. Eles defendem que não há como 'esconder' a rede do desenvolvedor em um espaço ativo, pois as falhas de comunicação e a indisponibilidade de servidores é comum nesse ambiente, e não pode ser tratada como exceção. Portanto, os programadores devem desenvolver aplicações conscientes de que os recursos da rede podem estar indisponíveis a qualquer momento ou até mesmo não existir em um determinado ambiente. O one.world apenas oferece ferramentas que tornem esse tipo de programação um pouco mais simples.

Para lidar com o terceiro tipo de problema, eles propõem que seja criada uma API comum para desenvolvimento de aplicações, e um código binário comum, que permita que o mesmo programa possa ser executado em ambientes diferentes (Windows, Unix, PalmOS) e em hardwares diferentes (Intel, Sparc, Motorola) sem necessidade de recompilação do programa ou criação de uma nova versão da aplicação.

2.5.1 Arquitetura

A arquitetura do one.world pode ser exemplificada pela figura abaixo [GDL$^+$01]:

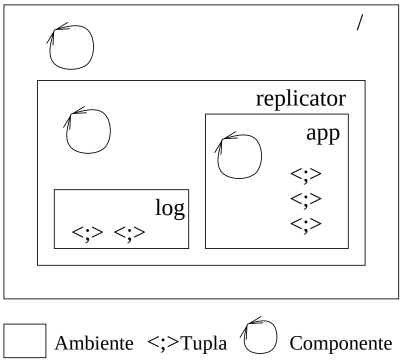

Figura 4: Estrutura do one.world

Cada equipamento roda uma instância do kernel do one.world. Estes equipamentos formam nós, que são totalmente independentes uns dos outros. As aplicações que rodam em cada nó são formadas por Componentes

[5] http://one.cs.washington.edu

e armazenam dados através de Tuplas. Os Ambientes fornecem estrutura e controle, servindo de contêiner para Componentes, Tuplas e outros Ambientes.

2.5.2 Serviços

O acesso aos serviços, tanto locais quanto remotos, é feito através de *leases*. Os *leases* limitam o tempo disponível para que as aplicações acessem os recursos e forçam as aplicações a renovarem seu interesse nos recursos solicitados. Esse mecanismo explicita as mudanças ocorridas no ambiente para as aplicações ao invés de escondê-las das aplicações, que é um dos princípios do one.world.

Em diversas aplicações em espaços ativos, os programas e dados devem seguir o usuário por onde ele for. Para facilitar essa tarefa, o one.world oferece o serviço de Migração. Através desse serviço, uma aplicação pode mover ou copiar um Ambiente e seu conteúdo – Componentes (código), Tuplas (dados) e estado de execução – para outro nó.

Outra necessidade de aplicações em espaços ativos é de comunicação entre os diversos componentes executando em nós diferentes pelo sistema. O serviço de Passagem Remota de Eventos (*Remote Event Passing* - REP) possibilita às aplicações enviarem eventos a nós remotos. Os eventos são assíncronos, e cada Componente deve declarar quais *event handlers* ele importa ou exporta para o sistema.

A Descoberta de Serviços no one.world é feita através do REP. Os Componentes exportam um *event handler* para Descoberta de Serviços. Essa ação se propaga pelo sistema e atinge o Servidor de Descoberta local. A Descoberta de Serviços suporta tanto *early binding* quanto *late binding*. Se estiver sendo usado o *late binding*, os demais componentes do sistema enviam uma consulta ao Servidor de Descoberta, que resolve a consulta. Senão, a consulta é feita diretamente aos componentes que exportaram o *event handler* para Descoberta de Serviços.

Para manter a disponibilidade dos dados, o one.world oferece um serviço de Replicação. Esse serviço permite que as Tuplas armazenadas em um nó estejam disponíveis em vários nós simultaneamente, estando estes nó conectados ou não.

Finalmente, o serviço de *Checkpointing* permite que se capture a pilha de execução de um Ambiente e se armazene a pilha em Tuplas. Dessa forma, um Ambiente pode retornar ao seu estado de execução simplesmente lendo as Tuplas armazenadas.

2.6 Jini

O Jini,[6] desenvolvido pela Sun Microsystems, é um sistema que procura tornar mais fácil a tarefa de administrar uma rede. A tecnologia Jini provê mecanismos para que equipamentos, serviços e usuários se juntem ou saiam da rede de forma simples e natural [Sun99].

Para que a arquitetura Jini funcione, pressupõe-se algumas características dos equipamentos e da rede em que o Jini é utilizado. A rede que conecta os computadores e os equipamentos deve ter uma velocidade razoável. Cada equipamento que vai se utilizar da tecnologia também deve ter um poder de processamento e uma quantidade de memória bons.

2.6.1 Arquitetura

A arquitetura de um sistema Jini pode ser descrita através da figura 5.

Provedor de Serviços: qualquer entidade (software, equipamento ou usuário) do sistema que deseje oferecer algum serviço aos outros membros da comunidade.

