ANDRÉ BIGONHA TOLEDO

PROTEUS: UM ARCABOUÇO PARA O PARTICIONAMENTO DE APLICAÇÕES ORIENTADAS POR OBJETOS NO AMBIENTE DA COMPUTAÇÃO PERVASIVA

Belo Horizonte
04 de setembro de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PROTEUS: UM ARCABOUÇO PARA O PARTICIONAMENTO DE APLICAÇÕES ORIENTADAS POR OBJETOS NO AMBIENTE DA COMPUTAÇÃO PERVASIVA

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ANDRÉ BIGONHA TOLEDO

Belo Horizonte

04 de setembro de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Proteus: Um arcabouço para o particionamento de aplicações orientadas por objetos no ambiente da Computação Pervasiva

ANDRÉ BIGONHA TOLEDO

Dissertação defendida e aprovada pela banca examinadora constituída por:

Dr. ROBERTO DA SILVA BIGONHA – Orientador

Dr. OSVALDO SÉRGIO FARHAT DE CARVALHO
Universidade Federal de Minas Gerais

Dr. MARCO TÚLIO DE OLIVEIRA VALENTE
Pontifícia Universidade Católica de Minas Gerais

Belo Horizonte, 04 de setembro de 2007

Resumo

Em um ambiente pervasivo, passamos a ter uma abundância de dispositivos computacionais capazes de se comunicarem. Estes dispositivos podem se encontrar muitas vezes em estado ocioso, o que os permite doar parte de seus recursos às aplicações em outros dispositivos. Esta dissertação apresenta um modelo de distribuição com foco neste ambiente e um Arcabouço para o desenvolvimento de soluções de distribuição para aplicações Java, que habilita aplicações centralizadas a se distribuirem pelos dispositivos do ambiente, aproveitando os recursos disponíveis.

Abstract

In a pervasive environment, we have an abundance of computational devices capable to intercommunicate. These devices can be many times in idle state, which allows it to donate part of its resources to applications in other devices. This master thesis presents a model of distribution and a framework for development of distribution solutions for Java applications in pervasive environment. The framework enables centralized applications to be distributed for the devices of the environment, taking advantage of the available resources.

Agradecimentos

Nada na vida conquistamos sozinhos. Sempre precisamos de outras pessoas para alcançar os nossos objetivos. Muitas vezes um simples gesto pode mudar a nossa vida e contribuir para o nosso sucesso.

Agradecer a todos que ajudaram a construir esta dissertação não é tarefa fácil. Meu maior agradecimento é dirigido a meus pais, Ruimar e Iracema, por terem sido o contínuo apoio em todos estes anos, ensinando-me, principalmente, a importância da construção e coerência de meus próprios valores e que sempre estiveram ao meu lado nas dificuldades, apoiando e acreditando e mim.

Agradeço aos meus tios Roberto e Mariza, por ter me acolhido em sua casa quando vim morar em Belo Horizonte. Agradeço o apoio recebido nesses últimos 2 anos, sempre presentes em minha vida. Eu devo muito a vocês. Muito obrigado.

Agradeço em especial ao meu orientador, Prof. Dr Roberto da Silva Bigonha, por ter me ensinado a arte de pensar o trabalho acadêmico com rigor e disciplina, propiciando-me a fundamentação básica, sem a qual este trabalho não teria sido escrito.

Agradeço à minha família, avós, tios, e primos e em especial a tia Rosana o carinho que sempre me dispensaram.

Devo agradecer também ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) o apoio financeiro, que possibilitou a minha estada em Belo Horizonte.

"O maior bem que podemos fazer aos outros não é oferecer-lhes nossa riqueza, mas levá-los a descobrir a deles" Louis Lavelle

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Na década de 60, os computadores se caracterizaram pelo alto custo e limitada capacidade. Nessa época, muitas pessoas utilizavam um mesmo computador e tinham que se locomover até os centros de computação. A partir da década de 80, com o início da disseminação dos computadores pessoais, as pessoas passaram a possuir seus próprios computadores, em casa ou no trabalho, mas esses ainda eram fixos e dependiam de o usuário se deslocar até eles. Após décadas de progresso do *hardware* e das redes sem fio, tornou-se possível a criação de dispositivos computacionais que até então eram tidos como futuristas: *handhelds*, *wearable computers* e dispositivos para aplicações de sensoriamento e controle. Esses dispositivos, em sua maioria móveis e portáteis, que podem acompanhar o usuário em seu deslocamento, são a base de um novo paradigma, o da Computação *Pervasiva*[1] [Satyanarayanan, 2001].

O paradigma da computação *pervasiva* refere-se a uma visão do futuro quando o poder computacional estará disponível em todo lugar. O conceito de Computação *Pervasiva* ainda não é um consenso. Diversos trabalhos utilizam os termos Computação Móvel [Augustin, 2004, Loureiro et al., 2003, Forman e Zahorjan, 1994], Computação *Pervasiva* [Satyanarayanan, 2001, Augustin, 2004] e Computação Ubíqua [Weiser, 2002, Satyanarayanan, 2001] ora como sinônimos, ora como diferentes estágios evolutivos do uso da computação. Seguimos a classificação utilizada por Iara Augustin [Augustin, 2004]. A Computação Móvel seria o primeiro estágio evolutivo, surgindo da convergência dos sistemas distribuídos com dispositivos portáteis e das redes sem fio. O usuário, agora, pode usar o dispositivo com mobilidade e acessar informações em todo lugar.

[1] Não existe tradução para o termo *pervasive* na língua portuguesa. Alguns pesquisadores utilizam o termo ubíquo, enquanto outros utilizam o neologismo *pervasiva(o)*. Uma vez que existem na literatura os termos *Ubiquitous Computing* e *Pervasive Computing*, e em nosso entendimento não são sinônimos, preferimos utilizar o neologismo *pervasiva(o)*.

No segundo estágio da Computação *Pervasiva*, não só a mobilidade física dos dispositivos é possível, como também a mobilidade de códigos e dados. Agora as aplicações também ganham independência de dispositivo. As aplicações também começam a interagir e se adaptar ao ambiente. A computação *pervasiva* surge da união de Computação Móvel, Computação em *Grid* [Foster, 2002] e a Computação Sensível ao Contexto [Chen e Kotz, 2000].

A Computação Ubíqua, também chamada de computação invisível, seria o próximo estágio de evolução. O conceito de Computação Ubíqua foi proposto originalmente em 1991 por Mark Weiser [Weiser, 2002] e cria o cenário de um mundo onde a computação ocorre de maneira transparente e natural ao usuário, em que os sistemas se tornam pró-ativos, passam a perceber os desejos dos usuários e automaticamente se ajustam e realizam tarefas.

Atualmente, os sistemas e dispositivos comercialmente disponíveis oferecem aos usuários uma experiência, principalmente, com as características descritas do primeiro estágio evolutivo, a Computação Móvel.

Nesta dissertação nos restringiremos aos desafios e características da Computação *Pervasiva*.

A possibilidade de o usuário interagir com diversos dispositivos computacionais e ter acesso contínuo a diferentes redes abre oportunidades para novos serviços e formas de acesso às informações. Serviços tornam-se mais dinâmicos e colaborativos e sensíveis ao contexto, como os serviços baseados na localização [Rao e Minakakis, 2003], isto é, serviços específicos oferecidos automaticamente para usuários segundo sua localização. A facilidade de se distribuir e interagir com novas aplicações cria um cenário de grande diversidade de aplicações especializadas disponíveis. Esse novo cenário de extrema conectividade e interatividade também deve permitir novas formas de interação entre o usuário e uma aplicação, tornando-a independente de dispositivo, permitindo assim ao usuário continuar seu trabalho mesmo que mude de dispositivo, como de um PDA para um *Desktop.* Essa migração da aplicação entre dispositivos deve ser feita da maneira mais transparente e simples possível.

A escassez de recursos nos dispositivos móveis, a diversidade de modelos existentes e o grande número de aplicações criam dificuldades para concretizar o acesso aos novos serviços, fazendo-se necessário, então, que esses dispositivos ou serviços se adaptem de forma a serem amplamente acessíveis.

A variedade de dispositivos impõe obstáculos à concretização da Computação *Pervasiva*, pois as aplicações podem necessitar de recursos que não estejam disponíveis como: saídas de áudio; captação de vídeo; tela colorida ou recursos escassos de processamento ou memória.

A grande heterogeneidade destes dispositivos, em termos de capacidade e arquitetura, torna difícil o projeto de software totalmente portátil e cria uma série de novos desafios para a comunidade científica. Diversas plataformas e sistemas de *middleware* que visam esconder ou minimizar tal complexidade e ainda prover novas abstrações e funcionalidades, estão sendo pesquisadas

[Geyer et al., 2006, Campbell et al., 2006, MIT, 2006, Zachariadis, 2005]. A mobilidade de código e a adaptação têm sido defendidas como um meio de obter todo o potencial da computação em ambientes móveis e dinâmicos [Augustin, 2004, Zachariadis, 2001, Zachariadis, 2005].

Em virtude da dinamicidade e heterogeneidade dos ambientes móveis, desenvolvedores devem se preocupar com características não funcionais desses ambientes como: capacidade de processamento, energia limitada, memória disponível, falhas de conexão à rede de comunicação, freqüência de entrada e saída de dispositivos no ambiente, robustez desses dispositivos.

Como vimos, as aplicações devem ser independentes de dispositivo. Estão habilitadas a serem executadas em uma grande variedade de aparelhos, como também devem poder deslocar-se entre eles. Dentre os vários problemas acarretados por esse objetivo, destaca-se o de como disponibilizar a execução de aplicações em dispositivos onde os recursos computacionais disponíveis, como memória e processamento, são inferiores à necessidade da aplicação.

1.1 O Problema

Um usuário não deveria ser impedido de realizar suas tarefas porque seu dispositivo não possui recursos de memória e processamento suficientes, quando em seu ambiente existe um excedente desses recursos.

Uma solução trivial é o usuário passar a utilizar a aplicação no dispositivo do ambiente que contém recursos disponíveis. Porém, esta solução é desconfortável, pois obriga ao usuário a procurar o dispositivo que possua recursos, e mais, o usuário deve possuir direito ao acesso a esse dispositivo e ele não deve estar ocupado com outro usuário. Além disso, o ambiente, como um todo, pode possuir recursos suficientes, mas individualmente em cada dispositivo eles não se encontram disponíveis. Por essas razões consideramos essa solução insuficiente.

Outra solução que pode ser empregada é a adaptação, troca ou até mesmo descarte de certas partes de software, provavelmente acarretando perda de funcionalidades. Apesar de ser uma estratégia interessante, existe a perda de funcionalidades e a adaptação pode não ser suficiente dependendo da quantidade de recursos disponíveis no dispositivo.

Em um ambiente *pervasivo*, passamos a ter uma abundância de dispositivos. Estes dispositivos podem se encontrar muitas vezes em estado ocioso, permitindo que parte de seus recursos sejam doados às aplicações em outros dispositivos. Com isso, em vez de descartarmos parte da aplicação ou de maneira complementar, poderíamos distribuí-la entre os dispositivos com recursos disponíveis. Enviando parte da aplicação para fora do dispositivo do usuário e evitando a perda de funcionalidades, assim, na visão do usuário, aparentemente, a aplicação continuaria local.

Esta abordagem leva a uma série de outros problemas e decisões de projeto:

- Que partes do software devem ser migrados e quando devem ser migrados?

  A maneira como as partes forem distribuídas interfere no desempenho da aplicação e no consumo de recursos. O acesso remoto a recursos é cerca de algumas ordens de grandeza mais lento que o acesso local [Busch, 2001]. Migrar parte da aplicação implica em um consumo de energia adicional já que o consumo de energia com a comunicação entre dispositivos sem fios normalmente é grande. Em compensação, se migramos um código que realiza um grande processamento e pouca comunicação para um dispositivo na rede fixa podemos economizar recursos.

  Realizar migrações mais freqüentemente pode melhorar o desempenho da aplicação, pois ajusta a distribuição às características atuais do ambiente e da própria aplicação; entretanto, aumenta-se o custo associado à execução do sistema de particionamento e consome-se mais recursos de banda e energia migrando objetos.

- Como descobrir os dispositivos disponíveis e quais selecionar para a migração?

  Antes de se realizar a distribuição, deve-se conhecer os dispositivos do ambiente e seus recursos excedentes (recursos de processamento, memória, banda de comunicação e disponibilidade de energia). Os dispositivos podem se tornar acessíveis ou inacessíveis ou podem ter uma grande variação nos recursos disponíveis, o que torna necessária a constante atualização desses dados. Deve-se fazer essa atualização, tentando-se minimizar o *overhead* de execução e o consumo de recursos, de maneira a não comprometer o desempenho das aplicações e os recursos disponíveis. Temos também que decidir para quais dispositivos distribuir. A escolha dos dispositivos leva a diferentes condições de desempenho, tolerância a falha e consumo de recursos.

- Como as partes do *software* devem interagir?

  As partes do *software* podem ser executadas remotamente ou apenas enviadas para fora e trazidas de volta quando necessário. No caso da utilização de chamadas remotas cria-se um novo problema de gerenciamento, pois a chamada pode requisitar mais recursos que estejam disponíveis no dispositivo remoto.

- Como garantir a segurança?

  Dados e códigos transmitidos podem sofrer alterações ou acessos indevidos no dispositivo destino, bem como serem agentes de danos nesses dispositivos. Além disso, deve existir um controle de acesso que determine quem tem direito de acessar quais dispositivos. Esse controle de acesso deve ser mais sofisticado que uma simples conta de usuário. Por exemplo, uma loja pode querer compartilhar seus recursos apenas com os clientes que

se localizem dentro do seu estabelecimento e negar esse acesso à loja vizinha, mesmo que compartilhem a mesma rede. Umas das formas de se implementar isto é utilizando a localização do usuário.

- Como lidar com uma eventual falha em alguns desses dispositivos?

  Dispositivos móveis são muito mais propensos a falhas que os dispositivos fixos. Alguns pontos de falhas comuns são um acidente físico, fim carga da da bateria, perda da conectividade temporária ou total. Após a distribuição da aplicação, o usuário deixa de ser sensível apenas a falhas no próprio dispositivo e passa a sofrer com falhas nos demais.

- Como lidar com *handoffs*[2]?

  Devido à mobilidade dos dispositivos que utilizam enlaces de comunicação sem fio, estes podem se distanciarem do ponto de acesso à rede utilizado. Com isso, para permanecerem conectados, devem mudar o ponto de acesso. Essa mudança de acesso cria problemas de roteamento de pacotes, uma vez que estes devem ser entregues em um novo endereço.

O presente trabalho teve como objetivo criar uma Ferramenta que permita explorar estratégias e configurações de distribuição de aplicações orientadas por objetos para o ambiente *pervasivo*. De forma complementar, implementamos uma ferramenta que permite agregar às aplicações Java centralizadas, de maneira automática, a capacidade de se distribuir entre os dispositivos do ambiente, quando existe uma situação de escassez de recursos computacionais como memória, energia e processamento.

Denominamos o conjunto Arcabouço e Ferramenta de Proteus. Proteus [Graves, 1990] é, na mitologia grega, um antigo deus do mar, filho de Oceanus e Thetis. Proteus é conhecido por ser mutável, versátil, ter a capacidade de assumir várias formas e prever o futuro. Devido a estas características, é considerado um sinônimo de versatilidade, flexibilidade e adaptabilidade. Características estas almejadas por este trabalho.

1.2 Modelo de Ambiente

Nesse trabalho, queremos possibilitar o caso de uso de um usuário que entra em um determinado centro de entretenimento e nesse centro, softwares estão disponíveis para *download* na rede local. Ele baixa o software e inicia a execução. Em determinado momento, recursos como a memória encontram-se insuficientes para a execução, a partir desse momento a aplicação antes centralizada, distribui-se entre os dispositivos do ambiente para conseguir recursos. Dependendo

[2]Processo de mudança de rede de conexão. A mudança de rede pode ser transparente ou não às aplicações.

das características da aplicação do ambiente ou da preferência do usuário, essa distribuição pode ser feita com base nos seguintes objetivos: melhor desempenho, menor consumo de energia, maior tolerância a falha.

As seguintes suposições foram feitas para o desenvolvimento deste trabalho:

- todos dispositivos do ambiente estão livres de defeitos e protegidos de quaisquer danos que impeçam seu funcionamento normal;
- todos os dispositivos permitem a obtenção de dados sobre seus recursos, como energia, processamento, memória e qualidade do sinal da rede sem fio;
- os recursos estão disponíveis no ambiente;
- o usuário não deseja modificar o ponto de acesso à aplicação, isto é, a interação com aplicação restringe-se somente pelo dispositivo onde iniciou-se sua execução;
- problemas de roteamento criados por modificação do ponto de rede *hand off* não são tratados nessa dissertação e supõe-se que são tratados transparentemente pelas camadas de transporte e rede.

A primeira suposição é aplicada durante a implementação para avaliação dos testes, mas durante a discussão do modelo proposto é desconsiderada.

1.3 Contribuições

Essa dissertação faz as seguintes contribuições:

- identificação dos problemas de distribuição de objetos em ambientes móveis e dinâmicos;
- desenvolvimento e avaliação de um novo modelo de distribuição que permite explorar com grande flexibilidade e generalidade estratégias de distribuição, que, além disto, permite considerar a diferença de capacidades dos dispositivos e requisitos de qualidade de serviço, tornando-o adequado ao ambiente *pervasivo*. Para isso foi desenvolvido um Arcabouço baseado em Java para explorar a distribuição automática de aplicações nesses ambientes. A distribuição automática tem benefícios sobre a distribuição manual tais como a menor propensão a erros, o aumento da produtividade do desenvolvedor, que não tem que se preocupar com as características de distribuição e o fato que aplicações “legadas” são facilmente portadas;

- proposta de um grafo representativo de objetos com granularidade para o particionamento de aplicações orientadas por objetos em tempo de execução;
- proposta de um gerenciamento distribuído para o particionamento da aplicação;
- proposta e avaliação de gerenciamento de particionamento distribuído.

1.4 Estrutura do Texto

No próximo capítulo são investigados trabalhos que tratam da distribuição de programas, avaliando a proximidade destes com os problemas encontrados no ambiente *pervasivo*. O Capítulo 3 é dedicado a descrição de um novo modelo de distribuição e de alguns algoritmos propostos. No Capítulo 4 é descrito Proteus, sua arquitetura e funcionamento. No Capítulo 5, descrevemos a implementação de Proteus, seu estado atual e limitações. E o último capítulo fazemos a conclusão do trabalho, descrevemos os problemas não tratados e os trabalhos futuros.

Capítulo 2

Soluções Para a Distribuição de Objetos

No paradigma de programação orientada por objetos, a distribuição de objetos visa permitir o compartilhamento de recursos pela disponibilização do acesso a objetos que se encontram em máquinas diferentes. Distribuir objetos é uma solução natural uma vez que confinam um estado e comportamentos relacionados. Objetos podem ser enviados ou criados em máquinas remotas e então acessados.

Muitas pesquisas já foram realizadas nessa área com foco na distribuição de objetos na rede fixa e resultaram em padrões e plataformas amplamente utilizados como CORBA [OMG, 2007] e JavaRMI [Microsystems, 2006c]. Esses trabalhos pressupõem características como:

- conexões de rede de alta velocidade e estáveis;
- topologia fixa da rede;
- dispositivos sempre acessíveis;
- alto poder de processamento;
- energia inesgotável;
- capacidade de memória abundante.

Essas suposições não são sempre válidas no ambiente *pervasivo*, onde também encontramos também as seguintes características:

- conexões de rede mais lentas e menos estáveis, erros e interferências são mais freqüentes devido ao uso de transmissões sem fio;

- topologia dinâmica da rede;
- dispositivos sujeitos a perda de conectividade;
- baixo poder de processamento dos dispositivos móveis se comparado com *Desktop*;
- energia limitada;
- capacidade de memória restrita.

Tudo isto pode tornar a simples transposição de soluções imprópria. As soluções podem se tornar ineficientes ou mesmo inviáveis, o que torna necessário modificações e adaptações nessas soluções, adequando-as ao novo ambiente.

Dadas estas características identificamos os seguintes requisitos como importantes para soluções visando um ambiente *pervasivo*:

- ter capacidade de avaliar o ambiente e a utilização de recursos necessários e disponíveis;
- possibilitar segurança da transmissão e execução;
- ter capacidade de distribuir a aplicação em busca de mais recursos;
- possibilitar a adaptação dinâmica da distribuição;
- possibilitar utilização de algoritmos diferenciados de acordo com a capacidade do dispositivo;
- ter capacidade de reagir a falhas em dispositivos.

A distribuição pode ocorrer de três formas: a primeira é feita de forma estática, isto é, antes da execução da aplicação, define-se em que locais estarão que partes da aplicação, e estas assim permanecem; a segunda seria a forma dinâmica em que a aplicação se distribui de forma automática pelos locais disponíveis, e a terceira seria um misto das duas soluções, inicia-se com a aplicação já distribuída e posteriormente a ajusta durante a execução.

A distribuição do ponto de vista do programador pode ser feita de duas maneiras: a distribuição explícita onde o programador insere manualmente o código responsável pela distribuição; ou por meio de distribuidores, que automaticamente inserem o código de distribuição na aplicação ou propiciam um ambiente de execução que trate da distribuição.

Nas próximas seções, descreveremos trabalhos que utilizam estas duas abordagens. Também descreveremos alguns dos ambientes de programação para dispositivos móveis, e por fim um arcabouço para criação de *middlewares* de distribuição orientados por objetos.

2.1 Soluções Explícitas

A vantagem da abordagem explícita é a possibilidade de se otimizar a distribuição, além de poder realizar o tratamento de erros de acordo com as necessidades da aplicação. Como desvantagens o programador deve aprender uma plataforma de distribuição, inserir códigos de distribuição e tratamento de erros, preocupar-se com o balanceamento de carga entre os dispositivos e mudanças na disponibilidade de recursos no ambiente, dentre outras. Estas tarefas não estão ligadas aos requisitos funcionais da aplicação, retirando assim o foco do programador da solução do problema para o funcionamento da aplicação.

A seguir são descritas algumas das principais soluções explícitas para distribuição.

2.1.1 CORBA

Common Object Request Broker Architecture (CORBA) é um padrão aberto para sistemas de objetos distribuídos definido pela OMG (Object Mangement Group) [OMG, 2006]. A OMG é composto por mais de 800 empresas com o propósito de definir padrões para aplicações orientadas por objeto.

CORBA foi criado para ser independente de hardware, linguagem de programação, sistema operacional e aplicações. CORBA define uma estrutura para prover acessos a objetos remotos.

Os objetos CORBA se comunicam por meio do chamado ORB (*Object Request Broker*). Esses objetos podem estar localmente ou remotamente localizados. ORB é uma camada de comunicação implementada como biblioteca para uma determinada linguagem.

Para permitir que objetos em diferentes arquiteturas possam se comunicar, é necessário especificar as características desses objetos de uma maneira uniforme. Para isso, CORBA define uma linguagem de definição de Interface (IDL). O programador descreve as características expostas do objeto na IDL CORBA e as implementam na linguagem de sua escolha. A partir da descrição das interfaces são gerados os *stubs* e *skeletons* que são entidades atuam como intermediárias entre o objeto cliente e o objeto remoto (veja Figura 2.1). Essas entidades realizam o mapeamento entre os tipos da arquitetura local e os tipos representados na IDL. Como a especificação da interface IDL é precisa, tantos os *stubs* como os *skeletons* podem ser compilados em linguagens diferentes, até mesmo os ORBs podem ser de fornecedores diferentes.

Cada linguagem de programação tem suas características próprias o que dificulta a descrição da interface de um objeto de maneira comum. Por exemplo, em IDL, *arrays* são tratados como tipos primitivos, enquanto que em Java são objetos.

A OMG padronizou o mapeamento da linguagem IDL para diversas linguagens como C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python e IDLscript.

Em CORBA todos os objetos têm uma referência única, uma identificação única.

A OMG padronizou o processo de invocação remota em dois níveis: em primeiro lugar, o cliente deve conhecer a interface do objeto que quer invocar. *Skeleton* e *subs* devem ser gerados da mesma IDL; e em segundo, o cliente ORB e o objeto ORB devem se comunicar por protocolo comum IIOP. Nesse protocolo é definido uma maneira comum de se especificar um objeto, operações e transportar os parâmetros. Esta arquitetura permite que objetos produzidos em linguagens e máquinas diferentes se comuniquem. Observe que temos uma separação do modelo em cliente e servidor: objetos servidores que são acessados remotamente e objetos clientes que acessam serviços desses objetos.

CORBA também define um conjunto de objetos que disponibilizam uma série de serviços, por exemplo, para controle do uso de objetos pelo usuário, serviços de segurança como autenticação e cifragem da comunicação e serviços de nomeação[1] *(naming).*

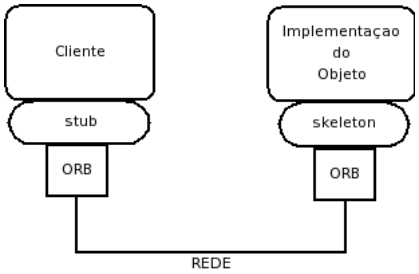


Figura 2.1: Cliente, Objeto Remoto e ORB

CORBA apenas permite que o programador torne seu código acessível e interoperável entre diversas máquinas, e tudo isso feito manualmente, isto é, por meio de programação. CORBA não possui mecanismos que permitam de maneira automática e transparente o rearranjo dos objetos em virtude das alterações no ambiente.

2.1.2 Invoção Remota de Métodos (RMI)

Invoção Remota de Métodos (*Remote Method Invocation ou* RMI) [Microsystems, 2006c] é uma tecnologia para permitir o acesso remoto a métodos de objetos Java distribuídos similarmente a tecnologia RPC [Coulouris et al., 1996]. Entrentanto com algumas diferenças notórias: RMI permite a transferência de objetos complexos, não apenas tipos primitivos, e não há a necessidade de se aprender uma linguagem de definição de interfaces (IDL) visto que RMI destina-se a interoperar apenas com a plataforma Java.

Não são todos os objetos que podem se tornar acessíveis remotamente, RMI possui algumas restrições como:

[1] Serviço que associa a referência a um objeto a um nome

- não é possível acessar campos, apenas os métodos são visíveis;

- o acesso remoto a métodos e variáveis estáticos também não é permitido.

O programador deve explicitamente tratar da distribuição em seu código. Para isso, todo objeto remoto deve possuir uma interface que estenda *java.rmi.Remote*. Nessa interface, devem ser definidos os métodos visíveis remotamente. Esses métodos podem lançar a exceção *RemoteException* para tratar problemas de rede que podem ocorrer. Um objeto de uma classe remota é passado por referência entre objetos de máquinas diferentes, enquanto objetos de classes não remotas são transportados por valor. Toda classe não remota cujos objetos são passados a objetos de classes remotas deve implementar a interface *java.io.Externalizable* ou *java.io.Serializable* para permitir que sejam serializados e enviados.

RMI permite o *download* das classes dos *stubs* que não se encontram localmente a partir de um servidor HTTP. Isso é feito especificando-se para a JVM as URLs *(Uniform Resource Locators*) onde essas classes estão disponiveis para *download.*

Apesar de possuir mecanismos de concorrência (*threads*) e distribuição (RMI) nativos, Java não suporta sincronização e coordenação de *threads* em um ambiente distribuído. Por exemplo, uma *thread* chama um método em um objeto remoto e este por sua vez chama um método em um objeto local. Para o sistema local é criada outra *thread* para tratar essa chamada remota, perdendo assim o acesso a monitores que se teria direito se os objetos fossem todos locais (chamados pela mesma *thread*) [Tilevich e Smaragdakis, 2004]. Assim não é possível, de maneira simples e direta, a transformação de uma aplicação que utilize esses recursos de centralizada em distribuída.

RMI também não suporta uma migração transparente de objetos remotos entre dispositivos, ao se mover um objeto, a atualização da nova localização e o redirecionamento de chamadas devem ser tratados pelo programador.

Essa tecnologia foi desenvolvida para ambientes distribuídos de rede cabeada, não sendo otimizado para o ambiente móvel sem fio. Por exemplo, toda comunicação é feita por meio do protocolo TCP, que não é apropriado para o meio sem fio, uma vez que sua detecção de congestionamento de rede confunde-se com a alta taxa de erros do meio sem fio, levando a uma queda acentuada na taxa de transmissão.

Como CORBA, RMI somente fornece os meios para a distribuição e não possui facilidades para adaptar a distribuição às características do ambiente.

2.1.3 JavaParty

JavaParty [Philippsen e Zenger, 1997] é um ambiente de programação orientada por objetos para aplicações distribuídas, onde a computação é realizada em um *cluster*[2]. Nesse ambiente o espaço de endereçamento é compartilhado, permitindo que objetos de classes identificadas como remotas pelo programador, sejam transparentemente visíveis a todas as partes do programa. Variáveis ou métodos estáticos são entidades únicas no ambiente. Outra característica desse ambiente é a transparência de localização, isto é, o programador não necessita de mapear objetos remotos ou *threads* para nós na rede, o compilador e o sistema de *runtime* cuidam dessa tarefa. Se o programador não especificar onde deseja posicionar o objeto remoto, este é mapeado de maneira aleatória. Durante a execução, um objeto pode ser migrado chamando-se uma função da API, mas essa migração falha se existir algum método do objeto a ser migrado na pilha de execução. JavaParty também preserva as semânticas de sincronização e controle de *threads* mesmo estas estando distribuídas.

A aplicação é desenvolvida em uma linguagem Java estendida, que é compilada para uma máquina virtual Java (JVM) convencional. Para sua execução necessita-se apenas de possuir o sistema de *runtime*, também implementado em Java. Nessa nova linguagem adiciona-se apenas a palavra chave *remote* à linguagem. Objetos de classes definidas como *remote* tornam-se acessíveis a todos os nós da rede.

Diferentemente de sistemas distribuídos convencionais, JavaParty não necessita de dividir o programa em porções cliente e servidor, lógicas complexas para criação de objetos remotos, ou tratamento de exceções de rede. A não ser pela palavra chave *remote*, a programação é feita de maneira tão simples como em Java puro.

JavaParty traduz transparentemente o programa fonte para Java puro mais RMI ou KaRMI (uma nova implementação similar ao RMI; porém mais eficiente), escondendo todo o trabalho do programador [Nester et al., 1999].

JavaParty supõe que os arquivos contendo código da aplicação são visíveis a todas as máquinas, via um sistema de arquivos distribuído ou por meio de cópias locais.

JavaParty, ao adicionar a palavra chave *remote,* simplifica ainda mais o desenvolvimento de aplicações distribuídas em comparação com RMI, além de prover transparência de localização e possibilitar o uso de *threads* distribuídos, mantendo-se as mesmas semânticas de sincronização definidas quando as *threads* são executas em um mesmo dispositivo. Entretanto, a transferência automática de objetos não considera quaisquer questões de balanceamento de carga e características e estados das máquinas disponíveis, o que pode tornar ineficiente esse mapeamento.

[2]Conjunto de computadores convencionais (*desktops*) interconectados por uma rede, trabalhando em conjunto como um único recurso de processamento e gerenciados por um controlador central.

JavaParty também não realiza a carga remota de classes, supondo que essas classes já estão disponíveis no sistema de arquivos. Outra desvantagem é a manipulação direta dos arquivos fontes, o que impossibilita alterar classes do sistema, pois são distribuídas apenas no formato binário(*bytecode*), impedindo assim que possam ser móveis e remotamente acessíveis. Também não há um tratamento para centralizar dispositivos de entrada e saída; assim uma saída de texto executada por um objeto é impressa em sua máquina e não na que iniciou a execução.

2.1.4 Emerald

Emerald [Jul et al., 1988] é um sistema distribuído desenvolvido pela Universidade de Washington. É formado por uma linguagem, um compilador e uma plataforma de execução. Todos os objetos em Emerald podem ser movidos. Cada objeto é formado por um nome único, dados primitivos, referências a outros objetos, um conjunto de operações e um processo opcional. Objetos com operações em execução possuem um processo.

Um processo são registros de operações que estão em execução em um determinado objeto. Uma *thread* pode possuir chamadas a operações em vários objetos. Cada objeto armazena as informações de contexto das *threads* que faz parte. Isto permite aos objetos serem migrados e garante o retorno das chamadas as operações.

Diferentemente de linguagem como Smalltalk e Java, objetos não são membros de uma classe e herança não é permitida. Objetos em Emerald conceitualmente carregam seu próprio código. A linguagem define primitivas que permitem ao desenvolvedor mover, fixar ou localizar um objeto. Além disso, permite que os objetos sejam associados de maneira que sempre migrem juntos. Emerald também permite a passagem de parâmetros por visita (*by visit*) ou por movimento (*by move*) . As duas são semanticamente equivalentes a passagem de parâmetros por referência, mas na passagem por movimento o objeto é migrado até o dispositivo onde é executado o método e lá permanece, enquanto na passagem por visita o objeto é migrado de volta ao fim da execução da operação.

Chamadas a objetos remotos ou locais são feitas de forma transparente ao programador. Referências locais são transformadas de forma automática em referências remotas quando necessário. Como o acesso a objetos locais é feita por meio de endereçamento direto, implica em nenhum *overhead* adicional. As referências remotas são na verdade *proxies* dos objetos remotos.

Emerald, por meio da linguagem definida, permite de forma simples o controle da distribuição de objetos. Além disso, permite uma execução eficiente ao referenciar objetos locais diretamente. Porém não provê herança, que é um poderoso mecanismo para reutilização de código e projeto de aplicações.

2.1.5 Conclusão

Nas seções anteriores descrevemos algumas soluções explícitas para a distribuição de objetos. A adaptação a mudanças no ambiente é uma questão fundamental em um ambiente *pervasivo*, e como podemos observar, essas soluções, apesar de esconderem questões relacionadas à comunicação e à rede do desenvolvedor, ainda demandam um grande esforço de desenvolvimento. Elas não possuem mecanismos que facilitam a adaptação da distribuição e a avaliação do ambiente de execução. Adicionar qualquer funcionalidade nesse sentido fica totalmente por conta do desenvolvedor.

Também é precário o suporte à própria migração dos objetos, sendo que, quando possível, interfaces e classes já devem estar pelo menos parcialmente no dispositivo de destino. Esses trabalhos devem ser adaptados e expandidos para que facilitem o desenvolvimento de aplicações com foco em ambientes pervasivos.

2.2 Soluções Implícitas

As soluções implícitas são soluções que são transparentes ao programador. Ao contrário das soluções descritas na seção anterior, o programador não precisa tratar da distribuição na linguagem de programação. Do ponto de vista do programador, temos um único espaço de endereçamento e um único dispositivo. Os benefícios da distribuição automática sobre a distribuição manual são: a correção, isto é, a menor propensão a erros; o aumento da produtividade do desenvolvedor que não tem que se preocupar com as características de distribuição; aplicações "legadas"são facilmente portadas. Como desvantagem temos uma possível perda de eficiência, uma vez que o desenvolvedor poderia otimizar a distribuição.

Nesta abordagem a distribuição é suportada automaticamente por um sistema de *runtime* que pode ser tanto de *software* como de *hardware*.