Cliente: qualquer entidade (software, equipamento ou usuário) do sistema que necessite de algum serviço da comunidade.

[6] http://www.sun.com/jini

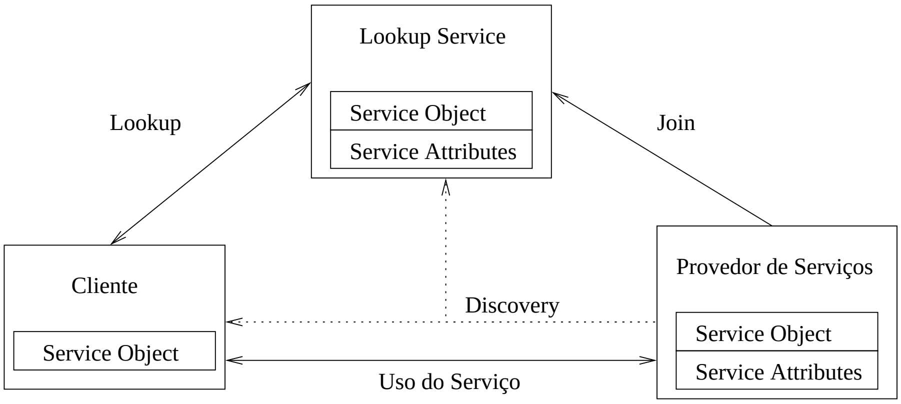

Figura 5: Estrutura do Jini

Lookup Service: serviço responsável por receber requisições de serviços dos clientes e identificar os provedores de serviço que podem satisfazer essas requisições.

Service Object: objeto Java que contém a interface para uso do serviço, incluindo os métodos que usuários e aplicações executam para utilizá-lo.

Service Attributes: atributos que descrevem o serviço oferecido, utilizados para identificar se o serviço satisfaz as necessidades do cliente.

A tecnologia Jini funciona resumidamente da seguinte forma:

- Quando um novo provedor de serviços é adicionado à rede, ele faz um *broadcast* à procura do *Lookup Service* daquela comunidade. Este é o processo de *Discovery*.
- Depois de receber a identificação do *Lookup Service*, o provedor de serviços se registra, enviando seu *Service Object* e seus *Service Attributes* ao *Lookup Service*. É a fase chamada de *Join*.
- O cliente consulta o *Lookup Service* sobre o serviço que ele necessita, passando o tipo do serviço mais alguns atributos opcionais. O *Lookup Service* identifica o serviço que se enquadra na consulta e envia o respectivo *Service Object* para o cliente, para que ele possa utilizá-lo. Esta é a fase de *Lookup* do Jini.
- De posse do *Service Object*, o cliente acessa diretamente o serviço através dos métodos definidos no objeto. Esse *Service Object* é válido por um certo período de tempo, determinado pelo *lease*. Passado esse período de tempo o cliente tem de renovar o *lease* para continuar tendo uma referência válida ao serviço.

2.6.2 Serviços

Basicamente a arquitetura Jini oferece dois tipos de serviço. O primeiro é o de Descoberta de Recursos. Através do *Lookup Service* todo cliente do sistema pode descobrir dinamicamente quais recursos estão disponíveis na rede num dado momento e escolher aquele que melhor se adapta às suas necessidades. No caso de falha de um serviço, o cliente consegue se recuperar graças ao mecanismo de *lease*.

O outro serviço oferecido pelo Jini é o de Segurança, do ponto de vista de acesso seguro aos serviços do sistema. A segurança do Jini é baseada no modelo do Java de entidades e listas de controle de acesso. Através deste modelo, serviços podem permitir que apenas alguns usuários ou outros serviços do sistema possam utilizá-lo, ou limitar as funcionalidades que cada usuário do sistema pode utilizar, dependendo de sua identidade.

3 Sistemas de Agentes Móveis

Um agente móvel é um programa que possui algumas características particulares que aumentam sua flexibilidade. Não existe um consenso sobre a definição de um agente móvel, cada autor possui a própria. De uma forma geral, um agente móvel apresenta as seguintes características [LO98]:

- Reage às mudanças do ambiente em que está sendo executado
- Possui controle das próprias ações
- É capaz de migrar de uma máquina a outra da rede

Graças a estas propriedades, a utilização de agentes móveis traz diversas vantagens em relação aos paradigmas tradicionais. Dentre elas, a redução do tráfego na rede, robustez, tolerância a falha, execução assíncrona e a capacidade de superar a latência da rede. Além disso, os outros paradigmas permitiam apenas transportar os dados para perto do código. Com os agentes móveis passou a ser possível transportar o código para perto dos dados, o que pode melhorar o desempenho de algumas aplicações específicas.

3.1 TACOMA

O TACOMA[7] (Tromsø And COrnell Moving Agents) é um projeto que visa desenvolver suporte do sistema operacional a Agentes Móveis [JvRS95]. O TACOMA suporta agentes capazes de se moverem por diversos pontos de uma rede, mas não oferece migração transparente de estado: o único meio de transferir o estado de um agente de um ponto para outro no TACOMA é explicitamente, através de variáveis.

Várias versões do TACOMA foram desenvolvidas, utilizando tecnologias diferentes:

- TACOMA 1.0 - Baseado em UNIX e Tcl-TCP. Agentes escritos em Tcl/TK
- TACOMA 1.1 - Baseado em UNIX e TCP. Agentes escritos em Tcl/TK ou C
- TACOMA 1.2 - Baseado em UNIX e TCP. Agentes escritos em Tcl/TK, C, Perl, Python ou Scheme
- TACOMA 2.0 - Baseado em UNIX e TCP. Agentes escritos em Tcl/TK, C, Perl, Python, Scheme, C++, ou Java

Em 2000 também foram lançadas versões para Windows, WinCE e PalmOS.

3.1.1 Estrutura do TACOMA

A estrutura dos agentes do TACOMA pode ser dividida em três partes: a parte responsável pelo código — chamada de *agent* — a parte responsável pelo armazenamento de dados — os *folders*, *briefcases* e *cabinets* — e a abstração responsável pela mobilidade dos agentes — a diretiva *meet*. Abaixo detalhamos cada um desses componentes.

agent: é a unidade computacional no TACOMA. Os *agents* podem funcionar como clientes, servidores, ou como um agente móvel, e são formados basicamente por uma porção de código e um estado.

Um *agent* tem a capacidade de se mover pelos servidores de uma rede. Além disso, o próprio *agent* é quem decide quando e para onde se mover.

folders: são unidades de dados acessíveis pelos *agents* tanto para gravação como para leitura. Os *folders* têm nomes relacionados normalmente com a natureza dos dados que eles armazenam. O *folder* CODE por exemplo, armazena o código do *agent*.

[7] http://www.tacoma.cs.uit.no

briefcases: são conjuntos de *folders* relacionados. A principal característica das *briefcases* é que elas podem ser carregadas pelo *agent*, enquanto ele se move pela rede. Num determinado host, por exemplo, o conteúdo do *folder* CODE poderia ser lido e executado.

cabinets: alguns *agents* não precisam se mover pela rede, mas precisam de alguma forma de armazenamento persistente de dados. Os *cabinets* são conjuntos de *folders* também, exatamente como as *briefcases*, mas não podem ser transportados pela rede. Ao contrário, os *cabinets* fornecem uma interface simples para armazenamento persistente de dados por parte dos *agents*.

meet: esta é a abstração que permite que os *agents* se movam. A sintaxe desta diretiva é `meet another_agent briefcase`, que significa que o *agent* que for se mover vai se encontrar com `another_agent` e lhe entregará a pasta `briefcase`. A pasta `briefcase` deve conter obrigatoriamente um *folder* chamado HOST com o endereço de destino do *agent*.

3.1.2 Exemplo de criação de agentes no TACOMA

O exemplo abaixo apenas envia uma frase para uma máquina remota. Na máquina remota existe um agente responsável por copiar a frase para a tela da estação. Queremos que a frase `''vamos comer?''` apareça na estação remota `mafalda`, que terá um agente local `ag_echo` responsável por copiar a frase para o `/dev/console`.

1. `bc_create out_bc`
2. `folder_store out_bc HOST mafalda`
3. `folder_store out_bc DATA ''vamos comer?''`
4. `folder_store out_bc OUTPUT /dev/console`
5. `meet ag_echo out_bc`

Na linha número 1, é criada a *briefcase* `out_bc`. Depois são criados três *folders*: HOST, contendo o endereço para onde o *agent* se moverá, DATA, que armazena a frase a ser mostrada remotamente, e OUTPUT, que guarda o local onde a mensagem deve ser mostrada. Na linha 5, finalmente, é dada a instrução para entregar a *briefcase* `out_bc` para o *agent* `ag_echo`. Através do *folder* HOST o sistema sabe que o *agent* `ag_echo` está localizado no computador `mafalda`.

3.2 Aglets

O Aglets é um sistema desenvolvido em Java para suportar o desenvolvimento e a implementação de agentes móveis. Inicialmente o *Aglets Software Development Kit* (ASDK) foi desenvolvido pela IBM[8], mas hoje o ASDK é um projeto de código aberto[9].

O ASDK, além de conter todas as classes necessárias para facilitar o desenvolvimento de agentes móveis utilizando Java, também possui uma aplicação chamada Tahiti. O Tahiti permite que um usuário seja capaz de enviar, receber e gerenciar os agentes do sistema através de uma interface gráfica, o que facilita a implementação de sistemas com agentes móveis.

A escolha da linguagem Java para desenvolvimento de um sistema de agentes móveis traz benefícios e prejuízos. Por um lado, a linguagem Java oferece transparência de plataforma, execução segura de programas de terceiros, carregamento dinâmico de classes, programação *multithread*, serialização de objetos e meios que permitam a reflexão do sistema. Por outro lado, a linguagem Java não permite que se controle a quantidade de recursos utilizados por uma aplicação, não permite controle nem proteção às referências públicas, e não possibilita a preservação do estado de execução e sua retomada posteriormente.