A primeira implementação desta solução foram os sistemas de memória compartilhada *(DSM)* [Li e Hudak, 1989]. Nestes sistemas, por meio de trocas de mensagens e técnicas de *caches,* a memória de todos os computadores é tratada como uma só. Isso envolve uma grande troca de mensagens. Atualmente a abordagem implícita utiliza técnicas baseadas em chamadas remotas de procedimentos, por serem mais eficientes que os *DSM*.

A seguir, apresentamos alguns trabalhos que realizam a distribuição implícita no ambiente convencional e com foco em ambientes móveis.

2.2.1 Pangaea

André Spiegel, em sua tese de doutorado [Spiegel, 2002], propõe o uso da análise estática da aplicação como indispensável para o particionamento e distribuição automática de programas orientado por objetos. A análise estática tem como objetivo determinar o que deve ou não se tornar acessível remotamente, isto é, quais classes devem receber todo o aparato que permita um objeto ser acessível remotamente. Uma solução simples seria tornar tudo remotamente acessível, mas essa solução tem um custo muito alto de desempenho, o que pode torná-la inviável.

Segundo Spiegel algumas das vantagens da análise estática são:

- identificar objetos imutáveis, isto é, objetos que podem ser livremente replicados pelo sistema e passados por valor;
- descobrir que certos objetos são utilizados por apenas uma parte do sistema e podem ser posicionados juntos em um mesmo dispositivo e não invocados remotamente;
- reconhecer oportunidades para migração síncrona de objetos (objeto migra seguindo o fluxo de execução do programa - move ou apenas visita) que é preferível a migração assíncrona (migração realizada feita pelo sistema de *runtime* independentemente do fluxo de execução).

Pangaea, que é a ferramenta construída para validar a proposta defendida acima, realiza um particionamento transparente e automático de código fonte Java puro centralizado, por meio de uma análise estática, sendo chamado de compilador distribuidor *(distributing compiler*).

Este compilador não é responsável por fazer a distribuição propriamente dita, mas pode ser usado com diversas plataformas de distribuição como JavaParty [Philippsen e Zenger, 1997], RMI [Microsystems, 2006c], dentre outras. O compilador é dividido em duas partes:

- *Front end*
  - analisa o programa e decide como distribuí-lo,
  - determina onde os objetos devem ser alocados,
  - decide quando realizar migrações;
- *Back end*
  - gera código,
  - esconde as características da plataforma de destino do analisador de distribuição usado no *front end*.

A ferramenta Pangaea possui uma interface gráfica que permite ao usuário, caso ache necessário, modificar a distribuição.

A análise estática é realizada mediante a criação de um grafo de objetos que representa a estrutura de execução da aplicação. Esse grafo é uma representação finita para um número potencialmente infinito de objetos da aplicação. Os vértices representam três tipos de conjuntos de objetos: vértices chamados `concretos`, que representam um conjunto unitário de objetos; vértices chamados `estáticos`, que também representam um conjunto unitário de vértices; e vértices denominados `indefinidos`, que representam um conjunto com possivelmente diversos objetos do mesmo tipo. As arestas podem ser de três tipos: uma aresta de criação, que liga um vértice que contém um objeto A a um vértice que contém um objeto B instanciado por A; uma aresta de referência, que representa em algum momento um objeto A que pode possuir uma referência ao objeto B; e, uma aresta de uso, que indica que um objeto A tem acesso a campos ou chama métodos de B.

Toda análise é feita desconsiderando-se o fluxo de execução.

A construção do grafo se dá da seguinte forma:

Etapa 1. Encontrar os tipos que constituem o programa:

A partir do método *main* da classe principal localizam-se todos os tipos que são referenciados.

Etapa 2. Construir um grafo de classes:

O grafo captura as relações do uso e de fluxo de dados no nível de classes, isto é, se uma classe A acessa campos ou faz chamadas a métodos de uma classe B, então é criada uma aresta de uso de A para B; se A propaga uma referência do tipo C para B, uma aresta de fluxo de dados C é criada entre A e B.

Etapa 3. Criar uma aproximação da população de objetos do programa:

Existem três tipos de objetos: `estáticos`, `indefinidos` e `concretos`. A idéia principal é separar as alocações iniciais das alocações não iniciais. As alocações iniciais são aquelas que garantidamente serão executadas apenas uma única vez na vida do objeto, são executadas nos construtores e estruturas de inicialização fora de um laço iterativo. As alocações iniciais geram objetos `concretos` e as não-iniciais, objetos `indefinidos`. Classes com campos `estáticos` geram um objeto estático que representa todos os campos e métodos estáticos dessa classe.

Em primeiro lugar, cria-se um objeto estático correspondente ao método *main*, iniciando-se a análise por ele e a cada instrução de alocação adiciona-se um objeto à população de objetos. Se a alocação somente puder ser executada um única vez (instrução em um

construtor fora de um laço), adiciona-se um objeto `concreto` à população de objetos. Senão adiciona-se um objeto indefinido, adiciona-se também arestas de criação e arestas de referência se aplicável. Observe que objetos indefinidos só podem gerar objetos `indefinidos` e um objeto só pode gerar um objeto `indefinido` do mesmo tipo. Para evitar a entrada em um laço infinito, dentro desta árvore criada pelos vértices e arestas de criação, antes de se criar um novo vértice de um tipo T, verifica-se se já existe um vértice antecessor do tipo T, caso exista cria-se uma aresta de retorno (aresta de criação) para ele ao invés de se gerar um novo objeto.

Etapa 4. Propagar referências no grafo de objetos:
Baseado nos fluxos de dados do grafo de tipos, adicionam-se arestas de referência.

Etapa 5. Criar as arestas de uso no grafo de objetos:
Se um objeto A conhece um objeto B e existe uma indicação de uso no grafo de tipos entre os tipos dos objetos, adicione uma aresta de uso entre os dois.

Uma vez gerado o grafo, Spiegel sugere a aplicação de algum algoritmo de particionamento de grafo, dividindo-o em um número de partições definidas. Ele também sugere adicionar pesos às arestas, pontuando o nível da possível interação entre os objetos. Todos os objetos que são acessados apenas internamente em suas partições não necessitam receber todo o "maquinário" para permitir seu acesso remoto, somente objetos acessados por objetos de outras partições.

Miriam Busch, em sua dissertação de mestrado [Busch, 2001], sugere ao projeto Pangaea um sistema de *runtime* responsável por realizar reposicionamento dinâmico dos objetos durante a execução com objetivo de reduzir a comunicação. Como o particionamento é feito *a priori* por particionamento estático, objetos com uma alta comunicação podem ter sidos separados. Busch usa uma solução baseada em uma análise local de cada objeto. Para cada objeto um objeto especial denominado *watcher* é associado e sua função é monitorar o acesso a métodos do objeto a que está associado e as partições de onde provém esses acessos. Com base nisso, o objeto *watcher* pode decidir migrar o objeto monitorado. A estrutura de implementação permite que diferentes estratégias sejam aplicadas a cada objeto, como definir quando será realizada a migração, após um determinado tempo ou número de chamadas, por exemplo. Apenas objetos acessíveis remotamente podem ser migrados. O sistema de *runtime* tem a característica de atuar mais como um sistema de ajuste do que propriamente de distribuição, que é realmente definido antes do inicio da execução; além disso, as características avaliadas para o ajuste, como número de chamadas são bastante limitadas e não levam em conta características como memória e capacidade de processamento das máquinas.

O objeto principal de Spegiel foi, ao se dividir uma aplicação entre várias máquinas, reduzir os custos de desempenho associados à comunicação entre as partes separadas. Criada para um modelo de ambiente fixo, onde antes da execução já se define como será o particionamento, a grande contribuição desse trabalho é o grafo representativo de objetos que permite uma melhor análise da aplicação. Nesta dissertação, vamos utilizar essas idéias adaptando-as para o ambiente dinâmico da computação *pervasiva*.

2.2.2 Infra-Estrutura Adaptativa Para Execução Distribuída (AIDE)

Adaptive Infrastructure for Distributed Execution (AIDE) é uma plataforma criada nos laboratórios da HP por Gu et al. [Messer et al., 2002, Gu et al., 2004], que, partindo do princípio que existe uma diversidade de dispositivos no ambiente, propõem uma plataforma distribuída para, transparentemente, distribuir uma aplicação entre os dispositivos do ambiente e assim tentar melhorar a utilização dos recursos disponíveis nesse ambiente (no atual estágio funciona com apenas um dispositivo do ambiente). O uso transparente de recursos do ambiente por um cliente é denominado por eles como *offloading*.

Segundo Gu et al., a distribuição da aplicação pode ser feita nas granularidades de objetos, classes ou componentes, e suas partes passam a ser acessadas por RPC. Nesse trabalho é avaliada a distribuição apenas na granularidade de classes e o recurso monitorado foi a utilização da memória.

O particionamento é feito de maneira dinâmica, construindo-se um grafo ponderado em tempo de execução e o particionando. Nesse grafo, os nós representam classes, e as arestas os relacionamentos entre elas, sendo que quanto maior a interação entre elas, maior o peso das arestas. Devido à decisão de particionamento se realizar sobre um grafo, a geração do grafo na granularidade de objetos torna-se uma alternativa pouco viável devido ao grande número de objetos que podem ser gerados. Como exemplo, citam um programa simples de edição de imagens que gera 16994 objetos distintos nos primeiros 174 segundos de execução [Gu et al., 2004] .

Um particionamento ótimo é um problema NP-Completo [Karypis e Kumar, 1998b]. Por esse motivo eles utilizam uma heurística baseada no algoritmo Mincut[Stoer e Wagner, 1997], onde alguns particionamentos são gerados e é escolhido o melhor entre eles. Feito o particionamento, os objetos são migrados e o acesso aos mesmos passa a ser feito por meio de RPC. Para a seleção do particionamento três medidas são utilizadas:

- *offloading delay - tempo gasto na migração, atrasos devido a acessos a dados e procedimentos remotos;*
- *atraso médio de interação - tempo de espera causado por chamada ou acesso a dados remotos;*

- *banda total - banda necessária para migração, e passagem de parâmetros a métodos remotos.*

Para realizar essa migração e transformações de chamadas locais em chamadas remotas de forma transparente, uma máquina virtual Java (*Java Virtual Machine* - JVM) modificada foi utilizada. Uma observação importante é que métodos nativos não podem ser migrados, pois estão em linguagem nativa e chamadas a esses métodos devem ser executados na JVM principal. O acesso a propriedades do sistema, como qual o sistema operacional, também devem ser executados na JVM principal.

Para realizar o particionamento não se deve apenas decidir como realizá-lo, mas também quando fazê-lo. Nesse trabalho, uma máquina de inferência baseada em lógica fuzzy é utilizada para esse fim. Segundo os autores, por meio dela, é possível uma maior adaptabilidade que a simples utilização de limites para a ocupação da memória. Com esse modelo pode-se utilizar na decisão, além dos níveis de memória outros indicadores como qualidade da banda de rede.

Ainda para melhorar a performance particionam classes que sejam maiores que 500 Kbytes criando novos vértices e distribuindo seus objetos entre eles.

Como trabalhos futuros, eles citam a avaliação de outros recursos como nível de bateria, e a utilização de múltiplos dispositivos para realizar o *offloading*.

Ao modificar a máquina virtual eles tornam qualquer aplicação Java passível de distribuição, mas tornam necessária a atualização e o desenvolvimento de novas máquinas virtuais para os diferentes dispositivos já existentes.

Este trabalho é um importante ponto de partida para o problema que investigamos. A partir dele observamos diversos pontos, que também poderiam ser investigados, como descentralizar as decisões de particionamento, distribuindo por exemplo o grafo de classes entre outros dispositivos, permitir que objetos da mesma classe sejam mapeados em vértices diferentes, eliminar a restrição de apenas um dispositivo como possibilidade para migração, e utilizar outros algoritmos de particionamento.

2.2.3 J-Orchestra

Nikitas Liogkas et al. [Liogkas et al., 2004], defendem o uso do particionamento automático como ferramenta para prototipação de aplicações ubíquas. Aplicações ubíquas têm caráter inerentemente distribuído, e dada a diversidade de dispositivos, essa distribuição tende a ser constantemente modificada. A abordagem utilizada permite ao desenvolvedor abstrair-se das questões de distribuição, desenvolvendo a aplicação de maneira centralizada, e depois, via ferramentas, realizar o particionamento da aplicação.

Defendem essa abordagem com foco apenas na prototipação, pois, segundo eles, requisitos dos usuários, como alta performance, balanceamento de carga, segurança ou comunicação assíncrona, são difíceis de serem incluídos de forma automatizada.

As abordagens tradicionais modificam uma aplicação centralizada para distribuí-la. *Middlewares* (por exemplo CORBA, DCOM, Java RMI) são utilizados para permitir que as diferentes entidades do programa comuniquem entre si. Entretanto, a programação do *middleware* requer que diversas convenções e restrições de programação sejam obedecidas, pois a execução de diversas ações familiares é radicalmente diferente, por exemplo, a construção do objeto acontece por meio de chamadas a uma fábrica de objetos remota; tais objetos necessitam ser registrados com um serviço especial para serem remotamente acessíveis, chamadas a procedimentos remotos normalmente não aceitam passagem por referência, dentre outros.

Nesse trabalho é realizada a inserção automática,do código de distribuição onde o desenvolvedor apenas indica via mecanismos de configuração quais e em quantas partes o programa deve ser particionado. Nessa abordagem é utilizada a granularidade de classe para o particionamento e a distribuição é feita mantendo-se a lógica e a estrutura equivalente a da aplicação original. Quando a aplicação acessa periféricos, como vídeo e som, o acesso a estes também fica distribuído e migra junto com as classes responsáveis pelo seu controle.

Para realizar a distribuição, as referências aos objetos passam a ser feitas por meios de *proxies*[3] que acessam verdadeiramente o objeto via chamadas locais ou, se o objeto for remoto, via RPC. Nem todas as classes podem ser modificadas para a utilização de *proxies*, por exemplo classes de sistema. Isso impede que algumas classes possam ser distribuídas de forma separada e as obriga a serem mapeadas em conjunto.

Esta abordagem foi validada por meio da implementação de uma ferramenta denominada J-Orchestra para o particionamento de programas Java. Na verdade esse trabalho ainda não permite a utilização de qualquer programa Java, por exemplo programas que utilizam *Multithreading* e sincronização podem apresentar comportamentos inesperados. J-Orchestra recebe como entrada os bytecodes da aplicação centralizada e os modifica para a distribuição. Não é necessário qualquer modificação da máquina virtual padrão.

Este trabalho não permite que a aplicação se reconfigure dinamicamente e não possui mecanismos de avaliação sobre condições do ambiente.

Apesar dessa abordagem ser voltada apenas para a prototipação, acreditamos, ao contrário do que os autores defendem, que a transformação automática pode ser bastante útil para produção de aplicações finais. Dada a complexidade dos sistemas pervasivos, se o desenvolvedor tiver que tratar todas as questões de balanceamento de carga, de segurança, e de diversidade de

[3]Um *proxy* é uma entidade que se passa por outra, atuando como um intermediáro entre a entidade cliente e a entidade a que representa.

dispositivos, apenas profissonais altamente especializados teriam capacidade de desenvolver tais aplicações. Pensando nisso, concluímos que o caminho para o futuro é permitir uma transformação automática flexível, que permita ajustes por parte do desenvolvedor, e auxiliada por um sistema de suporte à execução disponível nos dispositivos, propiciando serviços que permitam a aplicação lidar de forma dinâmica com tal complexidade e heterogeneidade.

2.2.4 Dynamic Juce

Teodorescu e Pandey [Teodorescu e Pandey, 2001] propõem um *middleware* denominado *Java for Ubiquitous Computing Environments* (JUCE), que permite a execução de código Java (traduzido na rede fixa para código nativo) em dispositivos de poucos recursos. Dispositivos requisitam o *download* de aplicações Java armazenadas na rede fixa. Antes de serem enviadas elas são traduzidas para código nativo por um compilador JIT e então são enviadas. O envio é feito na granularidade de métodos, e somente são enviados quando são invocados. A principal contribuição desse trabalho é a criação de uma infra-estrutura para computação móvel que permite a execução eficiente de código Java (uso de código nativo), menor utilização da memória e recursos de rede (migração no nível de método, e somente os utilizados) e economia de processamento no dispositivo (migração de tarefas como inicialização de classes e *link* dinâmicos para dispositivos da rede fixa). A tradução de *bytecodes* para código nativo dificulta a migração de aplicações no ambiente móvel em dois aspectos:

- o tamanho do código em seu formato nativo é em média de seis a oito vezes maior do que código em *bytecodes* [Zhang e Krintz, 2004]. No ambiente sem fio, a comunicação é propensa a erros. Assim quanto mais código a ser transmitido, maior a probabilidade de erros e também maior o consumo de energia;
- a utilização de código nativo também dificulta a migração de métodos e dados para execução em arquiteturas diferentes.

Athansiu et al. [Athanasiu et al., 2004], baseado em [Teodorescu e Pandey, 2001], criaram o *Dynamic* JUCE, que utiliza o conceito de coleções de código como um serviço do componente *runtime system* pertencente à arquitetura JUCE. Ele permite um gerenciamento do código nos dispositivos descartando métodos caso haja pouca memória, e requisitando-os novamente quando necessário. A principal contribuição desse trabalho foi o algoritmo de coleção de código, que cria um índice para cada método baseado nas métricas: número de invocações do método, número de invocações do método que o chama e quando foi chamado pela última vez. Com base nesses parâmetros determina-se o que será descartado. Experimentos medindo o desempenho do *middleware* mostraram que o principal "gargalo" é exatamente a compilação remota. Isso

pode ser resolvido criando-se uma coleção de código pré-compilados no servidor. O tempo de comunicação foi considerado um fator de baixo impacto, porém os testes foram realizados via rede *fast ethernet* sem congestionamento o que acarreta um baixo número de erros, divergindo do ambiente móvel. Também não foi tratada a questão da memória ocupada pela alocação de objetos.

Este trabalho, diferentemente dos tratados anteriormente, não tem como função distribuir objetos pelo ambiente e sim o código. Supondo a existência de um servidor de código, ele propõe um controle mais fino do código a ser enviado ao dispositivo e propõe um algoritmo para o seu descarte, se necessário. Em um ambiente *pervasivo*, onde a aplicação se espalha pelo ambiente, enviar apenas parte do código a ser utilizado pode reduzir o consumo de recursos e ganhar em desempenho. No que tange a distribuição do código em uma granularidade fina, por ser um trabalho complementar, essa solução pode ser integrada aos trabalhos discutidos anteriormente, como também, à proposta defendida nesta dissertação.

2.2.5 Particionamento Adaptativo (K+1)

Shumao Ou, Kun Yang e Antonio Liotta [Ou et al., 2006] propuseram um algoritmo denominado *Adaptive (k+1) Partitioning*, para o particionamento dinâmico de uma aplicação a ser executado em tempo de execução no ambiente *pervasivo*.

Esse trabalho, como o anterior, também não é um sistema de distribuição de objetos, mas está intimamente ligado a distribuição de objetos, uma vez que trata de um algoritmo de decisão, que decide como deve ser essa distribuição.

O algoritmo tem como estrutura de dados fundamental um grafo com múltiplos custos, isto é, cada vértice está associado a uma n-tupla cujos valores representam uma medida de custo do uso de recursos pela aplicação (memória, uso do processador,uso de largura de banda). As arestas também são ponderadas e representam a interação entre partes do programa e os valores, a freqüência dessa interação. Cada vértice do grafo representa uma classe, ou melhor, todos objetos de uma mesma classe. O algoritmo de particionamento é aplicado sobre grafo, e os objetos do programa são distribuídos segundo o vértice que os representa.

O algoritmo se baseia nos algoritmos de particionamento denominados algoritmos de múltiplos níveis (*multilevel algorithms*). Nestes algoritmos, o particionamento é feito da seguinte forma: primeiro diminui-se recursivamente o tamanho do grafo, por meio da fusão de vértices, então particiona-se o grafo e depois recursivamente se separam os vértices novamente até obter os vértices originais. Durante esse processo, os vértices são movidos entre as partições a fim de se realizar um refinamento.

As modificações propostas por Shumao Ou et al. são realizar a fusão dos vértices até que

o número de vértices atinja o número de partições e, dado o requisito de tempo real, ignorar a última etapa (refinamento), que possui uma elevada complexidade computacional.

De uma maneira simplificada, a fusão em cada passo é realizada da seguinte forma: vai se selecionando vértices aleatórios, escolhe-se seu vértice vizinho ainda não selecionado e realiza a fusão dos dois. O critério de escolha do vizinho é denominado por eles como *Heavy-Edge Light-Vertex (HELVM),* onde o vértice com a aresta de maior peso é selecionado e caso exista outras com o peso máximo é selecionado o vértice com o menor custo. Esse custo é dado por uma função que mapeia os valores da tupla em um número real. O grafo gerado em cada passo é usado pelo passo posterior.

Shumao Ou et al. comparam o algoritmo com o algoritmo *Mincut* utilizado no projeto AIDE [Messer et al., 2002], descrito anteriormente. Também comparam o *HELVM* com outros critérios de escolha de vértice: *Random Matching* (RM), *Heavy Edge Matching* (HEM), *Heavy Edge Matching* (LEM) e *Heavy Clique Matching* (HCM) propostas em [Karypis e Kumar, 1998a, Karypis e Kumar, 1998b]. Segundo seus resultados, seu algoritmo proporciona um menor tempo de resposta e um menor custo total de arestas entre partições.

Para permitir a distribuição, desenvolveram uma ferramenta que instrumenta classes Java. A ferramenta adiciona mecanismos para comunicação remota, monitoração de recursos e gerenciamento do particionamento.

Este algoritmo é interessante por se destinar a dispositivos de poucos recursos, comuns em um ambiente *pervasivo*, sendo assim, oportuno na investigação desta dissertação. Deve-se observar que dada a diversidade de dispositivos, em alguns destes (dispositivos com mais recursos) poderiamos utilizar algoritmos mais sofisticados e obter melhores resultados. Além disso, apesar de que nos (poucos) experimentos [Ou et al., 2006] realizados o algoritmo tenha obtido o melhor resultado, isso não garante que seja o melhor em todas as situações.

2.2.6 Um Compilador e Uma Infra-estrutura Para Distribuição Automática de Programas

Roxana E. Diaconescu et al. [Diaconescu et al., 2005] apresentam um compilador e um sistema de execução para distribuição automática de programas. Esse trabalho visa a distribuição de aplicações de propósito geral. Possui semelhanças aos trabalhos de Athanasi et al. [Messer et al., 2002, Gu et al., 2004] e Spigel [Spiegel, 2002], também realiza o particionamento baseado em uma representação em forma de grafo da aplicação.

Utiliza um modelo de ambiente onde os dispositivos alvo da distribuição podem ser dispositivos móveis de poucos recursos, mas exige um servidor de alta disponibilidade de recursos, que é responsável pelo particionamento uma vez que os algoritmos utilizados não são adequados

para dispositivos móveis e demandam recursos elevados de processamento e memória.

Basicamente o compilador recebe uma aplicação em *bytecode* Java e a transforma em duas representações intermediárias, *quad* (Quadrupla que se assemelha a uma representação baseada em registradores) e *bytecode*. A partir dessas representações é criado o grafo que representa a aplicação, que eles denominam de grafo de dependência do objeto. Este grafo também modela o uso de recursos pela aplicação. Aplica-se, sobre o grafo, a ferramenta de particionamento Metis [Karypis, 2006]. Essa ferramenta permite a utilização de diferentes algoritmos de particionamento. A aplicação é então modificada para permitir a comunicação entre as partes definidas pelo particionamento. Também é adicionado um sistema de monitoramento que coleta estatísticas de uso para que em trabalhos futuros sejam ultilizadas em um reparticionamento adaptativo.

O algoritmo para construção do grafo de dependência de objetos é baseado no grafo de André Spiegel [Spiegel, 2002] e é descrito em detalhes em [Diaconescu et al., 2003]. As principais diferenças são:

- O trabalho de Spiegel trata apenas distribuição por chamada remota síncrona, já nesse trabalho o modelo inclui a concorrência e a comunicação assíncrona.

- O uso de uma representação intermediária permite que aplicações em linguagens diferentes de Java sejam analisadas. Como trabalhos futuros, descreve que o código seja gerado novamente para a plataforma nativa alvo.

- Segundo eles, utilizam-se técnicas avançadas de análise para conseguir uma maior precisão e generalidade do grafo de tipos e de objetos. Com isso é possível representar um grafo com mais detalhes e próximo do grafo de real da aplicação.

Um dos primeiros passos do algoritmo é a aplicação do RTA (*rapid type analysis*) que é um algoritmo para criação de um grafo, que representa o fluxo de chamadas de métodos possíveis, em programas descritos em linguagens orientadas por objetos. O RTA[Bacon, 1998] foi proposto por David F. Bacon com parte de sua tese de doutorado.

Utilizando-se o grafo de chamadas e analisando-se os atributos e métodos das classes é criado o grafo de tipos, que descreve o relacionamento entre as classes. Como no algoritmo de Spiegel, os relacionamentos entre as classes também são classificados como arestas de uso, importação e exportação. O grafo de objetos também é gerado. Nele, são representadas as arestas de criação, uso e referência.

O particionamento atribui partições aos vértices. De maneira a diminuir a comunicação entre as partes e respeitar um conjunto de restrições a serem observados em cada partição.

Esse trabalho é uma adaptação aprimorada das idéias de Spiegel a uma infra-estrutura focada em dispositivos móveis, mas ainda não propicia a reconfiguração da distribuição dinamicamente, e é dependente da existência de um servidor central de alto processamento.

Apesar do foco em dispositivos móveis, a inexistência da reconfinguração do particionamento em tempo de execução pode limitar a utilização em um ambiente *pervasivo* real. Contudo, no contexto desta dissertação, esse algoritmo pode ainda ser utilizado como a base para definir o grafo utilizado no particionamento adaptativo em tempo de execução.

2.2.7 Conclusão

Cada uma das soluções apresentadas nesta seção trata de algumas das características que desejamos neste trabalho: a inserção automática dos mecanismos de distribuição e técnicas e algoritmos para o particionamento de uma aplicação orientada por objetos. Os trabalhos analisados ainda possuem uma aplicabilidade limitada em um meio *pervasivo*, pois não estão preparadas, de maneira geral, para lidar com o grau de variabilidade nas condições desses ambientes e pecam algumas vezes em não permitir adaptação dessa distribuição, ou realizar a distribuição com a adaptação sem considerar as condições do ambiente ou ainda utilizando um ambiente mais restrito.

2.3 Ambientes de Programação Para Dispositivos Móveis

A maioria dos dispositivos móveis conta com um compilador C/C++ e um conjunto de bibliotecas próprias, mas devido à variedade de dispositivos e sistemas operacionais diversas plataformas foram projetadas visando facilitar o desenvolvimento para esses dispositivos. As principais plataformas para desenvolvimento de aplicações para dispositivos de poucos recursos são normalmente versões simplificadas de plataformas de desenvolvimento para *desktops.* A seguir, descreveremos algumas:

2.3.1 Java

A plataforma de desenvolvimento Java[Microsystems, 2006b] é composta por uma linguagem, uma máquina virtual, bibliotecas e um conjunto de ferramentas. A linguagem Java é interpretada, isto é, seus fontes não são compilados para uma arquitetura alvo padrão, e sim para um formato especial de uma máquina de pilha virtual. Esta máquina é simulada por um interpretador, a chamada Java Virtual Machine (JVM). Existem JVMs para diferentes arquiteturas e sistemas operacionais, o que permite a execução de um mesmo programa Java em diversas arquiteturas sem a necessidade que esse programa seja alterado e recompilado. Java possui três

edições, cada edição com um enfoque de aplicações diferenciados. São elas: Java Standard Edition (Java SE) para aplicações *desktop; Java Enterprise Edition* (Java EE) para aplicações empresariais e Java Micro Edition [Microsystems, 2006a] (J2ME ou Java ME), direcionada para dispositivos de recursos limitados.

J2ME possui uma máquina virtual mais "leve", sendo um subconjunto dos componentes do Java *Standard Edition*. J2ME não é uma solução única, mas é formada por um conjunto de configurações e perfis que se adequam a certas classes de dispositivos. Dada a grande variedade de dispositivos, existe um compromisso entre portabilidade do código, funcionalidades e características específicas disponíveis nesses dispositivo. Assim um código Java ME é seguramente compatível apenas entre dispositivos que forneçam os mesmos perfis e configurações.

As configurações definem uma plataforma mínima, definindo recursos suportados pela linguagem, pela máquina virtual e suas bibliotecas básicas. Existem duas configurações: uma para dispositivos conectados em rede, com mais recursos de memória e processamento, chamada *(CDC Connected Device Configuration)*. A CDC define uma máquina virtual compacta denominada *Compact* Virtual *Machine* (CVM) que é totalmente compatível com a máquina virtual padrão, mas é otimizada para esses dispositivos. Os requisitos mínimos para um dispositivo suportar uma máquina CDC são:

- processador 32 bits;
- 2 MB disponíveis na memória principal;
- 2.5 MB de ROM;
- Algum tipo de conexão a rede.

A outra configuração, chamada *Connect Limited Device Configuration* (CLDC), se destina a dispositivos com maiores restrições de recursos e conexão e que também apresentam mobilidade. A CLDC define uma máquina virtual reduzida a Kilo Vitual Machine (KVM), atualmente na versão 1.1. A KVM tem tem os seguintes requisitos de recursos:

- processadores de 16 ou 32 bites;
- mínimo de 160 KB de memória persistente (182KB na versão 1.1);
- mínimo de 32 KB de memória volátil.

Como pode-se perceber, os dispositivos podem possuir recursos bastante restritos. Para acomodar uma máquina virtual nesses requisitos, foram feitas as seguintes restrições:

- não tem suporte a ponto flutuante (suportado na versão 1.1);
- o método finalize foi removido de *Object*;
- conjunto limitado de exceções;
- não tem suporte ao JNI (Java Native Inteface);
- carregador de classes personalizado não suportado;
- não possui suporte à reflexão computacional;
- sem *thread group*;
- não possui referência fraca (parcialmente adicionada na versão 1.1);
- não possui suporte a serialização de objetos;

Um perfil disponibiliza funcionalidades específicas (APIs) para uma categoria de dispositivos, eles são implementados para uma configuração em particular. As aplicações escritas para um perfil são totalmente compatíveis apenas com dispositivos que implementem aquele perfil. Para CDC temos os perfis *Foundation Profile* para dispositivos sem interface gráfica; *Personal Basis Profile,* um superconjunto do *Foundation Profile* que dentre outras funcionalidades passa a suportar um conjunto mínimo de recursos de interface (AWT) e *Personal Profile* , bastante semelhante ao JavaSE, também um super conjunto do *Personal Basis Profile*. Enquanto para o CLDC, os perfis disponíveis são o *Mobile Information Device Profile* (MIDP), para dispositivos móveis com *display* como celulares e *Information Module Profile* (IMP) para dispositivos similares ao MIDP sem interface gráfica.

Atualmente, JavaME [Microsystems, 2006a] encontra-se disponível em bilhões de dispositivos móveis.

2.3.2 Brew

Brew [QUALCOMM, 2006] é a solução da Qualcomm para desenvolvimento, execução e distribuição de aplicações e conteúdos para dispositivos móveis. BREW é encontrado principalmente em celulares CDMA.

O ambiente de execução situa-se entre as aplicações e o sistema operacional e disponibiliza uma série de APIs ao desenvolvedor, de maneira que ele não necessite conhecer as características específicas do dispositivo. Possui uma arquitetura extensível e aberta permitindo a terceiros estenderem e implantarem o sistema em qualquer dispositivo e rede.

O Brew é baseado em C/C++, suas aplicações são distribuídas em código nativo. É permitido ao desenvolvedor acessar diretamente funções do dispositivo se necessário, diferentemente de Java que limita o acesso ao sistema do dispositivo. Isto permite mais segurança, impedindo que aplicações mal-intencionadas danifiquem o dispositivo. Brew contorna o problema da segurança por meio do *Brew Delivery System* descrito adiante.

Brew pode ser estendido para outras linguagens interpretadas como Java, Flash e XML. Algumas máquinas virtuais já foram implementadas sobre BREW.

O sistema de distribuição Brew denomina-se *Brew Delivery System* (BDS) e fornece as operadoras um modelo de negócios integrado, que permite às operadoras o controle sobre a seleção, o gerenciamento, a política de preços, o acompanhamento de uso e a cobrança dos aplicativos.

O BDS cria um mercado virtual onde os desenvolvedores podem submeter seus aplicativos no mercado mundial. Brew não permite a instalação de softwares diretamente a não ser pelo BDS. Essa restrição também é utilizada para fornecer segurança aos usuários, uma vez que os aplicativos são testado e recebem um certificado da Qualcomm.

2.3.3 O Arcabouço .Net Compact

O Arcabouço *.Net* [Microsoft, 2007] é a solução Microsoft para desenvolvimento da próxima geração de softwares e XML *web services*. Possui componentes que facilitam a integração e troca de informações e funcionalidades sobre a rede. Disponibiliza protocolos independentes de plataforma como SOAP, HTTP, XML.

Consiste em dois componentes principais:

- Linguagem Comum de Tempo de Execução (*Common Language Runtime*): Esse componente é responsável por fornecer um ambiente de execução, tratando do gerenciamento e execução do código e provendo serviços como, gerenciamento de código (impedir "vazamento" de memória (*Memory Leak*), tipos fortes, *bad references*) e *threads;*
- conjunto de bibliotecas.

Programas desenvolvidos em .Net podem ser implementados em diferentes linguagem, como VB e C#. Estas linguagem são compiladas para um linguagem denominada MSIL (*Microsoft Intermediate Language*), essa nova linguagem é transformada, via um compilador JIT, em código nativo quando é invocada sua execução. Isso quer dizer que os programas escritos em .Net não são interpretado como em Java (determinadas JVMs também fazem uso desse recurso), e sim em código nativo, compilado antes da execução.

O Microsoft .NET Compact Framework é versão do .Net Framework adaptada para os dispositivos inteligentes, que possuem restrições e recursos limitados. Utiliza um compilador JIT

de alta performance. Possuem as mesmas classes básicas do .Net mais algumas que são específicas para a mobilidade. O Microsoft .NET Compact Framework necessita de dispositivos com mais recursos se comparados com os dispositivos suportados pelo JavaME CLDC.

2.3.4 Mobile AJAX

AJAX ou *Asynchronous JavaScript and XML* não é uma tecnologia, mas sim um conjunto de tecnologias atuando em conjunto. São elas:

- XHTML e CSS como padrões para apresentação;
- Document Object Model para exibição e interação dinâmica;
- XML and XSLT para troca e manipulação de dados;
- XMLHttpRequest para requisição assíncrona de dados;
- Javascript para implementar, coordenar e integrar as tecnologias acima.

AJAX permitiu que conteúdo exibido por navegadores Web tomassem forma semelhante a de aplicações *desktop,* como planilhas de cálculo e processadores de texto. Com essas tecnologias é possível que a geração da página HTML seja feita localmente no navegador e que este se comunique de forma assíncrona com um servidor de dados. Assim o modelo tradicional é expandido. Anteriormente, as requisições de informações eram feitas apenas pelo usuário, agora, com AJAX, o navegador pode requisitar sozinho informações em segundo plano. O navegador também fica dispensado de gerar uma nova página HTML por completa, quando apenas parte da página foi modificada, requisitando e transferindo do servidor apenas as informações necessárias. Com isso o usuário passa a ter uma experiência mais rica.

A Opera[Opera, 2006b] recentemente lançou um navegador Web chamado Mobile Opera e uma plataforma [Opera, 2006a] com suporte a AJAX para dispositivos móveis. A plataforma denominada *Opera Plataform* permite que aplicações AJAX sejam armazenadas no dispositivo. Além disso, permite que essas apicações acessem as funções do telefone como lista de contatos, SMS e email.

Ainda em desenvolvimento, Mojax[mFoundry, 2006] é um *framework* para o criação de aplicações AJAX, como *Opera Plataform*. As aplicações são desenvolvidas para serem armazenadas no dispositivo. Mojax também permite acesso a funcionalidades do dispositivo como Câmera API, *Push Messaging, Bluetooth, Location Services*, contatos e dentre outras.

As principais vantagens de se utilizar AJAX em dispositivos móveis seriam: A utilização de padrões e tecnologias Web já difundidas, e um melhor desempenho das aplicações, uma

vez que o tráfego de informações pela rede diminui se comparado com acesso a páginas Web convencionais. Uma desvantagem é que ainda existem poucos navegadores para dispositivos móveis que suportam essa tecnologia.

2.3.5 SuperWaba

SuperWaba[SuperWaba, 2006] é uma plataforma para desenvolvimento de aplicações para dispositivos móveis criada em 2000. Possui sintaxe semelhante a Java, mas possui uma máquina virtual e bibliotecas próprias.

Algumas de suas características são: suporte para SQL, leitor de código de barras, protocolo GPS, sockets, serial/IR, Bluetooth, protocolo HTTP, tratamento de imagens, suporte aos tipos double e long, parser XML e HTML e uma bibloteca de Jogos nativas.

SuperWaba possui implementações para dispositivos Palm OS, Windows CE, Pocket PC, Windows Mobile e Symbian OS.

O SDK SuperWaba é distribuído em duas versões uma paga: com mais recursos e outra livre nas licenças GPL ou LGPL.

2.4 Arcademis

Desenvolvido na tese de mestrado de Fernando M. Q. Pereira, Arcademis[Pereira et al., 2006, Pereira, 2003] é um arcabouço, escrito em Java, para construção de sistemas de objetos distribuídos, que servirá como estrutura base para o presente trabalho. De maneira geral, ele descreve uma estrutura para permitir a localização e a invocação remota de métodos de objetos.

Como vimos Java provê RMI, um mecanismo de alto nível que habilita aplicações a realizarem chamadas a métodos remotos. Todavia, como discutimos na Seção 2.1.2, esse mecanismo não é adequado à comunicação sem fio. O fato de ser um mecanismo fechado cria outro empecílio à utilização de RMI. Assim, não há possibilidade de realizar nele adaptações diretamente. O uso de um mecanismo aberto (Arcademis) possibilita um maior controle e monitoramento sobre a comunicação. Em J-Orchestra (Seção 2.2.3), que utiliza RMI, uma entidade extra (*Proxy*) foi criada por não ter acesso a implementação de RMI.

O objetivo de Arcademis é permitir a criação de *middlewares* flexíveis e reconfiguráveis. Sua utilização nesse trabalho possibilita a criação de *middlewares* personalizados para computação *pervasiva*, por meio do uso de protocolos de comunicação adequados, adição de mecanismos de segurança e monitoramento.

Arcademis é formado por um conjunto de classes e interfaces que devem ser estendidas para criação do *middleware.* Variações como chamadas assíncronas, uso de *caches* e protocolos de

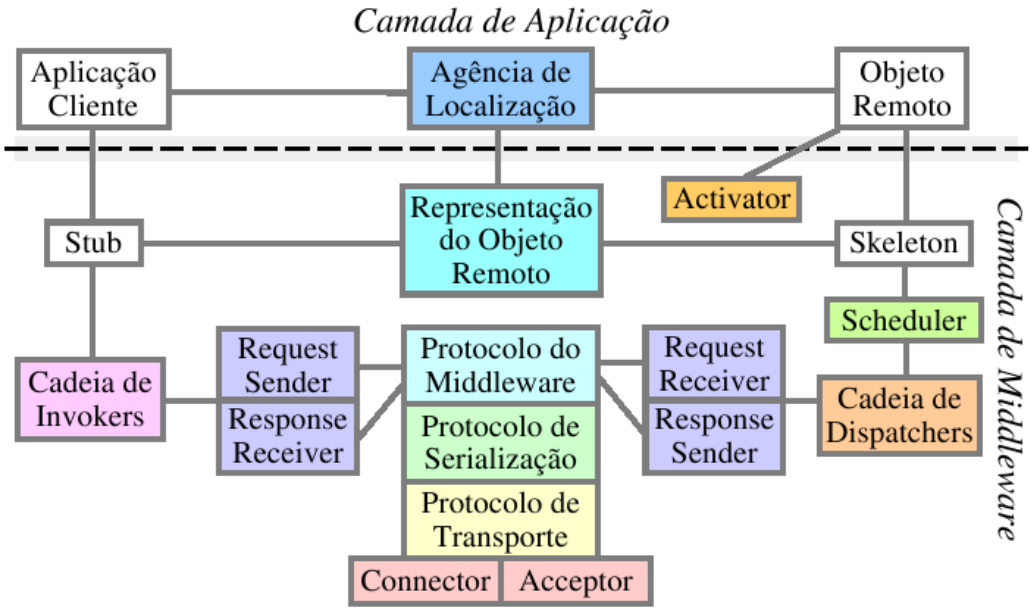


Figura 2.2: Principais componentes de Arcademis [Pereira, 2003, pg.34]

comunicação diversificados são passíveis de serem implementadas com Arcademis. A Figura 2.2 mostra seus principais componentes. Nem todos os componentes definidos precisam ser utilizados, mas existem dois que necessariamente precisam estar em qualquer instância de Arcademis, são eles o *stub* e o *skeleton.* O primeiro funciona como um *proxy* do objeto remoto, encaminhando as chamadas dos objetos cliente para o objeto remoto, transmitindo parâmetros e recebendo os resultados dos métodos invocados. O papel do *skeleton* é receber as chamadas remotas e repassá-las ao seu destino e enviar de volta o resultado. Os outros componentes, descritos na Figura 2.2, têm o papel de garantir a comunicação entre esse dois componentes. Nesta figura alguns componentes não representam apenas um único, mas sim um conjunto de componentes.

Existem 13 constituintes básicos, que podem ser definidos a partir de Arcademis, são eles:

- **transporte de dados:** esse parâmetro define os mecanismos utilizados para transmitir os dados. Diversos protocolos podem ser utilizados como TCP, UDP e HTTP por exemplo;

- **estabelecimento de conexões:** componentes desse grupo determinam como as conexões entre clientes e servidores são estabelecidas. O estabelecimento da conexão e seu processamento são separados utilizando o padrão *acceptor-connector*[Schmidt, 1997];

- **tratamento de eventos:** componentes desse grupo permitem definir a forma de reconhecimento e tratamento de eventos. O modelo de tratamento de eventos é baseado no padrão *Reactor* [Schmidt, 1995];

- **estratégia de serialização:** Arcademis define seus próprios componentes de serialização independente dos mecanismos existentes em Java. Por meio desses componentes é permitido especificar como os objetos serão convertidos em uma seqüência de *bytes.* Com isso podemos realizar migrações mesmo utilizando JavaME CLDC, que não permite a serialização de objetos;

- **protocolo do *middleware*:** componentes desses grupo permitem definir as mensagens que caracterizam as interações entre os objetos;

- **semântica de chamadas remotas:** é possível escolher a garantia de execução de uma chamada remota, como a semântica do melhor esforço, no máximo uma execução, pelo menos uma execução[Coulouris et al., 1996];

- **representação de objetos remotos:** componentes desse grupo permitem definir como os objetos remotos serão identificados. Um objeto pode ser identificado de diversas maneiras, por exemplo em RMI os objetos são identificados por uma URL;

- **serviços de localização de nomes:** estes serviços definem como os objetos podem ser encontrados em um sistema distribuído;

- **ativação de objetos remotos:** determina com um objeto é ativado, isto é, torna-se apto a receber chamadas remotas;

- **estratégia de invocação:** define como serão realizadas as invocações despachadas pela aplicação. Pode-se por exemplo reaproveitar conexões anteriores, adicionar funcionalidades extras como realizar *caching* das chamadas, uso de *buffers* e monitoramento do desempenho da aplicação;

- **despacho de requisições:** realiza a contra-parte da estratégia de invocação e define como serão entregues ao *skeleton.* Pode-se entregá-las diretamente ou inseri-las em filas de prioridade. Pode-se utilizá-lo para o monitoramento do servidor como também para verificar a autenticidade das mensagens recebidas. Também é possível definir a política de *threads* do servidor para a execução de cada chamada remota;

- **política de prioridades:** pode-se definir em que ordem serão processadas a chamadas enviadas a um objeto;

- **tipos de exceções:** permite definir quais as situações anormais que podem ocorrer no sistema distribuído.

Qualquer objeto que se deseje tornar remoto por meio de uma instância de Arcademis, deve implementar uma interface que define os métodos que podem acessados remotamente e esta interface deve estender a interface *arcademis.Remote.* O objeto deve estender a classe abstrata *RemoteObject* de Arcademis (normalmente deve estendê-la indiretamente utilizando-se uma classe da instância do arcabouço que implementa os métodos abstrados de *RemoteObject*). Essa classe adiciona ao objeto a capacidade de receber chamadas remotas. Além disso, duas classes, que estendem respectivamente *Skeleton* e *Stub,* devem ser geradas para cada classe acessada remotamente. Os objetos remotos devem se registrar em um serviço de diretório que permite que sejam pesquisados e que disponibilize os *stubs*.

Arcademis utiliza o padrão de projeto *abstract factory [Freeman et al., 2004]* que permite ao desenvolvedor alterar os principais componentes do *middleware* alterando apenas as fábricas desses componentes. Uma instância de Arcademis é configurada por meio do componente ORB, onde as fábricas dos diversos componentes podem ser registradas e acessadas. Antes de iniciar a execução de qualquer programa, o ORB deve ser configurado.

2.5 Conclusão

Apesar de, em termos de *hardware* e redes, já termos o suporte para implantar a computação *pervasiva*, falta o desenvolvimento de padrões, técnicas e ferramentas para desenvolvimento de softwares que explorem com totalidade todo o potencial dessas tecnologias.

A codificação da distribuição de maneira direta pelo desenvolvedor é um obstáculo para a disseminação de aplicações pervasivas. Primeiro, porque não permite o uso de aplicações já existentes, estas necessitam ser modificadas manualmente; segundo, passa a existir uma maior oportunidade para ocorrência de erros, além disso o programador tem que se preocupar em tratar todas as possíveis variações que podem ocorrer no ambiente de execução. O que não ocorria com uma distribuição automática. Em contrapartida, a distribuição automática não permite que sejam feitas otimizações e que a própria aplicação trate as possíveis falhas oriundas da distribuição. Além disso, nenhum dos trabalhos analisados que automatizam a distribuição permitiram uma exploração completa do ambiente *pervasivo*, levando em conta toda dinamicidade desse ambiente em termos de variabilidade de condições do ambiente.

Vimos diversas abordagens para a distribuição de aplicações, e nenhuma delas captura todas as características descritas acima.

Acreditamos que o melhor caminho é um misto entre as duas soluções. No Capítulo 4, apresentamos uma Ferramenta de Distribuição Automática e um Sistema de Suporte à Execução, que permitem, caso o desenvolvedor ache necessário, customizar a Ferramenta com códigos específicos de tratamento de erros, influenciar no particionamento e especificar objetivos principais de otimização, como desempenho, tolerância a falha ou menor consumo de energia. Também permite a utilização de algoritmos específicos tanto para os dispositivos, com características diferenciadas, como para cada aplicação. Os algoritmos podem ser selecionados em tempo de execução. Com isso, pretendemos disponibilizar um modelo flexível que se adapte às necessidades de um ambiente *pervasivo*.

Antes de descrevermos a Ferramenta de Distribuição Automática e o Sistema de suporte à Execução no Capítulo 4, descreveremos no próximo capítulo alguns algoritmos baseados em trabalhos descritos neste capítulo e criamos um novo modelo de distribuição.

Capítulo 3

Modelo de Distribuição

Uma aplicação pode ser particionada e distribuída entre outros dispositivos ao necessitar de mais recursos do que disponíveis no dispositivo em que se encontra. Existem muitas questões de gerenciamento neste processo e precisamos de uma estrutura de dados que represente a aplicação, sobre a qual os processos de decisão serão aplicados. Grafos são estruturas que podem representar entidades e seus relacionamentos, onde os vértices representam as entidades, e as arestas, os relacionamentos. Uma aplicação pode ser vista como um conjunto de partes que se interagem trocando informações entre si para atingir um objetivo. Em uma aplicação orientada por objetos, essas partes são formadas por objetos. Os objetos interagem por meio de chamadas de métodos e acessos a atributos. Assim podemos ver uma aplicação orientada por objetos como um grafo, onde os objetos são representados pelos vértices e suas interações representadas pelas arestas. Podemos utilizar também conjuntos disjuntos de objetos como vértices e a interação entre esses conjuntos (entre seus objetos) como as arestas. Esta estrutura de dados também é comumente utilizada para o particionamento de aplicações nos trabalhos descritos no Capítulo 2.

A distribuição de uma aplicação é feita pela migração de suas partes para outros dispositivos. Existem várias granularidades de migração:

- migração de classes [Gu et al., 2004, Ou et al., 2006]. Migram-se todos os objeto da mesma classe;
- migração de componentes [Zachariadis, 2005]. Migram-se módulos da aplicação;
- migração de agentes móveis [Soldatos et al., 2007, Spyrou et al., 2004]. São agentes de software que possuem a capacidade de se moverem de forma autônoma entre os dispositivos. Cada aplicação pode ser formada um agente ou por um conjunto de agentes;
- migração de processos [Smith, 1988]. Migra-se toda a aplicação;

Neste trabalho, aliada a representação em grafo descrita, propomos uma nova granularidade de migração, a granularidade de conjuntos de objetos. Esta unidade de migração complementa a lista acima, propriciando uma unidade intermediária entre classes e componentes. Todas as unidades citadas acima possuem diferentes nichos de aplicabilidade. Ao variarmos a granularidade de migração, estamos ajustando três objetivos: a qualidade da distribuição, isto é, como a aplicação será particionada entre os dispositivos; a quantidade mínima de recursos a ser migrado, o que permite ou limita os dispositivos que podem receber a migração; o custo computacional de se decidir por uma distribuição. Uma granularidade menor permite um espaço de soluções maior e, possivelmente, permite a migração para dispositivos com menos recursos, mas também pode aumentar o custo para se encontrar a melhor distribuição. Além disso, formas alternativas de migração utilizam mecanismos distintos de gerenciamento e decisão. Por exemplo, agentes móveis são unidades autônomas e responsáveis por sua migração, atuando de forma independente, em contrapartida a migrações por classes que normalmente dependem de um sistema de gerenciamento "externo"que tem uma visão global de todas as classes da aplicação, e realiza as tomadas de decisão.

Neste capítulo, vamos descrever três propostas com intuito de aprimorar trabalhos anteriores de particionamento de aplicações onde um grafo representa a aplicação. Essas otimizações foram aplicadas sobre a arquitetura que será descrita no Capítulo 4. A primeira é um algoritmo para criação de um grafo representativo de objetos para ser utilizado no particionamento de aplicações. Diferentemente dos trabalhos anteriores, a utilização do grafo é feita em tempo de execução da aplicação. Com isso, visamos otimizar a adaptação do particionamento durante a execução. A segunda proposta é a modificação de um algoritmo baseado no MinCut[Stoer e Wagner, 1997] proposto por Alan Messer et al. [Messer et al., 2002], estendendo-o para ser utilizado com múltiplos dispositivos. A terceira e última proposta é a utilização de uma decisão de particionamento distribuído, em vez de se utilizar um particionamento coordenado por um único dispositivo que contém o grafo, propomos distribuir essa tarefa entre os vários dispositivos, visando diminuir a comunicação e o tempo destinado ao gerenciamento.

Antes de iniciarmos a exposição dessas propostas, descreveremos na próxima seção algumas definições e funções utilizadas.

3.1 Definições

Nesta seção estão descritas algumas funções e notações relevantes para este capítulo.

Para se adaptar às mudanças no ambiente, algumas informações sobre os dispositivos e a comunicação entre eles são necessárias. Por isso definimos as seguintes funções:

- *d:* representa um dispositivo computacional capaz de se comunicar com outros dispositivos e compartilhar seus recursos;
- *A:* representa um conjunto de vértices, cada vértice por sua vez, corresponde a um conjunto de objetos que compõe o programa em execução;
- *Banda (d):* retorna a largura de banda de comunicação disponível em um dispositivo;
- *Mem (A)*: retorna o tamanho ocupado na mémoria em *bytes* pelos objetos representados pelos vértices do conjunto A;
- *Livres (d)*: retorna o número de bytes livres na memória do dispositivo;
- *Energia (d)*: quantifica o nível de energia em um dispositivo;
- *Latência (d)*: quantifica o tempo de latência de comunicação do dispositivo corrente ao um dispositivo d.
- *Processamento (d)*: quantifica em porcentagem a capacidade de processamento livre em um dispositivo.
- *CPU (d)*: inteiro que pontua o poder de processamento de um dispositivo d, pode-se utilizar a freqüência da CPU ou o número de instruções por segundo.

Ao se distribuir uma aplicação, precisamos de critérios para escolhermos quais dispositivos receberão os objetos da aplicação. Para isto, escolhemos mapear diversos critérios em um valor numérico, que denominamos de pontuação do dispositivo. Os dispositivos com as maiores pontuações recebem prioridade para serem os destinos das partes da aplicação. Para criarmos essa pontuação, definimos a função $Pontos$. Esta função é uma soma linear dos valores do seguintes recursos: memória, energia, latência, banda de comunicação e processamento. Esses recursos foram escolhidos por influenciarem no desempenho da aplicação, e influenciarem na possibilidade da aplicação ser particionada e migrada novamente. Na função, dividimos o valor atual de cada recurso pelo maior valor disponível, assim confinamos todos os valores no mesmo intervalo fechado $[0, 1]$. Ao igualarmos todos dentro de um mesmo intervalo de valor estamos atribuindo a mesma importância a cada parcela. Com o propósito de permitir que cada recurso receba uma importância diferente, cada parcela é multiplicada por um peso. O ajuste desses pesos permite, por exemplo, beneficiar dispositivos com alto índice de energia, para aplicações onde tolerância a falha seja mais importante ou com alto poder de processamento quando aplicações necessitam de respostas rápidas. A função $Pontos$ é definida matematicamente abaixo:

- $Pontos(d)$: Função que pontua um dispositivo segundo algumas características, como tamanho, capacidade de processamento, energia, latência e banda. O impacto de cada uma dessas medidas pode ser ajustado via pesos, podendo, por exemplo, escolher privilegiar o desempenho contra o consumo de energia:
  $Pontos(d) = Recursos(d) + Comunicacao(d)$, onde:
  $Recursos(d) = v_1 \frac{livres(d)}{livres_{max}} + v_2 \frac{Energia(d)}{Energia_{max}} + v_3 \frac{Processamento(d)}{Processamento_{max}}$.
  $Comunicacao(d) = v_4(1 - \frac{Latencia(d)}{Latencia_{max}}) + v_5 \frac{Banda(d)}{Banda_{max}}$
  Os coeficientes v_1,v_2, v_3, v_4e v_5 são os pesos.

Apresentaremoas agora algumas notações utilizadas neste capítulo:

- $G(N, E)$: representa um grafo, onde N é um conjunto de vértices, e E conjunto de pares (a, b), $a \in N$ e $b \in N$, que definem as arestas. As arestas e os vértices possuem conjuntos de valores (ou pesos) associados a eles. Cada vértice possui valores associados ao consumo de recursos da parte da aplicação que ele representa, e cada aresta um valor que representa a intensidade de interação entre cada parte.
- as letras A e B denotam conjuntos de vértices do grafo G;
- letra minúsculas, com exceção do d, denotam vértices ou arestas;
- qualquer função com subscrito max retorna o maior valor dentro do seu contra domínio. Em algumas funções que são usadas para comparação, como funções de pontuação e custo, o subscrito max denota o maior valor entre todos os elementos comparados;
- D representa uma lista de dispositivos:

  $D = \{(d_0, d_1..., d_{k-1}, d_k) \mid d_0$ é o dispositivo local, e $Pontos(d_i) > Pontos(d_{i+1})\}$,

  onde $1 \leq i \leq k$;

  - d_j dispositivo na posição j da lista;
- P_s e I são referências a conjuntos de vértices;

As funções descritas a seguir recuperam informações do grafo G:

- $f(a, b)$: retorna o peso da aresta $(a, b) \in E$, que representa a freqüência de chamadas entre a e b;

- $FreqTotal(A, B) = \sum_{a \in A, b \in B} f(a, b)$ onde $A \subset N, B \subset N$ e $(a, b) \in E$;

- $FreqTotalP(A) = \sum_{a \in A, b \in A} f(a, b)$ onde $A \subset N, B \subset N$ e $(a, b) \in E$; Esta função pode ser utilizada para estimar o custo de processamento de um conjunto de objetos. Supõe-se que objetos com uma maior freqüência de acessos entre seus métodos provavelmente farão um maior uso de processamento, isto é, estarão sendo executado com mais freqüência. Essa medição pode divergir do comportamento da aplicação, uma vez que métodos contendo laços e não contendo chamadas a outros métodos podem ocupar boa parte do processamento.

- $Energia(A, B)$: sendo A e B conjuntos de vértices em um mesmo dispositivo. Esta função faz uma aproximação do custo de energia adicionado aos dispositivos para tornar o conjunto B remoto em relação ao conjunto A, sendo A o conjunto que permanece no dispositivo. Na verdade, o que se pretende com esta função é poder comparar conjuntos a serem migrados. O objetivo é determinar qual conjunto terá, possivelmente, um menor impacto nos níveis de energia dos dispositivos ao ser migrado, e não mensurar de maneira precisa este valor. O custo energético adicionado é igual à energia gasta com a transmissão (envio/recepção) dos objetos, somada a energia gasta com a comunicação posterior entre os dois conjuntos, isto é, as chamadas remotas. Há também uma economia de energia em processamento no dispositivo local que é transferido para outro dispositivo, essa economia só existe realmente em termos globais se o dispositivo destino possuir uma fonte "inesgotável" de energia. Desconsideramos esta hipótese aqui, como também abstraímos os custos de se ligar e desligar as interfaces de rede para a transmissão. Como a energia consumida é proporcional ao número de *bytes* transmitidos, contabilizamos o custo de transmissão por meio do número *bytes* transferidos na migração dos objetos e posteriormente na comunicação remota. Como é difícil prever quanto tempo os objetos permanecerão remotamente em execução, vamos supor um valor cuja função seria atuar como um peso, isto é, para decidir qual critério (migração ou comunicação) será mais relevante. Definimos a função como: $Energia(A, B) = Mem(B) + FreqTotal(A, B) \cdot z \cdot e$, z é o número de *bytes* médios transmitidos em uma chamada remota obtidos experimentalmente, e e é o peso que define a importância da comunicação na fórmula;

- $Custo(A, B)$: sendo A e B conjuntos de vértices em um mesmo dispositivo. Esta função faz uma pontuação do custo geral adicionado aos dispositivos para tornar o conjunto B remoto em relação ao conjunto A, sendo A o conjunto que permanece no dispositivo. Assim quanto maior o custo adicionado menos interessante se torna um conjunto para ser mo-

vido. De maneira geral, conjuntos que se comuniquem muito (perda de desempenho), que na migração consumam muita energia, e que não façam bom uso da capacidade de processamento remoto (mover objetos mais ativos para um dispositivo de menor capacidade ou vice-versa), inserem um "custo" maior ao desempenho global da aplicação. Utilizaremos esses critérios no cálculo do custo. Definimos a função $custo$ como a soma dos custos individuais destas características: interação, energia e processamento. Na função, dividimos o valor atual de cada característica pelo maior valor disponível para cada uma, assim confinamos todos os valores no mesmo intervalo fechado $[0, 1]$. Ao igualarmos todos dentro de um mesmo intervalo, estamos atribuindo a mesma importância a cada parcela. Com o propósito de permitir que cada recurso receba uma importância diferente, cada parcela é multiplicada por um peso. O ajuste desses pesos permite escolher quais parcelas terão o impacto maior no custo. A última parcela pode possuir um valor negativo, toda vez que o dispositivo remoto possuir um poder de processamento maior que o dispositivo local, assim previlegiamos a migração de objetos com maior uso de processamento, para dispositivos de maior capacidade.
$Custo(A, B) = w_1 \frac{FreqTotal(A,B)}{FreqTotal_{max}} + w_2 \frac{Energia(A,B)}{Energia_{max}} + w_3 (\frac{CPU(d_0) - CPU(d_j)}{|CPU(d_0) - CPU(d_j)|}) \cdot \frac{FreqTotalP(A)}{FreqTotalP_{max}}$
onde w_1, w_2,w_3 são pesos que permitem escolher a importância de cada termo, d_0 o dispositivo local e d_j o dispositivo remoto;

- $NoMax(A, B)$: Retorna o vértice n em B cuja a soma do valor das arestas conectadas à vértices de A é a maior. Isto é:
$\forall g \in B$ e $n \in B$, onde $\sum_{t \in A} f(t, n) \geq \sum_{y \in A} f(y, g)$

- $Conjunto_x(d_i)$: Retorna um conjunto de vértices associados ao dispositivo d_i. Um dispositivo pode possuir mais de uma associação, para indicar uma associação diferente da principal, rotulamos com um identificador no lugar de x. Definimos o operador $=$ para realizar a adição de vértices ao conjunto. Se esse vértice já pertencer a outro conjunto associado a um dispositivo pela mesma função, ele o remove desse conjunto antes de adicioná-lo ao outro. Ex: $Conjunto(d_i) = \{a, b, d\}$. O identificador $Conjunto$ sem os parâmetros retorna a imagem da função.

3.2 Grafo Representativo de Objetos

Os trabalhos de Shumao Ou et al. e Gu et al., descritos nas Seções 2.2.5 e 2.2.2 respectivamente, utilizam-se de um grafo para particionar a aplicação. Nesse grafo cada vértice representa todos os objetos de uma mesma classe, isto é, quando um vértice é selecionado para uma partição,

então todos os objetos desta classe devem ser migrados para essa partição. Isso gera alguns inconvenientes, pois alguns objetos de uma mesma classe podem ser usados de maneira distinta por partes diferentes da aplicação e estariam mais bem posicionados em partições distintas. O problema em utilizar um grafo com a mesma granularidade dos objetos é o elevado número de objetos que uma aplicação pode gerar em apenas poucos segundos de execução, tornando a análise do grafo lenta, o que vai contra o requisito de que o particionamento seja feito em tempo real.

Para contornar esse problema em vez de se utilizar uma granularidade tão fina como a de objetos e tão grossa como a de classes, um grafo representativo dos objetos é criado. Esse grafo baseia-se no grafo proposto por André Spiegel, descrito na Seção 2.2.1. Por meio desse grafo, temos uma imagem simplificada da estrutura dos objetos da aplicação durante a execução, permitindo melhor distribuí-los. Além disso, o tamanho do grafo é fixo, independente do número de objetos. No trabalho de Spiegel, esse grafo é usado de maneira estática, isto é, define-se o particionamento antes da execução da aplicação. Como no ambiente *pervasivo*, o número de partições (dispositivos) e as características de cada uma são altamente dinâmicas, essa abordagem estática se mostra insuficiente. Para resolver este problema, modificamos então o grafo criado por Spiegel e o utilizamos para o particionamento dinâmico, isto é, durante a execução da aplicação.

Grafo estático

O grafo representativo é criado com base na análise estática dos *bytecodes* Java. São realizadas as seguintes etapas para a construção do grafo:

- Etapa 1. Encontrar os tipos que constituem o programa;
- Etapa 2. Construir um grafo de tipos;
- Etapa 3. Criar uma aproximação da população de objetos do programa;
- Etapa 4. Propagar referências no grafo de objetos;
- Etapa 5. Criar as arestas de uso no grafo de objetos;
- Etapa 6. Ajuste do grafo;
- Etapa 7. Redução do tamanho do grafo.

Agora discutiremos em detalhe cada etapa:

Etapa 1: encontrar o tipos que constituem o programa A partir do ponto de início de execução da aplicação (o método *main* da classe principal) encontram-se todos os tipos (Classes) que são referenciados a partir daí. Essa seleção permite eliminar métodos e classes não referenciadas pela aplicação e assim diminuir o consumo de memória. Quando utilizamos bibliotecas de terceiros e não necessitamos de todas as funcionalidades, é comum a existência de métodos e classes "inúteis". Para se ter um conjunto de classes mínimo, primeiro capturamos todas as classes referenciadas em qualquer ponto da aplicação, mesmo em métodos não utilizados, (isso é feito pesquisando-se as tabelas de constantes dos arquivos *.class* de Java que contém referências a todas as classes referenciadas no arquivo). Tendo determinado todas as classes, obtém-se o relacionamento de herança entre elas e então iniciamos uma segunda etapa de descoberta: A partir do método *main*, navega-se por cada função referenciada ou construtor, ainda não visitados no método analisado, assim recursivamente até que todos os métodos referenciados já tenham sido visitados. Como uma referência a um objeto de uma classe pode designar um objeto de uma subclasse, também navegamos pelos métodos das subclasses que sobrescrevem o método em análise. Durante a análise dos métodos, adicionamos as classes (e as subclasses dessas classes) neles referenciados como parte da aplicação e marcamos os métodos e atributos referenciados. Existem métodos que mesmo não sendo diretamente referenciados devem ser analisados e adicionados. Estes métodos não são diretamente chamados pela aplicação e sim pelo sistema (principalmente métodos que tratam interrupções). No fim, temos um conjunto de classes e métodos que são utilizados pela aplicação. É importante ressaltar que aplicações que utilizam reflexão computacional não podem ser analisadas dessa forma, pois a aplicação pode manipular objetos de qualquer tipo e acessar qualquer função (pública) por meio desse mecanismo.

Etapa 2: construir um grafo de tipos Esse grafo pode ser definido como G (V,E_u,E_e,E_i), onde V é um conjunto de vértices, cada vértice representa um tipo encontrado na etapa anterior. E_u é um conjunto de arestas (a,b,c) onde $a \in V$, $b \in V$ e $c \in \mathbb{N}$ tal que a faz pelo menos uma chamada a um método ou acesso a algum campo de b, c é um número que representa uma estimativa da interação entre as duas classes, que é calculada da seguinte forma: c inicia-se com valor zero e para cada chamada a um método ou acesso a um campo de b soma-se 1**,** se essa chamada se encontra em um if divide-se por 2 o valor atual de c e se estiver em um *loop* multiplica-se por 10. Esses valores foram definidos experimentalmente na tentativa de estimar a coesão entre duas classes, mas esses valores podem divergir na execução da aplicação.

E_e é um conjunto de arestas (a, b, d) , denominadas arestas de exportação, onde $a \in V$, $b \in V$ e $d \in V$ tal que:

- a chama um método de b passa pelo menos um parâmetro da classe d, ou

- a atribui uma referência a um campo de b, cujo o tipo é d, ou
- b é um *array* e a atribui referência do tipo b em d.

E_i é um conjunto de arestas (a, b, d),denominadas arestas de importação, onde $a \in V$, $b \in V$ e $d \in V$ tal que:

- a chama um método de b que possui d como tipo de retorno, ou
- a lê a um campo de b, cujo o tipo é d, ou
- b é um *array* e a lê uma referência do tipo b em d

Essas arestas capturam informações de uso e fluxo de dados entre as classes.

Etapa 3: criar uma aproximação da população de objetos do programa Existem dois tipos de vértices: estáticos e vértices comuns. Em uma ambiente distribuídos, referências a entidades estáticas devem manter-se consistentes por toda distribuição. Para isso, separamos a parte estática das classes em outra classe para criar um objeto em separado. Durante as análises descritas mais a frente qualquer referência a um campo estático é mapeada em um objeto **único** associado àquela classe. Para criarmos os vértices comuns, procedemos da seguinte forma. Primeiro, cria-se um vértice estático para a classe que contém o método *main*. Inicia-se a análise por esse método, a cada alocação de um objeto de um tipo nesse método adiciona-se um objeto a população de objetos. Adiciona-se também uma aresta de criação entre o vértice criador e o vértice criado. Cada aresta define um ponto de criação, sendo identificado pelo identificador e tipo do vértice criador e a instruções *new* que disparam a criação. Associa-se cada vértice ao tipo dos objetos associados ao vértice. Para cada vértice criado, avalie as alocações na classe que represente seu tipo e repita o processo, avalie apenas os métodos marcados como utilizados na Etapa 1. Observe que para alocações de objetos de um mesmo tipo a partir de um mesmo vértice criador, cria-se apenas um vértice. Dependendo do tamanho da aplicação pode-se optar por criar mais de um vértice para objetos de um mesmo tipo. Permitindo assim uma simplificação menor e uma representação mais real do que seria o grafo de objetos gerado pela aplicação. Alguns das outras opções avaliadas são: associar a alocação de objetos do mesmo tipo em um método (ao invés de ser em uma classe) ao mesmo vértice e associar a alocação de objetos a partir de uma mesma instrução *new* ao mesmo vértice.
Temos então a criação de grafo em formato de árvore. Observe que pode existir uma recursão na criação, criando-se objetos do mesmo tipo que algum dos objetos predecessores. Isso é eliminado pela criação de uma aresta de retorno para o objeto do mesmo tipo que existe no ramo ao invés de se criar outro vértice.

Criam-se também as arestas de referência, normalmente a criação implica em referência, a não ser por exemplo quando a alocação é feita como parâmetro de um construtor, nesse caso o objeto criado deve ser referenciado pelo vértice do objeto do construtor.

Cada vértice recebe um identificador único.

Etapa 4: propagar referências no grafo de objetos Baseado nos fluxos de dados do grafo de tipos propagam-se as arestas de referência. Isto é, se existe a aresta de fluxo de dados (a, b, c) no grafo de tipos e no grafo representativo de objetos existem as arestas de referência (v_a, e_b) e (v_a, f_c) [ou (e_b, f_c)], onde o subscrito indica o tipo associado ao vértice, então cria-se (e_a, f_c) [ou (v_a, f_c)].

Etapa 5: criar as arestas de uso no grafo de objetos Se um objeto *a* conhece um objeto *b* e existe uma indicação de uso no grafo de tipos entre os tipos dos objetos, adicione uma aresta de uso entre os dois.