[8] http://www.trl.ibm.com/aglets
[9] http://aglets.sourceforge.net

3.2.1 Estrutura da API de um Aglet

Segue abaixo a descrição dos principais componentes que fazem parte da API para desenvolvimento de agentes móveis [LO98]:

Aglet: é um objeto Java móvel que visita pontos da rede habilitados para recepção de Aglets. Ele é autônomo, pois tem sua própria *thread* de execução, e reativo, pois é capaz de responder a mensagens.

Proxy: o Proxy é um representante do Aglet, que impede o acesso direto de outros agentes ao Aglet, como forma de proteção. Além disso, o Proxy e o Aglet podem estar em pontos diferentes da rede, permitindo transparência de localização do agente. Nesse caso o Proxy recebe todas as requisições e as encaminha para a localização atual do Aglet.

Context: o *Context* é o ambiente de execução de um Aglet. Cada nó da rede pode ter vários *Contexts* sendo executados simultaneamente. Cada *Context* possui configurações específicas de segurança e ambiente para os Aglets que são executados sob sua supervisão.

Identifier: é a identificação do Aglet. Esta identificação é globalmente única e não se altera durante o ciclo de vida do Aglet.

3.2.2 Operações realizadas por um Aglet

Várias operações básicas ao funcionamento de um Aglet foram implementadas no ASDK. Um Aglet é capaz de realizar as seguintes operações:

Criação: a criação de um Aglet ocorre dentro de um *Context*. O Aglet recebe um identificador, é adicionado ao *Context* e inicializado.

Clonagem: a Clonagem produz uma cópia quase idêntica do Aglet clonado. O novo Aglet tem um novo identificador, e sua execução é reiniciada. O estado de execução não é clonado.

Envio: retira o Aglet do seu *Context* de origem e adiciona o Aglet ao *Context* de destino, onde ele é reiniciado. O estado de execução não é enviado.

Retorno: quando um Aglet é solicitado para retornar ele é retirado do *Context* em que está e é adicionado ao *Context* de onde partiu o pedido de retorno.

Ativação e Desativação: a desativação interrompe temporariamente a execução de um Aglet e armazena seu estado em disco. A ativação continua a execução do Aglet no mesmo *Context* em que ele foi interrompido.

Destruição: quando um Aglet é destruído ele interrompe a sua execução e se apaga do *Context* em que está.

3.2.3 Comunicação entre Aglets

O ASDK fornece mecanismos para comunicação entre os agentes móveis do sistema. Existem três abstrações responsáveis por viabilizar a troca de mensagens entre Aglets:

Messages: uma *message* é um objeto passado de um Aglet para outro. Existem mecanismos para passagem de mensagens tanto de forma síncrona quanto assíncrona.

Future Reply: o objeto de *future reply* é utilizado numa troca de mensagens assíncrona para permitir que o Aglet que enviou a mensagem possa receber uma reposta no futuro.

Reply Set: um *reply set* é um conjunto de *future replies*, usado para obter as respostas conforme elas apareçam, sem ter de esperar que todas fiquem prontas para recebê-las.

3.2.4 Segurança

O ASDK tem um sistema de segurança bastante completo, baseado em permissões e entidades, que possibilita que tanto o servidor que hospeda aglets quanto os próprios aglets se protejam de diversos tipos de ataque. Além disso a configuração é bastante flexível, permitindo vários níveis de controle de acesso. As principais permissões que podem ser configuradas são:

Permissão de Acesso a Arquivos: pode-se controlar o acesso dos Aglets a um arquivo ou diretório, permitindo-se acesso apenas para leitura ou para leitura e escrita.

Permissão de Acesso à Rede: o acesso à rede também é sujeito a controle. Pode-se controlar o acesso do Aglet a servidores e a certas portas de comunicação.

Permissão de Acesso ao Sistema de Janelas: pode se dar ou não permissão a que um aglet abra janelas.

Permissão de Acesso ao Contexto: pode se controlar o acesso de um aglet aos serviços oferecidos pelo *Context*, incluindo os métodos para criar, clonar, enviar, retornar, ativar e desativar aglets.

Permissão de Acesso ao Aglet: os métodos oferecidos por um aglet também podem ter seu acesso controlado. Isso impede que um aglet qualquer possa acessar os métodos de um outro aglet.