Etapa 6: ajuste do grafo As arestas de uso deixam de ser direcionadas (ou adiciona-se uma aresta com mesmo peso no sentido inverso), pois do ponto de vista do particionamento o importante é a interação entre os objetos e não a sua direção. Se dois objetos possuem arestas de uso mútuo, a aresta passa a ser única e não direcionada e o peso passa a ser igual à soma dos pesos das duas arestas. Descartam-se as arestas de referência.

Etapa 7. Redução do tamanho do grafo Após geração do grafo alguns vértices podem ser unidos visando diminuir o tamanho do grafo com base para satisfazer uma restrição de um tamanho máximo μ do grafo. Pois no pior caso, N classes instanciam objetos das N classes gerando então um grafo com $1 + \sum_{i=1}^{N-1} \frac{(N-1)!}{(N-1-i)!}$ vértices (Lembrando que vértices não geram vértices de sua própria classe e de classes cujos objetos o precedem, mas geram apenas uma referência de retorno, com uma aresta de criação). Essa fórmula é válida quando um vértice cria apenas um vértice para cada tipo. Essa possibilidade de explosão na população de objetos cria a necessidade de reduzi-lo.

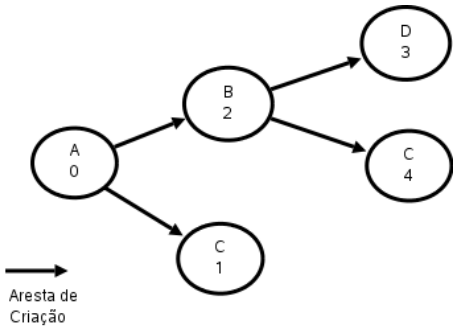
A redução do tamanho do grafo pode ser feita pela fusão de vértices. A estratégia de fusão utilizada é a seguinte: Encontra-se um conjunto de arestas de uso independentes (sem vértices em comum) escolhendo-se vértices aleatoriamente e adicionando a aresta de uso com maior peso ao conjunto, tal que não existam vértices em comum já adicionados. Os vértices pertencentes as arestas escolhida passam a ter o mesmo identificador (para cada par), indicando que representam o mesmo vértice. Para cada novo vértice que substitui os outros dois, atualizamos as arestas uso. Se dois vértices possuem um vizinho em comum a aresta resultante da união desses dois

deve possuir a soma do peso das arestas individuais. Se até o final das fusões ainda se não tenha atingido o tamanho determinado repete-se o processo na grafo gerado na iteração anterior.
Nesse trabalho, utilizamos apenas a fusão de vértices que não possuam tipos em comum. O identificador do vértice e o tipo do objeto criado são utilizados para definir qual será o vértice a ser associado ao novo objeto. Unido-se vértices com o mesmo tipo passamos a ter um único identificador o que reduz os mapeamentos possíveis. Assim temos que unir também vértices criados pelos vértices unidos, unindo assim dois ramos inteiros da árvore representada pelas arestas de criação. Se essa união for feita no início da árvore teremos uma redução muito grande no número de vértices. A utilização de outras formas de mapeamentos poderiam resolver esse problema, mas não são avaliadas pelo presente trabalho.

Temos agora um grafo de tamanho finito que captura melhor a organização dos objetos em tempo de execução do que o grafo de tipos. A partir desse grafo modificamos o código da aplicação para que seus objetos sejam associados a cada vértice corretamente no grafo dinâmico.

Grafo dinâmico

O grafo da fase dinâmica possui os mesmos vértices correspondentes ao do grafo gerado na fase estática, e possui apenas as arestas de uso e criação. A partir do grafo da fase estática é criado o código de associação entre objetos e os vértices do grafo da fase dinâmica. Por exemplo, na Figura 3.2 temos um grafo gerado no final da fase estática com as arestas de criação (outros tipos de arestas foram retirados para simplificar a figura). As letras representam os tipos e os números são o identificadores dos vértices. Objetos associados ao vértice 0 sempre criam objetos associados aos vértices 1 e 2, e objetos associados ao vértice 2 sempre criam objetos associados aos vértices 3 e 4.



Tanto os vértices como as arestas possuem valores associados a eles, os vértices são associados a uma tripla que representa o uso de memória, processador e largura de banda dos objetos

representados por eles. As arestas são associadas a valores que representam a freqüência de interação entre os objetos de cada vértice. Esses valores devem ser contabilizados durante a execução da aplicação.

Observe que um vértice somente será alocado na memória a partir da criação de um objeto que deve ser associado a ele.

Quando um objeto acessa um campo ou método de outro objeto cria-se a aresta de uso entre os vértices que os representa. O peso da aresta entre os vértice é inicializado com 0 e incrementado a cada acesso.

3.3 Algoritmo Multi-Dispositivos

O algoritmo aqui descrito é inspirado no algoritmo de distribuição de grafos utilizado por Xiaohui Gu et. al. [Messer et al., 2002, Gu et al., 2004]. Estendemos o algoritmo para permitir o particionamento entre vários dispositivos.

O particionamento é feito em função da disponibilidade de recursos no dispositivo, escolhendo quais objetos a migrar e para onde migrá-los. Cada dispositivo possui uma pontuação que determina sua prioridade em receber objetos. O dispositivo de onde serão movidos os objetos possui sempre a maior pontuação, isso porque queremos minimizar o número de migrações.

Em linhas gerais o algoritmo vai particionando o grafo e atribuindo uma partição a cada dispositivo. Tratamos como dispositivo atual aquele que será alvo de migração dos objetos em cada passo. São realizadas as seguintes etapas:

- Etapa 1. Geram-se vários particionamentos (em dois conjuntos) do grafo representativo de objetos. De maneira que pelo menos uma partição (conjunto) satisfaça as restrições de recursos do dispositivo atual. Se não for possível encontrar tal conjunto, descarta-se o dispositivo atual e seleciona-se outro dispositivo com maior pontuação disponível ainda não utilizado;

- Etapa 2. Escolhe-se, entre todos os pares de conjuntos formados, um conjunto satisfatório ao dispositivo atual, que minimize alguns critérios como comunicação, custo de envio do objetos, entre outros. Tenta-se associar o outro conjunto ao dispositivo com maior pontuação disponível ainda não utilizado; se não for possível associar um conjunto, volte a aplicar a etapa 1 sobre esse conjunto, utilizando o dispositivo selecionado como o atual;

- Etapa 3. Desconectam-se as partições, e as conecta a uma cópia (vazia) dos vértices dos outros conjuntos com quem faziam fronteira, esses vértices formam conjuntos que possuem a mesma associação a dispositivos que seus respectivos conjuntos de origem;

- Etapa 4. Cada dispositivo recebe sua partição do grafo e as associações entre dispositivos e vértices.

As etapas 1 e 2 estão relacionadas a associação dos vértices aos dispositivos. As etapas 3 e 4 fazem a interligação deste algoritmo com o proposto na próxima Seção, onde serão detalhados.

Agora, vamos detalhar as etapas 1 e 2 do algoritmo, que tem como entradas:

- grafo ponderado $G(N, E)$ gerado na fase estática. (Os valores das arestas são modificados em tempo de execução);
- lista D dos k dispositivos visíveis, ordenados decrescentemente por uma pontuação, onde d_i representa um dispositivo em D na i-ésima posição;
- conjuntos de vértices $Conjunto(d_i)$ associados aos dispositivos, onde $0 \leq i \leq k$, normalmente inicializados vazios;
- M_i nível desejado de ocupação máxima da memória para o dispositivo d_i.

A saída do algoritmo contém:

- grafo G particionado entre os dispositivos;
- conjuntos de vértices associados aos dispositivos atualizados.

Etapa 1: cria vários particionamentos do grafo, cada um separando em 2 conjuntos

//Inicialização
1. $i := 0$, k:= número de dispositivos visíveis,
$X = \{n | n$ é um vértice cujos objetos não podem ser movidos no momento$\}$
2. Se $i = 0$
    2.1 $I := Conjunto(d_i) \bigcup X$
    se não
    $I := Conjunto(d_i)$
3. $P_s := N - \bigcup_{j=0}^{k} Conjunto(d_j) - X$
4. $L := \emptyset$ //Conjunto de soluções

```
//Criação de pares de conjuntos candidatos, exemplificado
//na Figura 3.1.
5. Enquanto existem vértices em P_s
   5.1 Se I ≠ ∅, então
        adicione a I o vértice NoMax(I, P_s)
   5.2 Se I = ∅, então
        adicione a I um vértice qualquer de P_s, tal que,
        i = 0 ⇒ Mem(I) ≤ M_i
        ou se i ≠ 0 ⇒ Mem(I) ≤ livres(d_i) , e
        Mem(P_s) ≤ Σ_{j=i+1}^{k} livres(d_j)
   5.3 Se i = 0 ⇒ Mem(I) ≤ M_i ou
          se i ≠ 0 ⇒ Mem(I) ≤ livres(d_i), e
          Mem(P_s) ≤ Σ_{j=i+1}^{k} livres(d_j) então,
             insira (I, P_s) atual em L
   5.4 se não parar laço

6. Se L = ∅,
      P_s := I
      i := i + 1
   6.1 Se i ≤ k
       vá para 4.
   6.2 se não falhe.
```

Etapa 2: escolha do conjunto e associação do dispositivo

```
//Escolha do par de conjuntos que satisfaçam
//os requisitos do dispositivo e que minimize
//a comunicação entre os conjuntos,isto é
//menor peso total entre as arestas
7. Escolher em L, (A, B) tal que
   7.1 Custo (A, B) seja mínimo, e satisfaça a
       seguinte restrição:
       se i ≠ 0 ⇒ w_2 = 0 em Custo

   7.2 se não for possível encontra L que satisfaça as
       restrições tente o próximo dispositivo (i + 1)
```

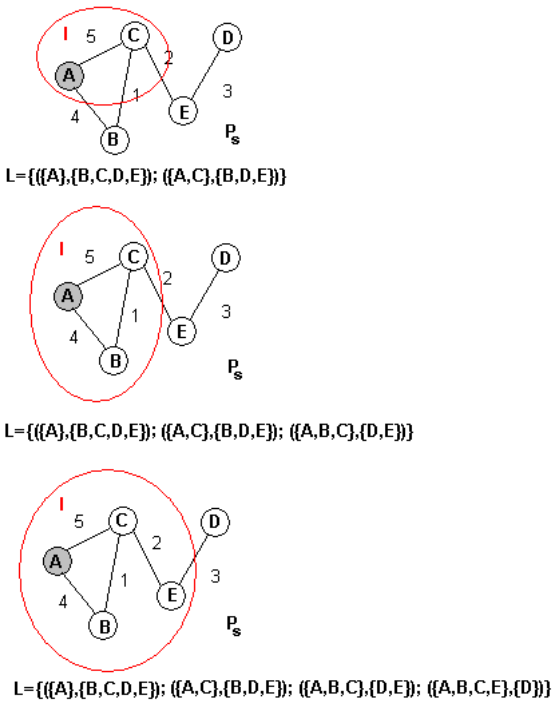


Figura 3.1: Exemplo da formação das soluções candidatas de particionamento.

se não houver mais dispositivos falhe

8. $Conjunto(d_i) := A$
9. $i := i + 1$
10. Se $livres(d_i) < Mem(B)$
    - 10.1 $I := Conjunto(d_i)$
    - 10.2 Utilize $N := B$
    - 10.3 vá para 3
11. Senão
    - 11.1 $Conjunto(d_i) := B$

3.4 Distribuição do Grafo

A utilização do grafo de maneira centralizada, isto é, totalmente contido em um único dispositivo, leva a alguns inconvenientes:

- necessidade de atualizar o grafo com dados de objetos que estão em outros dispositivos (freqüência de acesso, consumo de memória, processamento, dentre outros), o que acarreta um consumo de energia com a transmissão de dados. Normalmente as interfaces de comunicação possuem um elevado consumo de energia;

- um dispositivo que esteja com falta de recursos deve enviar essa informação para o dispositivo do usuário para que ele ajuste o particionamento;

- vértices que representam objetos de outros dispositivos permanecem ocupando recursos no dispositivo do usuário.

A fim evitar esses inconvenientes, propomos a distribuição do grafo. Com isso, cada dispositivo pode gerenciar seus próprios objetos. Cada dispositivo fica livre para contactar outros dispositivos caso também necessite de mais recursos e pode atualizar os valores do grafo diretamente. A técnica de distribuição aqui sugerida pode ser usada em conjunto com os algoritmos de particionamentos para o grafo centralizado com apenas pequenas modificações.

Existem três situações a serem tratadas:

1. O particionamento do grafo e seu envio para os demais dispositivos;

2. A atualização do grafo quando recebe mais vértices;

3. A devolução de parte do grafo quando requisitada (Recursos disponíveis no dispositivo de origem, início da migração do dispositivo para outro ambiente).

Para não perdemos as informações de interação entre os vértices locais e remotos, criamos o conceito de vértices fantasmas de fronteira, que são alguns vértices que ainda permanecem no dispositivo (como cópia) mesmo que seus objetos estejam em outro dispositivo. Esses vértices são os vértices visíveis por uma partição nas demais. Eles também são utilizados para tentar garantir que os grafos em cada dispositivo sejam conexos, pois permite indicar ao algoritmo de particionamento um conjunto de vértices que estão em determinado dispositivo e enviar a este apenas vértices conexos aos que já estão no outro dispositivo. Quanto menor a conectividade entre os vértices de um grafo em um dispositivo, maior será sua conectividade com o vértices em outros dispositivos (supondo o número total de conexões constante), o que pode levar a uma maior comunicação entre os dispositivos e com isso perda de desempenho e um maior consumo de energia. Se os vértices tiverem uma alto grau de conectividade, teremos um grande número de vértices fantasmas ocupando a memória, o que pode comprometer o terceiro objetivo listado anteriormente.

Durante a execução do algoritmo a associação entre vértices fantasmas e seus dispositivos podem ser corrompidas, pois os vértices que eles representam podem ser movidos para outros dispositivos. Pode se fazer com que em toda movimentação de vértices, sejam informadas a novas posições aos dispositivos com suas versões fantasmas. Essa solução pode falhar quando dois dispositivos realizam a migração em paralelo, pois a localização do vértice fantasma também pode migrar. Outra solução é utilizar um serviço de registro de localização, se o acesso a um objeto falhar por não se encontrar naquele local, requisita-se ao serviço de localização em qual dispositivo se encontra o objeto e atualiza-se também a localização do vértice a ele associado.

Denominamos o dispositivo de origem do grafo como dispositivo mestre e os que recebem o grafo como dispositivo escravo. Se um dispositivo escravo particiona seu grafo e o distribui, ele passa a ser denominado mestre em relação aos dispositivos que receberam seu grafo.

A seguir serão apresentados algoritmos para tratar cada uma das três situações descritas. Todas as três situações recebem como entrada:

- o grafo $G(N, E)$ local do dispositivo;
- os conjuntos de vértices associados aos dispositivos, que definem o particionamento da aplicação. Cada conjunto é definido pela função: $Conjunto(d_i)$ onde $0 \leqslant i \leqslant k$ e k é o número de dispositivos visíveis. Esses conjuntos e associações foram gerados por um algortimo de particionamento qualquer.

Situação 1

Particiona o grafo e prepara as informações que devem ser enviadas a cada dispositivo.

1. Para $k \geqslant u \geqslant 0$, onde k é o número de dispositivos visíveis
   - 1.1 Crie $G'_u(N', E')$
   - 1.2 $N' := Conjunto(d_u)$
   - 1.3 Se $N' = \emptyset$
     - 1.3.1 continue 1
   - 1.4 $E' = \{(e, f) \mid e \in Conjunto(d_u), f \in Conjunto(d_u)$ e $(e, f) \in E\}$
   - 1.5 para cada $Conjunto(d_x)$ tal que $\exists(a, b), \exists d_x$ onde $a \in Conjunto(d_x)$, $b \in Conjunto(d_u)$ e $(a, b) \in E$
     - 1.5.1 Crie $A' = \{a \mid a \in Conjunto(d_x), b \in Conjunto(d_u))$ e $(a, b) \in E\}$
     - 1.5.2 Crie $X' = \{(a, b) \mid a \in Conjunto(d_x)$

$b \in Conjunto(d_u)$ e $(a,b) \in E\}$

1.5.3 Crie a associação $A' = Conjunto'_u(d_x)$

1.5.4 $E' := E' \bigcup X'$

1.5.5 $N' := N' \bigcup A'$

1.6 Se $u = 0$

1.6.1 $G(N,E) := G'(N',E')$

1.6.2 Para $0 \leqslant j \leqslant k$, onde k é o número de dispositivos visíveis

1.6.2.1 $Conjunto(d_j) = Conjunto'_u(d_j)$

1.7 senão envie para o dispositivo **u**

1.7.1 $G'_u(N',E')$

1.7.2 Os conjuntos associados por $Conjunto'_u$

1.7.3 Os objetos associados a $Conjunto'_u(d_u)$

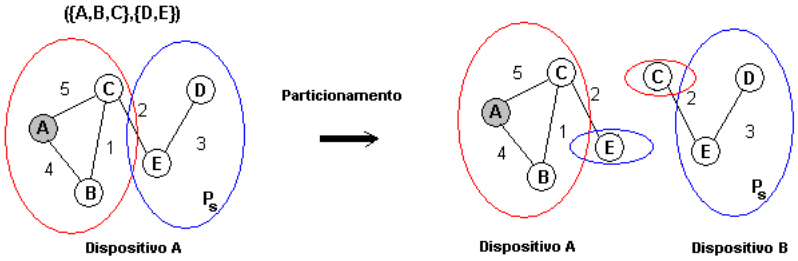


Figura 3.2: Exemplo de particionamento de grafo

Situação 2

Os próximos passos ocorrem no dispositivo d_z que está recebendo objetos migrados de um dispositivo d_w, e realiza a atualização dos vérices fantasmas:

1. Faça um merge entre o grafo recebido e o grafo já existente;
2. Atualize os conjuntos associados aos dispositivos. Para cada dispositivo d_i:

2.1 se $i \neq w$, então $Conjunto(d_i) := Conjunto(d_i) \bigcup (Conjunto'(d_i) - Conjunto)$,

2.2 se $i = w$, então $Conjunto(d_i) := Conjunto(d_i) \bigcup Conjunto'(d_i)$

Situação 3

Essa etapa é semelhante a etapa de separação no dispositivo, a única diferença é que não há a necessidade de se migrarem vértices para dispositivos além do dispositivo mestre. O algoritmo de particionamento pode apenas selecionar os vértices que devem ser devolvidos segundo os recursos disponibilizados. Após o envio dos vértices e objetos, no dispositivo mestre, aplica-se a etapa de atualização.

A seguir descrevemos uma adaptação do algoritmo descrito na seção anterior para realizar a seleção dos vértices a serem devolvidos.

O dispositivo escravo d_w recebe de d_m dispositivo mestre a informação que possui memória disponível.

1. $i := m$
2. $P_s := Conjunto(d_w)$
3. $I := Conjunto(d_i)$
4. $L := \emptyset$
5. execute os items 5 a 8 da seção anterior
6. $Conjunto(d_w) := B$

3.5 Conclusão

Com objetivo de avaliar novas possibilidades e formas mais adequadas de distribuição de objetos em ambientes pervasivos desenvolvemos três algoritmos que envolveram três aspectos da distribuição: A forma de distribuição, sugerindo uma organização da distribuição mais " natural" da aplicação no ambiente, utilizando como base da decisão da distribuição o relacionamento entre os objetos (aproximado pelo grafo representativo); a decisão da distribuição, dividindo a responsabilidade pela decisão de distribuição para todos os dispositivos a fim de torná-la mais eficiente e com menor custo de comunicação; e a seleção de objetos: criando um algoritmo de particionamento baseado no MINCUT, que suporta distribuir aplicativos entre vários dispositivos.

Essa modificações tiveram o objeto de diminuir o consumo de energia, uso de memória e melhorar a eficiência global da distribuição.

Capítulo 4

A Arquitetura de Proteus

Neste capítulo, propomos e descrevemos uma Ferramenta e um Sistema de Suporte à Execução para distribuição automática no ambiente *pervasivo*, que denominamos de Proteus. Por meio dele, baseado no modelo descrito no capítulo anterior, adicionamos a aplicações centralizadas a capacidade de se distribuir entre os dispositivos do ambiente e se adaptar às mudanças nesse ambiente. As aplicações passam a ter a capacidade de enviar suas partes a outros dispositivos, adaptando a forma da distribuição segundo a demanda por recursos computacionais e a disponibilidade destes recursos nos dispositivos.

Proteus permite que uma aplicação seja executada em dispositivos cujos recursos são insuficientes para a mesma.

O ambiente vislumbrado é mostrado na Figura 4.1, onde podemos ter diferentes dispositivos, como *smartphones,* PDAs, *laptop* e *desktops,* comunicando-se entre si, por enlaces sem fio ou cabeado. Esses dispositivos podem variar em capacidade e em recursos. Dispositivos móveis podem entrar e sair do ambiente e não há, a priori, a obrigação de existir algum servidor central. Estes dispositivos executam aplicações de seus usuários e permitem compartilhar seus recursos com os demais.

Em cada dispositivo, existe instalado o Sistema de Suporte à Execução (SSE), que deve ser iniciado como um serviço de sistema no dispositivo, e é responsável por iniciar a execução de aplicações locais, como também receber aplicações do ambiente. Para iniciar a execução de uma aplicação, o usuário fornece ao SSE o nome da classe principal, um conjunto de URLs determinando a localização das classes que podem estar ser localmente ou remotamente disponíveis, e seleciona objetivos de qualidade desejados como: desempenho, redução do consumo de energia e tolerância a falhas. O Sistema de Suporte à Execução de posse dessas informações inicia a execução e o monitoramento da aplicação.

O Sistema de Suporte à Execução gerencia a distribuição, mas não é responsável pela capaci-

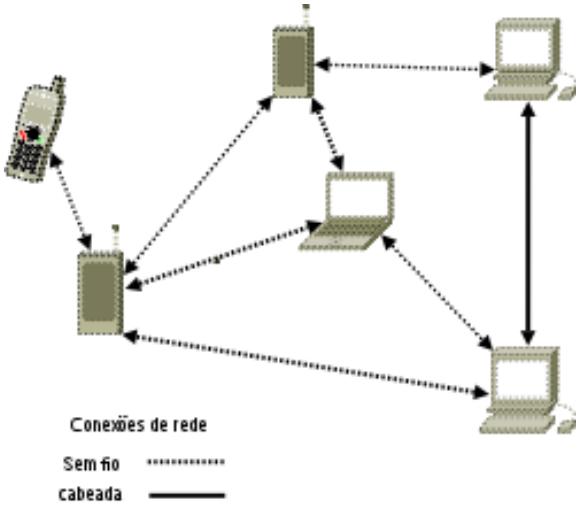


Figura 4.1: Ambiente de execução

dade de distribuir e nem por definir o que pode ser distribuído. A própria aplicação deve fornecer esta funcionalidade, pois ninguém melhor que a própria aplicação (ou melhor o desenvolvedor) "conhece" sua estrutura interna. Para permitir, de maneira simplificada, a inserção destas funcionalidades na aplicação, propomos o uso de uma ferramenta que instrumenta a aplicação (o código binário) antes de sua execução, inserindo todo o aparato necessário à distribuição. Essa abordagem facilita o reúso de aplicações centralizadas já existentes. Assim o próprio desenvolvedor pode disponibilizar sua aplicação com a capacidade de distribuição ou o usuário, por meio de uma ferramenta de instrumentação compatível com Proteus, pode modificar a aplicação.

A Ferramenta e o Sistema de Suporte à Execução aqui propostos têm como objetivo serem facilmente customizáveis para permitir a avaliação de diversas estratégias de distribuição baseadas em particionamento de grafo. As estruturas e relacionamentos entre as parte da aplicação são representadas por um grafo.

A versatilidade aqui almejada vai desde estratégias gerais de distribuição, isto é, a configuração da distribuição sobre os dispositivos, como no uso de técnicas particulares em cada dispositivo de acordo com recursos nele disponíveis.

Tanto a Ferramenta como o Sistema de Suporte à Execução foram projetados com o intuito de formar um arcabouço, isto é, um conjunto de classes cooperantes que constroem um projeto reutilizável. Isso possibilita que se modifique de maneira simples alguns comportamentos de Proteus, como os algoritmos de decisão, de particionamento, e seleção de dispositivos. Por exemplo, como particionamento ótimo de grafo é um problema NP-Completo[Karypis e Kumar, 1998b], uma instância do Proteus em um dispositivo com maior capacidade de processamento, pode uti-

lizar heurísticas mais elaboradas.

Nos referiremos a uma instância do Sistema de Suporte à Execução como *middleware,* por funcionar como uma camada independente entre a aplicação e o ambiente de execução.

A distribuição automática compreende duas decisões:

1. determinar a configuração do particionamento;
2. definir quando se alterar essa configuração.

Essas decisões devem ser tomadas em tempo de execução, pois dependem de informações sobre o programa e sobre o ambiente, que podem sofrer grandes alterações durante a execução, dadas as características dinâmicas do ambiente *pervasivo*. Informações sobre o ambiente, como número de dispositivos, disponibilidade de recursos e qualidade do sinal de comunicação, não podem ser previstas.

O comportamento de uma aplicação pode variar segundo sua finalidade, as configurações e as entradas do usuário no caso de aplicações deterministas. Aplicações não-deterministas podem apresentar variações em seu comportamento independentemente dos itens acima. Em aplicações deterministas, nas quais se conhecem sua finalidade e configurações, é mais fácil que algumas informações sejam coletadas antes de sua utilização pelos usuários. Essas informações podem ser coletadas via análise dos componentes da aplicação e/ou por meio da realização de execuções modelo, isto é, executa-se a aplicação com algumas entradas padrões e pressupõem-se que sempre terá um comportamento semelhante. Com isso, espera-se capturar informações aproximadas sobre quais são os componentes que se interagem, qual a intensidade dessa interação e qual o espaço ocupado pelo código e estruturas de dados. Neste trabalho, por enquanto não são utilizadas execuções modelo. Informações como uso de banda de comunicação, uso do processador e o consumo de memória somente podem ser avaliados precisamente durante a execução. É difícil capturar essas informações apenas pela análise do código e sem o conhecimento de informações sobre a finalidade da aplicação. Por isso a solução aqui proposta possui duas fases, uma realizada antes da execução e outra durante a execução, aqui denominadas fase estática e dinâmica respectivamente.

Na fase estática, encontra-se a base da decisão de particionamento da aplicação que é realizada via particionamento do grafo representativo dos objetos da aplicação, que tem como função ser a versão simplificada da estrutura de objetos em tempo de execução da aplicação. O uso de uma estrutura simplificada tem como objetivo reduzir o tempo de análise da aplicação, uma vez que milhares de objetos podem ser gerados em apenas alguns segundos de execução. Durante a execução, os objetos são associados aos vértices do grafo, e qualquer operação sobre um vértice é refletida sobre todos objetos associados a ele. O grafo e esta associação são definidos de maneira estática antes da execução da aplicação, o que permite que o desenvolvedor possa sugerir

a melhor maneira de se agrupar os objetos no grafo ou que ferramentas de prospecção analisem o código, e com base nas informações coletadas gerem o grafo. O grafo também pode ser alterado durante a execução caso observe-se a necessidade. Por exemplo, se o dispositivo possuir um baixo poder de processamento e o grafo determinado pela aplicação for demasiadamente grande, o dispositivo pode passar a associar novos objetos a vértices do grafo já existentes a fim de reduzir seu tamanho e o tempo de processamento. Na Seção 3.2 descrevemos uma solução para gerar o grafo baseada no algoritmo proposto por André Spiegel, descrito anteriormente na Seção 2.2.1.

A construção desse grafo é associada ao código da aplicação de maneira que grafos diferentes podem ser gerados especificamente para uma aplicação de maneira que a melhor represente.

A fase dinâmica utiliza o construtor de grafo gerado na fase estática para criar o grafo em tempo de execução e realizar o particionamento e a distribuição da aplicação. A fase dinâmica captura e atualiza durante a execução informações sobre a execução dos objetos (uso da memória, processamento, banda de comunicação e interação entre os objetos) sumarizadas em cada vértice. Essas informações são utilizadas para permitir uma distribuição que otimize o uso dos recursos do dispositivo.

Dada a popularidade da plataforma Java em aplicações de rede (incluindo a computação *pervasiva*), Java foi escolhida como plataforma alvo de implementação; contudo, os princípios discutidos aqui permanecem válidos para qualquer linguagem orientada por objetos. Além de sua grande popularidade, outras características levaram a escolha de Java como ambiente de desenvolvimento:

- independência de arquitetura, por ser interpretada, garante uma homogeneidade de execução em arquiteturas diferentes e favorece a migração de código e dados entre dispositivos diferentes;
- a necessidade de manipular diretamente o código binário e a existência de bibliotecas que facilitem essa manipulação;
- a existência de uma versão específica para dispositivos móveis e sua ampla utilização, atualmente, em bilhões de dispositivos[Microsystems, 2006a].

Durante o mapeamento do modelo da arquitetura aqui proposto para linguagem Java, encontramos algumas dificuldades que serão descritas no decorrer do texto, e também são citados processos atuais de desenvolvimento que visam remover tais dificuldades em um futuro próximo.

Na Figura 4.2 temos uma visão geral da arquitetura da solução composta do Instrumentador e do Sistema de Suporte à Execução;

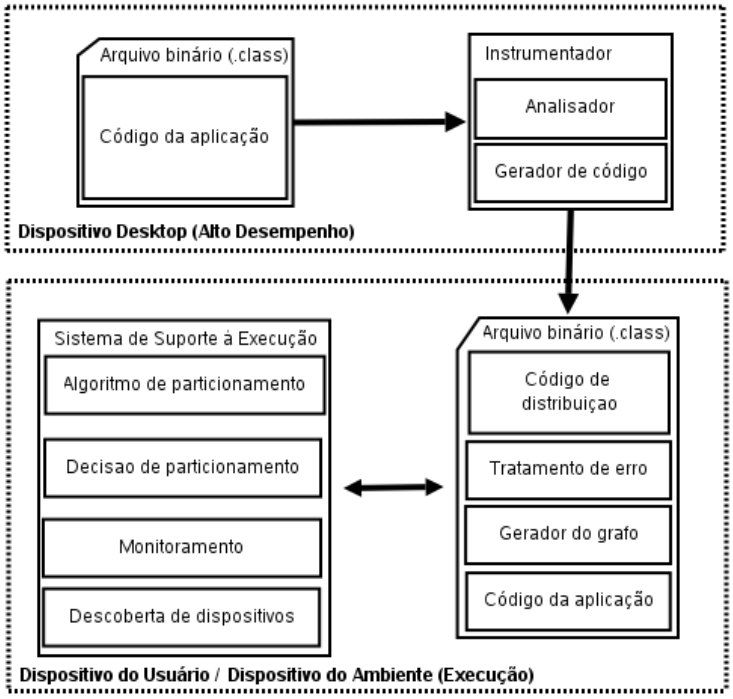


Figura 4.2: Instrumentador e Sistema de *suporte à Execução*

O Instrumentador analisa os *bytecodes* da aplicação, cria o grafo representativo de objetos definindo assim a granularidade de migração, e insere o código para geração do grafo, comunicação remota, interação com o Sistema de Suporte à Execução e instrumentação da aplicação.

O Sistema de Suporte à Execução tem a função de gerenciar a execução da aplicação, é responsável por capturar informações do ambiente e da aplicação, decidir como e quando particionar, descobrir os destinos possíveis e decidir para onde enviar cada objeto.

O Sistema de Suporte à Execução tenta otimizar suas decisões em função de três objetivos que podem tanto ser definidos pelo usuário como também sugeridos pela própria aplicação. Esses objetivos são um melhor desempenho, tolerância a falhas ou um menor consumo de energia.

Existem na instância do Arcabouço quatro níveis de modificações possíveis, tanto sobre o comportamento dinâmico da distribuição, quanto a características estruturais do Sistema de Suporte à Execução e do Instrumentador:

1. Nível de usuário: o usuário (uma pessoa ou um processo que requisita a execução de uma aplicação) pode escolher entre os objetivos de execução: melhor desempenho, maior tolerância a falhas e menor consumo de energia, como também definir novos objetivos.

2. Nível de aplicação: modificando-se o Instrumentador é possível alterar a geração do grafo

e modificar como será o agrupamento do que pode ser movido. Dessa maneira estruturas especializadas podem ser criadas para cada aplicação.

3. Nível de distribuição: é possível utilizar diferentes algoritmos de particionamento, sendo estes configurados em tempo de execução, isso permite que dispositivos diferentes usem algoritmos que trabalhem segundo suas capacidades e também que algoritmos específicos sejam executados segundo o objetivo de otimização. Também é possível definir diferentes algoritmos de decisão de particionamento e seleção de dispositivos destino.

4. Nível de sistema: permite ao desenvolvedor do *middleware* definir alguns parâmetros globais de comportamento, como estratégias de alocação de objetos, distribuição ou centralização da decisão de particionamento.

Para pemitir que as partes da aplicação se comuniquem e sejam movidas, como também para permitir que as instâncias do Sistema de Suporte à Execução troquem informações, e cooperem entre si, precisamos de um mecanimo de comunicação e conjunto de protocolos. A fim de permitir que a comunicação seja feita de maneira eficiente, e seja integrada ao Sistema de Suporte à Execução, utilizamos o arcabouço Arcademis [Pereira et al., 2006, Pereira, 2003], que possui uma grande versatilidade, o que permite sua adequação aos propósitos investigados nessa dissertação.

Arcademis é utilizado como um componente do Sistema de Suporte à Execução para formar um arcabouço maior. A partir deste Arcabouço, é possível a criação de diferentes *middlewares* para distribuição de objetos em sistemas pervasivos. Enquanto Arcademis fornece uma estrutura para definir as estratégias de comunicação, o SSE fornece a estrutura para definir as estratégias de distribuição e gerenciamento.

A Figura 4.3 mostra o posicionamento de Arcademis dentro da arquitetura. O Sistema de Suporte à Execução faz uso e interage com alguns componentes do *middleware* derivado de Arcademis, pois toda comunicação é feita por meio dele. Tudo isso executando sobre uma máquina virtual Java, que garante portabilidade de código e dados entre os diversos dispositivos, escondendo as características específicas de *hardware* e sistema operacional (SO).

A aplicação também deve utilizar componentes do *middleware* derivado de Arcademis para se comunicar e para permitir serialização e migração de suas partes. O SSE necessita reconhecer os objetos remotos e seus stubs para manipulá-los (gerenciar a distribuição), por isso é importante que eles estendam as classes de Arcademis que definem estas semânticas.

Como especificamos na Seção 2.4, Arcademis é configurado por uma entidade chamada ORB, onde são definidas as fábricas que determinam os componentes de comunicação. Tanto o Sistema de Suporte à Execução como a aplicação devem configurar essa entidade. Proteus

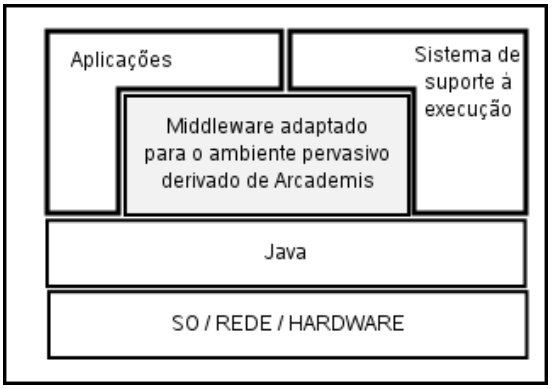


Figura 4.3: Arquitetura

garante que do ponto de vista da aplicação, como do SSE, cada um possua um ORB próprio. Essa separação é feita durante a carga da aplicação. Isto permite que os mecanismos de comunicação sejam configurados independentemente.

Mais a frente será descrito o Instrumentador, que tem a sua função de gerar o código de distribuição, e que fará uso intensivo da arquitetura provida por Arcademis.

4.1 Sistema de Suporte à Execução

O Sistema de Suporte à Execução (SSE) coordena a distribuição das aplicações, cooperando com Sistemas de Suporte à Execução em outros dispositivos no ambiente, e realiza particionamentos e migrações das aplicações. O SSE serve para separar uma aplicação de seu gerenciamento, permitindo que cada dispositivo trate à sua maneira, e dentro de suas capacidades do gerenciamento do particionamento, e simplificando o desenvolvimento das aplicações.

Cada dispositivo no ambiente deve possuir uma instância em execução do SSE, possivelmente iniciada ao se ligar cada dispositivo. O SSE deve gerenciar todas as aplicações distribuídas no dispositivo e dar todo o suporte à distribuição. São suas funções:

De maneira específica para uma aplicação:

- permitir iniciar/finalizar a aplicação;
- particioná-la;
- distribuí-la;
- monitorar uso do processamento, memória, interação entre seus componentes, uso de largura de banda.

De maneira geral:

- monitorar o ambiente;
- receber pedidos para receber uma aplicação;
- aceitar/negar reservar recursos para uma determinada aplicação;
- monitorar uso de recursos de maneira geral pelas aplicações.

A arquitetura proposta e seus principais componentes e relacionamentos podem ser visualizados de forma simplificada nas Figuras 4.4 e 4.5, são eles:

- *Objective:* essa classe define o objetivo ou a qualidade que se deseja na distribuição de uma aplicação, isso permite ao usuário ou à aplicação definir variações no comportamento da distribuição. São definidos inicialmente três objetivos: desempenho, tolerância a falhas e redução do consumo de energia. Essa classe pode ser estendida a fim de conter parâmetros específicos de cada algoritmo e para adicionar novos objetivos;
- *DevicesElection:* classes que implementam esta interface são responsáveis por, dada uma lista de dispositivos e seus recursos, ordená-los em ordem decrescente de qualidade. O conceito de qualidade do dispositivo varia segundo o objetivo de otimização. Por exemplo, se o objetivo for tolerância a falhas, dispositivos com altos níveis de energia e que tenham em seu histórico baixa perda de conectividade seriam considerados de melhor qualidade;
- *PartitioningAlgorithm:* classes que implementam esta interface são responsáveis por particionar o grafo que representa uma aplicação. Particiona o grafo e associa os vértices aos dispositivos destinos. Para isto, recebem do SSE o grafo representativo da aplicação, os dispositivos em ordem decrescente de qualidade e o objetivo de otimização para definir o comportamento do algoritmo;
- *PartitioningDecision:* classes que implementam esta interface são responsáveis por decidir quando realizar o particionamento, normalmente interagem com os serviços de monitoramento;
- *ObjectNodeAssociator:* classes que implementam essa interface são responsáveis por definir a estratégia da associação de objetos ao grafo representativo. Ela permite um controle do dispositivo sobre o tamanho do grafo e as associações entre objetos e vértices sugeridas pela aplicação;

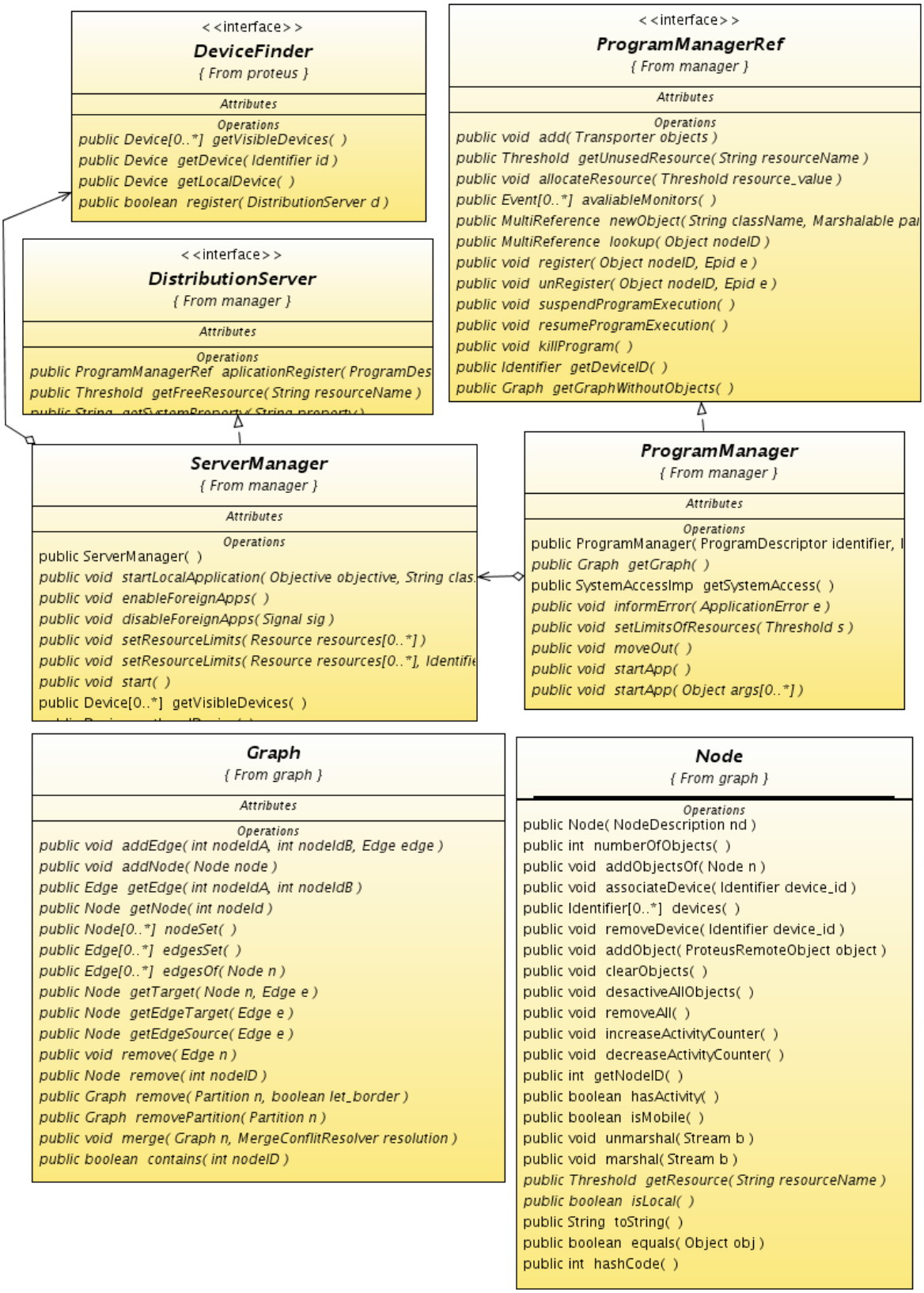


Figura 4.4: Diagrama de classe dos principais componentes do SSE (Parte 1)

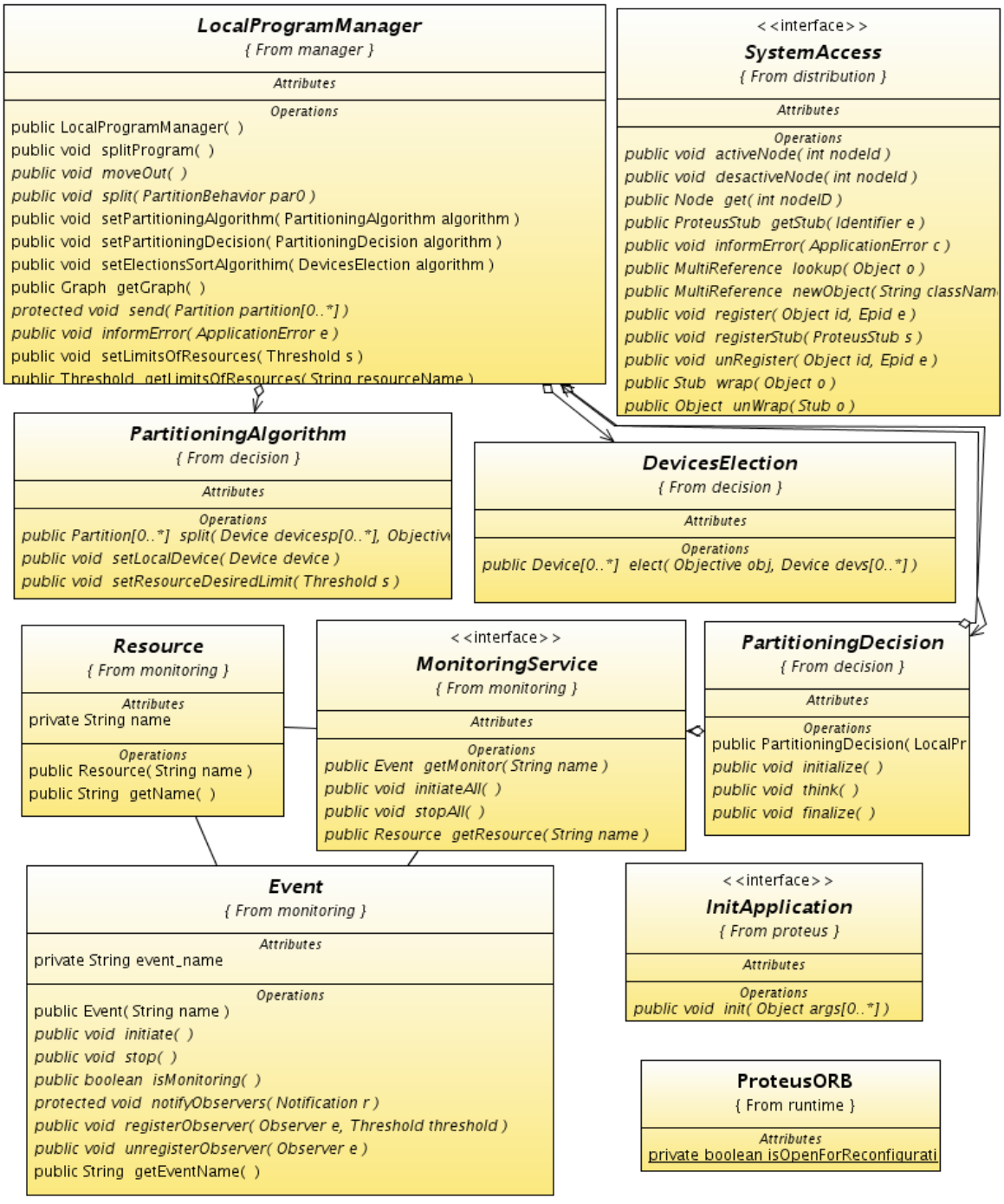


Figura 4.5: Diagrama de classe dos principais componentes do SSE (Parte 2)

- *ProgramManager*: são os gerenciadores da aplicação. Para cada aplicação distribuída em um dispositivo existe um *ProgramManager.* Eles agregam informações sobre os objetos locais e permitem realizar operações sobre eles. Algumas das operações básicas sobre a aplicação são a migração de objetos e sua finalização. Vários de seus métodos podem ser acessados remotamente. Os métodos acessíveis remotamente estão definidos pela interface *ProgramManagersRef*;

- *LocalProgramManager:* esta classe herda ProgramManager e a adiciona a capacidade de decisão de gerenciamento, mais detalhes serão descritos na Seção 4.1.1;

- *Resource:* representa um recurso (por exemplo, memória, processamento, áudio) e possibilita acesso as informações e atributos desses dispositivos, como o nível de recursos disponíveis;

- *Events*: são entidades responsáveis por monitorar mudanças nos dispositivos ou no ambiente e notificar mudanças nos entes monitorados. Entidades que desejam ser informadas devem se inscrever. É utilizado o padrão de projeto *Observer* [Freeman et al., 2004]. Nele todo observador deve implementar a interface *Observer* que possui o método *notify,* invocado caso haja alguma modificação no item observado. Todo monitor deve implementar a interface *Subject (*no nosso caso *Event),* que permite o registro de observadores interessados em serem notificados. A estrutura desse padrão pode ser observada na Figura 4.6. Alguns dos monitores já estão definidos no modelo, são eles: *MemoryEvent* que monitora os níveis de memória, *EnergyEvent* que monitora o nível de energia e *SignalEvent* que monitora a qualidade do sinal de comunicação. Ao registrar-se em um monitor, um observador pode passar um filtro que determina quando ele deve ser notificado. No caso da memória, por exemplo, pode-se determinar quais os níveis de memória ele deve notificar. *Events* e *Observers* são acessíveis remotamente, o que permite o monitoramento remoto de aplicações e dispositivos;

- *DistributionServer*: interface que disponibiliza acesso aos recursos de um dispositivo para os dispositivos do ambiente. O sistema de descoberta de serviços é responsável por encontrar e disponibilizar as referências a esse componente. O *DistributionServer* informa sobre as características do dispositivos e seus recursos disponíveis como também informações sobre o ambiente de execução, como qual a máquina virtual utilizada, versão do SSE, dentre outros. Também define métodos para receber requisições de recursos para uma aplicação de outro dispositivo. Se o pedido for aceito, ele reserva o recursos se possível, cria e retorna uma referência ao *ProgramManager* responsável por gerenciar a aplicação

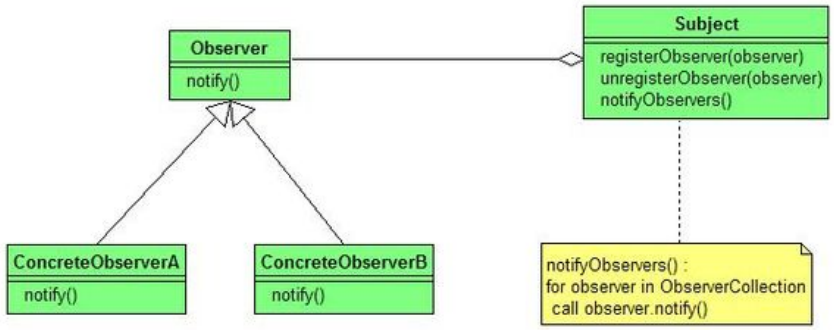


Figura 4.6: Padrão de projetos *Observer*

no dispositivo. Ao aceitar a requisição, o dispositivo também recebe um referência ao dispositivo que a originou. Nesse modelo, quem requisita recursos de um dispositivo externo tem acesso ao controle dos objetos a ele enviados. Assim podemos tanto distribuir o controle, permitindo que cada dispositivo gerencie, requisite recursos e interaja com dispositivos que cooperam diretamente com ele, ou isso pode ser feito por apenas por um único dispositivo;

- *ServerManager:* componente responsável pela carga de aplicações locais e pelo o gerenciamento dos recursos do dispositivo entre as aplicações em execução. Essa classe gerencia os recursos requisitados por meio do *DistributionServer;*

- *ProgramDescription*: contém informações sobre a aplicação como seu identificador e objetivos de otimização. É enviado ao dispositivo a que se deseja registrar. Pode ser estendido para conter outras informações, por exemplo, de onde buscar o código para carga, ou a qual dispositivo deve ser enviado informações sobre o estado da aplicação, ou ainda onde buscar pelo endereço de objetos. As informações nele armazenadas dependem da forma como o desenvolvedor implementa o modelo aqui proposto.

- *SystemAccess:* classe que fornece métodos para interação entre a aplicação e o sistema de suporte a execução, como a requisição para criação de objetos, acesso a *stubs* de objetos, acesso a informações do grafo, (des)ativação da mobilidade de um vértice. Em cada dispositivo, deve existir uma instância de SystemAccess para cada aplicação hospedada. Os objetos da aplicação devem sempre referenciar a instância do dispositivo onde estão localizados. Projetada originalmente com métodos estáticos para facilitar o acesso de todas as classes do programa às funções do sistema, esta opção só é viavel se o ambiente de execução permitir a separação do espaço de endereçamento de cada aplicação gerenciada

pelo mesmo SSE. A fim de impedir o acesso de aplicações diferentes a funções reservadas às demais aplicações, um objeto sem métodos estáticos pode ser utilizado no lugar e passado ao programa quando iniciado. O próprio programa se encarrega de garantir o acesso a esse objeto pelos objetos do próprio programa. Em Java, o carregador das classes (*ClassLoader*) pode realizar essa tarefa e disponibilizar a referência a instância correta da classe *SystemAccess* a cada aplicação;

- *InitApplication*: interface que deve ser implementada pela aplicação e seus métodos possuem a finalidade de configurar e iniciar a execução da aplicação. Um objeto que implementa esta interface é criado em cada dispositivo alvo e chamado para que configure os mecanismos de conunicação, registrando as fábricas dos componentes de Arcademis. Em uma instância de Arcademis, as fábricas dos componentes são acessadas por meio de métodos estáticos, não sendo assim serializadas juntamente com a aplicação e devem ser configuradas remotamente.

- *DistributedClassLoader:* esse componente é responsável por carregar as classes da aplicação e permitir o acesso a um objeto do tipo *SystemAccess* Local. A localização das classes deve ser informada pelas URLs *(Uniform Resource Locators*);

- *DeviceFinder:* interface que possibilita ao Arcabouço ter acesso a informações dos dispositivos disponíveis no ambiente. Como salientamos no início desta dissertação, não exploramos a descoberta dos dispositivos no ambiente, apenas supomos a existência de tal serviço, sendo ele acessado por esta interface;

- *Graph:* classe que permite o controle do grafo cujos vértices referenciam e sumarizam conjuntos de objetos da aplicação. Os vertices são utilizados nas decisões de particionamento e no controle da alocação e desaloção dos objetos;

- *ProteusORB*: Componente que registra as fábricas dos principais componentes do SSE, isso permite que os componentes sejam trocados de forma fácil apenas registrando-se uma nova fábrica;

- *Node:* classe que representa um vértice.

Esses são os principais componentes do SSE. No capítulo anterior foram descritas algumas propostas de modificações relacionadas com trabalhos anteriores. Essas modificações foram implementadas em uma instância da Ferramenta e do Sistema de Suporte à Execução aqui descritos.

A Figura 4.7 exemplifica parte do diagrama de colabaração entre os objetos de uma aplicação e alguns componentes do SSE na memória em dispositivo. O *ProgramManager* acessa o grafo

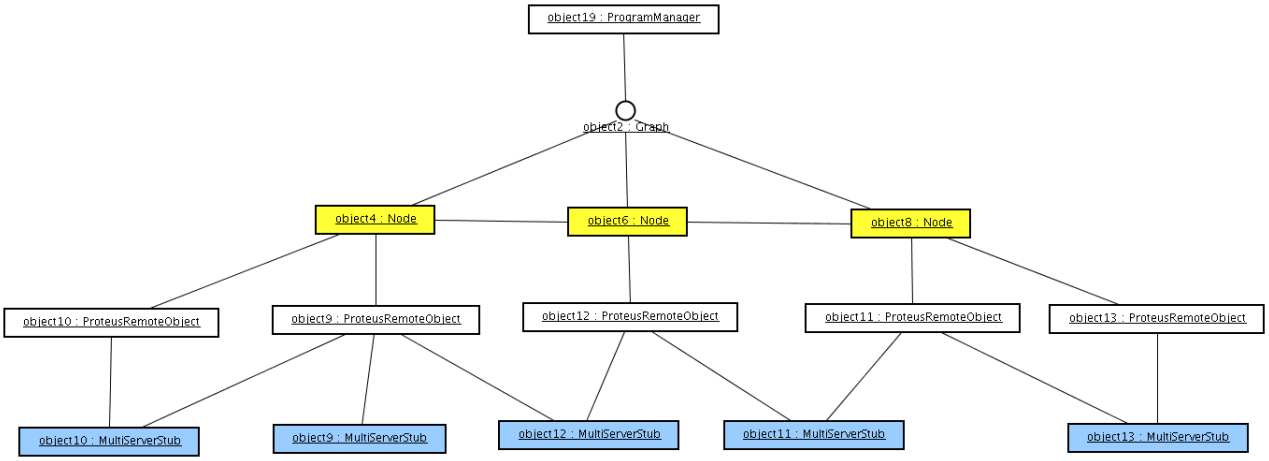


Figura 4.7: Diagrama de colaboração entre objetos de uma aplicação

(objeto da classe *Graph*) para aplicar suas operações de gerenciamento. O grafo por sua vez é formado por um conjunto de vértices (objetos da classe *Node*) que mantêm referências aos objetos da aplicação (representados por objetos da classe *ProteusRemoteObject*)). Os objetos da aplicação comunicam-se por meio de *proxies (Stubs).*

Nas Figuras 4.8,4.9,4.10, são ilustradas, de forma simplificada, as interações entre componentes do Sistema de Suporte à Execução.

A Figura 4.8 exibe a criação de um um novo objeto por um objeto da aplicação. Na verdade, a aplicação cria o *stub* que requisita a criação do objeto ao SSE, por meio do objeto *SystemAccess* local. O Sistema de Suporte à Execução consulta ao NodeAssociator e descobre a qual vértice do grafo ele deve ser associar o objeto. Se o vértice se localizar localmente, ele cria o objeto e o associa ao vértice, senão ele localiza o dispositivo onde está o vértice e requisita a criação do objeto. Em uma implementação com tolerância a falhas por replicação, o ProgramManager local propagaria a requisição de criação para todos os dispositivos que devem conter cópias do objeto. No final os endereços (se houver cópias) dos objetos são retornados ao Stub.

Na Figura 4.9, exemplificamos o processo de particionamento da aplicação quando a memória torna-se escassa. O processo de particionamento normalmente inicia-se quando um monitor, neste caso o *MemoryEvent* que monitora a memória, gera um evento notificando a um objeto da classe *PartitioningDecision* alguma alteração no recurso monitorado. *PartitioningDecision* é reponsável por determinar quando realizar o particionamento, se julgar necessário ele chama o método *split* do *ProgramManager,* iniciando assim o processo de particionamento. Uma vez notificado o *ProgramManager* obtém informações sobre os dispositivos disponíveis no ambiente e as delega ao *DeviceElection* para pontuar os dispositivos e definir a preferência de envio

dos objetos a eles. O grafo representativo de objeto é particionado pelo *PartitioningAlgorithm*, que associa os objetos aos dispositivos. Após negociar com os dispositivos os recursos necessários, os objetos da aplicação são serializados e enviados como parâmetros pelo método *add* dos *ProgramaManagers* remotos.

A inicialização de uma aplicação é demonstrada na Figura 4.10. O usuário requisita ao gerenciador de aplicações (*ServerManager*) a criação de uma aplicação. Primeiramente, ele cria o *ProgramaManager* associado a aplicação e determina quanto de recursos serão destinados à aplicação e depois ele requisita ao *ProgramaManager* iniciar a execução do programa. O *ProgramManager* localiza o código da aplicação e cria o objeto *InitApplication* que contém o método *configure*, chamado para incializar os componentes do Arcademis utilizados pelo programa, e o método *InitApplication*, que dá início a execução da aplicação.

Nas próximas seções detalharemos as funcionalidades principais do SSE. Outras classes e interfaces importantes também serão detalhadas.

4.1.1 Gerenciamento da Aplicação

O gerenciamento de uma aplicação é realizado pelo *ProgramManager* no SSE. Sua interface define funções para:

- sumarizar as informações de monitoramento;
- receber e enviar objetos do dispositivo;
- servir de interface para a requisição de mais recursos no dispositivo;
- realizar funções de finalização ou suspensão da execução da aplicação;
- permitir o registro de observadores externos sobre o estado da aplicação local;
- permitir interação com outros *ProgramManagers* de uma mesma aplicação. A interface *ProgramManager* herda a interface *Remote* de Arcademis, que indica que seus métodos devem ser visíveis remotamente. No entanto, não é definido *a priori* como deve ser a interação entre eles. O controle sobre o particionamento pode ser realizado tanto remotamente por uma unidade centralizada ou de maneira distribuída. O particionamento distribuído do grafo pode ser visto em detalhes na Seção 3.4.

A classe abstrata *LocalProgramManager* (ver figura 4.4) implementa a interface *ProgramManager*. Além das funções básicas herdadas de *ProgramManager,* define as funções de controle sobre o particionamento e integra os componentes de monitoramento, particionamento e lógica

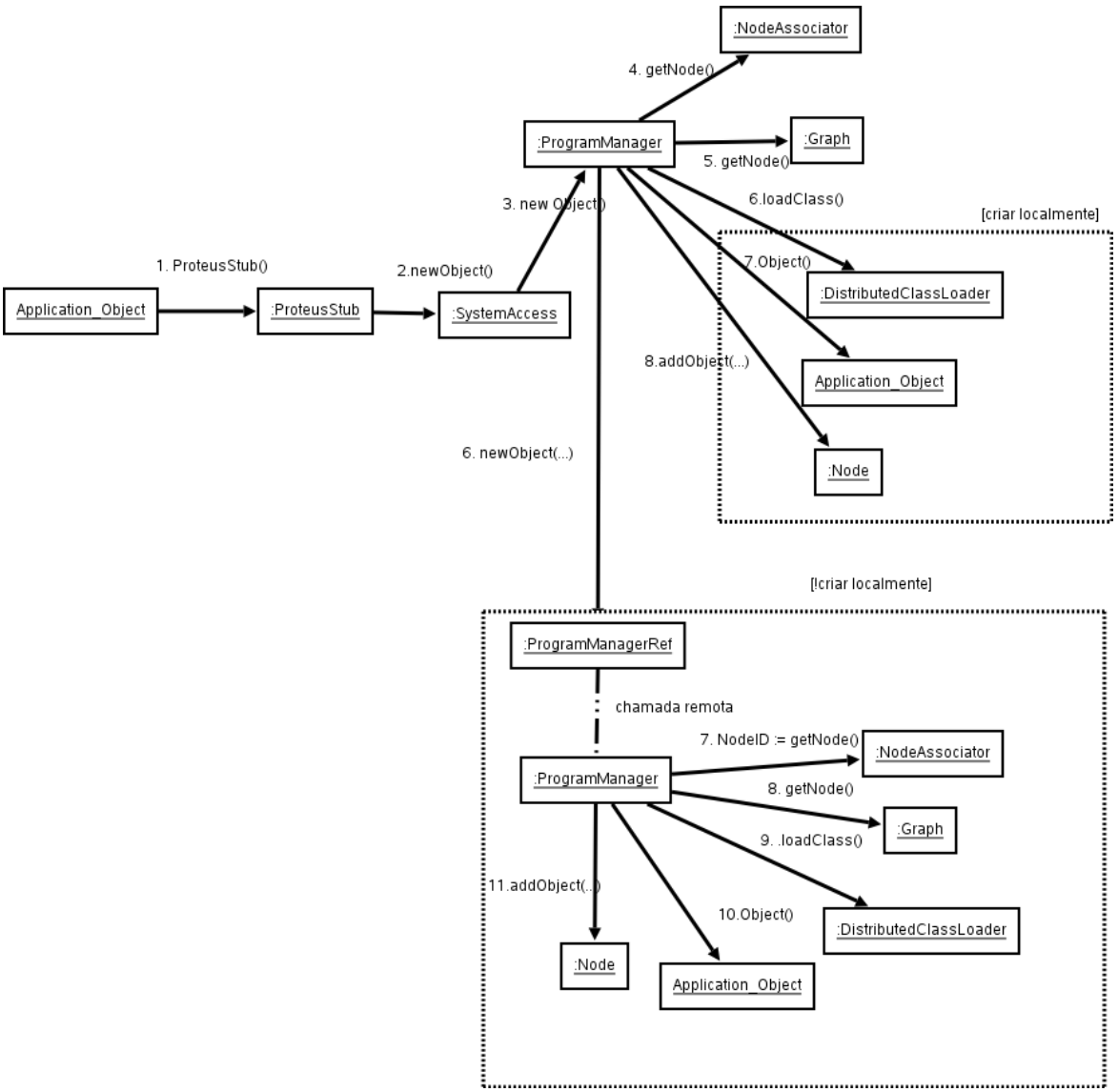


Figura 4.8: Colaboração entre objetos da aplicação e do SSE pra criação de um novo objeto

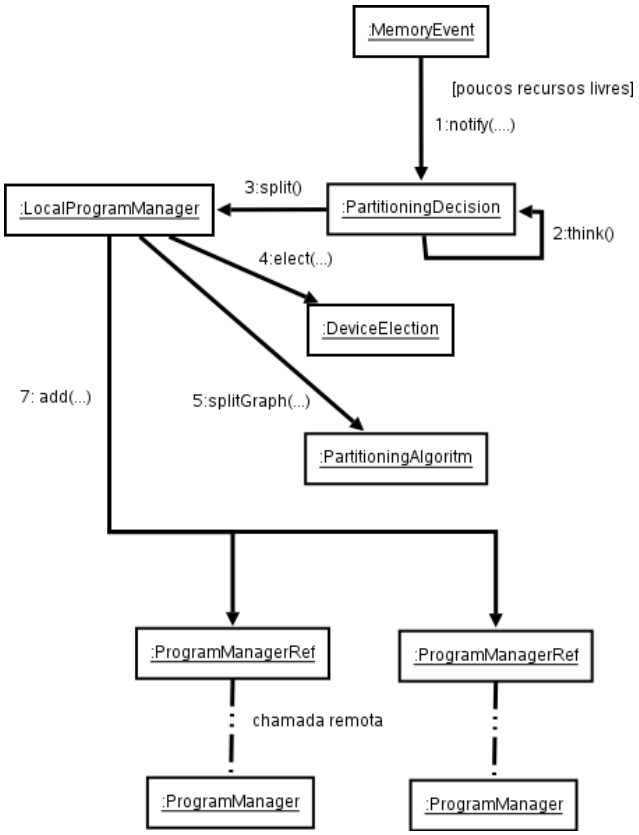


Figura 4.9: Colaboração entre os objetos do SEE para oparticionamento da aplicação devido ao baixo nível de memória

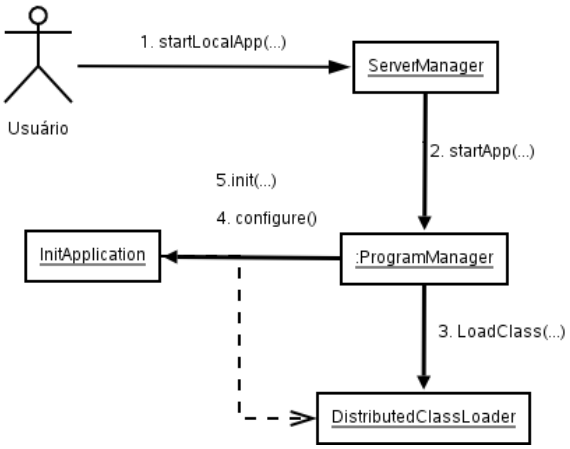


Figura 4.10: Colaboração entre os objetos do SSE pra inicialização de uma aplicação

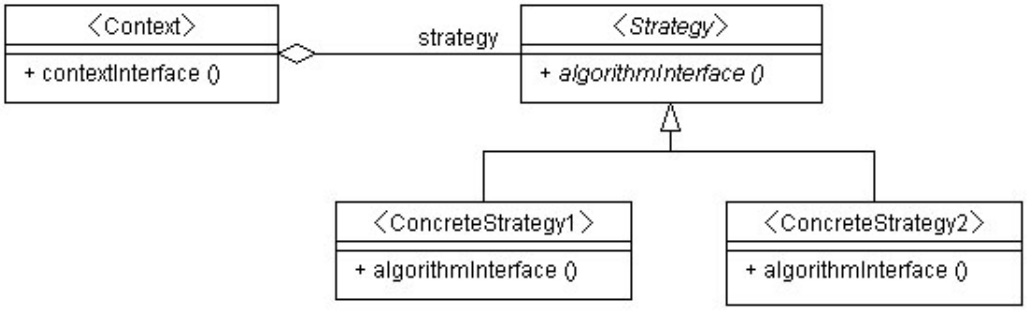


Figura 4.11: Padrão de projetos *Strategy*

de decisão. A interface *ProgramManager* apenas define quais operações podem ser operadas sobre os objetos contidos no dispositivo. A classe *LocalProgramManager* insere métodos que tomam decisões sobre a aplicação. Os métodos *set* permitem ajustar dinamicamente quais os algoritmos que tratam do particionamento, da decisão de distribuição e da criação do grafo. Isso permite que cada dispositivo utilize algoritmos diferenciados, de maneira que possam adequar a precisão e qualidade dos algoritmos segundo a capacidade dos dispositivos. Para isso utilizamos o padrão de projeto *Strategy*[Freeman et al., 2004]. A estrutura desse padrão pode ser vista na Figura 4.11. No SSE o *context* é representado pelo *LocalProgramManager* e se relaciona com três interfaces do tipo *Strategy,* são elas: *PartitioningAlgorithm, DevicesElection, PartitionAlgorithm e ObjectNodeAssociator.* O método *split*, que é responsável por gerenciar o processo de particionamento, pode requisitar informações sobre os dispositivos e partes da aplicação que estão no ambiente, alocar recursos externamente, manipular o grafo e aplicar os algoritmos de particionamento e seleção de dispositivos. Esse método define a interação entre os dispositivos, define se o particionamento se dará de forma distribuída ou centralizada, aplicando suas decisões sobre os demais dispositivos. No fim do processo deve enviar os vértices do grafo e os objetos a seus destinos por meio do método *send,* ou requisitar movimentações nos demais dispositivos. Outro método importante é o *notify*, que recebe informações sobre mudanças no ambiente e no dispositivo e convoca o algoritmo de decisão de particionamento, que deve decidir ou não, se realiza um novo particionamento.

O gerenciamento de uma aplicação também deve realizar-se sobre o controle de uso de seus recursos. Na próxima seção serão discutidas questões sobre o gerenciamento de recursos.

4.1.2 Gerenciamento de Recursos

Para que a distribuição de aplicações no ambiente seja satisfatória, é necessário o estabelecimento de políticas de qualidade de serviço e políticas de segurança, as quais garantem que, uma

vez que um dispositivo se ofereça para compartilhar seus recursos a uma aplicação, outras aplicações, que porventura venham a utilizar os recursos, não o façam de maneira prejudicial, ou até mesmo danosa às demais aplicações. Uma aplicação, por exemplo, poderia alocar indefinidamente memória até consumi-la por completo, podendo assim ocasionar a finalização de outras aplicações. Para garantir que isso não ocorra, deve ser possível distribuir entre as aplicações quotas máximas de utilização dos recursos, que devem ser impostas ao se permitir a migração da aplicação para o dispositivo, e que eventualmente poderiam ser renegociadas. Também seria interessante a utilização de prioridades na obtenção de recursos entre as aplicações. O proprietário de um dispositivo provavelmente não ficaria satisfeito se suas aplicações fossem prejudicadas, ou impedidas de executar por compartilhar recursos com aplicações externas.Tudo isso gera uma demanda para a realização de um gerenciamento de recursos.