3.2.5 Exemplo de agente utilizando o ASDK

A seguir temos um exemplo de agente móvel utilizando as bibliotecas do ASDK. Neste exemplo é criado um agente que é enviado a uma máquina remota para capturar o conteúdo de um diretório, e retorna à origem sozinho para mostrar a listagem adquirida.

```
public class ListingAglet extends Aglet {
    boolean back = false;
    File dir = new File(‘‘/tmp/PUBLIC’’);
    String[] list;
    URL origin = null;

    public void onCreation(Object o) {
        addMobilityListener(
            new MobilityAdapter() {
                public void onArrival(MobilityEvent me) {
                    if(back) {
                        for(int i=0; i<list.length;)
                            System.out.println(i + ‘‘: ‘‘ + list[i++]);
                        dispose();
                    }
                    else {
                        try{
                            list = dir.list();
                            back = true;
                            dispatch(origin);
                        }
                        catch(Exception e) {
                            dispose();
                        }
                    }
                }
```

```
            }
        );
        origin = getAgletContext().getHostingURL();
        try {
            dispatch(new URL(``atp://killi.genmagic.com''));
        }
        catch(Exception e) {
            System.out.println(``Failed to dispatch Aglet'');
        }
    }
}
```

O aglet tem 4 variáveis de instância: `back`, que assume o valor `true` antes do aglet retornar à sua origem, `dir` que contém o caminho do diretório que será listado pelo aglet, `list` que contém a listagem do diretório propriamente dita, e `origin` que contém o endereço de origem do aglet.

Ele implementa o método `onCreation` que instala um `MobilityListener` para lidar com os eventos de mudança de local de um aglet. Quando o aglet chega no seu destino, `back=false` e o aglet obtém a listagem do diretório solicitado. Quando a listagem é completada, `back=true` e o aglet é mandado de volta ao seu endereço de origem. O aglet então volta para o seu ponto de partida e mostra a listagem adquirida.

4 Aplicações de Agentes Móveis em Ambientes de Computação Ubíqua

Baseado nas informações apresentadas nas seções anteriores, foi possível identificar diversas necessidades de sistemas de computação ubíqua que podem ser supridas de forma eficiente através da utilização de agentes móveis.

No restante do documento apresentamos algumas destas aplicações. A lista que segue não é de forma alguma exaustiva, e novas formas de utilizar agentes móveis em ambientes de computação ubíqua podem ser identificadas durante o andamento de nossa pesquisa. Nos ítens seguintes, procuramos descrever o objetivo da aplicação proposta, como seria seu funcionamento em linhas gerais, e quais as vantagens que ela traria sobre as abordagens tradicionais, caso existam tais abordagens.

4.1 Atualização de Componentes

Um ambiente de Computação Ubíqua é composto por centenas e até milhares de componentes com capacidade de processamento. Uma das tarefas mais críticas num sistema como esse é a de manter o software de cada equipamento atualizado e consistente com as demais versões ativas no sistema, pois além dos equipamentos existirem em grande número, eles não estão disponíveis na rede o tempo todo. As versões utilizadas pelos componentes do sistema devem ser sempre as mais atuais possíveis, pois alterações de implementação de protocolos ou novas funções implementadas podem tornar o sistema mais eficiente.

Em [Kon00] propõe-se a utilização de agentes móveis para atualização de componentes do sistema. Foi demonstrado que a arquitetura cliente-servidor é muito dispendiosa em termos de banda de comunicação necessária a essas atualizações. No caso de equipamentos ubíquos, este problema se torna ainda mais grave, já que a transmissão de dados é atividade responsável pelo maior gasto de energia desses equipamentos.

De acordo com os testes apresentados em [Kon00], quando são utilizados agentes móveis para realizar este tipo de tarefa, o consumo de banda é bem menor. Como esta é uma atividade bastante freqüente num espaço ativo, tanto pela quantidade de equipamentos existentes quanto pela diversidade desses equipamentos, esta redução da utilização da banda de comunicação é essencial ao bom funcionamento do sistema.

4.2 Negociação de Requisitos

Aplicações de Computação Ubíqua precisam de algumas garantias do sistema acerca da qualidade dos serviços oferecidos por ele. Uma das formas de conseguir estas garantias é através da reserva de recursos pela aplicação, como proposto em [Gup99]. Além disso a aplicação também deve avisar o sistema de quais serviços ela precisa receber informações para poder funcionar. Em um dado instante, inclusive, para que uma aplicação adicional tenha condições de ser executada pelo sistema, pode ser necessário negociar os requisitos de hardware alocados pelas aplicações que já estão rodando.

Num ambiente de computação ubíqua isso pode ser ainda mais complicado, já que cada aplicação pode estar sendo executada em equipamentos em lugares diferentes. Para que essa negociação possa ocorrer com segurança e consistência é necessário um subsistema como o implementado em [NCN97].

Neste ponto os Agentes Móveis surgem como uma opção interessante. Um agente poderia ser programado para percorrer um caminho formado por todos equipamentos que possam ter seus recursos renegociados, e um certo objetivo necessário ao sistema operacional (como liberar mais 1kbps de banda de comunicação).

O agente passaria por todas as aplicações já rodando sobre o S.O. e negociaria a liberação dos recursos com cada uma. Dessa forma, aumentariam as probabilidades de uma nova aplicação conseguir ser executada pelo sistema. Se o objetivo fosse alcançado, o agente retornaria ao seu ponto de partida e comunicaria o fato ao kernel do sistema. Em caso contrário, se necessário, o agente poderia voltar às aplicações que tiveram seus recursos reduzidos devolvendo-os finalmente.