A arquitetura do SSE define métodos em nível de aplicação, e em nível de sistema para o controle dos recursos. Abordaremos agora as dificuldades de se implementar tais funcionalidades na plataforma Java.

Tradicionalmente, a plataforma Java funciona em um único processo do sistema, isto é, mesmo que se execute diferentes aplicações a partir de uma mesma JVM, do ponto de vista da plataforma temos apenas uma aplicação. A única facilidade da gerência de recursos fornecida pela máquina virtual é o coletor de lixo. O coletor de lixo se responsabiliza por gerenciar a memória de todos objetos em uma mesma JVM, independentemente a que aplicação eles pertençam.

Java, particularmente JavaME, não possui mecanismos nativos e expostos ao programador para permitir o gerenciamento de recursos como memória, uso da CPU e consumo de energia, uso de *sockets* e acesso aos arquivos. A única informação disponível é o total de memória e uma aproximação da memória disponível. JavaSE, na sua versão 5.0, incluiu um pacote *java.lang.management* que oferece informações sobre o sistema operacional, tempo de CPUs de *threads,* e monitoramento do uso da memória. Essas informações são obtidas de maneira geral, sem nenhuma diferenciação explícita entre as aplicações em execução, além disso, não existem maneiras de se especificar limites para o uso dos recursos, bem como forçar sua liberação caso necessário. O que torna difícil por parte do sistema de suporte a execução criar restrições e disponibilizar com exatidão recursos aos diversos processos. Existe na comunidade Java atualmente processos para a especificação de APIs para o gerenciamento de recursos em Java, são eles os processos JSR 278[Process, 2006b] para JavaME e JSR 284[Process, 2006c] para Java SE e Java EE.

A fim de minimizar essa limitação, é possível instrumentar a aplicação, inserindo o código necessário para capturar as informações para o gerenciamento. Essa não é uma solução completa, pois códigos nativos são difíceis de serem instrumentados, além disso essa instrumentação

deve ser feita em tempo de carga da aplicação, pois aplicações mal-intencionadas poderiam não fornecer a instrumentação correta. A instrumentação em tempo de carga nos Ambientes Java não são amplamente permitidas, uma vez que JavaME CLDC não permite a utilização de carregadores (*loaders*) customizados.

Descrevemos agora algumas estratégias e dificuldades para se obter em Java informações sobre os seguintes recursos: uso da memória, processamento e banda de comunicação.

Para se obter o consumo de memória de uma aplicação, devemos inspecionar seus objetos, a fim de obter o espaço utilizado. Se a aplicação se encontra sozinha em uma máquina virtual, em vez disso, podemos utilizar as funções que retornam o total de memória livre, e a partir disso calcular o consumo. A segunda solução no caso de múltiplas aplicações pode ser alcançada lançando-se para cada aplicação outro processo totalmente independente, isto é outra JVM. Essa solução é mais simples, mas não pode ser utilizada em J2ME CLDC, que não permite que processos externos sejam lançados. Além disso, nas outras plataformas, é uma solução dependente do ambiente que está sendo executado, pois a plataforma requisita a criação de um novo processo ao sistema operacional, e é necessário passar o comando que inicializa a nova JVM. Alguns dispositivos podem também não permitir multi-processos. Outra desvantagem é que o lançamento de uma nova JVM consome mais recursos que o uso de uma mesma compartilhada, pois é necessário a carga das estruturas de dados e bibliotecas básicas novamente. Java SE 5.0 minimiza isso compartilhando algumas biblitotecas dinamicamente, se possível. O processo JSR 121[Process, 2006a], denominado *Application Isolation API,* já finalizado e aprovado, permite que aplicações distintas executem em uma mesma JVM, comportando-se com se estivesse em processos separados, cada um com seu próprio espaço de endereçamento. Entretanto, essa API ainda não está disponível nas implementações da JVM comumente utilizadas.

É necessário inspecionar os objetos ao se utilizar múltiplas aplicações em uma mesma JVM, cada objeto deve informar seu tamanho para que se possa calcular individualmente o tamanho ocupado por cada aplicação. Java não possui um operador equivalente ao *sizeof* de C/C++ que informa o tamanho de um objeto. O consumo de memória de cada objeto é formado pela soma do espaço ocupado pelos objetos, tipos primitivos e referências a outros objetos contidos nesse objeto. Java, diferentemente de linguagens como C++, não possui outros objetos contidos, mas apenas referências a objetos. Um *array* também é tratado como um objeto em Java, assim com exceção deste tipo de objetos, o tamanho ocupado pelos demais é totalmente previsível a *priori*.

Consideramos apenas os atributos da classe, que são armazenados no *Heap*, e não variáveis existentes no escopo dos métodos, pois essas variáveis (tipos primitivos ou referências a objetos) são alocadas na pilha de execução.

Pode-se utilizar reflexão computacional para examinar os atributos da classe e calcular o tamanho do objeto. Note que apesar de, para o programador, os tipos primitivos e referências

Java possuírem um tamanho definido independente de plataforma, o seu mapeamento é feito transparentemente pela JVM em estruturas locais, e que podem ocupar um espaço maior. Em JavaME CLDC, os recursos de reflexão computacional não estão disponíveis, as classes devem ser instrumentadas antes da execução e inseridos métodos que calculam e retornam o tamanho do objeto.

Para se utilizar reflexão computacional de uma maneira mais precisa é necessário que todos os objetos sejam referenciados pelo componente de análise, pois não é possível realizar um caminhamento a partir de um objeto, já que não são acessíveis o conteúdo de atributos privados do objeto. Atualmente, esta é a forma implementada do Arcabouço. Pode-se também analisar a classe de objeto privado e estimar o seu tamanho. A questão é: em qual profundidade deve-se realizar essa análise, uma vez que o atributo do objeto pode não estar inicializado ou referenciando outro objeto já contabilizado.

Para se garantir um limite na utilização do uso da memória pode-se, por meio da instrumentação, alterar todos os comandos de alocação para que antes de alocar requisitem autorização ao sistema de gerenciamento. E caso não haja recursos o sistema pode realizar a alocação em outro dispositivo ou liberar recursos da memória migrando objetos.

Não se deve esquecer também do espaço ocupado pelo código de cada objeto. Todos os objetos de uma mesma classe compartilham o mesmo código, que pode ser contabilizando pelo *Classloader*[1] da aplicação.

O uso do processamento por uma aplicação pode ser medido pelo tempo uso da CPU, JavaSE 5.0 permite capturar o tempo de uso da CPU das *threads* em execução. Para monitorar o uso individual de cada aplicação, pode-se modificar o código e inserir um coletor de referências às *threads* de cada aplicação, ou como é permitido dar nomes a *threads*, modificar a criação destas nomeando-as de forma que se possa identificar as *threads* de cada aplicação. As dificuldades encontradas nessas duas abordagens são: primeiro que não podemos modificar *threads* criadas pelo sistema e segundo, se a aplicação já utilizar nomes próprios para suas *threads* não podemos alterá-los. Para medir a utilização da CPU por partes da aplicação, isto é, a utilização dos métodos associados a um objeto, podemos medi-los de uma maneira menos precisa, marcando se o tempo de início e término da execução de um método. Na verdade, medimos o tempo em que um método permanece ativo, na pilha de execução. Por enquanto, nenhuma medição de uso de CPU foi implementado, tem-se apenas a informação fornecida pelo JavaSE 5.0.

Para capturar a informação de comunicação com aplicações externas, podemos trocar as classes Java de acesso a rede por classes especiais, que passam a funcionar como *proxies* das classes Java, mas que realizam a medição da comunicação e antes de repassá-las ao sistema.

[1] Entidade responsável em Java por carregar na memória o código da aplicação

Algumas das classes *proxies* para classes dos pacotes java.io e java.net já estão disponíveis no Arcabouço nos pacotes *proteus.runtime.distribution.*io e *proteus.runtime.distribution.net*. Mas os Instrumentador ainda não realiza o redirecinamento para essas classes.

Vimos que até o término e a incorporação das APIs para o gerenciamento e isolamento de aplicações ainda não existe uma solução perfeita para nossos propósitos, pois dependemos de funções internas a máquina virtual.

As técnicas até agora discutidas têm sido tratadas no escopo de aplicações, isolando-se o consumo de recursos de cada uma e tentando limitá-lo. Uma análise mais pontual do consumo dos recursos pelos componentes da aplicação devem ser também avaliadas mesmo que de maneira imprecisa. Essa análise é de fundamental importância para se realizar o particionamento de uma aplicação de maneira que permita minimizar seu consumo e maximizar seu desempenho, pois permite um melhor mapeamento entre dispositivos, seus recursos e os recursos requeridos por cada parte da aplicação.

4.1.3 Interrupção da Execução

Objetos que possuem métodos ativos, isto é, em execução, não podem ser movidos durante a distribuição, pois Java não permite que se migre a pilha de execução. Com isso, ao se analisar quais grupos de objetos serão movidos, temos que excluir da análise os que possuem os métodos ativos. Para impedir que métodos não ativos sejam ativados durante o processo de análise temos que garantir que a execução da aplicação em análise seja temporariamente interrompida.

Java permite que *threads* em execução sejam suspensos por meio do método *suspend* da classe *Thread,* mas esse método consta na documentação como *Deprecated,* isto é, está funcional mas que será descontinuado em versões futuras de Java, além disso JavaME CLDC não possui esse método. Além de impedir que as *threads* correntes continuem a executar, é importante impedir que chamadas externas sejam temporariamente processadas. Caso o objeto alvo de uma chamada tenha se movido, pode-se redirecionar a requisição ou gerar um erro informando que o objeto foi movido, pode se também informar a nova posição. O objeto que requisita a chamada, se não notificado, deve tentar localizar a nova posição do objeto destino.

Para possibilitar uma implementação independente da edição de Java e compatível com futuras edições, supondo que todos objetos se comuniquem com objetos de outros vértices via Arcademis; podemos adicionar barreiras de sincronização à aplicação que sejam acionadas pelo *middleware*, assim garantimos que novas chamadas devem esperar até que os objetos sejam movidos. Observe que execuções de métodos entre objetos associados a um mesmo vértice não precisam necessariamente cessar, pois se existe pelo menos um objeto ativo em um vértice todos objetos desse vértice recebem essa mesma classificação e não poderão ser movidos.

4.1.4 Coleta de Lixo Distribuída

Linguagens como Java não possuem desalocação explícita de um objeto. Para liberar o espaço de um objeto utilizam o coletor de lixo, que remove um objeto que não seja mais alcançável pelo código. Ao se distribuir um programa, seus objetos passam a ser referenciados em máquinas distintas. O coletor de lixo não tem conhecimento dessas referências, assim se um objeto for referenciado apenas por objetos em outras máquinas, o coletor de lixo tentará recolhê-lo. Para evitar isso, cada objeto remoto passa a ser referenciado pelo Sistema de Suporte à Execução e a possuir um contador de referências. A cada *proxy* criado, o contador é incrementado; quando um *proxy* é recolhido pelo coletor de lixo, o contador é decrementado. Somente quando o contador for igual a zero, o Sistema de Suporte à Execução libera a referência do objeto permitindo que seja recolhido pelo coletor de lixo. Para decrementar o contador, deve-se determinar quando um objeto *proxy* vai ser coletado, para isto pode-se utilizar o método *finalize* em Java, que é executado pelo coletor de lixo antes do objeto ser reclamado. Todavia a execução desse método não é garantida pela linguagem, além disso ao se terminar uma máquina virtual, normalmente esse método não é executado, podendo gerar inconsistências na demais máquinas. Uma solução alternativa e menos eficiente em termos de processamento é manter referências fracas[2] a esses objetos, e verificá-las periodicamente. Se não utilizássemos Java, mas outra linguagem também dependente de coleta de lixo e que não dispusesse de mecanismo algum que permitisse verificar a "morte" de objetos, torna-se-ia impraticável a implementação do modelo de distribuição aqui previsto, uma vez que a memória não mais utilizada, nunca seria liberada.

4.1.5 Criação do Grafo

A classe *CreationPoint* especifica o ponto de criação de um objeto. Um ponto de criação pode ser definido como um conjunto de atributos que identificam uma ou mais instantes da execução de uma aplicação, isto é, são atributos que passam a caracterizar a instrução de criação de um objeto. Essa instrução pode ser associada a atributos estáticos, como a classe, o método ou a linha de código onde se localiza, e a atributos dinâmicos como a pilha de execução atual ou o vértice do grafo associado ao objeto invocado. Tudo isso pode ser associado a uma lógica de decisão que define o identificador do vértice a ser atribuído ao novo objeto.

Os pontos de criação tem um papel fundamental no mapeamento entre objetos e vértices do grafo e são definidos pela aplicação. Por meio das informações contidas neles, o SSE determina o vértice a que o objeto criado deve ser associado. Objetos criados por pontos de criação com os mesmos atributos, a princípio, devem ser associados a um mesmo vértice. Com isso, temos

[2]Uma referência fraca é uma referência a um objeto alcançável pelo código, mas que pode ser reclamado pelo coletor de lixo, caso não exista alguma outra referência (forte) a ele.

uma grande flexibilidade, deixando a cargo da aplicação a função de determinar como será o seu grafo representativo. Observe que o grafo pode possuir um tamanho máximo pré-definido ou indefinido, isso devido aos atributos dinâmicos que permitem que novas informações sejam utilizadas. Após a determinação dos valores dos atributos, estes são resumidos um valor numérico, que o identifica. Um objeto (*CreationPoint*) contendo esse identificador e o identificador do vértice a que pertence o objeto "pai" é passado ao Sistema de Suporte à Execução e usados para associar esse ponto de criação a um vértice.

A associação entre o vértice e o ponto de criação é responsabilidade do componente *ObjectNodeAssociator,* sua função é realizar o mapeamento entre o identificador de um vértice e o ponto de criação. O vértice que pode ser tanto local ou remoto ou criar um novo vértice (novo identificador), também é responsável por aplicar restrições ao tamanho do grafo em um determinado dispositivo e definir a estratégia de alocação de objetos.

Após o particionamento da aplicação, novos objetos podem ser alocados durante a execução, essa alocação pode se dar remotamente ou localmente e esse objeto deve ser associado a um nó.

O primeiro procedimento a ser feito é determinar onde o objeto será criado. O desenvolvedor do *middleware* pode tratar a alocação de uma forma variada.

Pode-se aplicar uma restrição, em que os objetos devem ser alocados no mesmo lugar onde os nós que possuem o mesmo ponto de criação se encontram. A vantagem de se obedecer essa restrição é que o objeto obedece ao critério de otimização escolhidos pelo gerador de grafos e pelo algoritmo de particionamento, em contra-partida toda alocação a um objeto remoto gera uma comunicação com outro dispositivo, consumindo energia e banda de transmissão. Existe também uma perda de desempenho, pois a uma alocação remota é muito mais cara que uma alocação local, a utilização de criação assíncrona poderia reduzir esse custo.

Eliminando-se a restrição acima, o desenvolvedor pode permitir que a alocação seja feita no próprio dispositivo, caso haja recursos. Dessa maneira o grafo local poderia ser expandido, associando-se o objeto a um nó já existente ou se criando um outro nó. Inversamente ao caso anterior, as vantagens dessa abordagem são: não gera uma comunicação com o ambiente, se o dispositivo possuir recursos abundantes; não consume recursos de outros dispositivos e possui um menor custo de execução no momento da criação. A desvantagem seria o fato de não se obedecer o critério de otimização escolhido pelo algoritmo de particionamento, podendo gerar um desempenho global pior.

4.1.6 Replicação

A replicação é uma forma de alcançar um certo nível de tolerância a falhas. Os dispositivos do ambiente podem falhar, a existência de cópias de objetos em mais de um dispositivo não garante,

mas diminui o risco de finalização de uma aplicação. Ao se criar cópias de objetos, todo acesso que modifique seu estado deve ser repetido em todas cópias.

Arcademis já permite associar um *stub* a vários endereços remotos, como isso basta implementar a replicação das chamadas para todas as cópias. Essa replicação de acessos é complexa uma vez que se cria diversos fluxos de execução, replicando-se as chamadas. Se os métodos chamados, também realizarem chamadas a outros objetos, teremos uma série de chamadas replicadas. Essa chamadas devem ser identificadas e não devem aplicar algum efeito sobre os objetos, mas devem retornar o mesmo valor da primeira requisição recebida. Observe que chamadas geradas por cada objeto replicado devem ter um mesmo identificador. As chamadas replicadas geram um aumento desnecessário de comunicação.

4.1.7 Sistema de Localização

Ao se movimentar objetos entre dispositivos, faz-se necessário atualizar os *proxies* que os referenciam. Um sistema de localização de referências é importante para permitir essa atualização.

A interface *SystemAccess* disponibiliza o método *lookup.* Este método recebe como parâmetro um identificador único do objeto a ser localizado (Objeto da Classe *Identifier* definida em Arcademis), e retorna um vetor com endereços do tipo *MultiReference* de Arcademis*,* identificando referências aos objetos nos dispositivos onde se encontram o objeto e suas cópias.

Existem várias formas de realizar a manutenção desse endereço. Primeiro, deve-se definir quando a atualização é feita: no momento em que um objeto é movido, que denominamos de forma ativa, ou quando a referência é utilizada, que denominamos de forma passiva.

Na forma passiva, um objeto tenta acessar um objeto movido e o acesso falha, nesse momento, quando o *proxy* chama pelo método *lookup,* inicia-se o processo de descoberta. Na forma ativa ao migrar os objetos, o *ProgramManager* deve informar a nova localização dos objetos. Mais especificamente algumas das formas de se obter os endereços são:

- uso de um algoritmo de *flooding* seletivo, onde o dispositivo envia uma pergunta aos dispositivos do ambiente com que interage, até que o dispositivo com o objeto responda com seu endereço. Nesta opção existe um grande consumo de recursos devido à alta comunicação.

- envio da informação sobre a nova localização aos demais dispositivos (que possuem referência) ao se realizar a movimentação. Isso requer o conhecimento de quais objetos referenciam e onde eles estão. Esta informação pode ser obtida com a utilização dos vé fantasmas[3] descritos no Capítulo 3. Nesse caso, se certos dispositivos não chamarem no-

[3]Vértices que não possuem objetos associados a ele, e sim representam vértices em outros dispositivos

vamente pelo objeto migrado, houve uma atualização desnecessária. Se houver migrações em paralelo em mais de um disposistivo, a aplicação pura dessa solução pode falhar, pois os vértices fantasmas ficarão inconsistentes;

- registro em um servidor "central" da nova localização, permitindo que os *proxies* ao chamarem o método *lookup* consultem esse servidor. Nesse caso deve-se consumir recursos de comunicação para consultar essa base e atualizar em toda movimentação, além disso o servidor deve possuir uma cópia de todos os vértices associados ao endereço dos dispositivos que o contém;

- algoritmos de *hash* distribuido (DHT), usados em redes *Peer-to-Peer*[Rocha et al., 2004]. Nesses algoritmos a localização dos endereços é distribuída pelos nós da rede. Cada nó fica responsável por manter um conjunto de pares (chave, valor). A localização de chaves necessita um número reduzido de mensagens trocadas em comparação com o algoritmo de *flooding*. Outra vantagem é não depender de um único dispositivo como servidor.

Foi implementado apenas o registro e busca utilizando-se um servidor central.

Para evitar repetidas falhas de acesso a objetos movidos e pertencente a um mesmo vértice, podemos associar todos *proxies* dos objetos de um mesmo vértice a um único objeto que contém o endereço do dispositivo, assim ao atualizar o conteúdo desse objeto na primeira falha, todos os demais são atualizados.

Vimos um modelo que fornece toda uma estrutura para dar suporte a execução de sistemas distribuídos no ambiente *pervasivo*. Na próxima seção, trataremos de como transformar uma aplicação centralizada em distribuída de forma automática.

4.2 O Instrumentador Distribuidor

O Instrumentador é responsável por capturar informações da aplicação, instrumentar o código e gerar todo o suporte a comunicação e distribuição da aplicação. Sua principal função é determinar o que pode ser tornado remoto e como isso é feito. Consiste em dois módulos: o módulo analisador, responsável pela captura de informações da aplicação; e o módulo gerador de código que instrumenta o código e gera todo o suporte à comunicação.

A capacidade de se distribuir objetos e a definição do que pode ou não ser migrado são de total responsabilidade da aplicação. O sistema que suporta a execução apenas define interfaces para que possa interagir e requisitar recursos. Isso permite que instrumentadores independentes realizem a transformação da aplicação centralizada em distribuída, da maneira que melhor lhes convier.

A estrutura descrita aqui pode ser utilizada em diversas linguagens, mas tratamos com maior proximidade da linguagem Java. As justificativas para escolha desta linguagem foram descritos no início deste capítulo. Dentre as edições Java, escolhemos mais especificamente JavaSE (*Standard Edition*), como a plataforma alvo do código analisado e gerado. Toda a análise é feita sobre o código binário Java o *bytecode.* A escolha dessa plataforma foi feita com base nas seguintes justificativas: (a) os trabalhos relacionados utilizam em sua maioria JavaSE, ou implementações de máquinas virtuais próprias compatíveis com JavaSE, o que permite uma melhor comparação de desempenho; (b) a constatação de que diversos recursos que analisamos na Seção 4.1 e que julgamos como requisitos para a distribuição de objetos no ambiente *pervasivo* ainda não estão disponíveis em platarformas mais limitadas como JavaME CLDC; e dada necessidade de manipular diretamente o código binário, a existência de bibliotecas que facilitem essa manipulação.

Ao utilizarmos essa plataforma, restringimos nossa capacidade de testes aos dispositivos com mais recursos, como *Desktops, notebooks, Tablet PC* ou *Ultra Mobile PCs*.

4.2.1 O Módulo Analisador

O módulo analisador varre o código da aplicação a fim de capturar as seguintes características:

- comunicação e o relacionamento entre as partes da aplicação;
- consumo de memória de um objeto de cada classe;
- identificação dos objetos de classes que não poderão ser migrados;
- identificação de classes do sistema (sem código nativo) que podem/devem ser substituídas por classes equivalentes. Como esse trabalho utiliza manipulação direta do *bytecode* Java é possível acessar e modificar essas classes e salvá-las com um novo nome para adicionar, por exemplo, mobilidade a objetos dessas classes;
- identificação dos objetos que são apenas acessados localmente, isto é, não serão referenciados remotamente.

Essas informações são coletadas e armazenadas em dois grafos:

- Um grafo de tipos que expõe informações sobre as classes utilizadas na aplicação, suas características (como quais métodos e atributos são utilizados, se possui métodos nativos, métodos estáticos, tamanho de um objeto, etc) e seus relacionamentos com outras classes;
- Um grafo representativo de objetos que tentar capturar uma imagem simplificada de como estarão organizados os objetos durante a execução, nesse grafo cada vértice representa um conjunto de objetos.

O grafo de tipos é utilizado para determinar a mobilidade das classes. Analisando o código e os relacionamentos indicados no grafo, realizamos a classificação das classes em: *modificáveis*, *não-modificáveis*, *móveis*, *fixas* e posteriormente com auxílio do segundo grafo, classes *acessíveis-remotamente* ou *não-acessíveis-remotamente* . O segundo grafo é construído com base na análise do código e nas informações do primeiro grafo, sendo utilizado para adicionar instruções no código da aplicação. O segundo grafo é utilizado para se criar um grafo semelhante a ele durante a execução, associar objetos aos vértices desse grafo e utilizá-lo no particionamento. Cada vértice deve conter um identificador.

Um algoritmo para criação desses grafos é descrito com mais detalhes na Seção 3.2. Um algoritmo para classificação é descrito ainda nesta seção.

No grafo representativo de objetos, algumas arestas (denominadas arestas de criação) contêm informações que identificam os pontos de criação (vide Seção 4.1.5). Além disto, nos grafos de tipos e representativo de objetos, as arestas podem possuir pesos que quantificam a interação (ou comunicação) entre as partes da aplicação. Existem dois tipos de comunicação: a comunicação entre os componentes do software, e a comunicação com entidades externas, como *webservices*, servidores *http*, dentre outros.

A comunicação com outras entidades não é passível de qualquer forma de previsão, a não ser que se descreva características da aplicação, como ser um *player* de *streams* vídeo ou um navegador *http*, e o comportamento do usuário.

A comunicação entre os componentes de software depende basicamente da freqüência de acessos a campos e métodos, dos parâmetros e resultados transmitidos. O comportamento de uma aplicação também é difícil de ser previsto, mas faz sentido que componentes que possuam mais referências entre si (ou referência dentro de *loops*) provavelmente se comuniquem mais. Segundo André Spiegel [Spiegel, 1998], se o número de parâmetros e o espaço ocupado por eles na memoria não forem muito grandes, o maior impacto na comunicação está na freqüência de acesso. Vamos, neste trabalho, descartar a análise dos parâmetros e analisar apenas as interações. Um critério para ponderar essas arestas é descrito na Seção 3.2.

Aplicações diferentes podem ser melhor representadas por grafos que utilizem conhecimentos específicos dessas aplicações, a fim de permitir a utilização de vários algoritmos para criação desses grafos separou-se a classificação das classes, a criação do grafo de tipos e do grafo representativo de objetos. De maneira que algoritmos diferentes possam ser utilizados e ser combinados. Para isso utilizou-se o padrão de projetos *factory*. Onde fábricas diferentes podem ser utilizadas para criar os grafos. Na Figura 4.12, temos de forma simplificada as interfaces que definem os principais componentes responsáveis pela análise. A classe *Analyser* implementa o controle do fluxo de informações entre os componentes e não precisa ser estendida. Os algoritmos para a classificação, criação do grafo de objetos e criação de tipos devem implementar a se-

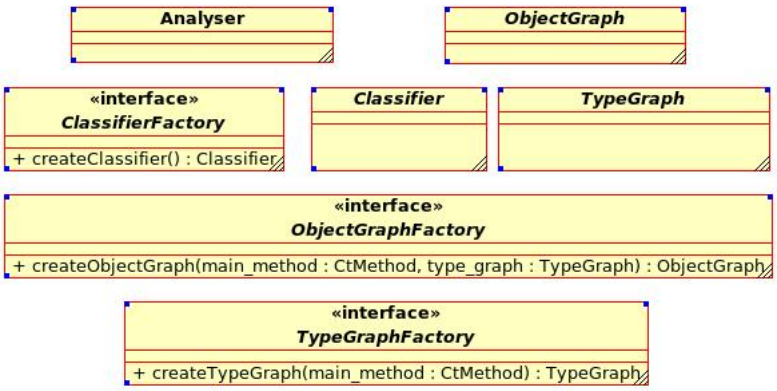


Figura 4.12: Principais interfaces do analisador

guintes interfaces *ClassifierFactory*, *ObjectGraphFactory*, *TypeGraphFactory* respectivamente. Para modificar um algoritmo, basta registrar a nova fábrica na classe *CompilerSYS* não listada na figura, que funciona como um contêiner de fábricas. Essa classe permite, facilmente, a configuração de novas fábricas como o acesso a estas pelas demais classes da arquitetura. As classes *Classifier, ObjectGraph* e *TypeGraph* são os produtos das fábricas e disponibilizam informações resultantes das análises. *Classifier* identifica que classes são móveis, fixas, modificáveis e/ou remota, *ObjectGraph* representa o grafo de representativo de objetos e *TypeGraph* representa o grafo de tipos.

Além dessas interfaces foram também criadas implementações concretas dessas interfaces e métodos para facilitar a análise e a transformação do código. Esses métodos e classes foram implementadas, utilizando-se como base a biblioteca *javassist[Chiba, 2006]* para manipular *bytecode*s.

Nem todos os objetos podem ser distribuídos. Objetos que fazem acesso direto a recursos de entrada e saída não podem migrados, uma vez que esse acesso deve ocorrer no dispositivo do usuário. Não faz sentido, por exemplo, que o usuário esteja ouvindo um tocador de MP3 e tenha o som direcionado para um outro dispositivo qualquer. Objetos que possuem código nativo também não podem ser migrados, uma vez que podem não ser compatíveis com o dispositivo destino. Classes do sistema (classes distribuídas com a JVM) também não podem ser modificadas, logo objetos passados por referência a essas classes não podem ser migrados, já que podem ter atributos acessados diretamente por essas classes.

J-Ochestra[Tilevich e Smaragdakis, 2002] utiliza essa mesma forma de distribuição e classifica as classes em três categorias:

- classes *não-modificáveis* e *fixas*: são classes com código nativo ou que são manipuladas

Algoritmo 1 Identifica as classes não modificáveis [Tilevich e Smaragdakis, 2002]

```
achar_juntos (A)
   AS := conjunto de todas as classes modificáveis
         do sistema e todos os tipos de vetores
   A := A ∪ SuperClasses (A) ∪ SubClasses (A)
   Faça
      AS := AS-A;
      AArg := argumentosDoMetodo (A)
      AArg := AArg ∪ SuperClasses (AArg) ∪
              SubClasses (AArg) ∪ Constituintes (AArg)
      Args := AS ∩ AArg
      A := A ∪ ArgS
  enquanto ArgS≠Ø
```

diretamente por classes não modificáveis. Para simplificar supomos que apenas as classes do sistema fazem acesso a código nativos;

- classes *modificáveis* e *fixas*: são classes que apesar de modificáveis não devem ser distribuídas e devem permanecer juntas a classes fixas. Essas classe estendem ou possuem campos públicos acessados por classes não modificáveis;

- classes *móveis*: são classes que não se encaixam nas duas categorias acima, e podem ser modificadas. Observe que classes do sistema que não possuem código nativo ou são acessadas por classes fixas podem ser copiadas para um pacote não pertencente ao sistema e modificadas.

O Algoritmo 1, descrito em pseudocódigo, é utilizado no J-Ochestra[Liogkas et al., 2004] para obter as classes que devem manter-se no mesmo dispositivo das classes do conjunto A.

As funções utilizadas são as seguintes*: Super (Sub)Classes (X),* que obtém as *super (sub)* classes das classes contidas em A. O método *argumentosDoMetodo (X)* que retorna todas as classes passadas como parâmetros; todas as classes que são tipo de retorno de todos os métodos existentes na classes contidas em X, *Constituintes (X)*, que retorna os tipos constituintes de todos os tipos de vetores contidos em X. Exemplo T[][], os tipos constituintes são T[] e T.

Esse algoritmo é extremamente conservador. Observe que classes *não-modificáveis* que recebam classes modificáveis como parâmetros e que somente acessem métodos e não acessem campos não necessitam ser colocadas em conjunto com as classes *não-modificáveis*. Durante a análise do grafo de tipos é possível detectar quais atributos e métodos são utilizados em cada classe, o que permite uma aplicação mais precisa desse algoritmo.

Além dos objetos pertencentes às classes fixas, alguns objetos se tornam impedidos de serem distribuídos ou de movidos após sua distribuição, pois possuem métodos na pilha de execução. Java não possui mecanismos para o acesso e migração do estado de execução. A migração de objetos e seu estado de execução é denominado migração forte. Java permite apenas a migração das informações armazenadas no *heap*, denominada migração fraca. Existem diversos trabalhos voltados para permitir a migração forte em Java, como exemplo, temos JavaThread [Bouchenak e Hagimont, 2002] que utiliza uma máquina virtual modificada. O trabalho de Hong Wang et. al. [Sekiguchi et al., 1999] permite simular a migração forte por meio do uso de primitivas de migração inseridas por eles na linguagem Java *(go* e *unlock*). Por meio de transformações no código fonte gera um código Java puro que permite que o contexto seja mantido. Recentemente, uma máquina virtual da Sun MicroSystem denominada Squawk [Simon et al., 2006] com foco em redes de sensores sem fio permite a migração de *isolates*. O que mostra ser uma funcionalidade futura da JVM.

Após a geração do grafo, algumas classes poderão ter seus objetos referenciados apenas dentro dos vértices que os contêm. Se todos os objetos de uma mesma classe não forem referenciadas por objetos em vértices diferentes, esses objetos não precisam se tornarem visíveis remotamente, não sendo necessário implementarem a interface *Remote* e podem ser referenciados diretamente. Não tornar uma classe remota torna sua execução mais eficiente, pois remove todo processamento inserido para intermediar a conexão entre o objeto cliente e o objeto remoto. Além disso, dispensa o processamento de criação requisitado ao SSE, que define onde será criado o objeto, uma vez que vai estar associado ao objeto remoto do vértice local. Na implementação, já disponibilizada da Ferramenta, todos os objetos móveis são considerados remotos.

4.2.2 Gerador de Código

O gerador de código, com base nas informações obtidas pelo módulo de análise, modifica o código da aplicação. As principais modificações são:

- inserir *proxies* e mecanismos de comunicação e distribuição;
- inserir tratamento de erros de comunicação;
- inserir código para a criação do grafo e associação entre objetos e vértices;
- inserir código de prospecção dos recursos utilizados pela aplicação;
- inserir interfaces para comunicação com o SSE e inicialização.

É importante ressaltar que a capacidade de se distribuir é responsabilidade da aplicação. Contudo, o mecanismo de distribuição deve ser derivado de Arcademis, com algumas modificações. Ao invés de estenderem diretamente as classes *Stub*, *RemoteObject* e *Skeleton* de Arcademis, devem estender as classes *ProteusRemoteObject, ProteusStub* e *ProteusSkeleton* respectivamente. Estas classes possuem atributos, métodos e construtores adicionais que permitem ao Sistema de Suporte à Execução informar ao objeto em sua criação o identificador do vértice associado a ele. Esse identificador pode ser útil para o objeto requisitar informações sobre seu vértice ao SSE, como também para definir o ponto de criação.

4.2.2.1 Inserir *proxies* e mecanismos de comunicação e distribuição

Neste trabalho, a distribuição é feita por meio de *proxies* que são inseridos como intermediários entre os objetos. Eles têm a função de direcionar o acesso a um objeto destino esteja ele local ou remoto. Outra abordagem seria a modificação do interpretador, a máquina virtual Java (JVM), essa abordagem elimina a necessidade de se gerar código de distribuição, mas perde em portabilidade da solução.

Na Figura 4.13, exemplificamos a transformação aplicada aos objetos em um programa centralizado. Na letra (a), temos um objeto A sendo referenciado por três objetos. Na letra (b), temos a inserção dos *proxies*, que ficam responsáveis por direcionar as mensagens ao objeto A. Como pode ser notado, todo objeto possui apenas um *proxy* por dispositivo. Se um objeto migrar de dispositivo, ele deve passar a referenciar os *proxies* locais (se já existirem) para garantir a mesma semântica nas operações de igualdade com referências. Isso é feito por meio de chamadas à interface *SystemAccess* definida no SSE, que registra e permite a localização dos *proxies* locais da aplicação.