Com isto pode-se otimizar a utilização dos recursos por parte das aplicações, permitindo que mais aplicações rodem simultaneamente, garantindo a qualidade do serviço prestado.

4.3 Serviços Adaptáveis

Uma das características necessárias a um ambiente de Computação Ubíqua é que ele deve ser altamente adaptável. Este tipo de ambiente deve estar sempre disponível aos usuários, independetemente de falhas de componentes, atualizações, picos de utilização e versões de software utilizadas.

Os Agentes Móveis são estruturas de software bastante flexíveis, que podem auxiliar bastante na adaptação dos serviços disponíveis em um espaço ativo. Utilizando-se Agentes Móveis, a idéia de *Proxies* como proposta no projeto Ninja [GWvB+00] e o conceito de Migração utilizado pelo one.world [GDH+01] pode-se obter a flexibilidade desejada para facilitar essa adaptação.

Os próprios serviços do espaço ativo podem ser implementados como agentes móveis. Cada serviço seria disponibilizado através de um agente móvel mestre. Este agente teria além da responsabilidade de executar o serviço disponibilizado, monitorar o funcionamento do servidor onde está sendo executado e o seu próprio funcionamento.

Este agente mestre seria capaz de identificar quando o seu servidor começa a ter poucos recursos disponíveis e poderia migrar para outra máquina do ambiente que estivesse operando com mais recursos livres, antes que a falta de recursos afetasse a qualidade do serviço prestado.

O agente também poderia monitorar o seu próprio funcionamento e tomar decisões baseadas nas suas observações. Esta técnica, também chamada de computação reflexiva, permitiria que o agente detectasse por exemplo um aumento muito grande no número de requisições ao seu serviço. A partir deste dado, o agente mestre poderia criar várias cópias escravas em outros servidores, que supririam esta demanda instantânea pelo serviço. Assim que a quantidade de requisições voltasse a se normalizar, as cópias escravas poderiam ser eliminadas.

Os serviços poderiam também implementar agentes *proxies* para realizar a interação entre o cliente do serviço e o agente que implementa o serviço propriamente. Quando um cliente tentasse utilizar um serviço qualquer do sistema, esse agente *proxy* seria enviado. Ele seria responsável por receber todas as solicitações feitas ao serviço e enviá-las para o agente mestre responsável por processá-las.

Este agente *proxy* permitiria que atualizações e melhorias na implementação do agente mestre pudessem ser feitas de forma transparente para as aplicações. Quando uma nova versão do servidor fosse executada, as aplicações que se utilizam da versão antiga iriam receber um novo agente *proxy* e passariam automaticamente a utilizar o novo servidor.

Além disso, como os agentes responsáveis por processar as requisições dos serviços podem se mover pelos servidores disponíveis na rede, o agente *proxy* seria responsável por passar as requisições ao servidor em que o agente estivesse sendo executado naquele dado instante, livrando a aplicação da preocupação de ter de procurar o agente que implementa o serviço desejado pela rede.

Os agentes *proxies* também podem ser implementados de forma diferente para diferentes plataformas de hardware e software, a fim de otimizar os recursos disponíveis numa dada arquitetura de um cliente qualquer.

Esta arquitetura oferece várias características interessantes. Ela se adapta facilmente a picos de utilização do sistema graças à capacidade dos agentes mestres em criar escravos clones ao detectarem um aumento na demanda pelo serviço, e da sua capacidade de moverem a si próprios quando detectam que a máquina em que estão sendo executados está perdendo recursos. Permite que os agentes servidores sejam atualizados sem que o sistema todo tenha de parar para realizar a atualização ao mesmo tempo; os clientes são atualizados conforme precisem do serviço. E finalmente, máquinas de diferentes arquiteturas podem ter seus recursos explorados ao máximo através da criação de *proxies* específicos para cada uma delas.

4.4 Busca de Informações

Um ambiente de Computação Ubíqua pode interligar muitas redes e oferecer aos seus usuários uma quantidade enorme de informações. Usuários do espaço ativo podem desejar buscar algum tipo específico de informação entre as redes interligadas pelo ambiente. Essa pesquisa pode levar bastante tempo para ser efetuada pois o universo de informações a ser pesquisado é muito grande. Se esta pesquisa estiver sendo realizada a partir de um equipamento que se comunique via rádio, trabalhar em modo conectado em casos como esse pode ser inviável. Além do mais, equipamentos utilizados em um espaço ativo entram e saem da rede de forma intermitente, e não possuem conexão confiável.

Os Agentes Móveis possuem características capazes de suprir estas deficiências do sistema. Um agente móvel pode trabalhar em modo desconectado, o que não ocorre normalmente na arquitetura cliente-servidor, por exemplo. Uma busca às informações do sistema poderia ser transformada num agente móvel e enviada para a rede. Depois disso, o usuário pode se desconectar da rede sem problemas: o agente fará a busca sem a interferência do usuário e obterá a resposta. Assim que ele estiver conectado ao sistema novamente, o agente retornará com o resultado da pesquisa.

Este tipo de aplicação é bastante natural em ambientes de computação ubíqua. Nestes ambientes, a troca de informações entre o usuário e o sistema deve ser realizada da forma mais rápida possível, pois os equipamentos de interface com o usuário podem falhar, ficar sem bateria, ou ser desligados. Com agentes móveis, essa transferência de informações é feita de forma rápida, sem o custo de manter uma comunicação aberta esperando pela resposta. O usuário pode até estar fora do ambiente enquanto a consulta é realizada, que uma nova comunicação ocorrerá apenas quando os dados estiverem prontos.

4.5 Controle de Acesso

Muitas aplicações de ambientes de Computação Ubíqua necessitam de alguma forma de segurança no acesso aos dados e aos serviços oferecidos. A Segurança destes sistemas normalmente é feita utilizando-se dois métodos: **criptografia** na comunicação entre os equipamentos do ambiente, para evitar que intrusos possam ter acesso a informações indevidas através da monitoração da comunicação, e **controle de acesso** baseado em permissões, para garantir que apenas as entidades que tenham privilégio de acesso a um tipo de dado ou serviço tenham este acesso, e que o acesso de eventuais intrusos seja negado.

No entanto esse modelo é muito dispendioso em termos de computação e consumo de banda. O cálculo de chaves para criptografia leva tempo para ser realizado e consome energia dos equipamentos. E a comunicação criptografada cria um *overhead* de dados muito grande, que resulta num consumo de banda muito acima do que seria normal. Muitas vezes esse processo todo é realizado mas a aplicação não possui os privilégios necessários para completar a operação.

Nesses casos, as aplicações poderiam implementar agentes de segurança, que armazenariam toda política de acesso aos recursos que elas oferecem. Estes agentes seriam enviados a todos equipamentos que desejassem acessar algum serviço daquela aplicação. De posse da lista de permissões, o controle de acesso poderia ser feito localmente em cada equipamento. Isso evitaria o consumo de processamento e o envio de dados pela rede que seriam barrados no controle de acesso da aplicação.

Algumas questões deveriam ser consideradas na implementação de um sistema de segurança conforme descrito acima. Neste modelo, o Servidor deve confiar no Cliente, pois parte do processo de autenticação seria feito por ele. Um Cliente mal-intencionado poderia, em teoria, interferir no agente enviado pelo Servidor e por conseqüência no processo de autenticação, criando uma brecha na segurança do sistema. Isto aumenta muito a complexidade de uma implementação prática do modelo proposto, e pode até torná-lo inviável. Uma pesquisa nesta direção teria que primeiro identificar uma forma de resolver este problema.

4.6 Armazenagem Transparente à Localização

Todo usuário de um Espaço Ativo possui informações particulares que devem ser armazenadas de forma persistente, e devem estar disponíveis a ele sempre que necessário. Como os dispositivos utilizados como interface entre os usuários e o ambiente possuem restrições de espaço para armazenagem de dados, estas informações devem ser guardadas em algum dos servidores fixos do ambiente. Mas a longo prazo esta abordagem pode ser ineficiente. A massa de dados particular do usuário tende a crescer com o tempo. Toda vez que fosse necessário acessar algum destes dados, eles teriam que ser transportados pelo sistema do servidor onde estão armazenados até o dispositivo de interação com o usuário. O transporte de grandes quantidades de dados, no entanto, é custoso pois leva tempo, e consome recursos do ambiente (particularmente energia dos dispositivos mais compactos).

Uma das principais motivações para a utilização de agentes móveis é a de transferir a computação aos dados, ao invés de transportar os dados à computação [JvRS95]. Neste caso temos uma aplicação onde esta característica pode ser aproveitada para melhorar o desempenho do sistema.

Suponha um usuário que deseja fazer uma compra através de seu *handheld*. Para completar esta operação, ele precisa se autenticar, utilizando uma foto com marca d'água digital[10]. Esta foto faz parte do perfil do usuário, e está armazenada em algum ponto do sistema. Ao invés de se enviar todos os dados do usuário para o *handheld*, e realizar então a compra de posse destes dados, pode-se construir um agente que saia do *handheld* com destino ao lugar de armazenagem destas informações. Lá o agente adquire os dados necessários para efetivação da compra e prossegue concluindo o processo.

Utilizando-se esta proposta, pode-se economizar recursos de várias formas. A banda de transmissão é menos utilizada, já que são transmitidos pela rede apenas os dados da computação que se deseja realizar, que normalmente são menores que os dados que serão utilizados pela computação. Diminui-se também a quantidade de recursos necessária para armazenagem temporária de dados nos equipamentos distribuídos pelo sistema. Em um ambiente onde banda de transmissão e área de armazenagem são recursos escassos, a economia proporcionada pelos agentes móveis é muito importante.

5 Conclusão

A pesquisa em espaços ativos é uma área muito ampla, e que começou a ter mais atenção da comunidade científica apenas recentemente. Com a evolução dos equipamentos computacionais, a construção de tais ambientes se tornou viável, o que alavancará ainda mais as pesquisas nesta área e a melhoria destes sistemas nos próximos anos.

Os agentes móveis também são uma tecnologia recente, apesar de madura. Diversas aplicações de agentes móveis já foram implementadas com sucesso, mas ainda existem muitas áreas nas quais não foi estudada a utilização de agentes e os benefícios que eles poderiam trazer.

[10] http://www.digimarc.com/

No caso específico da aplicação de agentes móveis em espaços ativos, pouca pesquisa foi realizada até o momento. Pelo que podemos vislumbrar, a utilização de agentes móveis nestes ambientes é promissora. Através deste artigo enumeramos algumas aplicações e seus benefícios. Pretendemos no futuro implementar algumas destas aplicações na prática e contribuir com o desenvolvimento e maturidade da tecnologia de espaços ativos.

Referências

[CHRC01] Renato Cerqueira, Christopher K. Hess, Manuel Román, and Roy H. Campbell. Gaia: A development infrastructure for active spaces. *Workshop on Application Models and Programming Tools for Ubiquitous Computing 2001*, Setembro 2001.

[CN97] Hao-hua Chu and Klara Nahrstedt. A soft real time scheduling server in UNIX operating system. *IDMS*, 1997.

[CUH+01] Renato Cerqueira, Cristina Ururahy, Christopher Hess, Dulcineia Carvalho, Manuel Román, Noemi Rodriguez, and Roy Campbell. Support for mobility in active spaces. *Workshop on Middleware for Mobile Computing 2001*, Novembro 2001.

[CZH+99] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. *Mobicom '99*, 1999.

[DA00a] Anind K. Dey and Gregory D. Abowd. The Context Toolkit: Aiding the development of context-aware applications. *Workshop on Software Engineering for Wearable and Pervasive Computing*, Junho 2000.

[DA00b] Anind K. Dey and Gregory D. Abowd. CybreMinder: A context-aware system for supporting reminders. *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, Setembro 2000.

[DA00c] Anind K. Dey and Gregory D. Abowd. Towards a better understanding of context and context-awareness. *2000 Conference on Human Factors in Computing Systems*, Abril 2000.

[Dey00] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications.* PhD thesis, Georgia Institute of Technology, Novembro 2000.

[GDH+01] Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetheral. Systems directions for pervasive computing. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Maio 2001.

[GDL+01] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetherall. Programming for pervasive computing environments. Technical report, University of Washington, Junho 2001.

[GGWB99] Ian Goldberg, Steven D. Gribble, David Wagner, and Eric A. Brewer. The ninja jukebox. *2nd USENIX Symposium on Internet Technologies and Systems*, Outubro 1999.

[Gup99] Manish Premchand Gupta. Reservation based distributed resource management. Master's thesis, University of Illinois at Urbana-Champaign, 1999.

[GWvB+00] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *Special Issue of IEEE Computer Networks on Pervasive Computing*, 2000.

[JvRS95] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An introduction to the TACOMA distributed system. Technical report, University of Tromsø and Cornell University, Junho 1995.

[Kon00] Fabio Kon. *Automatic Configuration of Component-Based Distributed Systems*. PhD thesis, University of Illinois at Urbana-Champaign, Maio 2000.

[LCN98] Chih-han Lin, Hao-hua Chu, and Klara Nahrstedt. A soft real-time scheduling server on the Windows NT. *2nd USENIX Windows NT Symposium*, 1998.

[LO98] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, Agosto 1998.

[NCN97] Klara Nahrstedt, Hao-hua Chu, and Srinivas Narayan. QoS-Aware resource management for distributed multimedia applications. *Journal on High-Speed Networking*, December 1997.

[Pra01] Madhavarapu Jnana Pradeep. Discovery service. Technical report, University of Illinois at Urbana-Champaign, Janeiro 2001.

[Ran00] Anand Ranganathan. Space repository and active space web browser. Technical report, University of Illinois at Urbana-Champaign, Dezembro 2000.

[RC00] Manuel Román and Roy H. Campbell. Gaia: Enabling active spaces. *9th ACM SIGOPS European Workshop*, Setembro 2000.

[RHR+01] Manuel Román, Christopher K. Hess, Anand Ranganathan, Pradeep Madhavarapu, Bhaskar Borthakur, Prashant Viswanathan, Renato Cerqueira, Roy H. Campbell, and M. Dennis Mickunas. GaiaOS: An infrastructure for active spaces. Technical report, University of Illinois at Urbana-Champaign, Maio 2001.

[SDOA99] Daniel Salber, Anind K. Dey, Rob J. Orr, and Gregory D. Abowd. Designing for ubiquitous computing: A case study in context sensing. Technical report, Georgia Institute of Technology, Julho 1999.

[Sun99] Sun Microsystems, www.sun.com/jini/whitepapers. *Jini Architectural Overview*, Janeiro 1999.