Como especificado anteriormente, utilizaremos uma instância do arcabouço Arcademis para implementar a comunicação entre objetos remotos e clientes. Isso permite, por exemplo, utilizar o próprio *stub* como *proxy*, diferentemente de trabalhos como J-Orchestra [Liogkas et al., 2004] e JavaParty [Philippsen e Zenger, 1997] que utilizam o mecanismo fechado de RMI. Como não podem modificar o *stub*, acabam por utilizar além do *stub* um outro *proxy*. Neste trabalho, diferentemente da forma convencional, o *stub* é criado pelo objeto cliente e não pelo objeto remoto. O *stub* requisita a criação ao sistema de SSE que retorna o identificador e o endereço do objeto remoto. A Figura 4.13 é apenas uma simplificação de toda estrutura de comunicação adicionada. A Figura 2.2 mostra mais precisamente o aparato necessário para a comunicação adicionado.

Como vimos, existem classes que podemos modificar e classes que não são modificáveis. Para permitir o acesso remoto a essas classes são feitas transformações organizadas em quatro

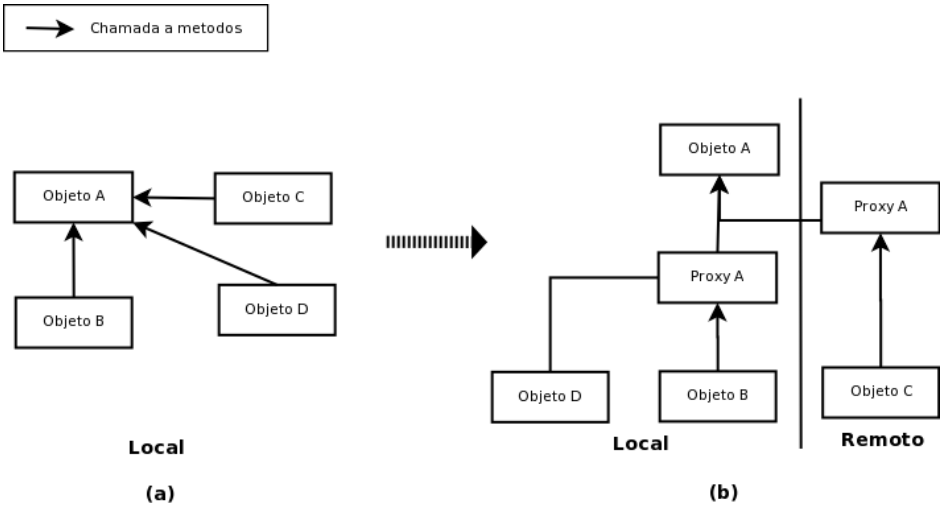


Figura 4.13: (a) Objetos se comunicando no programa original. (b) Objetos se comunicando por meio de *proxies*

categorias:

Categoria-I: a classe do objeto cliente (que possui a referência) e classe o objeto remoto são *modificáveis*:

- todos os atributos públicos da classe remota passam a ter métodos *get* e *set* associados a eles;
- classes remotas são renomeadas e recebem o sufixo _remoto, e um conjunto de *stubs (proxies)* com a mesma estrutura hierárquica e métodos dos objetos remotos são criados. Os *stubs* são criados com o nome original das classes. Veja as classes geradas na Figura 4.14.
- classes clientes que acessam os campos públicos da classe remota passam a acessá-los pelos métodos *set* e *get;*
- as ocorrências do operador *new* em classes clientes passam a referenciar a classe *stub* da classe original.

Categoria-II: a classe do objeto cliente é *modificável* e a classe do objeto remoto é *não-modificável*:

- cria-se outra classe *proxy (proxy* servidor*).* Ela fica posicionada em conjunto com o objeto que se quer acessar remotamente. Essa classe fica responsável por receber todo o aparato para permitir seu acesso remoto;

- a classe cliente é alterada como se estivesse referenciando a classe *proxy* ao invés do objeto *não-modificável.*

Categoria-III: a classe do objeto cliente é *não-modificável* e a classe do objeto do remoto é *modificável*:

- semelhante a primeira categoria, mas como não é possível alterar o operador *new,* uma outra classe *Proxy,* que passa a ter o nome original da classe remota original, fica responsável por requisitar sua alocação e repassar as chamadas para o *stub;*
- essa estratégia de modificação não é possível quando o cliente referencia um campo do objeto remoto, nesse caso objetos das duas classes não podem ser criados em dispositivos diferentes.

Categoria-IV: a classe do objeto cliente e a classe do objeto remoto são *não-modificáveis:*

- não existe solução, objetos das duas classes que não podem ser distribuídos e devem permancer juntos.

As classes que possuem atributos estáticos são separadas em parte estática e dinâmica. Uma vez que a parte estática (métodos e atributos) deve ser única e visível a todas as partições, um objeto remoto especial deve ser criado para representá-la. Como Java permite acesso ilimitado entre as partes dinâmica e estática, todos os métodos e variáveis da parte estática devem se tornar públicos.

Objetos estáticos são alocados no início da execução da aplicação e têm suas referências gerenciadas pelo Sistema de Suporte à Execução.

O Sistema de Suporte à Execução impõe à aplicação que todos os objetos, associados a vértices do grafo e móveis (objetos que podem ser migrados), devem implementar a interface *Marshalable* de Ardecamis, isto é, cada objeto é responsável por sua serialização e desserialização; também devem implementar a interface *Active,* a qual permite a um objeto realizar as inicializações necessárias para torná-lo acessível remotamente em um dispositivo, como também para realizar as desvinculações necessárias antes de ser migrado para fora do dispositivo. A execução dessas atividades é de total responsabilidade da aplicação e inseridas pelo Instrumentador.

4.2.2.2 Inserir tratamento de erros de comunicação

Diversos erros de comunicação passam a existir ao se distribuir uma aplicação centralizada. A comunicação pode falhar momentaneamente ou ser interrompida para sempre. As causas podem

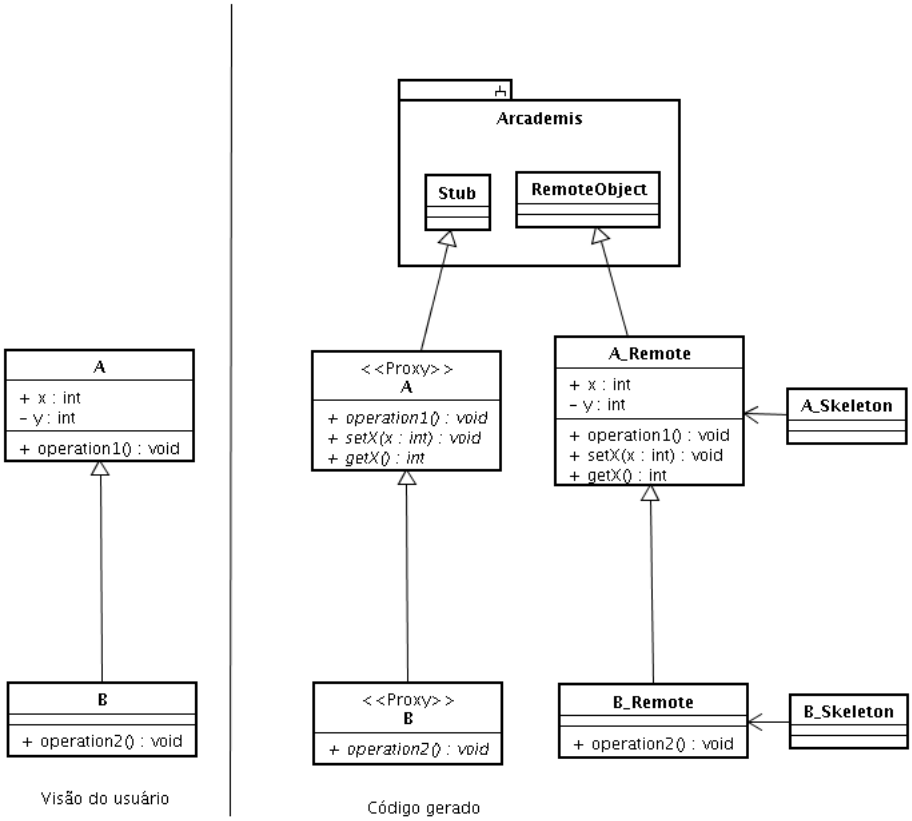


Figura 4.14: Geração transformação de classes

ser as mais variadas, por exemplo, uma interferência no sinal, o dispositivo foi desligado, travou, sofreu algum dano físico ou a bateria exauriu-se.

Ao tornar uma aplicação capaz de se distribuir, devemos torná-la também capaz de lidar com esses erros. Por meio do módulo analisador, agrupamos objetos em vértices e determinamos as arestas de uso entre esses vértices, que indicam possíveis interações remotas. Com esta informação, um programador ou a ferramenta de geração de código podem determinar quais interações podem falhar e tratá-las de forma diferenciada.

Todo erro de comunicação somente pode ser percebido pelo objeto cliente, isto é, o objeto que realiza a chamada, uma vez que o objeto remoto não sabe quando será requisitado. Assim, ao modificarmos uma aplicação centralizada, delegamos aos *stubs* o tratamento dos erros de comunicação. Isto é conveniente, dado o relacionamento estreito entre o *stub* e o tipo do objeto remoto. Além do tipo do objeto, outra informação é disponibilizada ao *stub*, o identificador do vértice remoto. Assim ele pode tratar cada vértice de forma própria. Arcademis já define

a exceção NetworkException para informar ao *stub* dos problemas da rede que deve tratar, e diferentemente de RMI, o stubs que geramos não lançam nenhuma exceção para a aplicação, já que esta não tem condições de tratá-las. Os stubs são gerados em código binário (*bytecode*) e também são gerados em forma de código fonte, o que permite ao desenvolvedor adicionar o tratamento de erro adequado a sua aplicação.

Por enquanto, o tratamento de erros adicionados pela Ferramenta é apenas sinalizar os erros, deixando totalmente por conta do desenvolvedor adicionar os mecanismos para o tratamento.

4.2.2.3 Criação do grafo e associação entre objetos e vértices.

O Sistema de Suporte à Execução oferece o método *newObject,* que permite criar um objeto, e sugerir associá-lo a um vértice no grafo. Cabe ao SSE determinar realmente qual vértice e em qual dispositivo será feita a associação e criação.

Do grafo obtido na fase de análise, temos as seguintes informações:

- um identificador único representando cada um vértice;
- as classes pertencentes aos objetos associados a cada vértice;
- o relacionamento de criação entre os vértices;
- informações que identificam o ponto de criação.

O nosso objetivo é determinar qual vértice do grafo um objeto a ser criado deve ser associado. As informações que utilizamos como atributos do ponto de criação são:

- o identificador do vértice do objeto que cria o novo objeto;
- o operador *new*;
- o tipo do objeto que executa o *new*.

Esses atributos devem caracterizar unicamente cada aresta de criação. Eles são usados para criar o mapeamento entre esses atributos cujos valores devem ser detectados durante a execução e o identificador do vértice indicado no grafo.

Para determinarmos o identificador do vértice associado a um objeto, em tempo execução, toda classe (móvel) passa a contar um atributo que armazena a identificação do seu vértice. Adicionamos também um parâmetro nos construtores para que ao ser criado, o SSE informe ao objeto a identificação. Com base nas informações citadas acima, em cada classe é inserido uma função que dado o ponto de criação define o identificador do vértice a que pertence o novo objeto .

4.2.2.4 Inserir código de prospecção dos recursos utilizados pela aplicação.

Existem duas utilizações diferentes da avaliação do uso dos recursos. A primeira que deve ser mais precisa, deve contabilizar o uso geral dos recursos pela aplicação local. Essa contabilização geral é importante para o gerenciamento de recursos por parte do SSE, limitando ou liberando recursos às aplicações de maneira segura. A segunda avaliação deve contabilizar o uso de recursos pelos objetos associados a cada vértice. Esta avaliação não necessita ser precisa, uma vez que sua função é disponibilizar informações ao algoritmo de particionamento, para melhor agrupar os vértices.

Informações globais são mais fáceis de serem obtidas por meio de APIs de gerenciamento, enquanto informações de partes da aplicação devem ser fornecidas por elas mesmas.

Recursos como a comunicação com entidades externas e com o usuário (interfaces de entrada e saída) podem ser medidos utilizando-se o padrão de projeto *Decorator* [Freeman et al., 2004]. Criam-se novas classes, chamadas classes decoradoras, que estendem as classes responsáveis pela comunicação da API Java e sobrescrevem seus métodos. As novas classes recebem como parâmetro, em seu construtor, um objeto do tipo da classe que estendem. Elas redirecionam as chamadas a seus métodos para este objeto, após realizarem o *log* das atividades. Na aplicação modificamos a criação desses objetos para que sejam passados ao decorador.

Para capturar recursos de processamento de uma aplicação local (como um todo), não é necessário instrumentá-la. O próprio SSE pode obter essas informações utilizando-se da API de gerenciamento *java.management,* desde que a JVM não compartilhe várias aplicações.

Para medir a interação entre partes da aplicação, a cada chamada de um método registramos os vértices origem e destino, contabilizando essa chamada. Pode ser registrar também o tempo decorrido entre chamadas para o cálculo de uma freqüência de invocação.

Para contabilizar o uso da memória como um todo, também pode ser feito apenas pelo SSE com uso de funções da API de gerenciamento *java.management.* Novamente, isto se não houver compartilhamento de aplicações em uma mesma JVM. Para obter-se a memória de regiões da aplicação, pode-se estimá-la com uso de reflexão computacional acessando as referências aos objetos referenciados pelo grafo, ou com inserção de métodos nos objetos que contabilizem seu tamanho.

Para se obter valores de consumo gerais da aplicação, existindo o compartilhamento de aplicações numa mesma JVM, pode se utilizar a soma dos valores informados pela instrumentação da aplicação, o que torna estes valores possivelmente inseguros e imprecisos. Essa técnica pode

ser utilizada apenas em sistemas experimentais, enquanto Java não disponibilizar as APIs de gerenciamento para aplicações compartilhadas.

4.2.2.5 Interfaces para comunicação e inicialização do Sistema de Suporte à Execução.

O Sistema de Suporte à Execução exige que um objeto da aplicação implemente a interface *InitApplication. A* interface *InitApplication* é responsável por entregar a cada aplicação o objeto do tipo *SystemAccess* associado a ela e chamar o método principal que inicia a execução da aplicação. A implementação dessa interface deve inicializar uma classe cujos métodos são estáticos e que tornam o objeto do tipo *SystemAccess* visível a todos os objetos da aplicação. Durante o processo de migração, qualquer referência a interface *SystemAccess* não é serializada, pois no processo desserialização, a referência a ele é substituida pelo SSE por uma referência a um objeto *SystemAccess* do dispositivo local.

4.3 Conclusão

Neste capítulo, descrevemos Proteus, um modelo e um conjunto de classes e interfaces que tem como finalidade facilitar a exploração de diversas estratégias para a distribuição de aplicações em ambientes pervasivos. Proteus simplifica o desenvolvimento de aplicações distribuídas, liberando o desenvolvedor de lidar diretamente com o código de distribuição, mas permitindo que ele insira o tratamento dos erros de comunicação adequados a aplicação, como também, por meio de Arcademis, escolha os protocolos de comunicação desejados. A estrutura do Sistema de Suporte à Execução permite a utilização de diferentes algoritmos de distribuição em cada dispositivo, adequando-os as capacidades de cada um. Além disso, defendemos um modelo que permite não apenas distribuir uma aplicação mas um conjunto de aplicações simultaneamente. Abordamos e listamos alguns requisitos para a distribuição nesses ambientes e as dificuldades de implementá-los sobre a tecnologia Java, que oferece uma capacidade limitada de gerenciamento de múltiplas aplicações em uma mesma JVM e vimos também que a evolução dessa tecnologia já mostra alguns esforços para superação dessas restrições.

Capítulo 5

A Implementação de Proteus

Neste capítulo, descrevemos uma implementação em Java do Arcabouço proposto no Capítulo 4. Como vimos, Proteus tem dois núcleos centrais: o Sistema de Suporte à Execução, que coordena a distribuição das aplicações e o Instrumentador, que modifica aplicações centralizadas, tornando-as capazes de se distribuirem e interagirem com o Sistema de Suporte à Execução. Antes de detalharmos a implementação, descrevemos algumas bibliotecas que utilizamos para a criação da instância do Arcabouço. Para criarmos o Instrumentador utilizamos, com pequenas modificações, a biblioteca de código aberto Javassist [Chiba, 2006], que permite a manipulação de *bytecodes*. Essa biblioteca permite acessar e alterar o conteúdo dos arquivos objetos Java (.class). Os grafos utilizados no Instrumentador foram criados pelas estruturas de dados disponibilizadas pela biblioteca JgraphT [Naveh, 2007], que implementa diversos tipos de grafos: grafos simples, digrafos, hipergrafos, dentre outros.

Por questões de tempo, desenvolvemos uma instância simplificada de Proteus, que ainda não é adequada para um ambiente real.

Como descrevemos na Seção 2.4, Arcademis define a arquitetura para a criação de *middlewares* que permitem a comunicação remota em um modelo similar ao RPC. Para criarmos uma instância de Proteus precisamos também criar uma instância de Arcademis com a finalidade de permitir a comunicação entre os componentes do Sistema de Suporte à Execução, como também a comunicação entre as partes da aplicação. Juntamente com Arcademis é disponibilizada uma instância denominada RME. Utilizamos essa instância com pequenas modificações para avaliarmos o Proteus.

Os fontes do Arcabouço estão organizados em pacotes da seguinte forma (ver a Figura 5.1): no nível superior temos os pacotes *proteus* e *instance.* O primeiro contém o Arcabouço, e o segundo as classes que configuram as fábricas de objetos desejadas do Arcabouço, isto é, define como será configurada uma instância de *Proteus.*

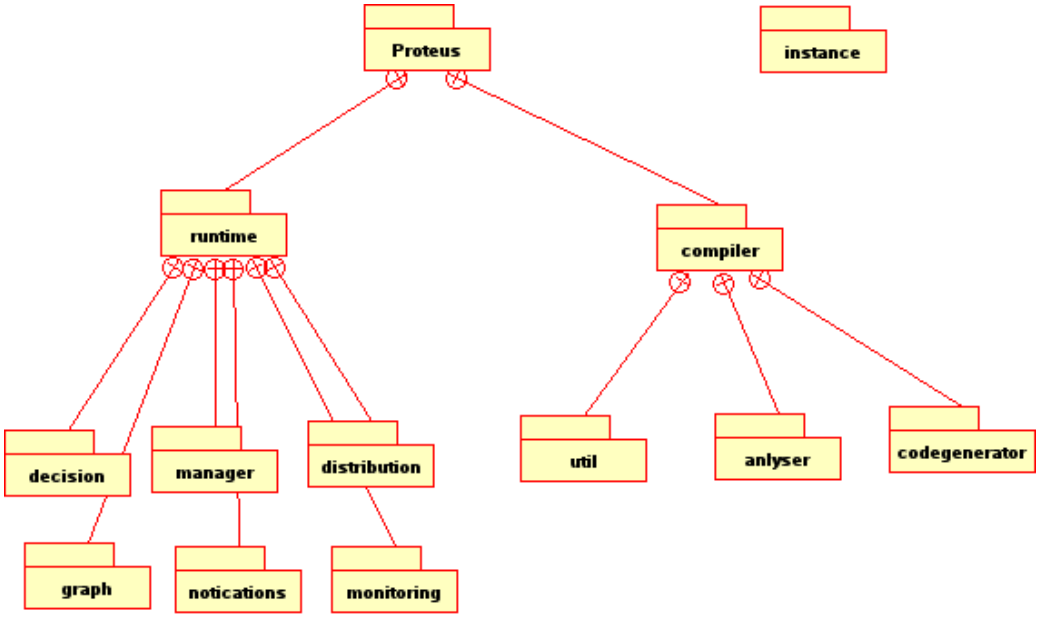


Figura 5.1: Digrama de pacotes: estrutura utilizada na implementação

Internamente o *Proteus* é composto pelos pacotes *runtime* e *compiler*:

O pacote *runtime* *Runtime* agrupa as classes, interfaces e algoritmos do Sistema de Suporte à Execução. É formado pelos seguintes subpacotes:

- *decision*: contém as classes, interfaces e algoritmos relacionados às decisões de particionamento da aplicação em tempo de execução;
- *distribution:* contém as classes relacionadas à comunicação e distribuição e que estendem classes de Arcademis e classes da API Java. Estas classes devem ser utilizadas pela aplicação no lugar das originais (estendidas) de *Arcademis e Java.*
- *graph:* contém classes e interfaces relacionadas ao grafo representativo de objetos utilizados durante a execução;
- *manager:* contém as classes responsáveis pelo gerenciamento das aplicações durante a execução;
- *monitoring:* contém as classes responsáveis pelo monitoramento das aplicações;
- *notifications:* contém as notificações definidas para a comunicação entre os SSEs em execução.

O pacote *compiler* Agrupa as classes, interfaces e algoritmos do Instrumentador. Possui a seguinte estruturação:

- *util:* contém funções para realizar transformações nas classes da aplicação, como adicionar parâmetros a métodos, criar *proxys* de classes, dentre outros;
- *analyser*: contém classes e interfaces para realização da análise da aplicação, criação do grafo de tipo, grafo representativo objetos e classificação das classes.
- *codegenerator:* contém as classes e interfaces responsáveis por definir e realizar a transformações no código da aplicação para torná-la distribuída.

Em cada pacote, temos ainda um pacote denominado *concrete.* Neste pacote está a implementação dos principais algoritmos disponibilizados. As classes que estão fora destes pacotes são, em sua maioria, classes de apoio aos algoritmos ou são classes abstratas ou interfaces que definem métodos que devem ser implementados caso se deseje customizar *Proteus* com algoritmos próprios.

A implementação é composta por cerca de 183 classes e interfaces (Instrumentador e o Sistema de Suporte à Execução), resumidas nas tabelas 5.1, 5.2, 5.3 e 5.4. Totalizam cerca de 12.000 linhas, desconsiderando-se comentários e espaços em branco. O espaço ocupado pelo Sistema de Suporte à Execução completo, incluindo o Arcabouço e a instância de Arcademis (baseada em RME), é aproximadamente 690KB em formato binário. Em cada tabela, exibimos as classes que compõem Proteus e o estágio de desenvolvimento de cada uma. Não exibimos as classes de RME, Arcademis e Javassist, salvo aquelas que sofreram grandes alterações. Na primeira coluna temos o nome completo da classe e os pacotes em que se encontram. A segunda coluna indica o número de linhas de código desconsiderando-se os comentários e campos em branco. A terceira coluna contabiliza o número de funções existentes. As duas colunas seguintes indicam o número de funções a serem feitas e uma estimativa do esforço de desenvolvimento necessário. Por último, marcamos com um $\surd$ as classes finalizadas e com *X* as que ainda necessitam serem finalizadas.

A classe do Instrumentador ainda inacabas as seguintes são:

- *proteus.compiler.util.TranformVectors:* responsável por substituir vetores por objetos remotos para que possam compartilhar seu conteúdo remotamente. Deve-se modificar as referências e gerar as classes que substituírão os vetores.
- *proteus.compiler.util.MonitoringCode:* responsável por adicionar ao código das aplicações funções de monitoramento dos objetos associados a um vértice, e assim estimar e mensurar o uso de recursos de cada vértice;

Classes	Linhas de código	Número de funções	Funções a fazer	Linhas a serem feitas	Completa √/X
proteus.compiler.rmec.MethodDecomposer	170	10	0	0	√
proteus.compiler.rmec.ClassDecomposer	99	7	0	0	√
proteus.compiler.rmec.ParameterDecomposer	19	5	0	0	√
proteus.compiler.rmec.TypeDecomposer	14	4	0	0	√
proteus.compiler.util.StubSkelGenerator	784	29	1	60	√
proteus.compiler.util.debug.Print	293	25	0	0	√
proteus.compiler.util.TranformVectors	0	5	5	400	X
proteus.compiler.util.CodeBrowser	111	19	0	0	√
proteus.compiler.util.CodeChangeLog	49	8	0	0	√
proteus.compiler.util.MonitoringCode	5	5	3	300	X
proteus.compiler.util.BehaviourInstructionsFinder	99	9	0	0	√
proteus.compiler.util.LimitedStack	42	5	0	0	√
proteus.compiler.util.StaticClassCreator	144	5	0	0	√
proteus.compiler.util.CodeInspection	494	15	2	180	X
proteus.compiler.util.ProxyStubCreator	8	0	6	700	√
proteus.compiler.util.CodeTransformer	352	12	2	150	√
proteus.compiler.util.ReplaceField	27	2	0	0	√
proteus.compiler.util.ByteCodeConstants	9	0	0	0	√
proteus.compiler.util.GenerateInitApp	30	3	0	0	√
proteus.compiler.CompilerException	9	3	0	0	√
proteus.compiler.NotInitiateClassException	5	2	0	0	√
proteus.compiler.codegenarator.CodeGenerator	22	3	0	0	√
...codegenarator.concrete.DefaultCodeGenerator	414	13	0	0	√
proteus.compiler...concrete.DefaultCodeGeneratorFc	4	1	0	0	√
proteus.compiler.codegenarator.ClassSets	15	2	0	0	√
...codegenarator.CodeGeneratorFactory	3	1	0	0	√
proteus.compiler.Distributor	43	1	0	0	√
proteus.compiler.CompilerSYS	47	11	0	0	√
proteus.compiler.Constants	14	0	0	0	√
proteus.compiler.CompilerConfigurator	1	0	0	0	√
proteus.compiler.analyser.ObjectVertex	36	9	0	0	√
proteus.compiler.analyser.ClassifierFactory	3	1	0	0	√
proteus.compiler.analyser.ObjectGraphFactory	3	1	0	0	√
proteus.compiler.analyser.TypeGraphFactory	3	1	0	0	√
proteus.compiler.analyser.ObjectGraph	24	17	0	0	√
...compiler.analyser.concrete.SimpleClassfierFc	5	2	0	0	√
...compiler.analyser.concrete.TypeGraph_impl	221	27	0	0	√
proteus.compiler.analyser.concrete.CollectMethod	66	3	0	0	√
...compiler.analyser.concrete.DefaultTypeGraphFc	412	15	0	0	√
...compiler.analyser.concrete.CreationEdge_impl	12	4	0	0	√
..compiler.analyser.concrete.DefaultObjectGraphFc	458	11	0	0	√
proteus.compiler.analyser.concrete.Graphviewer	70	5	0	0	√
...compiler.analyser.concrete.ObjectGraph_impl	138	23	0	0	√
proteus.compiler.analyser.concrete.SimpleClassifier	146	9	0	0	√
proteus.compiler.analyser.FrequencyEdge	28	7	0	0	√
proteus.compiler.analyser.CreationEdge	1	0	0	0	√
proteus.compiler.analyser.Classifier	11	7	0	0	√
proteus.compiler.analyser.TypeGraph	38	27	0	0	√
proteus.compiler.analyser.Type	493	101	0	0	√
proteus.compiler.analyser.TypePool	97	9	0	0	√

Tabela 5.1: Classes implementadas - parte 1

Classes	Linhas de código	Número de funções	Funções a fazer	Linhas a serem feitas	Completa √/X
appdistribution.AppEpidFc	31	3	0	0	√
appdistribution.AppConfigurator	43	1	0	0	√
appdistribution.server.AppDispatcher	35	5	0	0	√
appdistribution.server.AppRemoteObject	45	5	0	0	√
appdistribution.server.AppDispatcherFc	38	3	0	0	√
appdistribution.server.AppActivatorFc	14	2	0	0	√
appdistribution.server.RmeAcceptorFc	18	3	0	0	√
appdistribution.server.AppActivator	62	5	0	0	√
appdistribution.AppRemoteRefFc	22	5	0	0	√
proteus.util.ProteusStream	531	48	0	0	√
proteus.util.Constants	39	0	0	0	√
proteus.util.Cloneable	3	1	0	0	√
proteus.util.StreamUtil	95	3	0	0	√
proteus.util.WeakTreeMap	57	3	0	0	√
proteus.util.WeakHashMap	57	3	0	0	√
proteus.util.IntContainer	31	9	0	0	√
proteus.InitApplication	3	1	0	0	√
proteus.runtime.graph.IntEdge	5	2	0	0	√
proteus.runtime.graph.GraphFc	3	1	0	0	√
proteus.runtime.graph.NodeDescription	8	2	0	0	√
proteus.runtime.graph.constructor.CreationPoint	16	6	0	0	√
proteus.run...ctor.ObjectNodeAssociatorFc	3	1	0	0	√
proteus.run...ctor.ObjectNodeAssociator	3	1	0	0	√
proteus.runtime.graph.Edge	1	0	0	0	√
proteus...graph.concrete.FooObjectNodeAssociator	5	2	0	0	√
proteus.runtime.graph.concrete.Graph_imp	147	20	0	0	√
proteus.runtime.graph.concrete.Graph_impFc	5	2	0	0	√
proteus.runti...crete.FooObjectNodeAssociatorFc	5	2	0	0	√
proteus.runtime.graph.concrete.Node_imp	80	10	1	60	X
proteus.runtime.graph.Node	132	22	0	0	√
proteus.runtime.graph.Graph	27	16	0	0	√
proteus.runtime.graph.NodeFc	3	1	0	0	√
proteus.runtime.graph.NodeID	8	2	0	0	√
proteus.runtime.manager.ProgramManagerFactory	3	1	0	0	√
proteus.runtime.manager.ProgramManager	27	8	0	0	√
proteus.runtime.manager.ProgramDescriptor	61	11	0	0	√
proteus....manager.concrete.ServerManagerImp	122	15	**6**	600	X
proteus.run...crete.DistributedProgramManager	236	23	2	150	X
proteus.run...crete.DistributedProgramManagerFc	5	2	**0**	0	√
proteus.runtime.manager.ProgramManagerRef	18	12	0	0	√
proteus.runtime.manager.ServerManager	22	9	0	0	√
proteus.runtime.manager.ProgramDescriptorFc	3	1	0	0	√
proteus.runtime.manager.AccessDeniedException	5	2	0	0	√
proteus.runtime.manager.RemoteProgramManager	4	1	0	0	√
proteus.runtime.manager.DistributedClassLoader	74	9	0	0	√
proteus.runtime.manager.Transporter	4	2	0	0	√
proteus.runtime.manager.DistributionServer	5	3	0	0	√
proteus.runtime.manager.LocalProgramManager	46	12	0	0	√

Tabela 5.2: Classes implementadas - parte 2

Classes	Linhas de código	Número de funções	Funções a fazer	Linhas a serem feitas	Completa √/X
proteus.runtime.apperror.ApplicationError	3	1	0	0	√
proteus.runtime.concrete.SystemAcessImpl	57	9	0	0	√
proteus.runtime.decision.concrete.GraphTransporter	39	6	0	0	√
proteus.run...sion.concrete.SimplePartitionDecision	46	7	1	20	X
proteus.runtime.decision.concrete.TwoSet	255	15	0	0	√
proteus.runtime.decision.concrete.MinCutExpanded	203	9	0	0	√
proteus.runtime.decision.concrete.SimpleElection	77	3	1	50	X
proteus.....decision.concrete.SendBackNotification	28	6	0	0	√
proteus.runtime.decision.Objective	7	1	0	0	√
proteus.runtime.decision.Partition	43	9	0	0	√
proteus.runtime.decision.DevicesElection	3	1	0	0	√
proteus.runtime.decision.PartitioningAlgorithm	5,4	3	2	300	X
proteus.runtime.decision.PartitioningDecision	14	4	1	200	X
proteus.runtime.decision.PartitionFc	3	1	0	0	√
proteus.runtime.monitoring.Device	45	10	0	0	√
proteus.runtime.monitoring.Threshold	5	3	0	0	√
proteus.runtime.monitoring.MonitoringService	6,75	4			X
proteus.runtime.monitoring.DeviceObserver	3	1	0	0	√
proteus.runtime.monitoring.Observer	3	1	0	0	√
proteus...monitoring.concrete.MonitorService_imp	24,3	6	2	10	X
proteus.runtime.monitoring.concrete.EnergyEvent	45	6	1	5	X
proteus...concrete.resources.MemoryResource	14	3	0	0	√
proteus...concrete.resources.CpuResource	50	5			X
proteus...concrete.resources.ResourceTypes	5	0	0	0	√
proteus...concrete.resources.EnergyResource	5	1		100	X
proteus...SignalResource	0	1		100	X
proteus....monitoring.concrete.MonitorConstants	7	0	0	0	√
proteus.runtime.monitoring.concrete.MemoryEvent	100	8	0	0	√
proteus.runtime.monitoring.IntThreshold	*41*	*9*	0	0	√
proteus.runtime.monitoring.Resource	*8*	*2*	0	0	√
proteus.runtime.monitoring.Event	*24*	*9*	0	0	√
proteus.runtime.monitoring.NotifyStopMonitoring	16	5	0	0	√
proteus.runtime.monitoring.SignalEvent	4	1		50	X
proteus.runtime.NotFoundNodeException	5	2	0	0	√
proteus.runtime.ProteusORB	259	53	4	20	X
proteus.runtime.ProgramIdentifier	1,35	0	0	0	√
proteus.runtime.Converter	63	3	0	0	√
proteus.runtime.SystemAccess	68	12	0	0	√
proteus.runtime.distribution.io.ProteusInputStream	59	13	0	0	√
proteus....distribution.io.ProteusDatagramChannel	*4*	*1*			*X*
proteus...me.distribution.io.ProteusOutputStream	*35*	*9*	0	0	√
proteus.runtime.distribution.net.ProteusSocket	*208*	*50*	0	0	√
proteus.runtime.distribution.net.Datagram	*154*	*37*	*10*	*650*	*X*
proteus....distribution.net.ProteusServerSocket	*107*	*25*	0	0	√
proteus.runtime.distribution.NoArgsParam	3	1	0	0	√
proteus.runtime.distribution.ProteusStub	18	5	0	0	√
proteus.runtime.distribution.server.ProteusSkeleton	5	1	0	0	√
proteus....distribution.server.ProteusRemoteObject	24	6	0	0	√
proteus.runtime.distribution.ObjectFactory	8	1	0	0	√
proteus.runtime.distribution.RmeIdentifierFc	9	3	0	0	√
proteus.runtime.distribution.Barrier	43	5	0	0	√

Tabela 5.3: Classes implementadas - parte 3

Classes	Linhas de código	Número de funções	Funções a fazer	Linhas a serem feitas	Completa √/X
proteus.....notifications.LowBandwithNotification	5	3	0	0	√
proteus.runtime.notifications.EnergyNotification	19	5	0	0	√
proteus.runtime.notifications.Notification	1	0	0	0	√
proteus.....notifications.LowMemoryNotification	38	7	0	0	√
proteus.....ResourceLimintsChangedNotification	5	3	0	0	√
proteus.....notifications.HighMemoryNotification	38	7	0	0	√
proteus.....notifications.ResourceLimitsChange	5	3	0	0	√
proteus.runtime.ProteusOrbAccessor	116	32	4	20	X
proteus.DeviceFinderFc	*3*	*1*	0	0	√
proteus.ssecommunication....SSERemoteObject	47	5	0	0	√
proteus.ssecommunication...SSESkeletonDecorator	39	8	0	0	√
proteus.ssecommunication....SSERemoteRefFc	22	5	0	0	√
proteus.concrete.DeviceFinderSimulator	54	7	0	0	√
proteus.concrete.DeviceFdFc	7	2	0	0	√
proteus.exceptions.IncompatibleParameter	5	2	0	0	√
proteus.exceptions.ResourceNotAvailable	5	2	0	0	√
proteus.exceptions.LeavingDevice	3	1	0	0	√
proteus.exceptions.CommunicationUnavailable	3	1	0	0	√
proteus.exceptions.NotFoundException	5	2	0	0	√
proteus.exceptions.ProteusException	7	2	0	0	√
proteus.exceptions.AppKilled	3	1	0	0	√
proteus.DeviceFinder	*7*	*4*	0	0	√
proteus.Constants	3	0	0	0	√
instance.compiler.Main	19	5	0	0	√
instance.compiler.Configurator	18	1	0	0	√
proteus.util.SimulationFunction	45	0	0	0	√
proteus.util.Map	10	0	0	0	√
instance.runtime.Main	19	5	1	30	X
instance.runtime.Configurator	18	1	1	100	X

Tabela 5.4: Classes implementadas - parte 4

- *proteus.compiler.util.CodeInspection:* essa classe possui métodos relacionados para determinar e classificar estruturas no código, como laços, condicionais, parâmetros reais de funções, dentre outros. Ainda não são analisados *switchs* e blocos *try* e *catch,* como também nem sempre são determinados precisamente os blocos de laço e condicionais. Essa funções são utilizadas ao se pontuar a possível freqüência de execução de determinada instrução, analisando-se as estruturas onde está localizada a instrução;

- *proteus.runtime.graph.concrete.Node_imp:* Classe cuja instância representa um vértice do grafo em tempo de execução. A cada instância são associados objetos da aplicação para o particionamento e informações sobre o uso de recursos dos mesmos. Nessa classe es-

tão por terminar as funções e estruturas de dados que são acessadas pela aplicação para o armazenamento dos dados relacionados ao monitoramento da aplicação (funções relacionadas com a classe *proteus.compiler.util.MonitoringCode* descrita acima);

- *proteus.manager.concrete.ServerManagerImp:* essa classe provê o gerenciamento das várias aplicações, implementando as funcionalidades definidas pela classe *ServerManager*, que podem estar em execução. Além das dificuldades descritas na Seção 4.1.1, falta implementar as políticas que definem quantos recursos podem ser utilizados por cada aplicação*;*

- *proteus.runtime.manager.concrete.DistributedProgramManager:* essa classe implementa as funcionalidades do *LocalProgramManager e* gerencia a aplicação local. Nela é necessário incluir os tratamentos de condições de falhas na comunicação e erros reportados pela aplicação;

- os pacotes *proteus.runtime.distribution.io* e *proteus.runtime.distribution.net* são correspondentes aos pacotes de Java *java.io* e *java.net*, responsáveis por prover funcionalidades de comunicação e entrada e saída. E devem conter correspondentes das principais classes desses pacotes. Essa classes devem ser usadas pela aplicação no lugar das fornecidas pelo Java (trocadas pelo *proteus.compiler.util.MonitoringCode*) para monitorar as comunicações da aplicação.

As próximas classes que descrevemos foram simplificadas demasiadamente a fim de permitir a simulação de componentes e devem ser futuramente substituídas para permitir um teste mais realístico.

- *proteus.runtime.decision.concrete.SimplePartitionDecision:* essa classe implementa a classe *PartitionDecision* e implementa o algoritmo de decisão de particionamento. A implementação atual é bastante simples. O algoritmo utiliza dois valores de uso da memória: um valor máximo e um valor mínimo, que quando ultrapassados iniciam o processo de distribuição. Também utiliza um valor mínimo de energia, que ativa o processo de migração dos objetos para fora do dispositivo. Uma abordagem mais completa pode utilizar os níveis e a qualidade de vários recursos, como freqüência de erros de comunicação, uso do processamento, dentre outros. Regras lógicas mais elaboradas também podem ser utilizadas. O uso de lógica *Fuzzy* parece ser interessante, uma vez que os conceitos de recursos muito utilizados ou pouco utilizados podem ser bastante subjetivos;

- *proteus.runtime.manager.concrete.SimpleElection:* essa classe implementa a classe *DeviceElection,* sendo responsável por analisar os dispositivos do ambiente e retorná-los em

ordem de preferência para a distribuição da aplicação. Sua implementa utiliza apenas um critério para cada objetivo de decisão. Para o objetivo *desempenho,* ordenam-se os dispositivos em função dos recursos de processamento. Para *tolerância a falhas*, e *economia de energia*, ordenam-se os dispositivos pelo nível de bateria.

- *proteus.runtime.monitoring.concrete.resources.EnergyResource:* classe responsável por disponibilizar acesso ao nível de energia disponível. Esta classe não realiza a medição real do nível de energia do aparelho, apenas simula esses valores. Os valores são gerados a partir da interpolação linear de um conjunto de tuplas (x,t) fornecidos à classe, onde x é o nível de energia no intervalo [0,100], e t é o tempo de execução em milisegundos.

- *proteus.runtime.monitoring.concrete.resources.CpuResource:* classe responsável por disponibilizar acesso ao nível de uso da CPU. Esta classe não realiza a medição real de uso de processamento, apenas simula esses valores. Os valores são gerados a partir da interpolação linear de um conjunto de tuplas (x,t) fornecidos a classe, onde x é o nível de utilização da CPU no intervalo [0,100], e t é o tempo de execução em milisegundos.

- *proteus.runtime.monitoring.concrete.resources.SignalResource:* classe responsável por disponibilizar informações sobre a qualidade do sinal de comunicação sem fio. Sua função é medir a taxa de erros de comunicação. Esta classe não realiza a medição real, apenas simula esses valores. Os valores são gerados a partir da interpolação linear de um conjunto de tuplas (x,t) fornecidos a classe, onde x é a taxa de sucesso na comunicação entre os dispositivos no intervalo [0,100], e t é o tempo de execução em milisegundos.

Além das restrições discutidas a cima, ainda há as seguintes:

- não é possível uso de reflexão computacional. A reflexão computacional permite o acesso a qualquer objeto e classe da aplicação de maneira não explícita no código, o que dificulta a geração do grafo e análise do comportamento e estrutura da aplicação;

- não é mantida sincronização entre múltiplas *threads* quando distribuídas;

Como discutimos na Seção 4.1, existem diversas dificuldades para a implementação de um gerenciamento de recursos separado por aplicação em Java. Em vista disso e também por uma questão de prazo, nos restrigimos a implementar apenas o monitoramento de todas as aplicações como um conjunto único.

No estágio atual de desenvolvimento do Instrumentador, todo objeto móvel da aplicação se torna também remoto, mesmo que seja referenciado apenas dentro de um mesmo vértice. Podemos ter um problema de escassez de portas de comunicação, uma vez que uma aplicação

```
public class A{
    public A(){
        B b = new B();
        b.method2();
    }
    public void method1( B s)
    {...}
}
```

(a) Classe A

```
public class B{
    public B()
    {...}

    void method2()
    {...}

    static A method1()
    {
        return new A();
    }
}
```

(b) B

```
public class Main{
    public static void main(String args[]){
        A a = new A();
        Main n = new Main();
        n.method();
    }
    public void method(){
         A a = new A();
         A s = B.method1();
         B b = new B();
         a.method1(b);
    }
}
```

(c) Main

Figura 5.2: classes analisadas

pode possuir milhares de objetos, e o número de portas é limitado. Para diminuir o número de portas de comunicação utilizadas, não associamos uma porta a cada objeto como originalmente em RME, e sim compartilhamos uma porta com todos os objetos de um mesmo vértice do grafo criado durante a execução. Utilizamos o identificador do objeto para determinar qual objeto deve receber a chamada. Vale lembrar que essa forma de acesso é totalmente atrelada à aplicação e independente do SSE, o que permite, por exemplo, que pequenas aplicações utilizem uma porta para cada objeto.

A instância do Instrumentador implementada recebe, por linha de comando, os parâmetros para a análise e geração de código. O resultado principal da análise são: o grafo de tipos e o grafo representativo de objetos (ver Seção 3.2). Ambos são exibidos de forma gráfica. Atualmente, apenas é possivel visualizar os grafos. Futuramente, pretende-se que seja possível editá-los manualmente e assim permitir que desenvolvedor realize os ajustes que julgar necessário. Para facilitar a visualização em grafos maiores, uma saída do grafo é gerada para cada tipo de arestas.

(a) Arestas de exportação (b) Arestas de exportação (c) arestas de uso

Figura 5.3: Grafo de Tipos

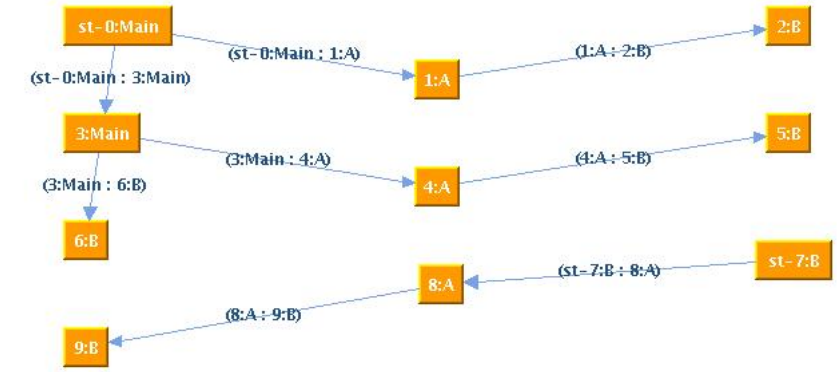


Figura 5.4: Grafo representativo de objetos: Arestas de criação

A Figura 5.3 exibe o grafo de tipos gerado pela análise das classes *Main, A* e *B* exibidas na Figura 5.2, onde são mostradas as arestas de exportação, importação e uso respectivamente. Cada vértice é rotulado com o nome do tipo que representa. As arestas de importação e exportação são rotuladas com os tipos que importam ou exportam. As arestas de uso exibem o valor que pontua a interação entre os vértices. As Figuras 5.4, 5.5, 5.6 exibem o grafo representativo de objeto gerado a partir do grafo de tipos e das classes *Main*, *A* e *B* exibidas na figura 5.2. Cada vértice é rotulado com um identificador único e nome dos tipos. Os vértices estáticos são exibidos com o prefixo *st.*

5.1 Conclusão

Neste capítulo, descrevemos a implementação de Proteus. Listamos suas classes e interfaces básicas, como também as classes que compõem uma instância simplificada do Arcabouço. Mostramos a dificuldade de implementação de cada uma e descrevemos futuros aprimoramentos.

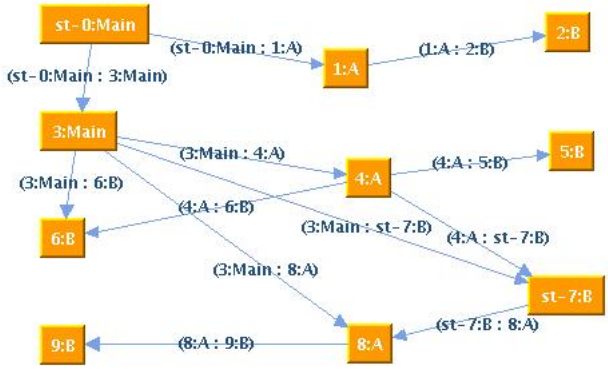


Figura 5.5: Grafo representativo de objetos: Arestas de referência

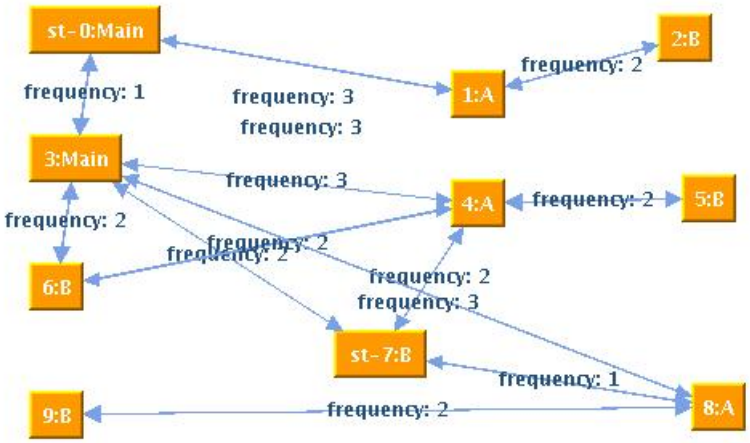


Figura 5.6: Grafo representativo de objetos: Arestas de uso

Para complementar e aprimorar o Instrumentador, precisamente o módulo gerador de código, caso se deseje realizar um grande número de modificações, recomendamos a utilização de uma biblioteca de manipulação de *bytecodes* alternativa à Javassit. Javassit mostrou-se ideal para aplicar pequenas modificações ou modificações não relacionadas, uma vez que esconde muitos detalhes de formatação dos arquivos binários Java. A dificuldade inserida por Javassit neste trabalho se deve ao fato de realizarmos muitas transformações e essas transformações estarem relacionadas. Javassit realiza verificações de integridade que não podem ser temporariamente falsas durante o processo de transformação. Ao constatar tal violação, Javassit lança uma exceção e impede a aplicação da transformação. Por exemplo, ao se tentar referenciar uma classe que ainda não existe ou ao se tentar redirecionar as chamadas a métodos, os quais já não existem mais na classe. Estas restrições tornaram complexa a implementação, pois geraram a necessidade do particionamento de várias transformações em etapas, que apesar de relacionadas e disponibilizadas por uma mesma classe, devem ser aplicadas de forma menos natural e intercaladas com outras transformações. A existência de dependências cíclicas entre as transformações geram a necessidade de se inserir e posteriormente remover entidades temporárias com a finalidade de garantir a integridade das referências. Também foi necessária a realização de alterações na própria biblioteca, possíveis graças ao seu código aberto. Certas modificações não eram diretamente permitidas, como por exemplo alterar o nome de uma classe sem que se altere todas as referências ao nome antigo. Assim, sugerimos que os próximos aprimoramentos sejam realizados utilizando-se alguma outra biblioteca, por exemplo, BCEL [Dahm, 2001].

A estrutura modular de Proteus permite de maneira simples que o gerador de código seja substituído. Apenas alterando-se sua fábrica, podemos utilizar diferentes geradores de código. A implementação do Sistema de Suporte à Execução disponibiliza os algoritmos discutidos no Capítulo 3. Devido às restrições de Java, a implementação do gerenciamento de múltiplas aplicações e recursos foram simplificados. Estas funcionalidades devem ser disponibilizados futuramente, assim que as novas funcionalidades de Java que estão em projeto forem disponibilizadas, conforme detalhado na Seção 4.1.1.

Capítulo 6

Conclusão

A mobilidade dos dispositivos, somada à sua diversidade, traz um conjunto de novas possibilidades de uso e de serviços dos recursos computacionais, permitindo formar um meio computacionalmente rico, interconectado, cooperante e dinâmico. Forma-se então o ambiente que denominamos de ambiente *pervasivo*. Todavia, para que essa realidade seja concretizada, diversos desafios devem ser vencidos para garantir a portabilidade e interoperabilidade das aplicações nesse universo heterogêneo.

Além da mobilidade física dos dispositivos, temos a mobilidade de código e/ou dados necessárias para disponibilização, adaptação e atualização dos serviços disponíveis nos dispositivos. Esta mobilidade gera questões de segurança e privacidade, pois para esses dados passam a trafegar pelo ambiente.

Diversos projetos [Geyer et al., 2006, Campbell et al., 2006, MIT, 2006, Zachariadis, 2005] avaliam diferentes aspectos desses desafios, que envolvem desde como perceber as mudanças e variações do ambiente, adaptar-se a elas, recuperar-se de falhas decorrentes dessas mudanças, e integrar-se a novos ambientes. Nesta dissertação, focamos em adaptar aplicações para aproveitar recursos do ambiente por meio da distribuição de seus componentes, quando estes encontram-se escassos em um dispositivo. Identificamos vários problemas de distribuição de objetos em ambientes móveis e dinâmicos. Vimos que sistemas distribuídos tradicionais pressupõem um ambiente de execução estacionário e confiável. As aplicações são iniciadas já distribuídas em uma estrutura pré-definida. Adaptar tais sistemas para ambientes dinâmicos não é aconselhável, principalmente por questões de confiabilidade, pois premissas em que esses sistemas se baseiam não são sempre válidas no ambiente *pervasivo*. Ao contrário de um ambiente estacionário, prever em que condições as aplicações serão executadas, torna-se uma tarefa mais árdua, se não impossível. É necessário, que essas condições sejam avaliadas, e as transformações necessárias aplicadas em tempo de execução. Para se realizar decisões em tempo de execução, necessitamos

de dados sobre esse ambiente e algoritmos para tomadas de decisão. Sabemos que esta decisão deve ser tomada em tempo hábil para não comprometer o tempo de resposta da aplicação ao usuário, e também para que o ambiente não mude de maneira que os dados sobre os quais a decisão é tomada se tornem inválidos. Como a capacidade computacional pode influenciar na qualidade dos resultados que um dispositivo pode computar em certo tempo, concluímos que se torna necessário o uso de algoritmos apropriados à capacidade de cada dispositivo, e que o modelo de gerenciamento deve permitir essa flexibilidade. Com a finalidade de permitir a exploração desse ambiente de forma flexível e eficiente desenvolvemos uma primeira versão de um arcabouço de distribuição que permite explorar estratégias de distribuição e gerenciamento, que, além disto, permite considerar a diferença de capacidades dos dispositivos e de requisitos de qualidade de serviço, tornando-o adequado ao ambiente *pervasivo*. Além do Arcabouço, foi desenvolvida uma Ferramenta, a ser aplicada antes da execução, para adicionar a capacidade de distribuição em aplicações centralizadas a partir da manipulação do seu código binário. Ambos, tanto o Arcabouço quanto a Ferramenta, foram desenvolvidas sobre a plataforma Java. Com isso atingimos os seguintes objetivos:

- separar a distribuição de seu gerenciamento, isto é, separar o que é distribuído (definido pela Ferramenta, e/ou otimizado pelo desenvolvedor), do como e quando distribuir (decidido pelo *middleware* derivado do Arcabouço em tempo de execução);
- permitir que aplicações determinem como se distribuir, otimizando assim a qualidade da distribuição;
- permitir o uso de algoritmos diferenciados em dispositivos com capacidades diferentes;
- permitir a avaliação de diferentes estratégias gerenciamento de distribuição;
- possibilitar ao usuário ou a aplicação solicitar certos níveis de qualidade de serviço;

Para permitir que uma aplicação seja distribuída, é adicionado um *overhead* em termos de processamento e memória, pois novas classes e objetos são criados para permitir a comunicação e para capturar informações durante a execução da aplicação. Assim, ela precisará de mais recursos do que a aplicação centralizada original, mas poderá aproveitar melhor os recursos do ambiente.

Além desse modelo de arquitetura também propusemos alguns algoritmos para avaliação:

- proposta de um algoritmo para particionamento de grafos. Expansão para uso em multi-dispositivos do algoritmo proposto por Xiaohui Gu et al. [Gu et al., 2004];

- proposta de descentralização do controle da distribuição. Ao invés de um dispositivo controlar e centralizar todas as informações, permitimos que cada um decida como reagir quando necessitar de mais recursos, gastando menos recursos com comunicação e tornando mais eficiente a decisão, uma vez que é tomada localmente;
- proposta de um algoritmo para geração do grafo usado em tempo de execução, permitindo um agrupamento dos objetos de maneira que aumente a qualidade da distribuição.

6.1 Trabalhos futuros

Durante o desenvolvimento do arcabouço, algumas importantes questões não foram avaliadas.

A primeira refere-se aos mecanismos de segurança que não foram avaliados. Quando se distribui aplicação, código e dados passam a se hospedar em dispositivos do ambiente. Durante esta situação temos a seguintes situações de risco:

- captura de informações durante a comunicação entre dispositivos;
- captura de informações pelo dispositivo hospedeiro;
- tentativa de causar algum dano ao dispositivo hospedeiro.

Além de impedir tais ocorrências também é importante a inserção de mecanismos de autenticação, que possibilitem limitar quais usuários ou que classes de usuários podem ter acessos a que dispositivos do ambiente. Brooks, em seu artigo "Mobile Code Paradigms and Security Issues" [Brooks, 2004], faz uma taxonomia das formas de mobilidade de código e suas vunerabilidades com mais detalhes.

A segunda questão trata dos mecanismos de localização e identificação dos dispositivos do ambiente também não foram avaliados. Diferentes tipos de rede requerem diferentes formas identificação, além de uma definição mais precisa do que é o ambiente do dispositivo, que pode ser, por exemplo, todos os dispositivos alcançáveis pela interface sem fio, como todos os dispositivos numa mesma sala, edifício ou célula de comunicação.

A Ferramenta ainda não possui suporte para manter a sincronização de *threads* distribuídas. Por exemplo, uma *thread* chama um método em um objeto remoto e este por sua vez chama um método em um objeto local. Para o sistema local é criada outra *thread* para tratar essa chamada remota, perdendo assim o acesso a monitores que se teria direito se os objetos fossem todos locais (chamados pela mesma *thread*). Uma solução para este problema é descrito por Eli Tilevich and Yannis Smaragdakis [Tilevich e Smaragdakis, 2004]. Um algoritmo de classificação mais preciso também deve ser adicionado, pois todas as classes móveis são declaradas remotas, e nem

sempre isso é necessário. Uma classe móvel, porém somente referenciada dentro dentro de um mesmo vértice, não precisa se tornar remota e pode ser referenciada diretamente. Com isto, ela é acessada de maneira mais eficiente (sem intermediários) e não é necessário requisitar ao SSE a sua criação.

Outro aprimoramento importante é adicionar suporte a migração forte (migração dos objetos e da pilha de execução). No estágio atual, aplicações que possuam objetos com métodos em execução (ativos) em um dispositivo não podem ser movidos. Se um dispositivo hospedeiro iniciar o processo de desconexão não será possível mover os objetos, e a aplicação terá que ser finalizada. Como é difícil determinar quanto tempo um método ficará ativo, isso pode impedir que um dispositivo que não deseje mais compartilhar seus recursos mova os objetos para outros dispositivos.

Por fim, é necessária complementar a Implementação descrita no capítulo anterior e realizar uma avaliação do seu desempenho e validar as hipóteses do modelo proposto.

Referências Bibliográficas

[Athanasiu et al., 2004] Athanasiu, L.; Raiciu, C.; Pandey, R. e Teodorescu, R. (2004). Using code collection to support large applications on mobile devices. In *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking*, pp. 16–29, New York, NY, USA. ACM Press.

[Augustin, 2004] Augustin, I. (2004). *Abstrações Para uma Linguagem de Programação Visando Aplicações Móveis em um Ambiente de Pervasive Computing*. PhD thesis, UFRGS.

[Bacon, 1998] Bacon, D. F. (1998). *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California.

[Bouchenak e Hagimont, 2002] Bouchenak, S. e Hagimont, D. (2002). Zero overhead java thread migration. Technical Report 261, L'Institut de Recherche en Informatique et Automatique.

[Brooks, 2004] Brooks, R. R. (2004). Mobile code paradigms and security issues. *IEEE Internet Computing*, 8(3):54–59.

[Busch, 2001] Busch, M. (2001). Adding dynamic object migration to the distributing compiler pangaea. Master's thesis, Freie Universitat Berlin.

[Campbell et al., 2006] Campbell, R. H.; Mickunas, D. M.; Reed, D.; Roman, M. e Hess, C. (2006). Gaia: Activespaces for ubiquitous computing. Último acesso em 30 de janeiro de 2006. Disponível em: http://gaia.cs.uiuc.edu/.

[Chen e Kotz, 2000] Chen, G. e Kotz, D. (2000). A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College.

[Chiba, 2006] Chiba, S. (2006). Javassist - a reflection-based programming wizard for java. . Último acesso em 30 de janeiro de 2006. Disponível em: http://gaia.cs.uiuc.edu/.

[Coulouris et al., 1996] Coulouris, G.; Dollimore, J. e Kindberg, T. (1996). *Distributed Systems - Concepts and Design*, volume 1. Addison-Wesley, 2 edição.

[Dahm, 2001] Dahm, M. (2001). Byte code engineering with the bcel api. Technical report, Institut fur Informatik, Freie Universit at Berlin.

[Diaconescu et al., 2003] Diaconescu, R. E.; Wang, L. e Franz, M. (2003). Automatic distribution of java byte-code based on dependence analysis. Technical Report 03-18, School of Information and Computer Science, University of California.

[Diaconescu et al., 2005] Diaconescu, R. E.; Wang, L.; Mouri, Z. e Chu, M. (2005). A compiler and runtime infrastructure for automatic program distribution. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, p. 52.1, Washington, DC, USA. IEEE Computer Society.

[Forman e Zahorjan, 1994] Forman, G. H. e Zahorjan, J. (1994). The challenges of mobile computing. *Computer*, 27(4):38–47.

[Foster, 2002] Foster, I. (2002). The Grid: A new infrastructure for 21st century science,. *Physics Today*, 55(2):42–47.

[Freeman et al., 2004] Freeman, E.; Freeman, E.; Bates, B. e Sierra, K. (2004). *Head First Design Patterns*. O' Reilly & Associates, Inc.

[Geyer et al., 2006] Geyer, C. F. R.; Augustin, I. e Yamin, A. C. (2006). Isam infra-estrutura de suporte às aplicações móveis. Último acesso em 29 de janeiro de 2006. Disponível em: http://www.inf.ufrgs.br/ isam/.

[Graves, 1990] Graves, R. (1990). *New Larousse Encyclopedia Of Mythology*. The Hamlyn Publishing Group, Yugoslavia.

[Gu et al., 2004] Gu, X.; Messer, A.; Greenberg, I.; Milojicic, D. e Nahrstedt, K. (2004). Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3(3):66–73.

[Jul et al., 1988] Jul, E.; Levy, H.; Hutchinson, N. e Black, A. (1988). Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

[Karypis, 2006] Karypis, G. (2006). Família metis de algoritmos de particionamento de múltiplo níveis. Último acesso em 10 de agosto de 2006. Disponível em: http://www-users.cs.umn.edu/ karypis/metis/.

[Karypis e Kumar, 1998a] Karypis, G. e Kumar, V. (1998a). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392.

[Karypis e Kumar, 1998b] Karypis, G. e Kumar, V. (1998b). Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129.

[Li e Hudak, 1989] Li, K. e Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359.

[Liogkas et al., 2004] Liogkas, N.; MacIntyre, B.; Mynatt, E.; Smaragdakis, Y.; Tilevich, E. e Voida, S. (2004). Automatic partitioning for prototyping ubiquitous computing applications. *IEEE Pervasive Computing*, 3(3):40–47.

[Loureiro et al., 2003] Loureiro, A. A. F.; Sadok, D.; Mateus, G. R.; Nogueira, J. M. S. e Kelner, J. (2003). Comunicação sem fio e computação móvel: Tecnologias, desafios e oportunidades. In *XXII Jornada de Atualização em Informática*, volume 2, pp. 195–244, Campinas, SP, Brazil. Sociedade Brasileira de Computação.

[Messer et al., 2002] Messer, A.; Greenberg, I.; Bernadat, P.; Milojicic, D. S.; Chen, D.; Giuli, T. J. e Gu, X. (2002). Towards a distributed platform for resource-constrained devices. In *ICDCS*, pp. 43–51.

[mFoundry, 2006] mFoundry (2006). Mojax framework. Último acesso em 21 de Dezembro de 2006. Disponível em: http://mojax.mfoundry.com.

[Microsoft, 2007] Microsoft (2007). .net framework develop center. Último acesso em 29 de Julho de 2007. Disponível em: http://msdn2.microsoft.com/pt-br/netframework/.

[Microsystems, 2006a] Microsystems, S. (2006a). Java 2 platform, micro edition (j2me). Último acesso em 29 de janeiro de 2006. Disponível em: http://java.sun.com/j2me/.

[Microsystems, 2006b] Microsystems, S. (2006b). Java tecnology. Último acesso em 20 de Dezembro de 2006. Disponível em: http://java.sun.com.

[Microsystems, 2006c] Microsystems, S. (2006c). Remote method invocation home. Último acesso em 16 de Junho de 2006. Disponível em: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[MIT, 2006] MIT (2006). Mit project oxygen. Último acesso em 30 de janeiro de 2006. Disponível em: http://www.oxygen.lcs.mit.edu/.

[Naveh, 2007] Naveh, B. (2007). Jgrapht: Um biblioteca que implementa estruturas e algoritmos da teoria de grafos. Último acesso em 25 de janeiro de 2007. Disponível em: http://jgrapht.sourceforge.net/.

[Nester et al., 1999] Nester, C.; Philippsen, M. e Haumacher, B. (1999). A more efficient rmi for java. In *JAVA '99: Proceedings of the ACM 1999 Conference on Java Grande*, pp. 152–159, New York, NY, USA. ACM Press.

[OMG, 2006] OMG (2006). Object management group. Último acesso em 21 de Dezembro de 2006. Disponível em: http://www.omg.org.

[OMG, 2007] OMG (2007). Catalog of omg corba/iiop specifications. Último acesso em 16 de Abril de 2007. Disponível em http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[Opera, 2006a] Opera (2006a). Opera platform: Enabling web applications on mobile phones. Último acesso em 20 de Dezembro de 2006. Disponível em: http://www.opera.com/products/mobile/platform/opera_platform_brochure.pdf.

[Opera, 2006b] Opera (2006b). Opera software. Último acesso em 20 de Dezembro de 2006. Disponível em: http://www.opera.com/.

[Ou et al., 2006] Ou, S.; Yang, K. e Liotta, A. (2006). An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In *Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications*, pp. 116–125, Washington, DC, USA. IEEE Computer Society.

[Pereira, 2003] Pereira, F. M. Q. (2003). Arcademis: Um arcabouço para construção de sistemas de objetos distribuídos em java. Master's thesis, UFMG.

[Pereira et al., 2006] Pereira, F. M. Q.; Valente, M. T. O.; Bigonha, R. S. e Bigonha, M. A. S. (2006). Arcademis: A framework for object-oriented communication middleware development. *Softw. Pract. Exper.*, 36(5):495–512.

[Philippsen e Zenger, 1997] Philippsen, M. e Zenger, M. (1997). Javaparty: Transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11):1225–1242.

[Process, 2006a] Process, J. S. (2006a). Jsr 121 java isolation api. Último acesso em 10 de Outubro de 2006. Disponível em: http://www.jcp.org/en/jsr/detail?id=121.

[Process, 2006b] Process, J. S. (2006b). Jsr 278 resource consuption management api for java me. Último acesso em 10 de Outubro de 2006. Disponível em: http://www.jcp.org/en/jsr/detail?id=278.

[Process, 2006c] Process, J. S. (2006c). Jsr 284 resource consuption management api. Último acesso em 11 de Outubro de 2006. Disponível em: http://www.jcp.org/en/jsr/detail?id=278.

[QUALCOMM, 2006] QUALCOMM (2006). Brew. Último acesso em 20 de Dezembro de 2006. Disponível em: http://www.qualcomm.com/brew.

[Rao e Minakakis, 2003] Rao, B. e Minakakis, L. (2003). Evolution of mobile location-based services. *Commun. ACM*, 46(12):61–65.

[Rocha et al., 2004] Rocha, J.; Domingues, M.; Callado, A.; Souto, E.; Silvestre, G.; Kamienski, C. e Sadok, D. (2004). Peer-to-peer: Computação colaborativa na internet. In *Minicursos SBRC*.

[Satyanarayanan, 2001] Satyanarayanan, M. (2001). Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17.

[Schmidt, 1997] Schmidt, D. (1997). Acceptor and connector: Design patterns for initializing communication services. *Pattern Languages of Program Design*, 3:191–229.

[Schmidt, 1995] Schmidt, D. C. (1995). *Reactor: an Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[Sekiguchi et al., 1999] Sekiguchi, T.; Masuhara, H. e Yonezawa, A. (1999). A simple extension of java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, pp. 211–226.

[Simon et al., 2006] Simon, D.; Cifuentes, C.; Cleal, D.; Daniels, J. e White, D. (2006). Java; on the bare metal of wireless sensor devices: The squawk java virtual machine. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pp. 78–88, New York, NY, USA. ACM Press.

[Smith, 1988] Smith, J. M. (1988). A survey of process migration mechanisms. *SIGOPS Operation System Review*, 22(3):28–40.

[Soldatos et al., 2007] Soldatos, J.; Pandis, I.; Stamatis, K.; Polymenakos, L. e Crowley, J. L. (2007). Agent based middleware infrastructure for autonomous context-aware ubiquitous computing services. *Computer Communications*, 30(3):577–591.

[Spiegel, 1998] Spiegel, A. (1998). Objects by value: Evaluating the trade-off. Technical report b-98-13, FB Mathematik und Informatik.

[Spiegel, 2002] Spiegel, A. (2002). *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin.

[Spyrou et al., 2004] Spyrou, C.; Samaras, G.; Pitoura, E. e Evripidou, P. (2004). Mobile agents for wireless computing: the convergence of wireless computational models with mobile-agent technologies. *Mobile Networks and Applications*, 9(5):517–528.

[Stoer e Wagner, 1997] Stoer, M. e Wagner, F. (1997). A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591.

[SuperWaba, 2006] SuperWaba (2006). Superwaba: The real power of mobile computing. Último acesso em 20 de Dezembro de 2006. Disponível em: http://www.superwaba.com.br/.

[Teodorescu e Pandey, 2001] Teodorescu, R. e Pandey, R. (2001). Using jit compilation and configurable runtime systems for efficient deployment of java programs on ubiquitous devices. In *UbiComp '01: Proceedings of the 3rd International Conference on Ubiquitous Computing*, pp. 76–95, London, UK. Springer-Verlag.

[Tilevich e Smaragdakis, 2002] Tilevich, E. e Smaragdakis, Y. (2002). J-orchestra: Automatic java application partitioning. Technical Report GIT-CC-02-17, 2002, Georgia Tech.

[Tilevich e Smaragdakis, 2004] Tilevich, E. e Smaragdakis, Y. (2004). Portable and efficient distributed threads for java. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pp. 478–492, New York, NY, USA. Springer-Verlag New York, Inc.

[Weiser, 2002] Weiser, M. (2002). The computer for the 21st century. *Pervasive Computing, IEEE*, 1(1):19–25.

[Zachariadis, 2001] Zachariadis, S. (2001). Towards a mobile computing middleware: A synergy of reflection and mobile code techniques. In *FTDCS '01: Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, p. 148, Washington, DC, USA. IEEE Computer Society.

[Zachariadis, 2005] Zachariadis, S. (2005). *Adapting Mobile Systems Using Logical Mobility Primitives*. PhD thesis, University of London.

[Zhang e Krintz, 2004] Zhang, L. e Krintz, C. (2004). Adaptive code unloading for resource-constrained jvms. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp. 155–164, New York, NY, USA. ACM Press.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo