UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Um *Middleware* Extensível para Disponibilização de Serviços em Ambientes Pervasivos

Emerson Cavalcate Loureiro Filho

Campina Grande, PB, Brasil
Julho de 2006

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Um *Middleware* Extensível para Disponibilização de Serviços em Ambientes Pervasivos

Emerson Cavalcate Loureiro Filho

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação (MSc).

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Modelos Computacionais e Cognitivos

Angelo Perkusich
Orientador
Evandro de Barros Costa
Orientador

Campina Grande, PB, Brasil
Julho de 2006

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

L892m 2006

Loureiro Filho, Emerson Cavalcante

Um middleware extensível para disponibilização de serviços em ambientes pervasivos/ Emerson Cavalcante Loureiro Filho. – Campina Grande, 2006

115f. il.

Inclui bibliografia

Dissertação (Mestrado em Informática) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadores: Angelo Perkusich e Evandro de Barros Costa

1– Computação - Ambientes Pervasivos 2– Computação - Middleware Extensível

CDU 004.382.731.75

"UM MIDDLEWARE EXTENSÍVEL PARA DISPONIBILIZAÇÃO DE SERVIÇOS EM AMBIENTES PERVASIVOS"

EMERSON CAVALCANTE LOUREIRO FILHO

DISSERTAÇÃO APROVADA EM 19.07.2006

PROF.ANGELO PERKUSICH , D.Sc
Orientador

PROF. EVANDRO DE BARROS COSTA , D.Sc
Orientador

PROF.DALTON DARIO SEREY GUERRERO , D.Sc
Examinador

PROF MARKUS ENDLER, Dr.
Examinador

CAMPINA GRANDE – PB

Resumo

Os diversos avanços nas tecnologias de *hardware* e redes sem fio vêm permitindo a concretização do paradigma conhecido por computação pervasiva, ou ubíqua. Nesse paradigma, a computação encontra-se embutida em objetos do dia a dia, como carros, televisores e roupas, de forma que a mesma possa integrar-se transparentemente às nossas vidas. É devido a esses avanços, portanto, que a pesquisa em computação pervasiva se vê hoje em ampla expansão, já que agora os primeiros ambientes e aplicações com suas características podem ser concebidos.

Nesse contexto, o desenvolvimento de aplicações voltadas à computação pervasiva envolve uma série de detalhes que não estão diretamente relacionados às mesmas. Comunicação via rede, aquisição de informações de contexto e segurança são alguns exemplos. No entanto, o mais óbvio, para o desenvolvedor, seria se concentrar inteiramente na lógica da aplicação, livrando-se de lidar com tais detalhes. É com esse propósito, portanto, que tem se utilizado infra-estruturas de *software* conhecidas por *middlewares*.

Nessa mesma linha de raciocínio, uma das abordagens que tem se utilizado no desenvolvimento de *middlewares* para computação pervasiva é a computação orientada a serviços. Além disso, características como ciência do contexto, descoberta de nós e evolução dinâmica aparecem também como elementos importantes no que se refere a tais *middlewares*. Dado isso, o objetivo principal deste trabalho consiste no desenvolvimento de um *middleware*, chamado de *Wings*, voltado à computação pervasiva, e que contemple as características citadas.

Abstract

The advances in hardware and wireless networking technology have enabled the realization of so-called paradigm of pervasive computing, also known as ubiquitous computing. In such a paradigm, computing is embedded in everyday objects like cars, televisions, and clothes, so that it can be transparently integrated into our lives. Due to such advances, research in the field of pervasive computing is increasing in a fast pace, as it is now possible to create the first environments and applications contemplating its characteristics.

Within this scope, the development of pervasive computing applications involves a set of intricacies not directly related to them. Networking communication, acquisition of context information, and security to name some. However, it is clearly more obvious to let developers concentrate entirely on the application logic, freeing them of dealing with such intricacies. With such an idea in mind, it has been proposed software infrastructures known as middlewares.

One of the approaches that has been successfully used in the development of middlewares for pervasive computing is the service oriented computing paradigm. Besides, features like context awareness, host discovery, and dynamic evolution also emerge as important elements concerning such middlewares. Given this trend, the main goal of this work is the development of a pervasive computing middleware, named of *Wings*, contemplating the mentioned features.

Agradecimentos

Agradeço, primeiramente, a meus pais, que sempre estiveram ao meu lado, me apoiando em vários aspectos da minha vida.

Agradeço também aos meus orientadores, Angelo Perkusich e Evando de Barros Costa, fundamentais nas várias etapas de desenvolvimento deste trabalho.

A Hyggo Almeida, Glauber Ferreira, Loreno Oliveira, Fred Bublitz, Nádia Milena, Elthon Allex (mais conhecido como Jesus), Leandro cabelo, Leandro bill, Mário Hozano, Willy Tiengo, Camila Nunes e Marcinho, com quem tive a oportunidade trabalhar em vários momentos do mestrado, contribuindo assim em diferentes aspectos da minha vida acadêmica.

Às várias reuniões do grupo nas quartas-feiras à noite no Bar e Restaurante Miúra, as quais contribuíram enormemente para o bom andamento deste trabalho e principalmente para o meu crescimento dentro do mundo acadêmico (acreditem, é verdade :-)).

Não poderia deixar de agradecer também à Cervejaria Bohemia Ltda., a qual foi fundamental para "refrescar" meus neurônios sempre que precisei.

Aos meus companheiros de gandaia, Thiago, Nilson, Argemiro e Leonildo, que me ajudaram bastante a manter meu bom humor frente às dificuldades e pressões enfrentadas ao longo do mestrado.

Ao café da Dona Inês, extremamente importante para me manter acordado ao longo desses dois anos de mestrado, principalmente durante as aulas.

À Aninha, que com toda a sua paciência também me ajudou muito ao longo desses dois anos.

Agradeço também à Capes, pelo apoio financeiro, mesmo com alguns atrasos no depósito do bendito dinheiro.

É provável que existam outras pessoas a serem citadas aqui, mas, por disfunção de minha memória, infelizmente as mesmas acabaram ficando de fora. A essas pessoas, minhas sinceras desculpas.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

"É sempre o início que requer o maior trabalho." (James C. Penney)

A computação encontra-se diante de uma importante revolução. Um exemplo disso é a miniaturização de componentes eletrônicos como memórias e processadores, permitindo assim a construção de dispositivos pequenos e portáteis, como os atuais telefones celulares, *PDAs*[1], *tablet PCs* e *Internet tablets*. Outro ponto interessante nesse escopo foi o aumento do poder de computação desses dispositivos. Essa característica possibilitou aos mesmos executarem aplicações cada vez mais complexas, como por exemplo, jogos eletrônicos, tanto em 2D quanto em 3D, aplicações de multimídia, dentre outras.

Analisando essa mesma revolução, agora sob o prisma das redes de comunicação, não podemos deixar de mencionar o fato das interfaces de rede terem evoluído até as soluções sem fio encontradas hoje em dia. Dentre essas soluções, *Bluetooth*[2] (Bray & Sturman, 2000) e *Wi-Fi*[3] (Reid & Seide, 2002) estão com certeza entre as mais populares. Um outro ponto que vale a pena observarmos, é o fato de que algumas delas são embutidas com técnicas sofisticadas para economizar energia. Tal fato criou condições bastante favoráveis para a disseminação de dispositivos móveis, já que estes funcionam à base de baterias com capacidade limitada, e portanto, há a necessidade de se prolongar ao máximo a duração das mesmas.

[1] *Personal Digital Assistants*

[2] http://www.bluetooth.org

[3] *Wireless Fidelity* (http://www.wifialliance.org)

Diante de todos esses avanços, é notável a popularidade alcançada pelos dispositivos móveis. Basta dar uma boa olhada ao redor para se perceber, por exemplo, que grande parte das pessoas carrega consigo um telefone celular. Colocando isso em números, só as redes de telefonia móvel baseadas na tecnologia *GSM*[4] contabilizam 1,67 bilhões de clientes em todo o mundo, espalhados por 213 países[5]. O uso de dispositivos móveis como *smart phones* e *PDAs* também tem crescido bastante em alguns setores. Esses últimos podem ser encontrados em soluções de *Supply Chain Management*[6], suporte a hospitais [7], dentre outras.

O que se pode perceber, diante de todos esses eventos, é que existe, de fato, uma migração da computação tradicional, baseada em computadores pessoais, para uma era de pervasividade, na qual diversos dispositivos eletrônicos estarão espalhados ao nosso redor, transparentemente integrados ao nosso modo de vida. É esse novo estágio da computação que os pesquisadores denominaram de *computação ubíqua*, hoje também conhecido como *computação pervasiva*. As idéias desse paradigma foram inicialmente expostas em 1991 por Mark Weiser (Weiser, 1991), então pesquisador do Centro de Pesquisas da Xerox em Palo Alto (*Xerox Palo Alto Research Center*). Em sua visão, Weiser prega, de forma geral, um mundo no qual a computação está embutida em objetos do dia-a-dia, como televisores, carros e roupas, estando os mesmos integrados às nossas vidas. Mais precisamente, tais objetos comunicam-se transparentemente uns com os outros, para nos apresentar informações e recursos a qualquer hora e em qualquer lugar, de acordo com nossas necessidades e preferências.

Observando mais profundamente o paradigma concebido por Weiser, uma importante característica que se pode perceber é que, em ambientes de computação pervasiva (i.e., ambientes pervasivos), qualquer dispositivo é um cliente ou provedor de recursos em potencial. Como conseqüência, esses ambientes tornam-se um repositório dinâmico de recursos, todos disponíveis aos usuários móveis através de seus dispositivos.

No entanto, em ambientes pervasivos, os nós podem entrar e sair a qualquer momento. Isso implica que a disponibilidade dos recursos torna-se não só dinâmica mas também imprevisível, à medida que mais e mais nós entram e saem de um ambiente. Esse fato causa

[4] *Global System for Mobile Communication*

[5] Fonte: *ComputerWorld*, 13 de Fevereiro de 2006 (http://computerworld.uol.com.br)

[6] Fonte: *SAP Global* (http://www.sap.com)

[7] Fonte: *Mimosa Medical Information Systems* (http://www.minosa.com)

grande impacto na maneira como os recursos são descobertos e acessados. Isso se deve ao fato de que, em ambientes dinâmicos, como os de computação pervasiva, há grandes chances de que um recurso não esteja mais disponível em um dado momento, pois seu provedor pode ter sido desligado ou simplesmente saído do ambiente. Um outro motivo pode ser o fato do dispositivo cliente ter migrado para um ambiente no qual o recurso não esteja acessível. Em situações desse tipo, as aplicações clientes precisam buscar, dentre os demais recursos, aqueles que podem ser utilizados para substituir os que não estão mais disponíveis.

É nesse ponto então que a abordagem da *computação orientada a serviços* aparece como uma solução essencial (Zhu, Mutka, & Ni, 2005). Mais precisamente, a idéia é utilizá-la para ligar dinamicamente recursos (chamados de *serviços*) e aplicações cliente (Bellur & Narendra, 2005), possibilitando, portanto, alterar essas ligações em tempo de execução. Em outras palavras, uma aplicação pode, por exemplo, utilizar um serviço *A* e posteriormente trocá-lo por um substituto *A'*, mesmo quando em execução.

Embora a abordagem de serviços seja considerada promissora no contexto da computação pervasiva, são necessárias ainda infra-estruturas de software (i.e., *middlewares*) que sirvam de suporte aos desenvolvedores de aplicações. Isso permitiria que os mesmos se concentrassem exclusivamente na lógica da aplicação, livrando-os de detalhes de mais baixo nível, como protocolos de comunicação e disponibilização de serviços, dentre outros. Como conseqüência, diminui-se consideravelmente não só o tempo de desenvolvimento das aplicações mas também a incidência de erros nas mesmas (Mascolo, Capra, & Emmerich, 2002).

1.1 Problemática

No escopo do que foi discutido até agora, é importante ressaltar o fato de que, diversas soluções podem ser utilizadas com o propósito de estabelecer uma rede para a disponibilização de serviços (i.e., uma rede de serviços). *Bluetooth*, *JXTA* [8] (Gong, 2001), *Zeroconf* [9] (Guttman, 2001) e *Web Services* [10] são alguns exemplos dessas soluções. Portanto, à medida que um nó se move entre diferentes ambientes, o mesmo pode se encontrar diante de diferentes redes de serviços. Como exemplo, enquanto em um certo ambiente os serviços são dispo-

[8] http://www.jxta.org

[9] http://www.zeroconf.org

[10] http://www.w3.org/2002/ws

nibilizados através de *Jini* (Waldo, 1999), em outro, isso pode ser realizado utilizando-se *UPnP* [11]. Note ainda que, mesmo em um único ambiente, diferentes redes para a disponibilização de serviços podem estar disponíveis. Permitir às aplicações pervasivas acessar serviços através dessas diferentes redes, possivelmente ao mesmo tempo, é certamente uma idéia interessante. Perceba que, com esse tipo de abordagem, poderia-se aumentar o número de serviços disponíveis às aplicações, permitindo, por exemplo, o acesso simultâneo a serviços disponibilizados através de *UPnP* e *Jini*, como ilustrado na Figura 1.1.

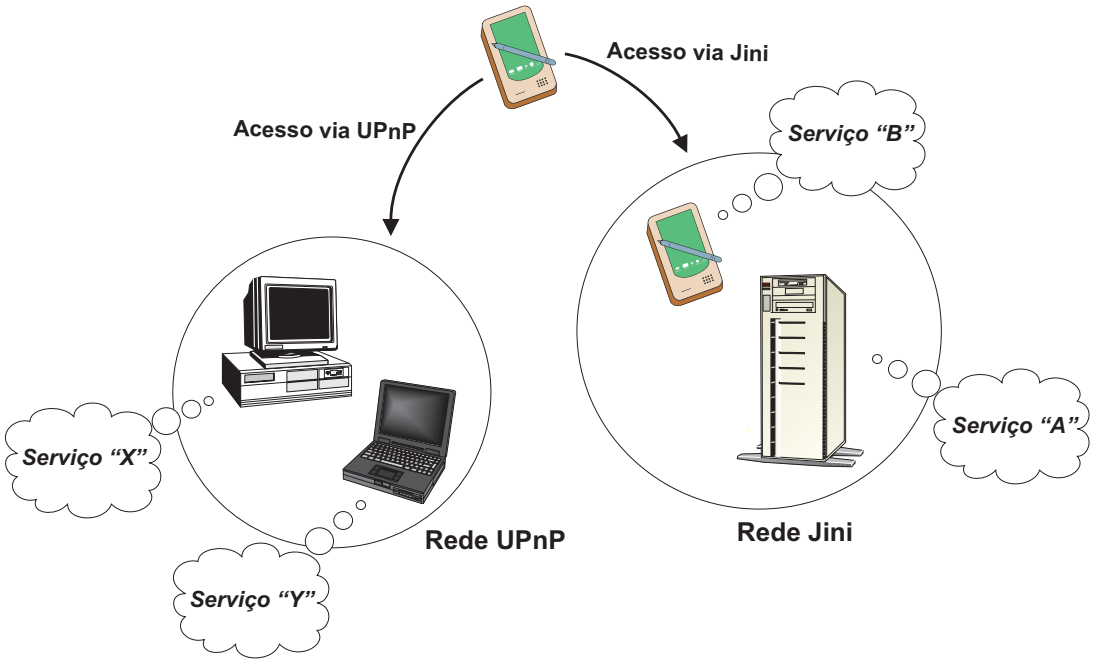


Figura 1.1: Acesso a serviços através de diferentes redes.

Embora a idéia de disponibilizar serviços através de redes heterogêneas seja interessante, uma pergunta que podemos nos fazer é "quais os protocolos e tecnologias devem ser embutidos no *middleware* de forma a satisfazer as necessidades das aplicações executando sobre o mesmo?". Responder precisamente a essa pergunta é sem dúvida uma tarefa nada fácil, já que as soluções para disponibilização de serviços utilizadas podem variar bastante de aplicação para aplicação. Nesse escopo, uma possibilidade seria embutir no *middleware* diversas dessas soluções, ou possivelmente todas. Essa idéia, no entanto, se mostra ineficiente, primeiramente devido às restrições de armazenamento apresentadas por muitos dispositivos móveis. Além disso, é possível que algumas das soluções embarcadas nunca sejam real-

[11] *Universal Plug and Play* (http://www.upnp.org)

mente utilizadas, apenas desperdiçando espaço em disco. Um outro problema que merece destaque é o fato de que, novas soluções para disponibilização de serviços que venham a ser desenvolvidas não estarão disponíveis às aplicações, já que os protocolos com esse propósito são acoplados ao *middleware* em tempo de compilação. Uma abordagem mais interessante seria permitir a inserção e remoção dos mesmos sob demanda, como em uma arquitetura baseada em *plug-ins* (Birsan, 2005). Portanto, os protocolos não utilizados poderiam ser removidos sem maiores problemas. Da mesma forma, novas soluções, nesse sentido, poderiam ser adicionadas e disponibilizadas às aplicações sempre que preciso.

Um outro aspecto que deve ser observado refere-se a uma característica fundamental das aplicações pervasivas, a *ciência de contexto*. O conceito de ciência de contexto consiste, de forma geral, em disponibilizar às aplicações informações como a situação do usuário (e.g., dirigindo), o ambiente no qual o mesmo está localizado, suas preferências, dentre outras. É a partir dessas informações que as aplicações podem procurar e disponibilizar serviços relevantes aos seus usuários. O problema nesse caso é que, sem um suporte necessário, cada aplicação terá que implementar seus próprios mecanismos para acesso a essas informações. Dessa forma, as informações de contexto tornam-se acopladas demais às aplicações, não podendo, portanto, ser reutilizadas. É devido a isso que os *middlewares* de computação pervasiva devem disponibilizar os mecanismos necessários para que aplicações e serviços possam recuperar as informações de contexto que precisam. Isso gera, no entanto, um questão interessante. Como prever as informações de contexto que cada aplicação utilizará? Isso talvez não chegue a ser um problema em *middlewares* voltados a um domínio de aplicação específico (e.g., casas inteligentes). No entanto, se considerarmos os de propósito mais geral, torna-se difícil responder esta pergunta precisamente. É por isso então que não se deve fazer nenhuma suposição com relação às informações de contexto que um *middleware* disponibilizará. É mais óbvio, portanto, concebê-lo com um mecanismo de ciência de contexto extensível, na qual novas informações podem ser disponibilizadas sempre que preciso.

O último aspecto que gostaríamos de ressaltar refere-se ao conceito de invisibilidade descrito por Weiser (Weiser & Brown, 1995). Em linhas gerais, no escopo da computação pervasiva, esse conceito refere-se à transparência com a qual os dispositivos eletrônicos se integram com o nosso dia-a-dia. Em outras palavras, tal invisibilidade implica que a presença desses dispositivos nos ambientes deve passar praticamente desapercebida. Os mes-

mos devem então, se comunicar entre si e executar suas tarefas com o mínimo de intervenção humana, ou do contrário, sua presença não será tão desapercebida quanto o desejado. Não é muito difícil concluir, portanto, que um *middleware* de computação pervasiva deve, preferivelmente, realizar suas tarefas sem importunar seus usuários. Isso quer dizer que qualquer atualização no *middleware* deve causar o mínimo de impacto possível, não só no restante de seus componentes mas também nas aplicações, e possivelmente serviços, em execução. Os usuários não devem, portanto, ser requisitados a reiniciar o *middleware* ou suas aplicações e serviços só porque algum módulo foi inserido ou removido. Dentro do que foi exposto até então, essa é uma característica importante, pois em determinados momentos o mesmo precisará adicionar ou remover módulos para a disponibilização de serviços ou ciência de contexto. Dessa forma, o *middleware* deve ser capaz de evoluir dinamicamente, permitindo ser atualizado de forma transparente, tanto para as aplicações e serviços em execução quanto para os usuários.

1.2 Objetivos

Baseado nos problemas levantados na Seção 1.1, o objetivo deste trabalho é desenvolver um *middleware* para a disponibilização de serviços em ambientes de computação pervasiva, com suporte à ciência de contexto. Esse *middleware*, denominado de *Wings*, deve fornecer, portanto, suporte ao desenvolvimento de aplicações e serviços, à publicação, descoberta e utilização desses últimos em ambientes pervasivos e ainda à aquisição de informações de contexto. Vale ressaltar ainda que, a disponibilização de serviços deve ser realizada através de redes heterogêneas, pelos motivos discutidos anteriormente.

É também objetivo incorporar ao *middleware Wings* um mecanismo que permita sua atualização em tempo de execução. Para isso, será utilizada alguma das soluções atuais de suporte à evolução dinâmica. Tal solução, no entanto, deve ser compatível com as especificações nas quais o *middleware* será baseado, mais precisamente, *Java CLDC* [12] e *CDC* [13].

Além do que foi definido, colocamos ainda como parte dos objetivos deste trabalho a definição de uma arquitetura que contemple soluções para as questões aqui abordadas, e na

[12] *Connected Limited Device Configuration* (http://java.sun.com/products/cldc)

[13] *Connected Device Configuration* (http://java.sun.com/products/cdc)

qual o *Wings* será baseado, e a implementação de algumas soluções de disponibilização de serviços. Enquanto que a arquitetura servirá de base para implementações do *middleware* em outras linguagens, as soluções a serem desenvolvidas ajudarão a validá-lo em um estudo de caso.

1.3 Relevância

A computação pervasiva, embora ainda não esteja presente em nosso dia-a-dia, desponta como um promissor paradigma para as próximas gerações. Nesse escopo, a pesquisa tem o papel fundamental de investigar os conceitos da computação pervasiva, fazendo assim surgir soluções cada vez mais utilizáveis na vida real. É com esse pensamento que muitos avanços têm sido realizados nas diferentes linhas de pesquisa associadas à computação pervasiva, como tecnologias de rede sem fio, protocolos de comunicação, *middlewares*, dentre outras. Esse trabalho aparece, portanto, como uma contribuição a esta última.

Vale ressaltar ainda que, à medida que as aplicações enfrentam cenários cada vez mais dinâmicos e abertos, a busca por soluções mais flexíveis, como a abordagem orientada a serviços, cresce em importância. Como os cenários de computação pervasiva também apresentam essas características, contribuições em termos de flexibilidade para sistemas pervasivos são portanto de grande valor.

Além disso, como iremos mostrar ao longo do trabalho, apesar de existirem muitas soluções de computação pervasiva, não são encontradas em nenhuma delas todas as características aqui consideradas. Portanto, dentro do escopo que estamos trabalhando, isso abre espaço para a definição e implementação de uma solução mais completa.

1.4 Estrutura da Dissertação

O restante deste trabalho está organizado da seguinte forma:

- **No Capítulo 2**, apresentamos uma visão geral acerca da computação pervasiva. Mais precisamente, apresentaremos alguns conceitos básicos envolvendo as redes de comunicação e a ciência de contexto em computação pervasiva.

- **No Capítulo 3**, introduzimos alguns conceitos relativos à computação orientada a serviços e às abordagens para disponibilização de serviços.

- **No Capítulo 4**, apresentaremos o modelo de componentes *Compor*, o qual foi a solução de evolução dinâmica utilizada na implementação do *middleware Wings*. Descreveremos, portanto, como são estruturadas as aplicações baseadas nesse modelo e como o mesmo possibilita a evolução dinâmica das mesmas.

- **No Capítulo 5**, o *middleware Wings* é apresentado. Mais precisamente, detalhamos sua arquitetura, modelagem e implementação.

- **No Capítulo 6**, o estudo de caso escolhido para validar o *middleware Wings* é apresentado.

- **No Capítulo 7**, estão descritas algumas das atuais soluções no contexto deste trabalho.

- **No Capítulo 8**, são apresentadas as considerações finais bem como os trabalhos futuros.

Capítulo 2

Computação Pervasiva

"Esteja em todo o lugar, faça tudo e nunca falhe ao surpreender o cliente." (Margaret Getchell)

Como já discutimos no Capítulo 1, os recentes avanços nas tecnologias de *hardware* e redes sem fio, têm alavancado a criação dos primeiros, e experimentais, cenários de computação pervasiva. Devido à crença que esses cenários serão parte integrante de nossas vidas em um futuro próximo, a pesquisa nesse campo vem crescendo em um ritmo acelerado.

A realização dessas atividades de pesquisa, no entanto, requer identificar e entender os conceitos relativos ao paradigma da computação pervasiva. Embora a idéia do mesmo seja bastante simples, o entendimento desses conceitos é uma tarefa que envolve diferentes áreas de conhecimento. Redes de computadores, sistemas distribuídos e engenharia de software são algumas dessas áreas.

Neste capítulo, portanto, iremos identificar e discutir algumas características da computação pervasiva, através de diferentes perspectivas. Mais especificamente, nossa discussão está concentrada nas *redes de comunicação pervasivas* e na *ciência de contexto*. Antes disso, porém, proveremos uma visão inicial sobre a computação pervasiva.

2.1 Uma Visão Inicial da Computação Pervasiva

Imagine-se entrando em um *shopping*, portando um *PDA*. Imagine agora que, ao entrar, você sente uma súbita vontade de tomar um *cappuccino*. "Que pena!", você pensa, pois até aonde

você sabe, nenhum estabelecimento desse shopping oferece *cappuccinos*. Felizmente, seu *PDA* "sabe" que você gosta de *cappuccinos*. Além disso, o mesmo, interagindo com outros dispositivos existentes no local, "descobre" que uma loja de café foi aberta há poucos dias no *shopping*. O *PDA* descobre também que, para sua sorte, essa loja vende *cappuccinos*. Baseado nessa informação, seu *PDA* instantaneamente o notifica a respeito dessa novidade. "Isto é ótimo", você pensa, pois agora poderá saborear o tão desejado *cappuccino*. O *PDA* ainda é "capaz" de lhe informar a localização da loja de café, e dessa forma, você decide ir até lá. Tendo feito seu pedido, você agora espera pacientemente pelo *cappuccino*, quando se lembra de um livro que você há muito tempo procura. Sem hesitar, você retira seu *PDA* do bolso de forma a realizar uma busca pelas lojas do *shopping* que tenham o livro desejado. Quando a busca termina, você é notificado acerca de duas livrarias que vendem o livro em questão. Além disso, ainda são retornadas informações acerca dos preços do livro em cada uma das lojas, possíveis descontos e formas de pagamento. Com todas essas informações à mão, você então, do seu *PDA*, seleciona a melhor oferta e requisita o livro. Dessa forma, você provê todas as informações necessárias para concretizar a compra, como por exemplo, o número do seu cartão de crédito, a quantidade de parcelas, dentre outras. Agora, tudo o que você precisa fazer é passar na livraria e pegar seu mais novo livro.

Interessante não ? Apenas portando um dispositivo móvel você foi capaz de saborear um *cappuccino* e ainda comprar o livro que desejava. E não apenas isso; ambas as tarefas foram realizadas de forma bastante natural, como se a computação tivesse sido completamente mesclada ao seu dia a dia. Esse é um típico exemplo de um cenário da computação pervasiva, que como mencionado no Capítulo 1, foi idealizada por Mark Weiser (Weiser, 1991).

Weiser afirmou que a computação pervasiva poderia ser alcançada através de três elementos principais: dispositivos baratos e com baixo consumo de energia, infra-estrutura de rede sem fio, para permitir a comunicação entre esses dispositivos, e aplicações pervasivas. Na época dessas afirmação, a tecnologia de hardware não estava preparada para suportar o paradigma da computação pervasiva. As redes sem fio, como as que temos atualmente, ou não estavam disponíveis ou não se encontravam embutidas nos dispositivos móveis. Como conseqüência, aplicações pervasivas não podiam ser desenvolvidas.

Este cenário começou a mudar com a introdução de dispositivos móveis mais poderosos, em termos de capacidade de memória e processamento, o que os permitiu executar aplica-

ções mais complexas (Nogueira, Loureiro, & Almeida, 2005). Ainda, a partir do momento em que foram embutidas interfaces de rede sem fio nos mesmos, tornou-se possível o desenvolvimento das primeiras aplicações móveis. Foi devido a esses avanços que pudemos dar os primeiros passos em busca da visão de Weiser. Com isso, diversos esforços têm sido empregados recentemente no campo da computação pervasiva. Podemos citar, como exemplo de tais esforços, protocolos de comunicação como *UPnP*, *Zeroconf* (Guttman, 2001) e *Mobile IP* (Perkins, 1997), projetos de pesquisa como *Oxygen*[1], *Smart Space* (Stanford, Garofolo, Galibert, Michel, & Laprun, 2003) e *Portolano* (Esler, Hightower, Anderson, & Borriello, 1999) além de *middlewares* como *Aura* (Garlan, Siewiorek, Smailagic, & Steenkiste, 2002) e *Wings* (Loureiro, Bublitz, Barbosa, et al., 2006).

2.2 Redes de Comunicação Pervasivas

No âmbito das redes de comunicação pervasivas, o foco está nos detalhes envolvidos na comunicação entre dispositivos em ambientes pervasivos. Portanto, estudos nessa área variam desde a criação de interfaces de rede sem fio até o desenvolvimento de protocolos como os de transporte, roteamento e mobilidade. Dentro do escopo deste trabalho, *mobilidade*, *descoberta de nós* e *disponibilização de serviços* são elementos fundamentais para a criação de ambientes pervasivos. Discutiremos nesta seção os dois primeiros, enquanto que a disponibilização de serviços será discutida separadamente no Capítulo 3.

2.2.1 Mobilidade

O conceito de mobilidade está relacionado à possibilidade de um nó migrar entre diferentes redes, e ainda assim manter as conexões estabelecidas com os nós da rede de origem. Para um melhor entendimento, imagine uma pessoa que esteja de mudança. Nesse processo, dentre várias outras preocupações, essa pessoa deve querer certamente alterar os endereços de entrega de correspondências como revistas e faturas de cartão de crédito. Sendo assim, a mesma deverá então entrar em contato com as editoras das revistas e a companhia do seu cartão de crédito, para notificá-las que seu endereço mudou, e que portanto, as correspondências devem ser entregues no novo endereço. Do contrário, suas revistas e faturas de cartão

[1] http://www.oxygen.lcs.mit.edu

de crédito serão enviadas ao endereço antigo.

De uma maneira geral, isso é o que acontece em cenários caracterizados pela mobilidade de seus nós. Quer dizer, ao se mover de uma rede para outra, um nó deve disponibilizar seu novo endereço, na rede de destino, àqueles com os quais possui conexões estabelecidas. Dessa forma, conexões pendentes com outros nós podem ser restabelecidas. Essa possibilidade de disponibilizar comunicação aos dispositivos, mesmo em movimento, permite às aplicações trabalhar em segundo plano, procurando invisivelmente por serviços de interesse dos usuários, à medida que estes se deslocam entre diferentes ambientes. No entanto, aplicações desse tipo encontram-se diante de um novo conjunto de problemas, os quais podem ser agrupados da seguinte forma (Satyanarayanan, 1996).

- Limitação de recursos: é fato que os dispositivos móveis são limitados em termos de recursos, quando comparados com computadores pessoais. A velocidade dos processadores e a capacidade de memória e disco são consideravelmente maiores nestes últimos do que nos dispositivos móveis.

- Restrições de energia: enquanto que os computadores pessoais são ligados a uma rede elétrica, os dispositivos móveis, diferentemente, dependem de baterias. Sendo as mesmas de capacidade limitada, as aplicações móveis devem preferivelmente contemplar técnicas para economizar energia.

- Variabilidade dos enlaces sem fio: a qualidade das redes sem fio ainda é bastante variável, tanto em termos de desempenho quando de confiabilidade. Enquanto alguns ambientes disponibilizam conexões confiáveis e com razoável largura de banda, em outros isso não acontece. Essa variabilidade torna-se ainda mais evidente em ambientes abertos, nos quais o enlace sem fio pode ser compartilhado por diversos usuários em certos momentos, enquanto em outros por um número bastante reduzido dos mesmos.

- Segurança: devido à natureza de difusão (i.e., *broadcast*) dos enlaces sem fio, torna-se mais fácil interceptar mensagens nos mesmos do que nos enlaces cabeados. Portanto, se segurança já é um aspecto importante nestes últimos, em enlaces sem fio isso ainda é mais crítico.

A mobilidade, no entanto, além gerar os problemas descritos, tem um efeito direto sobre a estrutura da rede. Dessa forma, a flexibilidade da mesma deve ser proporcional à mobilidade de seus nós. Em redes *ethernets* cabeadas, por exemplo, os nós são estáticos. Portanto, apenas em situações esporádicas torna-se necessário alterar o endereço de rede de um nó. Nesse caso, protocolos como *DHCP*[2] resolvem transparentemente o problema de reconfiguração de endereço dos nós. No outro extremo estão as redes populadas por nós completamente móveis. Esse nível de mobilidade permite aos usuários se moverem para áreas que não possuem cobertura de rede. Em cenários desse tipo, redes ponto a ponto sem infra-estrutura física fixa, conhecidas como redes *ad hoc* (Chlamtac, Conti, & Liu, 2003), são mais apropriadas. Ou seja, os nós deveriam ser capazes de estabelecer conexões diretamente uns com os outros, sempre que preciso, sem depender de nenhuma infra-estrutura física.

Nesse escopo, como Sun and Sauvola (2002) já afirmaram, três modos de comunicação podem ser diferenciados, com relação ao grau de mobilidade: *nomádico*, *celular* e *pervasivo*. No primeiro modo nenhuma comunicação é necessária quando um nó está migrando de uma rede para outra. Um exemplo típico da comunicação nomádica é quando um usuário utiliza um computador portátil (i.e., um *notebook*) para se conectar a uma rede em seu trabalho e outra em sua casa. Perceba que, não há necessidade de manter as conexões de rede enquanto o usuário está se movendo do trabalho para sua casa. Já no modo de comunicação celular, a rede é organizada em células, adjacentes umas às outras. Além disso, cada célula tem um elemento central, o qual provê conectividade para todos os nós na mesma. Um nó pode, portanto, mover-se entre diferentes células e, mantendo contato com seus respectivos centralizadores, tornar-se acessível aos outros nós. As atuais redes de telefonia móvel são um exemplo desse modo de comunicação, nas quais as Estações Rádio Base (*ERBs*) atuam como centralizadoras. Por fim, o modo de comunicação pervasivo pode ser caracterizado tanto pela descentralização como pela falta de infra-estrutura de rede fixa, diferentemente dos outros dois modos. A rede é, portanto, formada espontaneamente, à medida que mais e mais nós se aglomeram em uma determinada área.

Dentre as atuais soluções no contexto de mobilidade podemos citar *Mobile IP* (Perkins, 1997), *GPRS*[3] e *Bluetooth*. Basicamente, os dois primeiros são soluções de mobilidade para

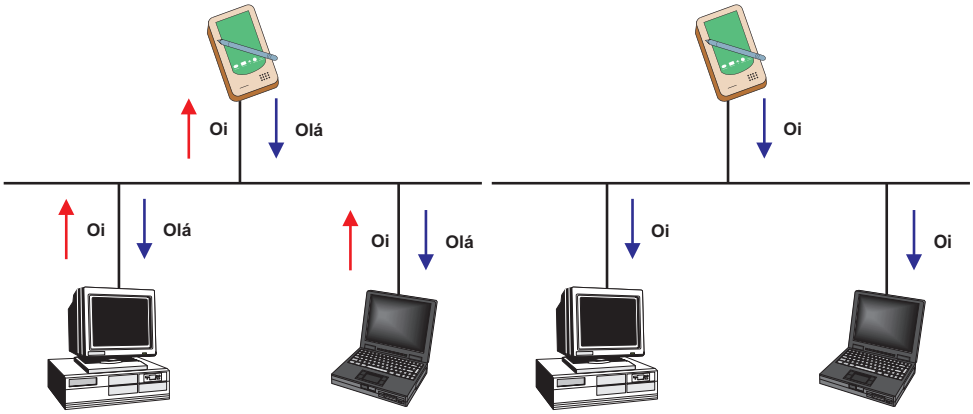
[2] *Dynamic Host Configuration Protocol*

[3] *General Packet Radio System*

redes *IP* e de telefonia móvel, respectivamente. *Bluetooth*, por outro lado, é uma tecnologia de baixo consumo de energia para a criação de redes *ad hoc* de curto alcance.

2.2.2 Descoberta de Nós

Colocando de maneira simples, a descoberta de nós está associada à capacidade de um nó em descobrir outros na rede, e de ser descoberto pelos mesmos. De uma maneira geral, esse processo pode ser realizado de duas formas: através de *consultas* e através de *notificações*. Na primeira, um nó envia mensagens de consulta de forma a descobrir os nós presentes na rede. Ao recebê-las, os mesmos podem então enviar uma mensagem de resposta ao requisitante, para informá-lo de sua presença na rede. Esse cenário é ilustrado na Figura 2.1(a). No outro modo de descoberta, ilustrado na Figura 2.1(b), um nó envia mensagens periodicamente para notificar aos demais que o mesmo está presente na rede. Note, portanto, que nessa abordagem nenhuma mensagem de resposta é necessária.



(a) Descoberta de nós utilizando *queries*. (b) Descoberta de nós utilizando notificações.

Figura 2.1: Abordagens para descoberta nós.

Esse conceito, apesar de simples, é bastante útil em ambientes de computação pervasiva, principalmente naqueles completamente descentralizados. Essa descentralização implica que os nós devem ser capazes de descobrir os demais no ambiente e manter conexão com os mesmos (L. Ma, 2005), para que possam então vir a fazer parte da rede. Dessa forma, um nó móvel poderia descobrir os dispositivos próximos para recuperar, por exemplo, os serviços e informações de contexto que mesmos disponibilizam (Loureiro, Oliveira, Almeida,

Ferreira, & Perkusich, 2005). Como mostraremos na Seção 7.1.1, esse é o mecanismo de descoberta de serviços utilizado pela tecnologia *Bluetooth*. Além disso, a descoberta de nós se torna bastante útil para nós recém chegados em um ambiente, seja o mesmo descentralizado ou não. Note que, em situações desse tipo, um nó não tem ciência de nenhum outro na rede. Portanto, o mesmo precisará primeiramente descobrir algum nó para que assim lhe seja permitido compartilhar e utilizar informações e serviços.

2.3 Ciência de Contexto

Como mencionamos anteriormente, uma funcionalidade fundamental de toda aplicação pervasiva é apresentar aos seus usuários informações e serviços relevantes, no lugar certo e na hora certa. Um turista, por exemplo, poderia ser informado sobre os principais pontos turísticos de uma cidade no momento em que saísse do avião, considerando suas preferências. Nesse processo, são necessárias informações como os interesses e preferências do usuário, o ambiente no qual o mesmo se encontra, os serviços disponíveis às aplicações, dentre outras. Essas informações, além de permitir às aplicações encontrar os serviços de maior interesse para os usuários, possibilita também determinar quais ações as mesmas devem tomar. Embora uma aplicação descubra que o calçado que você estava procurando acabou de chegar em uma das lojas do *shopping*, a mesma não deve perturbá-lo com essa informação se você estiver no cinema.

2.3.1 Definindo Contexto

Nossa discussão inicial nos dá pelo menos uma primeira impressão do significado real de contexto. Na literatura de computação pervasiva, contexto tem sido definido de diferentes maneiras, algumas delas buscando categorizar as diferentes informações relacionadas ao mesmo. Gwizdka (2000), por exemplo, identifica dois tipos de contexto, o *interno* e o *externo*. O primeiro está associado a informações referentes ao estado do usuário, como seu estado emocional. O contexto externo, por outro lado, se refere ao ambiente no qual o usuário está inserido, informando, por exemplo, sobre o nível atual de barulho ou iluminação do mesmo. Já no trabalho de Petrelli, Not, Strapparava, Stock, and Zancanaro (2000), dois tipos de contexto são identificados: *material* e *social*. O contexto material está associado com a

localização (e.g., em casa), dispositivos (e.g., *PDAs* e telefones celulares) ou infra-estrutura disponível (e.g., redes de comunicação). O contexto social, por sua vez, encapsula informações sobre a atual situação do usuário, por exemplo, "em reunião" ou "no cinema". Outro trabalho nessa mesma linha é o de Schilit and Theimer (1994), o qual define três categorias para agrupar informações de contexto: *contexto computacional*, *contexto do usuário* e *contexto físico*. Um refinamento dessas categorias é apresentado por Chen and Kotz (2000), através da adição de uma quarta categoria, *contexto temporal*. Algumas das informações relacionadas com essas categorias são apresentadas a seguir.

- *Contexto computacional*: largura de banda, recursos disponíveis no ambiente (e.g., impressoras, *displays* e estações de trabalho) e custos envolvidos na comunicação com outros dispositivos (e.g., consumo de bateria);
- *Contexto do usuário*: pessoas nas proximidades e localização, perfil e situação social do usuário;
- *Contexto físico*: níveis de iluminação, temperatura e barulho de um ambiente;
- *Contexto temporal*: hora do dia, dia da semana bem como mês e estação do ano.

É importante notar que nos trabalhos citados não se define realmente o que é o contexto em computação pervasiva. Ao invés disso, os mesmos tentam lhe dar um significado enumerando as informações relacionadas ao mesmo. Um problema com esse tipo de abordagem é que, em alguns casos, pode ser difícil afirmar quais informações fazem parte do contexto. Dessa forma, uma definição de propósito mais geral certamente permitiria entendermos melhor não só o contexto em si mas também seu papel na computação pervasiva. Portanto, no escopo deste trabalho, consideramos como contexto tudo aquilo que se encaixa na seguinte definição (Dey, 2001).

> "Contexto é qualquer informação que pode ser utilizada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto considerado relevante para a interação entre o usuário e a aplicação, incluindo o próprio usuário e a aplicação."

2.3.2 Aplicações Cientes do Contexto

Considerar o contexto atual para determinar as ações a serem tomadas ou as informações relevantes é algo bastante natural para nós. Normalmente utilizamos informações como o lugar e as pessoas ao redor para guiar nossas ações. Por exemplo, falar alto ou atender o telefone no cinema é bastante incômodo para os espectadores, e portanto, a maioria das pessoas evita tais ações nesse ambiente em particular. De uma maneira geral, aplicações que fazem uso desse tipo de informação para guiar suas decisões são chamadas de aplicações cientes do contexto. Isso requer, no entanto, mecanismos para adquirir e interpretar as informações de contexto, e baseando-se nisso, tomar decisões. Mais precisamente, três tarefas estão presentes nesse processo (Loureiro, Ferreira, Almeida, & Perkusich, 2006): *aquisição do contexto*, *representação do contexto* e *raciocínio sobre o contexto*.

Aquisição do Contexto

A aquisição de contexto está associada com a forma na qual as informações são obtidas, podendo ser *sentida*, *derivada* ou *explicitamente provida* (Mostéfaoui, Rocha, & Brézillon, 2004). A primeira é realizada a partir de sensores, como os de luminosidade e de temperatura. A aquisição derivada é computada sob demanda, como a hora do dia e o número de pessoas nas proximidades. Finalmente, no último tipo de aquisição, a informação de contexto é provida explicitamente pelo usuário à aplicação. Esta forma de aquisição de contexto pode ser encontrada, por exemplo, em aplicações que utilizam formulários para que os usuários os preencham com suas preferências (e.g., tipos de livros e filmes preferidos).

Representação do Contexto

Uma vez adquirida, a informação de contexto precisa agora ser disponibilizada às aplicações interessadas. Isso implica que a mesma deve ser representada em um formato específico, possibilitando assim que as aplicações possam "entender" a informação recebida. As soluções atuais para representação do contexto utilizam, por exemplo, pares chave-valor, documentos *XML* [4] (Boyera & Lewis, 2005; Ryan, 1999), modelos orientados a objetos (Henricksen, Indulska, & Rakotonirainy, 2002) e baseados em ontologia (H. Chen, Finin, & Joshi, 2003;

[4] *Extended Markup Language*

Masuoka, Labrou, Parsia, & Sirin, 2003; Henricksen, Livingstone, & Indulska, 2004)

Raciocínio Sobre o Contexto

Considerando que o contexto é representado em um formato que as aplicações compreendem, é possível agora fazer uso das informações disponíveis no mesmo, ou seja, realizar raciocínio sobre o contexto. De forma geral, esse processo consiste em utilizar eficientemente as informações de contexto, utilizando, por exemplo, cláusulas *if-then-else*, raciocínio baseado em casos (Nishigaki, Yasumoto, Shibata, Ito, & Higashino, 2005) e em regras (T. Ma, Kim, Ma, Tang, & Zhou, 2005), dentre outros mecanismos. Um exemplo de raciocínio sobre o contexto é, como já discutimos, a notificação de um serviço de interesse do usuário. Sendo uma peça fundamental na caminhada rumo à visão de Weiser (Satyanarayanan, 2001), essa característica permite às aplicações agir de forma personalizada, aumentando assim a satisfação do usuário com relação às mesmas (Tiengo, Costa, Tenório, & Loureiro, 2006).

Uma importante característica de sistemas que raciocinam sobre o contexto é a habilidade de lidar com informações imprecisas. Como muitas dessas informações são adquiridas através de sensores (i.e., informações sentidas), as mesmas estão sujeitas a erros, causados, por exemplo, por problemas na leitura da informação a partir do ambiente. Portanto, ao realizar esse tipo de raciocínio, as aplicações devem considerar a qualidade da informação adquirida. Com este propósito, diferentes abordagens têm sido propostas, utilizando técnicas como redes bayesianas (Gu, Pung, & Zhang, 2004) e lógica nebulosa (Ranganathan, Muhtadi, & Campbell, 2004).

Capítulo 3

Computação Orientada a Serviços

"Se você é flexível...existem realmente poucas coisas que você não conseguirá fazer em sua vida." (Anthony Robbins)

Tradicionalmente, aplicações distribuídas têm sido desenvolvidas definindo-se em tempo de compilação não só os componentes remotos que as integravam mas também os locais de onde os mesmos deveriam ser recuperados (Zhu et al., 2005; Bellur & Narendra, 2005). No entanto, à medida que essas aplicações começam a executar em cenários mais dinâmicos e abertos, essa abordagem mostra-se falha, devido à possível entrada, saída e indisponibilidade de nós nos mesmos. Por exemplo, na Internet, um nó contendo uma das partes integrantes de uma aplicação distribuída pode simplesmente sair do ar ou tornar-se inalcançável. Portanto, em situações desse tipo, a mesma não estará habilitada a executar, a menos que o nó em questão torne-se disponível novamente.

Nesse contexto, um paradigma que vem sendo considerado como o próximo passo no desenvolvimento de aplicações distribuídas é a *computação orientada a serviços* (Papazoglou, 2003a), a qual aparece como ferramenta essencial em áreas de pesquisa como computação autonômica (Kephart & Chess, 2003) e pervasiva (Zhu et al., 2005). Isso se deve ao fato de que, em tal paradigma, as relações entre as aplicações e seus componentes remotos integrantes são definidas em tempo de execução (Huhns & Singh, 2005), o que garante um alto grau de flexibilidade às mesmas. Isso permite, por exemplo, que uma aplicação utilizando inicialmente um recurso *X* o troque por um equivalente *X'*, mesmo em tempo de execução. Assim, em situações nas quais um determinado serviço não esteja mais disponível, pode-se

procurar por um substituto, de forma a integrá-lo dinamicamente à aplicação.

Sendo a computação orientada a serviços um dos principais conceitos envolvidos no desenvolvimento do *middleware Wings*, neste capítulo, portanto, apresentaremos uma visão geral sobre a mesma. Para isso, apresentaremos inicialmente os conceitos básicos envolvidos nessa abordagem. Posteriormente, ao falarmos de *arquiteturas orientadas a serviços*, descreveremos as entidades que compõem aplicações baseadas em serviços e como as mesmas interagem entre si para permitir a disponibilização de serviços. Finalmente, mostraremos ainda como se pode-se construir serviços mais complexos através da *composição de serviços*.

3.1 Visão Geral Sobre a Computação Orientada a Serviços

Na computação orientada a serviços, de maneira geral, pode-se dizer que aplicações distribuídas são construídas através da integração dinâmica de recursos remotos chamados de *serviços*. Nesse contexto, um serviço nada mais é do que uma entidade de software que implementa um conjunto de funcionalidades. Um serviço de biblioteca, por exemplo, poderia implementar funcionalidades para a busca de livros, por autor e título, registro de livros de interesse, dentre outras. Cada uma dessas funcionalidades, juntamente com os tipos de seus parâmetros e retorno, constituem a interface do serviço. Além disso, cada serviço possui uma *descrição*, a qual mantém informações sobre sua interface, nome, propósito de cada uma de suas funcionalidades, dentre outras. Como mostraremos mais à frente, é através dessa descrição que uma aplicação pode encontrar serviços em tempo de execução, para assim utilizá-los.

Deve-se observar, no entanto, que quatro operações são necessárias para que aplicações venham a utilizar serviços: *publicação*, *descoberta*, *seleção* e *ligação* (Papazoglou & Georgakopoulos, 2003). A publicação é o mecanismo que possibilita o compartilhamento de serviços com outros nós. Uma vez compartilhados, estes serviços podem então ser descobertos, utilizando para isso um conjunto de critérios (McGovern, Tyagi, Stevens, & Mathew, 2003), como algumas palavras chave representando a(s) funcionalidade(s) desejada(s). No entanto, mais de um serviço implementando tais funcionalidades pode ser descoberto. É nesse momento, portanto, que se faz necessária a operação de seleção, na qual apenas um serviço será

selecionado, dentre todos os que foram descobertos. Quando isso acontece, pode-se finalmente ligar-se ao serviço, para dessa forma utilizar as funcionalidades disponibilizadas pelo mesmo.

Um aspecto a ser considerado nesse contexto, é que a utilização das funcionalidades de um serviço pode ser realizada por um outro serviço. Assim, pode-se definir dois tipos de serviço: os *simples* e os *compostos*. Em linhas gerais, serviços simples são aqueles que não fazem uso de outros serviços na execução de suas funcionalidades. Os compostos, por outro lado, necessitam integrar certos serviços para executar algumas de suas funcionalidades. Abordaremos mais detalhadamente estes últimos na seção na Seção 3.3.

3.2 Arquiteturas Orientadas a Serviços

Na computação orientada a serviços, três entidades estão envolvidas nas operações de publicação, descoberta, seleção e utilização de serviços: *cliente*, *provedor* e *registro*. Essas entidades, juntamente com a descrição e *proxy* de um serviço, descrito mais à frente, constituem o que se conhece como uma *arquitetura orientada a serviços*. Em tais arquiteturas, o cliente é aquele que deseja utilizar um serviço, podendo ser uma aplicação ou mesmo outro serviço. O provedor, por sua vez, disponibiliza um conjunto de serviços, e suas respectivas funções, a clientes em potencial. Já o registro é onde os provedores publicam seus serviços, utilizando, para isso, a descrição dos mesmos. Com esse propósito, os provedores passam a esses registros a descrição de cada serviço a ser publicado, permitindo assim que os clientes as descubram, e conseqüentemente os serviços associados, em tempo de execução. Uma vez que isso ocorre, os clientes podem então ligar-se ao serviço, recebendo assim um *proxy* para o mesmo, o qual age como seu representante local. Esse *proxy*, portanto, retransmite todas as requisições dos clientes para o serviço, permitindo que estes o utilizem sem que saibam seu protocolo de comunicação. Essa comunicação com o serviço pode tanto ser realizada diretamente com seu provedor como através de um nó intermediário, o qual realiza uma série de conversões de forma a aliviar o processamento das requisições vindas dos clientes (Oliveira, Loureiro, Almeida, & Perkusich, 2006). O processo completo de publicação, descoberta, ligação e utilização de serviços é ilustrado na Figura 3.1, na qual são retratadas as entidades cliente, provedor e registro bem como as interações entre as mesmas. A partir desse ponto,

iremos nos referir a esse processo por *disponibilização de serviços*.

Em uma arquitetura orientada a serviços, o processo de disponibilização dos mesmos pode ocorrer de duas maneiras (Nickull, 2005) (Loureiro, Bublitz, Oliveira, et al., 2006), através do modo *pull* ou do modo *push*, os quais serão descritos nas seções seguintes.

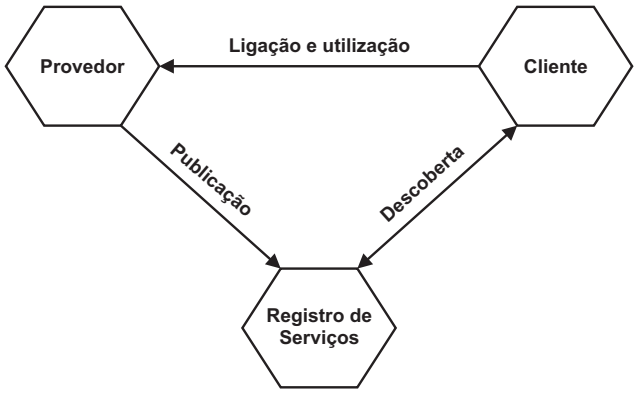


Figura 3.1: Arquitetura para a publicação, descoberta, ligação e utilização de serviços.

3.2.1 Disponibilização de Serviços Através do Modo *Push*

Nessa abordagem, os provedores publicam seus serviços diretamente aos clientes. Em outras palavras, os mesmos enviam as descrições dos serviços que desejam publicar para todos os nós da rede, estando estes interessados nos serviços ou não. Daí origina-se, portanto, o nome desse modo de disponibilização de serviços, *push*, em uma alusão à forma como as descrições dos serviços são "empurradas" aos demais nós. Sendo assim, sempre que tais descrições são recebidas, os mesmos as armazenam em um registro local, de forma que, uma vez que um cliente precise descobrir algum serviço, tudo que se precisa fazer é buscar sua descrição nesse registro. Um exemplo da disponibilização de serviços através do modo *push* é ilustrado na Figura 3.2. Veja que, nesse exemplo, o provedor, nó *A*, envia as descrições dos serviços a serem publicados diretamente para os nós *B* e *C*, como ilustrado no passo 1. No momento em que recebem tais descrições, *B* e *C* as armazenam em seus respectivos registros locais (passo 2). Dessa forma, se o nó *C*, por exemplo, precisar de alguma funcionalidade, este irá então buscar nesse registro por descrições de serviços que as implementem, como ilustrado no passo 3. Finalmente, considerando que essas descrições tenham sido encontradas, e que um serviço em particular tenha sido selecionado, o nó *C*

então solicitará ao provedor do mesmo (nó *A*) por um *proxy* (passo 4), permitindo-o, portanto, utilizar o serviço.

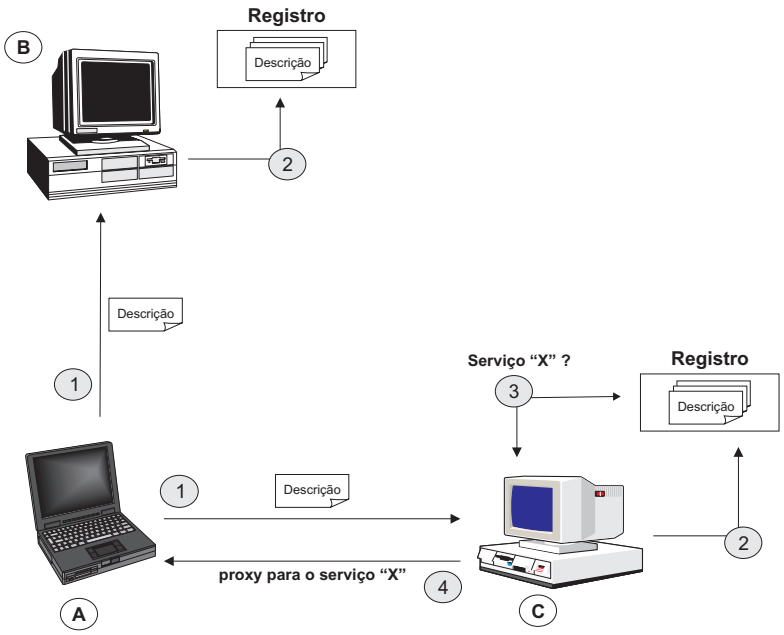


Figura 3.2: Esquema ilustrando a disponibilização de serviços no modo *push*.

3.2.2 Disponibilização de Serviços Através do Modo *Pull*

Nesse modo de disponibilização de serviços, as descrições dos serviços só são enviadas aos clientes no momento de descoberta. A denominação dessa abordagem, portanto, é uma tentativa de refletir a forma na qual um cliente "puxa" para si as descrições dos serviços que o interessa. Isso, no entanto, pode ser realizado utilizando tanto uma abordagem de registros centralizados quanto distribuídos, as quais serão abordadas a seguir.

Disponibilização Centralizada

Como seu nome indica, na disponibilização de serviços centralizada, os registros são mantidos em servidores específicos da rede. Dado isso, para publicar um serviço, o provedor precisa, inicialmente, descobrir pelo menos um desses servidores. Posteriormente, o mesmo determinará então em qual servidor, ou em quais, o serviço em questão será publicado. Certamente que, uma vez que tais servidores tenham sido encontrados, o passo inicial não é mais necessário, a menos que todos tornem-se inacessíveis. De maneira similar, na desco-

berta de serviços, o primeiro passo que o cliente deve realizar é descobrir os servidores que hospedam um registro de serviços. Feito isso, então, o cliente poderá procurar, dentre as descrições armazenadas pelos mesmos, por serviços que satisfaçam suas necessidades. Alguns exemplos de tecnologias para a disponibilização de serviços baseadas na abordagem *pull* centralizada são *Jini* e *Web Services*, os quais serão apresentados no Capítulo 7, onde são abordados outros trabalhos nesse contexto. Para um melhor entendimento sobre esse modo de disponibilização de serviços, apresentamos um exemplo simples, ilustrado na Figura 3.3. No primeiro passo, temos que tanto o nó *A* quanto o *B* publicam seus serviços em um servidor central *C* (estamos considerando que *A* e *B* já tenham descoberto o nó *C*). Com isso, as descrições de cada serviço publicado são mantidas em um registro de serviços gerenciado por *C* (passo 2). Sendo assim, considerando agora que o nó *D* já tenha encontrado o servidor central, o mesmo pode então descobrir os serviços de seu interesse (passo 3). Com isso, um conjunto de descrições será retornado para o nó *D* (passo 4), a partir do qual o mesmo pode selecionar um serviço em particular, interagir com seu provedor (nesse caso o nó *A*), para assim recuperar um *proxy* para o serviço em questão (passo 5).

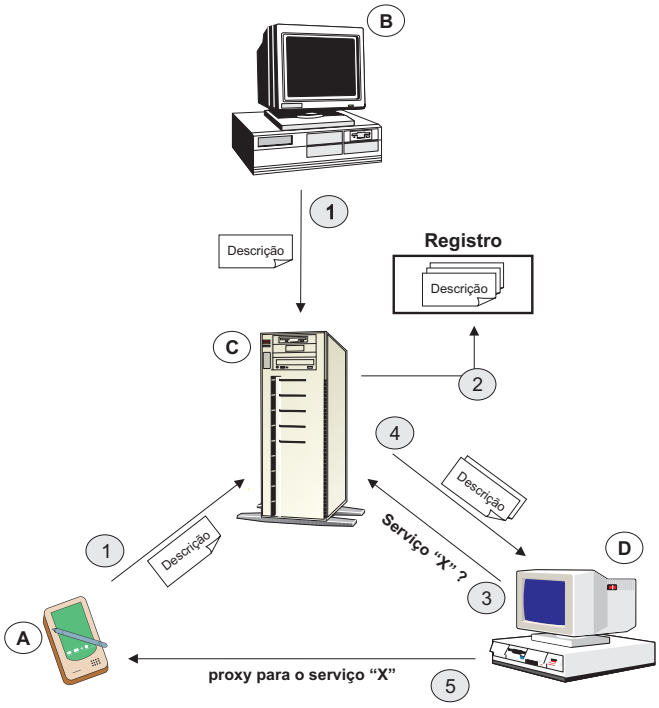


Figura 3.3: Esquema ilustrado a disponibilização de serviços no modo *push* centralizado.

Disponibilização Distribuída

Na disponibilização distribuída, a publicação de serviços é realizada em registros locais a cada provedor. Como se pode perceber, nesta abordagem, a tarefa de publicação é sem dúvida mais simples de ser realizada do que nas demais, já que as descrições dos serviços não precisam ser enviadas a registros remotos. Por outro lado, o processo de descoberta torna-se mais complexo, pois deve ser realizado em cada nó da rede, ou pelo menos até que um serviço relevante seja encontrado. Essa abordagem de disponibilização de serviços é ilustrada na Figura 3.4. No exemplo apresentado em tal figura, inicialmente cada nó publica seus serviços em registros locais (passo 1). Quando um cliente deseja descobrir algum serviço, no exemplo considerado, o nó *A*, o mesmo deverá buscar em todos os nós pelo serviço desejado (passo 2). É importante notar que é possível redirecionar as requisições de descoberta de serviço para nós que não estejam ao alcance do cliente. Na Figura 3.4 isso é realizado pelo nó *B*, quando o mesmo redireciona o pedido de *A* para *C*. Quando um dos nós possui um serviço de interesse do cliente, este receberá então sua descrição (passo 3). É a partir dessa descrição que o cliente pode, por fim, recuperar o *proxy* para o serviço, como ilustrado no passo 4.

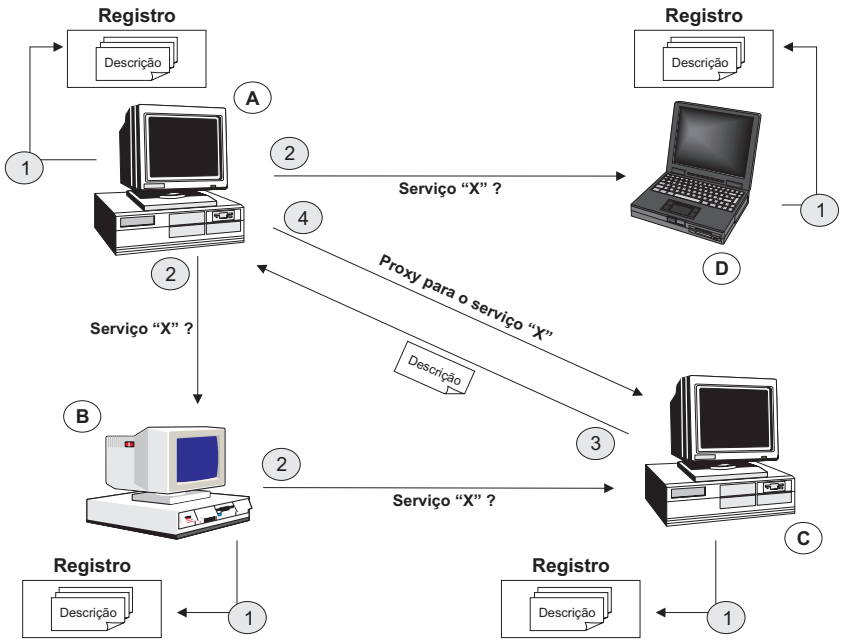


Figura 3.4: Esquema ilustrando a disponibilização de serviços no modo *push* distribuído.

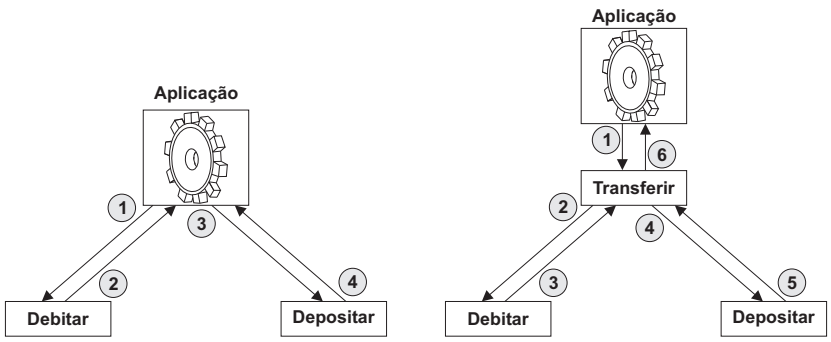
3.3 Construindo Serviços Mais Complexos: A Composição de Serviços

A idéia de integrar dinamicamente serviços de diferentes provedores é sem dúvida uma abordagem interessante para a construção de aplicações distribuídas. No entanto, em alguns casos, certas funcionalidades podem ser executadas através da coordenação de diferentes serviços. Tome como exemplo os serviços *debitar* e *depositar*, os quais, respectivamente, debitam e depositam uma quantia em dinheiro em uma conta bancária. Se em algum momento uma aplicação cliente precisasse realizar uma transferência de uma conta para outra, a mesma teria que debitar o dinheiro de uma conta e depositá-lo em outra, invocando os serviços *debitar* e *depositar*. Isso, é claro, não é uma solução desejável, já que a aplicação teria que coordenar manualmente a execução desses serviços. Mais precisamente, a mesma precisaria invocar o serviço *debitar*, esperar por seu retorno, invocar o serviço *depositar*, para finalmente retornar um resultado. Incluem-se ainda, em tal situação, possíveis erros ocorridos na invocação de um dos serviços, os quais também teriam que ser gerenciados pela aplicação.

E assim surge a necessidade de integrar e encapsular diferentes serviços em um novo, que é o que se conhece por composição de serviços. Voltando ao cenário descrito, poderia-se compor, por exemplo, um serviço *transferir*, utilizando para isso os serviços *debitar* e *depositar*. Assim, quando esse serviço fosse invocado, o mesmo debitaria certa quantia em dinheiro de uma conta, através do serviço *debitar*, para logo depois depositá-la em outra, através do serviço *depositar*. Não é difícil perceber, portanto, que devido a essa composição, toda a complexidade envolvida nesse processo fica completamente escondida do cliente (Cirne & Neto, 2005).

Essas abordagens para integrar e coordenar diferentes serviços (i.e., com e sem composição) são ilustradas respectivamente nas Figuras 3.5(a) e 3.5(b). Note que, na primeira, a aplicação lida diretamente com os serviços integrantes. Na segunda figura, por outro lado, a aplicação desconhece quantos e quais serviços são utilizados pelo serviço *transferir*. Além disso, possíveis erros que venham a ocorrer na invocação dos serviços *debitar* ou *depositar* são gerenciado internamente pelo serviço *transferir*. Assim, tudo que a mesma precisa fazer é invocar esse serviço e esperar por uma resposta, simplificando bastante, do seu ponto de

vista, a integração e coordenação de serviços.



(a) Sem composição de serviços. (b) Com composição de serviços.

Figura 3.5: Coordenação de serviços com e sem composição.

3.3.1 Abordagens para Composição de Serviços

De acordo com Yang, Papazoglou, and Heuvel (2002), a composição de serviços pode ser realizada de três formas: *fixa*, *semi-fixa* e *exploratória*. Na primeira delas, tanto a estrutura do serviço composto como seus serviços integrantes são fixos, ou seja, definidos em tempo de compilação. Por estrutura, ou *plano de composição*, entende-se como a lógica envolvida na execução dos serviços integrantes. Na composição semi-fixa, por sua vez, apenas a estrutura do serviço composto é definida em tempo de compilação. Quer dizer, embora os serviços integrantes não sejam especificados, determina-se *a priori* as sub-tarefas do serviço composto bem como a lógica de execução das mesmas. A composição exploratória, por fim, é a mais desafiadora de todas. Isso se deve ao fato de que, em tal abordagem, o plano de composição deve ser gerado automaticamente, por exemplo, a partir das necessidades do usuário. Apenas para ilustrar esse tipo de composição, considere um usuário que tenha em seu *PDA* um documento formatado no *Word*, e que deseja imprimir esse documento formatado em *PDF*. Na composição exploratória, essa tarefa seria automaticamente dividida, por exemplo, em duas sub-tarefas: gerar um documento *PDF* a partir do arquivo do usuário e posteriormente imprimi-lo. A partir daí, buscaria-se então por serviços que realizassem as sub-tarefas desejadas.

3.3.2 Transações e Serviços Compostos

Uma última questão que vale a pena elucidar, no contexto de serviços compostos, refere-se ao fato de que, em alguns casos, erros na execução dos mesmos podem levar à inconsistência dos dados gerenciados pelos serviços integrantes. Considerando novamente o exemplo descrito no início desta seção, imagine que, durante uma execução do serviço *transferir*, ocorra algum erro na invocação do serviço *depositar*. Note que, no entanto, o serviço *debitar* executou corretamente, de forma que o dinheiro, portanto, já foi debitado. Porém, como o serviço *depositar* não terminou de executar com sucesso, o dinheiro debitado seria "perdido".

É visando solucionar problemas desse tipo que tem-se utilizado o conceito de transações na execução de serviços compostos. O problema que acabamos de levantar, por exemplo, poderia ser facilmente resolvido utilizando-se o protocolo de transação de duas fases, o que garantiria a atomicidade do serviço *transferir*. Dessa forma, se houver erro na execução de algum de seus serviços integrantes, as operações realizadas até o ponto de falha serão desfeitas. Garante-se assim que, uma vez que o serviço *transferir* seja invocado, o mesmo não irá gerar inconsistência. Esse é, no entanto, apenas um dos diferentes cenários aos quais os serviços estão sujeitos. Em algumas situações, por exemplo, a garantia de atomicidade não é necessária (Papazoglou, 2003b), tornando necessária a utilização de outros protocolos de transação. Uma discussão mais aprofundada sobre esse tópico, no entanto, está fora do escopo deste trabalho.

Capítulo 4

O Modelo de Componentes *Compor*

"Não tema as mudanças, abrace-as." (Anthony J. D'Angelo)

Não é nenhuma novidade o fato de que, por muitas vezes, um *software* necessite de mudanças após seu primeiro *release*. Tais mudanças envolvem a correção de *bugs* ou ainda a adição, remoção e alteração de funcionalidades. A realização de atividades desse tipo tem recebido o nome de *evolução de software*, a qual vem ocupando o primeiro lugar nas despesas envolvidas no desenvolvimento de aplicações. Para se ter uma idéia, é estimado que entre 60% e 95% do custo de um *software* estejam associados a sua evolução (Lehman & Ramil, 2006).

Nesse escopo, uma das questões que vem sendo abordadas recentemente é a evolução de software não antecipada (Ebraert, Vandewoude, Cazzola, D'Hondt, & Berbers, 2005), ou evolução dinâmica, que consiste em permitir às aplicações serem modificadas em tempo de execução. Dessa forma, pode-se conceber um software com base em seus requisitos iniciais e evoluí-lo conforme os mesmos mudam, sem no entanto pará-lo ou reiniciá-lo. Esse conceito vem a ser fundamental para sistemas considerados críticos, como controladores de tráfego aéreo, processadores de transações financeiras, dentre outros.

É justamente com esse propósito que se definiu o modelo *Compor* (Almeida et al., 2006), uma solução baseada em componentes para a evolução dinâmica de software, inicialmente concebido como suporte ao desenvolvimento de sistemas multiagentes (Almeida et al., 2003). De forma geral, pode-se dizer que o modelo *Compor* permite a inserção, re-

moção e atualização de componentes em uma aplicação sem causar impacto nos demais, mesmo que estas operações sejam realizadas em tempo de execução. A utilização da abordagem de componentes nesse caso é providencial, pois permite que componentes de *software* pré-fabricados possam ser facilmente reutilizados, característica que tem levado soluções de diferentes campos a adotarem a abordagem de componentes (Ferreira et al., 2004; Sales et al., 2006). Não é difícil perceber, portanto, que o reúso de componentes pode acelerar consideravelmente o processo de evolução de um *software*.

A partir dessa discussão inicial, neste capítulo iremos apresentar o modelo de componentes *Compor*, bem como sua estratégia para permitir a evolução dinâmica de aplicações. Começaremos abordando os conceitos básicos de tal modelo, para a partir daí, detalharmos, como se dá a inserção, remoção e atualização de componentes em uma aplicação *Compor*. Por fim, abordaremos os dois mecanismos de interação entre componentes definidos pelo modelo, serviços e eventos.

4.1 Conceitos Básicos

O Modelo de Componentes *Compor* é baseado no padrão de projeto *Composite* (Gamma, Helm, Johnson, & Vlissides, 1995), como ilustrado na Figura 4.1, definindo duas entidades principais: *componentes funcionais* e *contêiners*. Componentes funcionais implementam as funcionalidades da aplicação, disponibilizando-as na forma de serviços. Além disso, componentes funcionais podem anunciar e receber eventos. Diferentemente, contêiners não implementam funcionalidades nem anunciam ou recebem eventos. Ao invés disso, os mesmos são compostos por componentes funcionais ou outros contêiners, gerenciando o acesso aos serviços providos por seus componentes filhos e redirecionando para os mesmos os eventos de seu interesse.

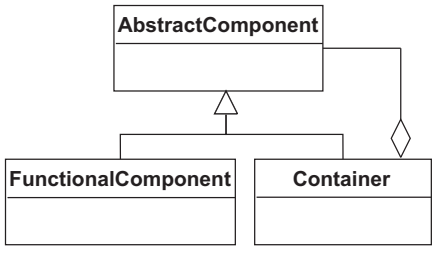


Figura 4.1: Principais entidades do modelo de componentes *Compor*.

Para possibilitar a composição de componentes em tempo de execução, o modelo *Compor* desacopla completamente os componentes funcionais entre si. Devido a isso, associa-se um identificador a cada serviço e evento disponibilizado pelos mesmos. Tal identificador deve então ser utilizado pelos demais componentes da hierarquia para invocar o serviço ou registrar-se como ouvinte do evento, sem tomar conhecimento de quem o provê. Como conseqüência, é possível trocar o provedor de um determinado evento ou serviço sem no entanto afetar os componentes restantes da aplicação.

Devido a esse desacoplamento entre componentes funcionais, cada contêiner precisa de duas informações: os *serviços providos* pelos seus componentes filhos e os *eventos de interesse* dos mesmos. Essas informações são armazenadas em duas tabelas, uma para os serviços providos e outra para os eventos de interesse, ambas atualizadas sempre que um componente é inserido ou removido da hierarquia. É com base nessas tabelas, portanto, que são realizadas as invocações de serviço e notificações de eventos. Com base nisso, detalharemos agora como é realizada a inserção e atualização de componentes bem como a invocação de serviços e notificação de eventos em uma aplicação *Compor*.

4.2 Disponibilização de Componentes

Já que os contêiners precisam estar cientes dos serviços providos e eventos de interesse de seus filhos, essas informações precisam ser atualizadas sempre que houver uma modificação na hierarquia. Portanto, sempre que um componente for inserido ou removido, as tabelas que guardam tais informações devem ser atualizadas, partindo do contêiner pai de tal componente, até a raiz da hierarquia. Com esse propósito, cada contêiner deve então recuperar os serviços providos e eventos de interesse do componente inserido/removido, para assim atualizar as respectivas tabelas. Esse processo pode ser melhor entendido através da Figura 4.2, que ilustra a inserção de um componente em uma hierarquia *Compor*. A remoção de componentes é similar à inserção, e dessa forma, não a ilustraremos aqui. Os passos presentes em tal figura são descritos a seguir.

1. Um componente, denominado de *X*, implementando o serviço *calcular*, é adicionado a *Contêiner 2*. Nesse ponto, as informações sobre os serviços providos e eventos de interesse do componente *X* serão recuperadas pelo seu contêiner pai.

2. De posse dessas informações, *Contêiner 2* atualiza sua tabela de serviços de forma a contemplar os serviços providos pelo componente recentemente adicionado. Note que uma das entradas de tal tabela relaciona o serviço *calcular* ao componente *X*. Embora não tenhamos ilustrado na figura, o mesmo aconteceria para a tabela de eventos, caso o componente *X* tivesse algum evento de interesse.

3. *Contêiner 2* irá então notificar ao seu contêiner pai, nesse caso o *Contêiner 1*, que o mesmo precisa atualizar suas tabelas de serviços e eventos.

4. Assim como no passo 2, *Contêiner 1* irá recuperar os serviços e eventos associados com o componente inserido para então atualizar suas as respectivas tabelas. Note que, como indicado na tabela de serviços de *Contêiner 1*, para o mesmo, *Contêiner 2* é o provedor do serviço *calcular*.

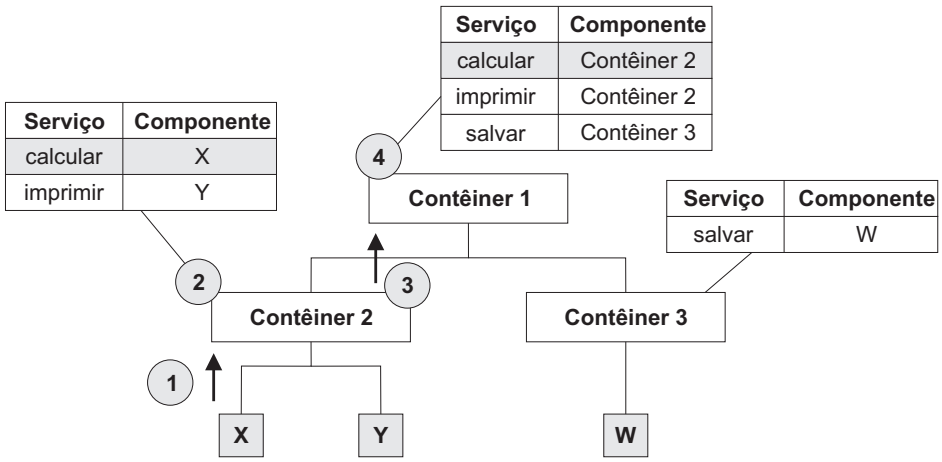


Figura 4.2: Inserção de componentes em uma hierarquia *Compor*.

Após a execução desses passos, os serviços e eventos disponibilizados pelo componente *X* podem ser acessados a partir de qualquer outro componente da hierarquia, sem nenhuma referência direta a *X* (observe que um componente só tem referência para o seu contêiner pai). Sendo assim, o componente *W* da Figura 4.2, por exemplo, pode invocar o serviço *calcular*, sem saber que *X* é o componente que o implementa. Esse componente *W* só precisaria, para isso, do identificador do serviço que o mesmo deseja invocar, nesse caso, "calcular". Essas mesmas idéias são aplicadas para um componente que deseja se registrar a um evento disponível na hierarquia.

4.3 Atualização de Componentes

A atualização de componentes em uma aplicação baseada no modelo *Compor* consiste, de maneira geral, em dois passos: i) inserção do novo componente na hierarquia e ii) remoção do componente antigo. Esses passos devem ser realizados exatamente nessa seqüência, pois do contrário, no intervalo entre as operações de remoção e inserção, um serviço do componente removido pode ser requisitado, não estando o mesmo mais disponível na árvore de componentes. Portanto, realizando a inserção antes da remoção, garantimos que os serviços providos pelo componente antigo continuam acessíveis, a partir do novo componente adicionado.

O processo completo de atualização de componentes em uma aplicação *Compor* é exibido na Figura 4.3, onde ilustramos a substituição do componente *Y* por um equivalente *Y'*. Note inicialmente que, qualquer invocação do serviço *imprimir*, ilustrada pela seta direcionada para baixo no quadro 1, é recebida pelo componente *Y*. A partir do momento em que um equivalente de *Y*, nesse caso *Y'*, é inserido na árvore de componentes (quadro 2), mais precisamente em *Contêiner 2*, a tabela de serviços do mesmo será atualizada, de forma a indicar que agora o serviço *imprimir* é implementado pelo componente *Y'*. Dessa forma, todas as requisições para o serviço *imprimir* serão então direcionadas para esse componente, e não mais para *Y* (observe a seta direcionada para *Y'* no quadro 2). O componente *Y* pode portanto ser removido da hierarquia, como ilustrado no quadro 3. Vale ressaltar que apenas as referências entre pai e filho são eliminadas, como ilustrado no quadro 4, e que portanto o componente removido pode continuar executando normalmente. Isso é útil quando, por exemplo, ainda existem invocações de serviço pendentes associadas ao componente removido, permitindo à aplicação aguardar até que todas estejam concluídas, para só então desativá-lo.

4.4 Invocação de Serviços

Como falamos anteriormente, a invocação de serviços, em uma aplicação baseada no modelo *Compor*, é baseada nos identificadores dos mesmos. Devido a isso, nesse processo, não se sabe *a priori* o componente que implementa o serviço requisitado. Portanto, na invocação de serviços, primeiramente é necessário realizar uma busca na árvore de componentes, com

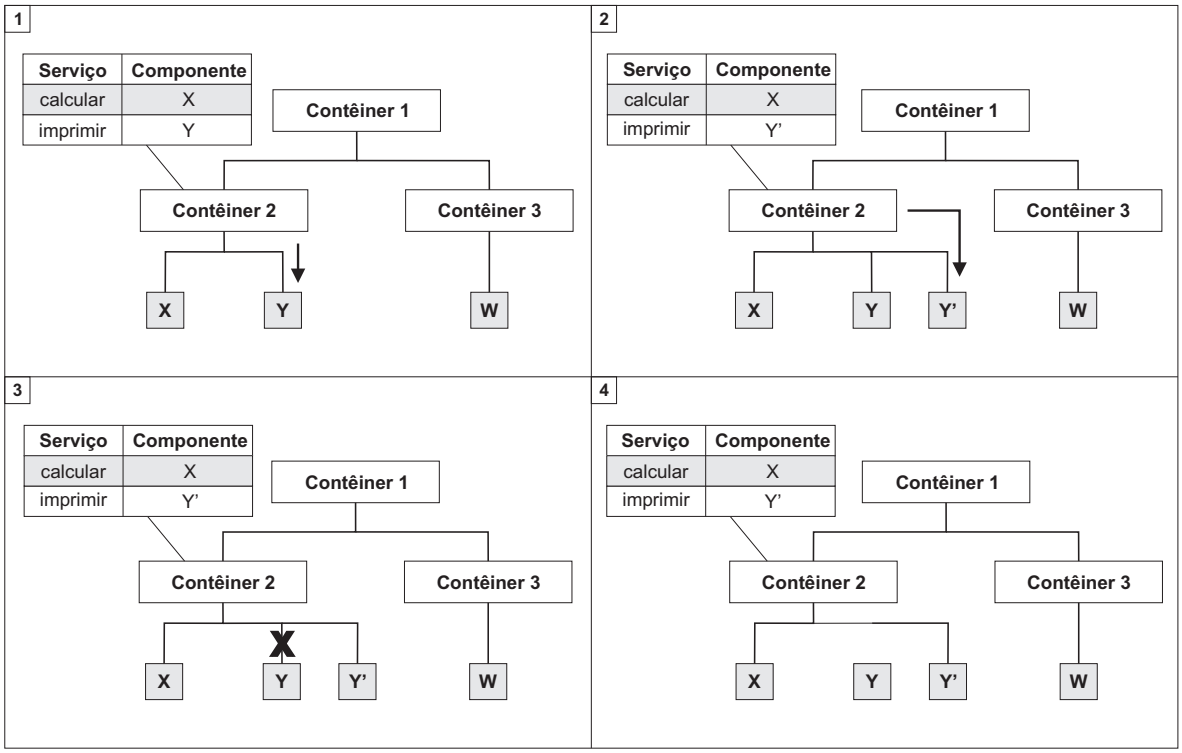


Figura 4.3: Atualização de componentes em uma hierarquia *Compor*.

o intuito de identificar o provedor do serviço que se quer executar. Essa busca inicia-se no contêiner pai do requisitante, subindo até a raiz da hierarquia, caso o provedor do serviço não seja encontrado em nenhum dos contêiners intermediários. Se assim for, o contêiner raiz re-direciona a busca para um de seus filhos, excluindo-se aquele de onde a mesma partiu, só parando no momento em que o componente funcional que implementa o serviço requisitado é encontrado. Uma vez que isso acontece, o serviço pode então ser invocado. Como exemplo, ilustramos na Figura 4.4 o processo de invocação de serviços em uma hierarquia de componentes *Compor*. Os passos apresentados em tal figura são descritos a seguir.

1. O componente *W* requisita a execução do serviço *calcular* ao seu contêiner pai, nesse caso *Contêiner 3*.

2. Baseado em sua tabela de serviços, *Contêiner 3* verifica que nenhum componente filho implementa o serviço *calcular*.

3. Dessa forma, *Contêiner 3* redireciona a requisição para o seu contêiner pai, *Contêiner 1*.

4. De acordo com sua tabela de serviços, *Contêiner 1* verifica que *Contêiner 2* implementa o serviço *calcular*. Portanto, para *Contêiner 1*, *Contêiner 2* é visto como o componente que implementa o serviço *calcular*.

5. *Contêiner 1* então redireciona a requisição do serviço para *Contêiner 2*.

6. *Contêiner 2* não implementa o serviço requisitado, mas, de acordo com sua tabela de serviços, tem uma referência para o componente que o implementa, nesse caso o componente *X*.

7. Sendo assim, *Contêiner 2* repassa a requisição para o componente *X*.

8. Ao receber a requisição, o componente *X* executa o serviço *calcular*, sendo seu resultado retornado para o componente *W*, seguindo o caminho inverso.

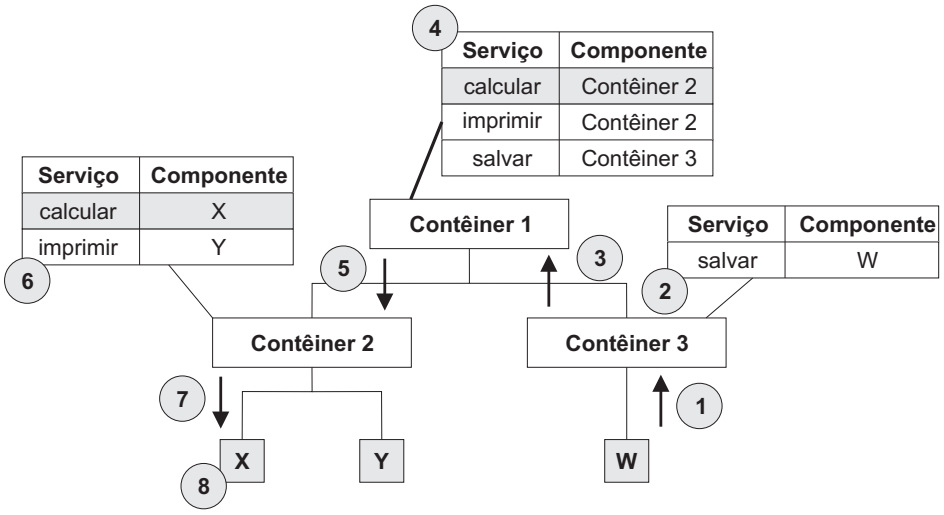


Figura 4.4: Invocação de serviços em uma hierarquia *Compor*.

4.5 Disparo e Notificação de Eventos

Para se disparar um evento em aplicações *Compor*, inicialmente, um componente precisa transmitir o mesmo ao seu contêiner pai. Este, ao receber o evento, irá verificar quais dos seus filhos estão interessados no mesmo, excetuando-se aquele que o transmitiu. O contêiner irá então notificar aos filhos interessados acerca do evento, posteriormente repassando o

mesmo para seu pai. Se o componente filho que receber o evento for um contêiner, este irá novamente verificar, dentre os seus filhos, quais estão interessados no evento, repassando-o para os mesmos, assim como no passo anterior. No entanto, neste ponto, o contêiner não transmite o evento para seu pai, pois se isso acontecesse a aplicação entraria em *loop*. Esse processo será repetido, pelo menos, até a raiz da hierarquia, para se ter certeza que todos os interessados recebam o evento, e continuará até que todos os interessados tenham sido notificados. Na Figura 4.5 apresentamos um exemplo da notificação de eventos, sendo os passos presentes na mesma descritos a seguir.

1. O componente *Y* dispara o evento *bateria baixa* para seu contêiner pai, *Contêiner 2*.

2. *Contêiner 2*, consultando sua tabela de eventos, verifica que o componente *X* está interessado no evento *bateria baixa*.

3. *Contêiner 2*, portanto, redireciona o evento para o componente *X*.

4. Além disso, *Contêiner 2* repassa para o seu contêiner pai, *Contêiner 1*, o evento anunciado pelo componente *Y*.

5. Dessa forma, *Contêiner 1* consulta sua tabela de eventos para verificar quais de seus filhos devem receber o evento, desconsiderando aquele que o transmitiu (*Contêiner 2*). Sendo assim, *Contêiner 1* verifica que apenas *Contêiner 3* deve receber o evento.

6. O evento é então repassado para *Contêiner 3*.

7. Ao receber o evento, *Contêiner 3* consulta a tabela associada e verifica que o componente *W* está interessado no evento *bateria baixa*.

8. Dessa forma, o evento é repassado para o componente *W*. Não tendo mais componentes interessados no evento *bateria baixa*, a notificação é, portanto, encerrada nesse ponto.

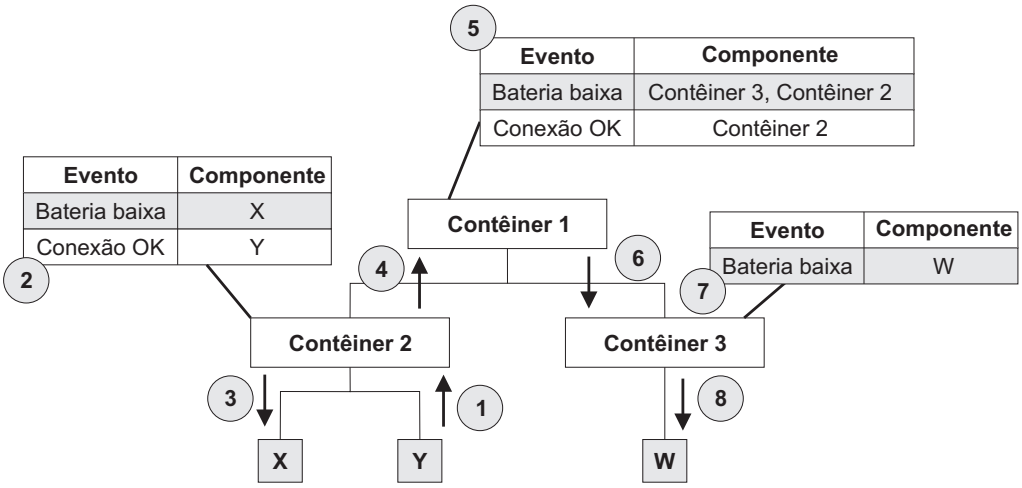


Figura 4.5: Notificação de eventos em uma hierarquia *Compor*.

Capítulo 5

O *Middleware Wings*

"Bem feito é melhor que bem dito." (Benjamin Franklin)

Neste capítulo iremos apresentar o objetivo principal deste trabalho, o *middleware Wings*. Com esse propósito, iniciaremos descrevendo os principais conceitos por trás do mesmo. Posteriormente, apresentaremos sua arquitetura, descrevendo seus módulos e os tipos de *plug-ins* definidos pelo *middleware*. A partir disso, iremos então apresentar a modelagem de cada módulo de sua arquitetura para então detalharmos os aspectos mais importantes em termos de implementação. Por fim, apresentaremos também a análise de performance que realizamos sobre o *middleware*.

5.1 Visão Geral Sobre o *Wings*

O *middleware Wings* baseia-se nos conceitos de *contexto*, *serviço* e *nó remoto*. O contexto, como já discutimos, encapsula informações utilizadas pelas aplicações para disponibilizar informações e serviços relevantes para o usuário. Tais informações podem ser, por exemplo, as pessoas ou restaurantes na proximidade, nível de temperatura, dentre outras. Um serviço, ao qual iremos nos referir como um serviço *Wings*, é uma entidade de software disponibilizando uma única funcionalidade (e.g., impressão). Como nosso propósito é disponibilizar serviços através de diferentes soluções, deve ser possível mapear um serviço *Wings* para qualquer tecnologia de serviços. Nesse escopo, um dos problemas é que as informações de

um serviço necessárias à sua publicação variam de tecnologia para tecnologia. Portanto, é desejável que esse mapeamento seja feito com máximo de compatibilidade possível. Dessa forma, serviços *Wings* provém apenas informações básicas sobre si, em uma tentativa de minimizar as diferenças entre os mesmos e os das demais tecnologias. Mais precisamente, as seguintes características estão associadas a um serviço *Wings*: *nome*, *descrição*, *lista de parâmetros* e *tipo de retorno*. A primeira característica representa o nome do serviço enquanto que a descrição provê informações sobre sua funcionalidade, sendo utilizada, por exemplo, no processo de descoberta para determinar sua relevância para o cliente. A lista de parâmetros descreve cada argumento do serviço, provendo, mais especificamente, informações sobre seus tipos (i.e., inteiro, *string*). Por fim, a última característica está associada ao tipo de retorno de um serviço. Informações como os tipos de parâmetro e de retorno podem ser úteis quando se precisa encontrar um substituto para um serviço que não esteja mais disponível. O último dos conceitos relacionados ao *Wings* são os nós remotos, os quais representam qualquer nó localizado remotamente. Tais nós provém duas informações, seu nome e a lista dos serviços que o mesmo provê.

5.2 Arquitetura

A arquitetura do *Wings*, ilustrada na Figura 5.1, consiste em quatro módulos: *Evolução dinâmica*, *Redes Pervasivas*, *Ciência de Contexto* e *Fachada*, os quais serão descritos nas seções seguintes.

Figura 5.1: Arquitetura do *middleware Wings*.

5.2.1 O Módulo de Evolução Dinâmica

O módulo de evolução dinâmica provê mecanismos para permitir que o *middleware* seja atualizado em tempo de execução. Mais precisamente, deve ser possível inserir, remover e atualizar funcionalidades no *middleware*, sem que seja preciso parar o mesmo nem as aplicações em execução. Como já mencionamos, no contexto deste trabalho, a solução de evolução dinâmica utilizada foi o modelo de componentes *Compor*, o qual foi apresentado no Capítulo 4.

5.2.2 O Módulo de Redes Pervasivas

Nesse módulo estão implementadas duas funcionalidades: *disponibilização de serviços* e *descoberta de nós*. Cada uma delas é implementada por um tipo específico de *plug-in*, respectivamente chamados de *Plug-in de Disponibilização de Serviços* (*PDS*) e *Plug-in de Descoberta de Nós* (*PDN*). Esses *plug-ins* são, na verdade, extensões da entidade componente funcional, definida no modelo de componentes *Compor*. Dado isso, o módulo de redes pervasivas em si é um contêiner, o qual também é baseado em tal modelo. É nesse contêiner, portanto, que os *PDSs* e *PDNs* são mantidos. Dessa forma, é possível inserir, remover e atualizar esses *plug-ins* de maneira transparente tanto para as aplicações quanto para o usuário, já que essas operações podem ser realizadas mesmo com o *middleware* em execução. Vale lembrar que, sendo os *PDSs* e *PDNs* tipos específicos de componentes funcionais, cada um deles provê um conjunto de serviços baseados no modelo *Compor*. Tais serviços serão descritos quando tratarmos da modelagem do módulo de redes pervasivas, na Seção 5.3.1.

A principal motivação para encapsular a disponibilização de serviços e descoberta de nós em *plug-ins*, é a possibilidade de adicionar diferentes *plug-ins* de um mesmo tipo, permitindo assim, acessar serviços e nós remotos utilizando diferentes soluções. Como exemplo, poderia-se ter dois *PDSs* inseridos no *middleware*, um implementado sobre *Bluetooth* e outro sobre *UPnP*, permitindo assim, publicar, descobrir e utilizar serviços através dessas duas soluções.

5.2.3 O Módulo de Ciência de Contexto

Este módulo implementa mecanismos para permitir às aplicações recuperarem informações de contexto, o que pode ser realizado utilizando *pares chave-valor* ou através de *eventos de contexto*. No primeiro, cada informação de contexto é associada a uma chave, a qual é utilizada para recuperar o valor atual da informação. Essas informações, no entanto, podem se apresentar de duas formas, *parametrizada* e *não parametrizada*. As informações parametrizadas sempre requerem um parâmetro para sua recuperação, representando o objeto do mundo real referente às mesmas. Considere como exemplo, uma informação de contexto cuja chave é *localização de*, que retorna o local em que uma determinada pessoa se encontra. Dessa forma, cada vez que essa informação precisar ser recuperada, deve-se passar como parâmetro o nome da pessoa a qual se quer saber a localização. As informações não parametrizadas, por outro lado, não necessitam de nenhum parâmetro. Um exemplo desse tipo de informação de contexto é o nível de bateria de um dispositivo, o qual poderia ser recuperado simplesmente através de uma chave, como *nível de bateria*, não requerendo, portanto, nenhum parâmetro. Uma outra forma de se obter informações de contexto é através dos chamados eventos de contexto, os quais permitem que aplicações recebam notificações acerca de certas informações de contexto. Para isso, no entanto, uma condição específica deve ser satisfeita. Essa condição, a qual é determinada pela aplicação, está associada a uma informação de contexto, definindo qual o valor da mesma a torna verdadeira. Dessa forma, por exemplo, uma aplicação poderia determinar uma condição *bateria baixa*, associada à informação de contexto cuja chave é *nível de bateria*, e que só é satisfeita quando essa informação indicar que restam menos de 5% de energia no dispositivo. Um outro exemplo seria uma condição *joão está na sala*, associada à informação parametrizada cuja chave é *localização de*, que retornaria verdadeiro sempre que a pessoa chamada "João" estivesse localizada na sala e falso caso contrário.

Mais especificamente, a disponibilização de informações de contexto é de responsabilidade de um tipo específico de *plug-in*, chamado de *Plug-in de Ciência de Contexto*, ou *PCC*. A idéia básica por trás dessa abordagem é que cada *PCC* esteja associado a um domínio específico (e.g., casas inteligentes, sistemas apoio a turistas), disponibilizando apenas informações e eventos de contexto referentes ao mesmo. Assim como acontece com os *PDSs* e *PDNs*, apresentados na seção anterior, os *PCCs* também são baseados no modelo de compo-

nentes *Compor*. Mais precisamente, esses *plug-ins* são componentes funcionais, provendo conseqüentemente um conjunto de serviços, sobre os quais falaremos na Seção 5.3.2. Dessa forma, o módulo de ciência de contexto é, assim como o de redes pervasivas, um contêiner baseado no modelo *Compor*, no qual todos os *PCCs* são mantidos.

É importante ressaltar que, com a idéia de *plug-ins*, as informações e eventos de contexto ficam completamente desacopladas das aplicações. Isso permite, portanto, que diferentes aplicações utilizem informações de contexto de um mesmo *PCC* (i.e., reúso de informações de contexto). Além disso, vale a pena frisar o quão simples é a tarefa de incrementar o conjunto de informações e eventos de contexto do *Wings*. Para isso, é preciso apenas adicionar, quando preciso, os *plug-ins* de ciência de contexto necessários. Como os mesmos estão baseados no modelo *Compor*, esse processo de adição pode ser realizado em tempo de execução.

5.2.4 O Módulo de Fachada

O módulo de Fachada tem como função prover um ponto único de acesso ao serviços dos *plug-ins* do *Wings*. A necessidade disso consiste no fato de que, como diferentes *plug-ins* podem coexistir no *middleware*, seria complicado para as aplicações invocar os serviços dos mesmos diretamente, pois as mesmas teriam que estar cientes de cada *plug-in* que fosse inserido e removido. Isso seria necessário para permitir que os serviços dos *plug-ins* recentemente adicionados pudessem ser invocados, bem como para impedir a invocação de serviços de *plug-ins* já removidos. O módulo de fachada, portanto, abstrai todos esses detalhes para as aplicações.

Dessa forma, a adição e remoção de *plug-ins* também é de responsabilidade do módulo de fachada. O mesmo se encarrega, portanto, de adicionar *PDSs*, *PDNs* e *PCCs* em seus respectivos contêiners. Além disso, o módulo de fachada possui um contêiner (*contêiner raiz*) no qual os módulos inferiores são mantidos, já que estes também são contêiners. Para se ter uma idéia mais precisa dessa organização, ilustramos na Figura 5.2 a hierarquia de *plug-ins* do *Wings*, de acordo com o modelo *Compor*.

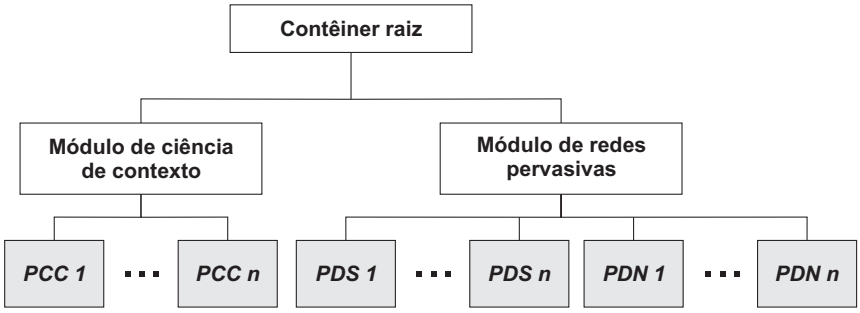


Figura 5.2: Hierarquia de *plug-ins* do *Wings*.

5.3 Modelagem

Nessa seção será apresentada a modelagem dos módulos de redes pervasivas, ciência do contexto e fachada. Já que o módulo de evolução dinâmica foi implementado seguindo o modelo de componentes *Compor*, sua modelagem já foi descrita no Capítulo 4.

5.3.1 Módulo de Redes Pervasivas

Como vimos na Seção 5.2.2, duas funcionalidades estão envolvidas no módulo de Redes Pervasivas: a disponibilização de serviços e a descoberta de nós. Nossa discussão sobre a modelagem desse módulo, portanto, será dividida de acordo com essas funcionalidades. Assim, descreveremos inicialmente as classes referentes à descoberta de nós para posteriormente apresentarmos aquelas associadas à disponibilização de serviços.

Descoberta de nós

No escopo da descoberta de nós, são três as classes que iremos apresentar, as quais estão ilustradas no diagrama de classes da Figura 5.3 e descritas a seguir.

- *HostDiscoveryPlugin*: classe abstrata que representa um *plug-in* de descoberta de nós (i.e., *PDN*), apresentado na Seção 5.2.2. Esse *plug-in* estende da classe *FunctionalComponent*, apresentada no Capítulo 4, e devido a isso, suas funcionalidades são disponibilizadas na forma de serviços, de acordo com o modelo de componentes *Compor*. Esses serviços são: *discoverHosts* e *stopHostDiscovery*. O primeiro inicia uma busca por nós, recebendo como parâmetro uma instância de *HostDiscoveryListener*, a qual é notificada sobre qualquer evento relacionado à busca (e.g., descoberta de um nó).

Além disso, esse serviço retorna um identificador para a busca recém iniciada. Para permitir que o *plug-in* execute buscas em paralelo, o serviço *discoverHosts* inicia uma nova linha de execução sempre que uma descoberta por nós é requisitada. O serviço *stopHostDiscovery*, por sua vez, encarrega-se de cancelar uma busca por nós, recebendo como parâmetro o identificador da busca a ser cancelada. Para cada um desses serviços a classe *HostDiscoveryPlugin* provê um método de mesmo nome, ou seja, *discoverHosts* e *stopHostDiscovery*. Esses dois métodos, portanto, implementam os serviços definidos para a classe *HostDiscoveryPlugin*. Além desses, um terceiro método, *discoverHostsImpl*, também é definido pela classe *HostDiscoveryPlugin*. Esse método é utilizado juntamente com o método *discoverHosts* para implementar o serviço de descoberta de nós. Mais detalhes sobre a implementação desse serviço serão apresentados na Seção 5.4.2.

- *RemoteHost*: essa interface representa um nó remoto, definindo quatro métodos: *getName* e *getProvidedServices*, *getAddress* e *openConnection*. O método *getName* retorna, como seu nome indica, o nome do nó remoto, se este estiver disponível. Já o método *getProvidedServices* retorna todos os serviços que o mesmo disponibiliza. Os métodos *getAddress* e *openConnection*, por fim, permitem respectivamente recuperar o endereço do nó remoto e abrir uma conexão de rede com o mesmo. Para este último, deve-se passar como parâmetro uma *string*, representando os parâmetros necessários à conexão.

- *HostDiscoveryListener*: interface para o recebimento de eventos relacionados à descoberta de nós. Esses eventos são recebidos através dos dois métodos da interface: *hostDiscovered*, *errorSearching*. O método *hostDiscovery* é invocado por um *plug-in* de descoberta de nós para notificar ao ouvinte que um nó foi descoberto. Como parâmetro, esse método recebe o identificador da busca e uma instância de *RemoteHost*, representando o nó descoberto. O método *errorSearching*, por sua vez, é utilizado quando um erro ocorreu durante uma busca. Além do identificador da busca na qual o erro ocorreu, esse método tem também como parâmetro o tipo de erro ocorrido.

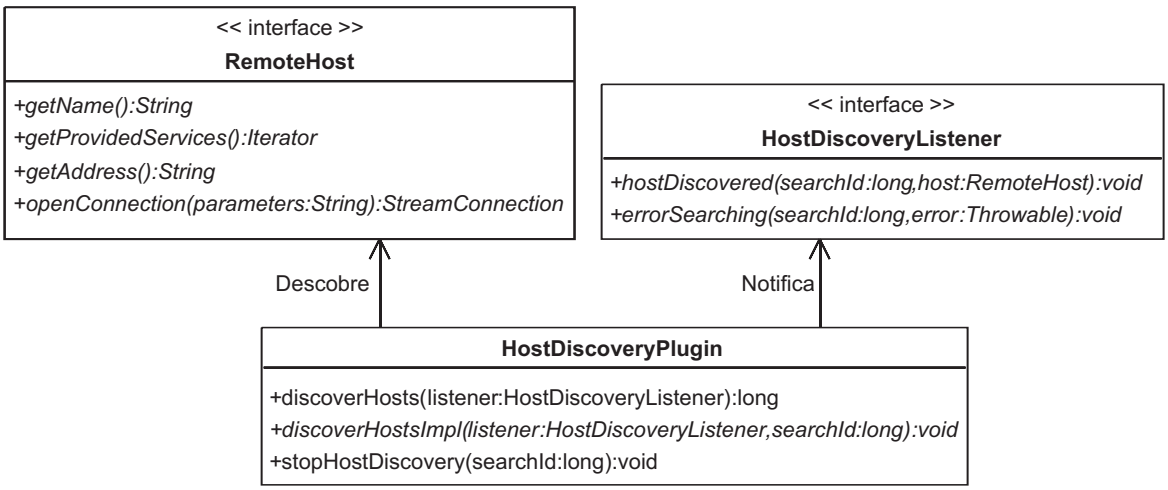


Figura 5.3: Principais classes relacionadas à descoberta de nós no *Wings*.

Disponibilização de Serviços

Iremos agora apresentar as principais classes associadas à disponibilização de serviços no *Wings*. Essas classes estão ilustradas na Figura 5.4, sendo cada uma delas descrita seguir.

- *ServiceProvisionPlugin*: classe que representa um *plug-in* de disponibilização de serviços (*PDS*), descrito na Seção 5.2.2, e estende de *FunctionalComponent*, parte do modelo de componentes *Compor*. As funcionalidades dos *PDSs* são, portanto, acessadas através de serviços baseados em tal modelo. Esses serviços são: *advertiseService*, *unadvertiseService*, *discoverServices* e *stopServiceDiscovery*. O serviço *advertiseService* implementa a publicação de serviços, recebendo como parâmetro o serviço a ser publicado. Vale frisar que tal serviço deve ser implementado pelo dispositivo local (i.e., uma instância da classe *LocalService*, descrita mais à frente). De maneira oposta, o serviço *unadvertiseService*, permite cancelar a publicação de um serviço. Para isso, o serviço cuja publicação será cancelada deve ser passado como parâmetro. O serviço *discoverServices*, por sua vez, implementa a descoberta de serviços. Esse processo, assim como a busca por nós, é executado em uma linha de execução separada, para permitir aos *PDSs* descobrir serviços paralelamente. Devido a isso, esse serviço tem como parte de seus parâmetros um objeto da interface *ServiceDiscoveryListener*, para a notificação de eventos referentes à busca. O outro parâmetro do serviço *discoverServices* é um *array* de *strings* representando as palavras-chave a serem utilizadas no processo de descoberta. Essas palavras-chave devem ser comparadas à descrição de cada

serviço remoto, de forma a determinar sua relevância para os possíveis clientes. Assim como o serviço *discoverHosts* da classe *HostDiscoveryPlugin*, *discoverServices* também retorna um identificador para a busca. O serviço *stopServiceDiscovery*, por fim, é utilizado para cancelar uma busca por serviços. Para isso, o identificador da busca, o qual é retornado do serviço *discoverServices*, deve ser passado como parâmetro. Para a implementação desses serviços a classe *ServiceProvisionPlugin* dispõe de quatro métodos, *advertiseService*, *unadvertiseService*, *discoverServices* e *stopServiceDiscovery*, cada um associado a um serviço de mesmo nome. Além desses, ainda existe o método *discoverServicesImpl*, utilizado em conjunto com *discoverServices* para implementar a descoberta de serviços. Mais detalhes sobre a implementação desses serviços serão dadas na Seção 5.4, na qual descrevemos a implementação do *Wings*.

- *Service*: interface que representa um serviço *Wings*. Essa interface define cinco métodos, dos quais quatro, *getName*, *getDescription*, *getParameters* e *getReturntype*, estão associados às características básicas de um serviço, nome, descrição, lista de parâmetros e tipo de retorno, descritas no início deste capítulo. O outro método dessa interface, *invoke*, tem o propósito de invocar o serviço, local ou remotamente. Esse método recebe como parâmetro um *array* de objetos da classe *Object*, representando os parâmetros necessários à invocação do serviço.

- *ServiceProxy*: a classe *ServiceProxy* representa um serviço provido por um nó remoto. Ou seja, são objetos dessa classe que um *PDS* passa para as instâncias de *ServiceDiscoveryListener* no momento em que um serviço remoto é descoberto. Dessa forma, todos os métodos dessa classe precisam fazer chamadas remotas ao serviço real. Vale lembrar que essas chamadas dependem dos protocolos associados ao *PDS*. Por exemplo, se o *PDS* é implementado sobre *UPnP*, suas chamadas remotas serão realizadas sobre *SOAP* [1]. Da mesma forma, se o *PDS* for implementado sobre *Jini*, essas mesmas chamadas serão agora implementadas sobre *RMI*. Devido a isso, essa classe não implementa nenhum dos métodos da interface *Service*, pois os mesmos dependerão dos protocolos associados ao *PDS*.

[1] *Simple Object Access Protocol*

- *LocalService*: essa classe abstrata implementa a interface *Service* e representa um serviço executando localmente. Dos métodos de sua super-interface, a classe *LocalService* implementa quatro: *getName*, *getDescription*, *getParameters* e *getReturnType*. O método *invoke*, por implementar a funcionalidade específica de um serviço, é deixada a cargo das sub-classes de *LocalService*. Além disso, *LocalService* ainda provê três métodos: *addService*, *getService* e *removeService*. O primeiro deles é utilizado para adicionar um serviço a outro. O serviço a ser adicionado é passado como uma instância da interface *Service*, permitindo assim a composição de serviços tanto remotos quanto locais. O outro parâmetro do método *addService* é um *alias* para identificar o serviço adicionado no escopo do serviço composto. Esse *alias* é utilizado posteriormente para recuperar um serviço adicionado, através do método *getService*. Assim, um serviço composto pode, quando necessário, recuperar seus serviços integrantes e invocar cada um deles na ordem desejada. O *alias* também é utilizado para remover um serviço previamente adicionado, utilizando, para isso, o método *removeService*.

- *ServiceDiscoveryListener*: interface para notificar aos interessados sobre eventos relacionados a descoberta de serviços, implementando, para isso, dois métodos: *errorSearching* e *serviceDiscovered*. O primeiro deles tem o mesmo propósito do método *errorSearching* da interface *HostDiscoveryListener*. Já o segundo método, *serviceDiscovered*, é utilizado para notificar os interessados sobre a descoberta de um serviço. O serviço descoberto é passado no parâmetro desse método, como uma instância de *ServiceProxy*. Além disso, esse método tem também como parâmetro o identificador da busca na qual o serviço foi encontrado.

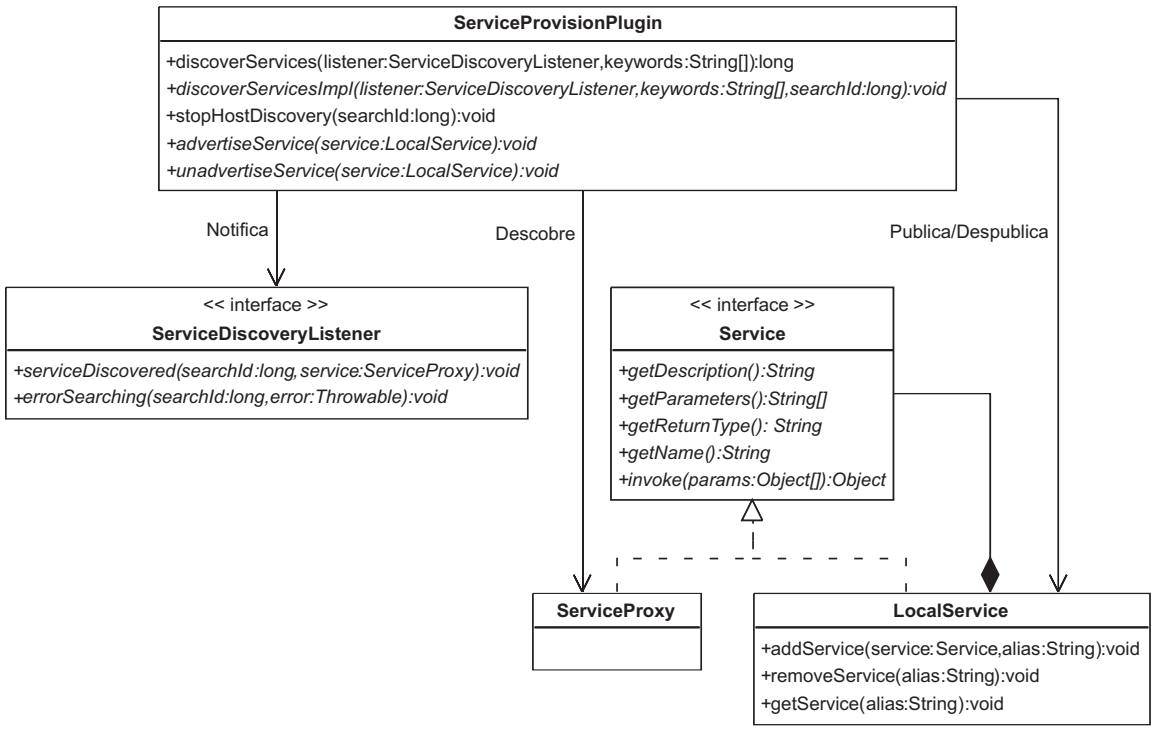


Figura 5.4: Principais classes relacionadas à disponibilização de serviços no *Wings*.

5.3.2 Módulo de Ciência do Contexto

Nesta seção apresentaremos a modelagem do módulo de ciência de contexto, cujas classes estão ilustradas na Figura 5.5, sendo descritas a seguir.

- *ContextAwarenessPlugin*: classe que representa um *plug-in* de ciência do contexto (*PCC*). Assim como as classes *HostDiscoveryPlugin* e *ServiceDiscoveryPlugin*, esta também estende de *FunctionalComponent*, disponibilizando três serviços, como especificado no modelo *Compor*: *retrieveContextInformation*, *registerContextListener* e *unregisterContextListener*. O primeiro permite recuperar informações de contexto através da abordagem de chave-valor. Dessa forma, esse serviço recupera tanto as informações de contexto parametrizadas quanto as não parametrizadas, recebendo como parâmetro uma *string*, representando a chave da informação de contexto a ser recuperada, e uma instância da classe *object*, representando o parâmetro associado à informação. Se a informação a ser recuperar for não parametrizada, este último parâmetro deve ser passado como *null*. O serviço *registerContextListener*, por sua vez, permite registrar ouvintes aos eventos de contexto de um *PCC*. Como parâmetro, esse serviço

recebe a condição que deve ser satisfeita para que o evento de contexto seja disparado e o ouvinte que deve ser notificado quando isso acontecer. Note que, para isso, o serviço precisa verificar de tempos em tempos se a condição está satisfeita ou não, para que o mesmo saiba quando o evento deve ser disparado. Para não bloquear o *plug-in* durante esse processo, o mesmo é realizado em uma linha execução separada. Por fim, o serviço *unregisterContextListener* remove um ouvinte de contexto de um *PCC*. Para isso, devem ser passados nos parâmetros o ouvinte e a condição da qual o mesmo será removido. Os serviços dos *PCCs* são implementados, respectivamente, pelos métodos *retrieveContextInformation*, *registerContextListener* e *unregisterContextListener*, tendo todos os mesmos parâmetros que os serviços associados. O método *getContextInformationKeys*, por sua vez, retorna as chaves das informações de contexto providas pelo *plug-in*. Finalmente, o método *addProvidedContextInformation* tem como função adicionar uma informação de contexto a ser provida pelo *plug-in*. Esse método recebe como parâmetro a chave da informação de contexto a ser adicionada e o método utilizado para recuperá-la (i.e., um objeto da classe *Method*). Mais detalhes sobre a utilização desses métodos serão dadas quando discutirmos a implementação do *Wings*, na Seção 5.4.

- *ContextEventListener*: essa é a interface que deve ser implementada para se receber eventos de contexto. Esses eventos são notificados através do método *receiveContextEvent*, para o qual é passado uma instância de *ContextEvent*, representando o evento disparado. O outro método dessa interface, *receiveError*, é utilizado para notificar o ouvinte acerca de erros na aquisição da informação de contexto associada ao evento.

- *ContextCondition*: a interface *ContextCondition* representa a condição que deve ser satisfeita para que um certo evento de contexto seja disparado. Essas condições têm associadas a si a chave de uma informação de contexto e possivelmente um parâmetro relacionado à mesma, através dos quais é possível extrair o valor atual da informação e assim verificar quando as condições são satisfeitas. Para isso, o método *isSatisfied* deve ser invocado, passando esse valor como parâmetro. O mesmo, por sua vez, é obtido utilizando-se a chave da informação de contexto e o parâmetro, se for o caso, associados à condição, retornados respectivamente através dos métodos *getContextIn-*

formationKey e *getContextInformationParameter*. Caso a condição não tenha nenhum parâmetro associado, este último método deve retornar *null*.

- *ContextEvent*: como o nome indica, essa classe representa um evento de contexto, o qual é recebido pelas instâncias de *ContextEventListener*. Os eventos de contexto provém duas informações, a chave e o valor da informação de contexto ao qual estão associados. Essas informações são recuperadas através dos métodos, *getContextInformationKey* e *getContextInformationValue*.

- *ContextEventThread*: essa classe tem como função monitorar uma condição de contexto específica e disparar eventos para o ouvinte associado, sempre que a condição for satisfeita ou algum erro ocorrer durante a aquisição da informação de contexto.

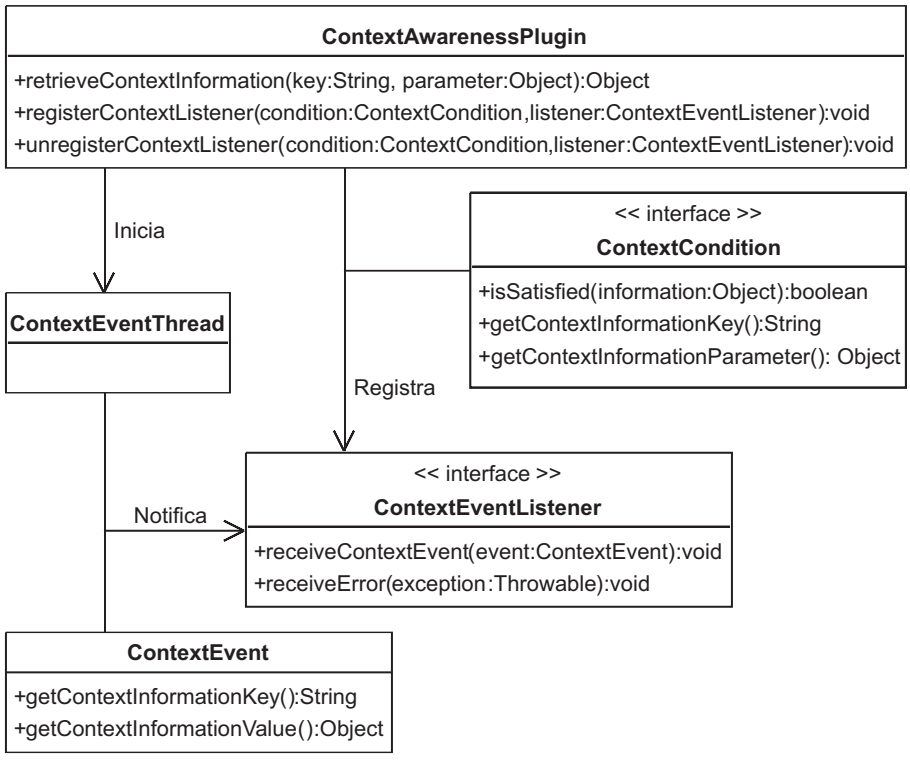


Figura 5.5: Principais classes do módulo de ciência de contexto.

5.3.3 Módulo de Fachada

Iremos agora apresentar a modelagem do módulo de fachada, cujas classes encontram-se ilustradas na Figura 5.6 e descritas logo a seguir.

- *MiddlewareFacade*: a classe *MiddlewareFacade* representa o módulo de fachada, disponibilizando ao todo dezessete métodos, dos quais nove servem para acessar os serviços dos *plug-ins* do *Wings*: *discoverHosts*, *discoverServices*, *stopHostDiscovery*, *stopServiceDiscovery*, *advertiseService*, *unadvertiseService*, *retrieveContextInformation*, *registerContextListener*, *unregisterContextListener*. Dessa forma, cada um desses métodos está relacionado a um serviço de mesmo nome, disponibilizado por *PDSs*, *PDNs* ou *PCCs*. Mais precisamente, a função desses métodos é delegar para o(s) *plug-in*(s) correto(s) a invocação do serviço associado aos mesmos. Por exemplo, o método *discoverServices* delega para todos os *PDSs* instalados a requisição de descoberta de serviços. Tais métodos possuem, portanto, os mesmos parâmetros e retorno dos serviços aos quais estão associados. A classe *MiddlewareFacade* disponibiliza ainda os seguintes métodos, voltados à adição e remoção de *plug-ins* nos módulos inferiores: *addServiceProvisionPlugin*, *addHostDiscoveryPlugin*, *addContextAwarenessPlugin*, *removeServiceProvisionPlugin*, *removeHostDiscoveryPlugin* e *removeContextAwarenessPlugin*. Os três primeiros servem, respectivamente, para inserir *PDSs*, *PDNs* e *PCCs*. Os três últimos, por outro lado, servem para removê-los de seus respectivos módulos.

- *RootContainer*: essa classe representa o contêiner raiz da hierarquia de *plug-ins* do *Wings*. Portanto, o mesmo encapsula os contêiners que representam os módulos inferiores (i.e., redes pervasivas e ciência de contexto), de forma que todas as invocações de serviço aos *plug-ins* do *Wings* são realizadas através dele.

Figura 5.6: Principais classes do módulo de fachada.

5.4 Implementação

Nesta seção será apresentada a implementação do *middleware Wings*. Nesse contexto, nossa discussão será conduzida através de suas funcionalidades básicas, ou seja, adição e remoção de *plug-ins*, descoberta de nós e publicação, "despublicação" e descoberta de serviços além da recuperação de informações de contexto. Além disso, abordaremos também alguns dos problemas encontrados ao longo do desenvolvimento.

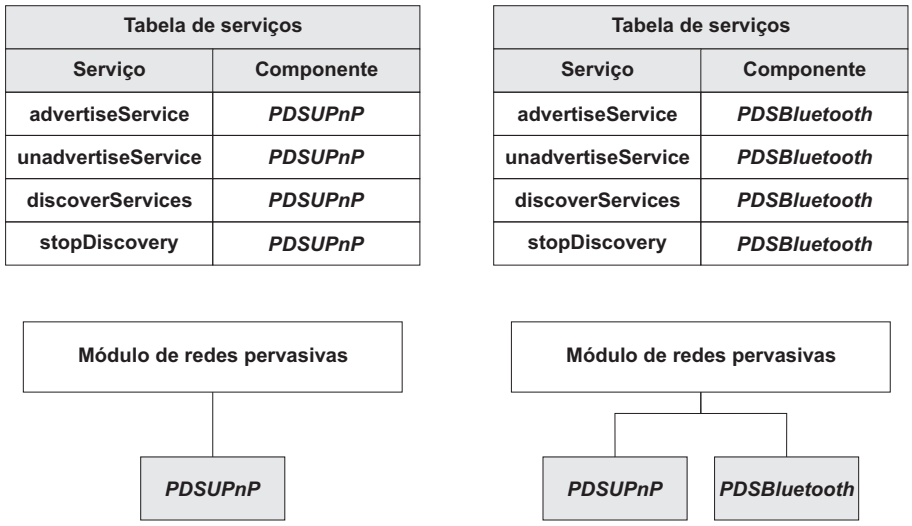
5.4.1 Adição e Remoção de *Plug-ins*

Iremos apresentar agora como foram implementadas a adição e remoção dos três tipos de *plug-ins* do *Wings*; *PDSs*, *PDNs* e *PCCs*. Mais especificamente, iremos mostrar aqui a implementação dos métodos *addServiceProvisionPlugin*, *removeServiceProvisionPlugin*, *addHostDiscoveryPlugin*, *removeHostDiscoveryPlugin*, *addContextAwarenessPlugin* e *removeContextAwarenessPlugin*, todos da classe *MiddlewareFacade*. Para os *PDSs* e *PDNs* as

operações de inserção e remoção são idênticas. Dessa forma, as discutiremos em conjunto.

Antes de apresentarmos as implementações desses métodos, no entanto, vale a pena abordarmos um problema referente à inserção dos três tipos de *plug-ins*. Tal problema está associado com o fato de que os *plug-ins* de um mesmo tipo disponibilizam serviços com o mesmo nome. Como exemplo, veja que, de acordo com o que foi apresentado na Seção 5.3.1, cada *PDS* disponibiliza serviços chamados *advertiseService*, *unadvertiseService*, *discoverServices* e *stopServiceDiscovery*. Os efeitos práticos disso no *Wings* é que, ao se inserir dois ou mais *plug-ins* de um mesmo tipo, os serviços do *plug-in* recentemente adicionado irão sobrescrever os do que já se encontra instalado. Para se ter uma idéia desse problema, imagine uma situação na qual existe inicialmente um *PDS* inserido no *middleware*, chamado, por exemplo, de *PDSUPnP*. Isso significa, portanto, que o módulo de redes pervasivas e sua tabela de serviços se encontram como ilustrado na Figura 5.7(a). Imagine agora que um outro *PDS*, chamado de *PDSBluetooth*, é inserido no *middleware*. De acordo com o mecanismo de disponibilização de serviços do modelo de componentes *Compor*, os serviços desse *PDS* irão sobrescrever os de *PDSUPnP*, como ilustrado na Figura 5.7(b) (note que, agora, cada serviço na tabela de serviços está associado a *PDSBluetooth* e não mais a *PDSUPnP*). O mesmo aconteceria com os serviços de *PDSBluetooth*, caso algum outro *PDS* fosse inserido no *middleware*. O que se pode concluir a partir disso é que, apenas os serviços de um *plug-in* de cada tipo vão estar acessíveis por vez. Em outras palavras, embora vários *plug-ins* possam estar instalados, apenas os serviços de um *PDN*, um *PDS* e um *PCC* estarão disponíveis. Para *PDNs* e *PDSs* isso ainda é mais crítico, já que a idéia é permitir ao *Wings* executar cada um de seus serviços em paralelo.

Para resolver esse problema utilizamos a conceito de *alias* de serviço, provido pelo modelo *Compor*. Esse conceito permite, dentro do escopo de uma aplicação, alterar o nome dos serviços de um componente. Ou seja, se um componente *A* disponibiliza um serviço chamado *salvar*, com o conceito de *alias* pode-se trocar o nome desse serviço para *salvarEmArquivo*, de forma que, a partir de então, todas as requisições ao mesmo devem utilizar esse novo nome. Isso nos permite, portanto, trocar o nome dos serviços de cada um dos *plug-ins* por um outro que os identifique unicamente no *middleware*. Sendo assim, todos os métodos do módulo de fachada associados com a inserção de *plug-ins* devem utilizar esse conceito de *aliases*, como forma de impedir a sobrescrita de serviços dos *plug-ins*.



(a) Módulo de redes pervasivas com apenas um *PDS* adicionado.

(b) Módulo de redes pervasivas com dois *PDSs* adicionados.

Figura 5.7: Configurações do módulo de redes pervasivas e de sua tabela de serviços.

Em termos mais práticos, os *aliases* são gerados concatenando-se o nome original do serviço à *string* "*At_*" e ao nome completo da classe do *plug-in* (i.e., nome do pacote mais nome da classe). Por exemplo, considere um *PDN* cuja classe chama-se *PDNBluetooth*, pertencente ao pacote *wings*. Como vimos na Seção 5.3.1, esse *plug-in* disponibiliza dois serviços, *discoverHosts* e *stopHostDiscovery*. Dessa forma, os *aliases* para seus serviços seriam, portanto: *discoverHostsAt_wings.PDNBluetooth* e *stopHostDiscoveryAt_wings.PDNBluetooth*.

Adição e Remoção de *PDSs* e *PDNs*

Em se tratando especificamente da adição de *PDSs* e *PDNs*, a estratégia de utilizar *aliases* para seus serviços gerou um outro problema. Veja que, sem a utilizarmos, o módulo de fachada sabia, *a priori*, os nomes dos serviços que devia invocar. Por exemplo, se uma aplicação requisitasse uma busca por serviços, o mesmo sabia que deveria invocar o serviço *discoverServices*. Alterando o nome dos serviços de cada *PDS* e *PDN* adicionado, impossibilita-se o módulo de fachada de invocar os serviços desses *plug-ins*, pois o mesmo desconhece os *aliases* associados aos serviços.

Em função disso, sempre que *PDSs* e *PDNs* são inseridos no *middleware*, os *aliases* utilizados para seus serviços são mantidos em um conjunto de tabelas (chamadas de *tabe-*

las de aliases). Mais precisamente, tem-se uma tabela de *alias* para cada serviço de um *PDS* ou *PDN*, totalizando seis tabelas. Suas entradas têm como chave o nome completo da classe do *plug-in* a ser adicionado e como valor o *alias* de um dos seus serviços. Por exemplo, considere novamente um *PDN* da classe *PDNBluetooth*, do pacote *wings*. De acordo com o que apresentamos no início da seção, os *aliases* de seus dois serviços (i.e., *discoverHosts* e *stopHostDiscovery*) seriam *discoverHostsAt_wings.PDNBluetooth* e *stopHostDiscoveryAt_wings.PDNBluetooth*. As entradas nas tabelas de *aliases* correspondentes a esses serviços, seriam então: <*wings.PDNBluetooth*, *discoverHostsAt_wings.PDNBluetooth*> para a tabela de *aliases* do serviço *discoverHosts* e <*wings.PDNBluetooth*, *stopHostDiscoveryAt_wings.PDNBluetooth*> para a tabela de *aliases* do serviço *stopHostDiscovery*. Com isso o *middleware* poderá recuperar da tabela correspondente os *aliases* dos serviços requisitados, para assim invocá-los em cada um dos *plug-ins* instalados. Como veremos mais adiante, essa abordagem também é útil no momento de remoção de um *PDS* ou *PDN*.

Pelo que foi apresentado, os métodos de *MiddlewareFacade* para a adição de *PDSs* e *PDNs* (i.e., *addServiceProvisionPlugin* e *addHostDiscoveryPlugin*) devem, portanto, gerar *aliases* para cada serviço dos mesmos, armazenar tais *aliases* nas tabelas correspondentes e adicionar o *plug-in* ao módulo de redes pervasivas. Mais especificamente, como ilustrado na Figura 5.8, a implementação desses métodos recupera inicialmente o nome completo da classe do *plug-in* a ser adicionado (passo 1). Posteriormente, geram-se os *aliases* para seus serviços (passo 2). Cada um desses *aliases* será então mantido na tabela de *alias* correspondente, utilizando como chave o nome completo da classe do *plug-in* (passo 3). O *PDS* ou *PDN* é então adicionado ao módulo de redes pervasivas (passo 4), de forma que sua tabela de serviços é atualizada utilizando os *aliases* gerados no passo 2 (passo 5). Caso ocorra algum erro ao adicionar o *plug-in*, os *aliases* que foram adicionados às tabelas durante o passo 3 são, portanto, removidos.

A partir do que foi discutido, fica claro que, na remoção de *PDSs* e *PDNs*, além de removê-los do módulo de redes pervasivas, é preciso remover também as entradas nas tabelas de *aliases* contendo referências para os mesmos. Como explicaremos mais adiante, isso se faz necessário, pois do contrário, o *middleware* poderia invocar serviços de *plug-ins* que já tivessem sido removidos. Dessa forma, os métodos de *MiddlewareFacade* voltados à remoção de *PDSs* e *PDNs*, *removeServiceProvisionPlugin* e *removeHostDiscoveryPlugin*,

devem implementar os passos apresentados na Figura 5.9. Ou seja, primeiramente, os mesmos devem recuperar o nome completo da classe do *plug-in* a ser removido (passo 1). De posse desse nome, para cada serviço do *plug-in*, remove-se a entrada na tabela de *alias* associada ao mesmo (passo 2). Após isso, o *plug-in* pode então ser removido do módulo de redes pervasivas (passo 3). Note que, com isso, as entradas de sua tabela de serviços referentes ao *plug-in* em questão também são removidas (passo 4).

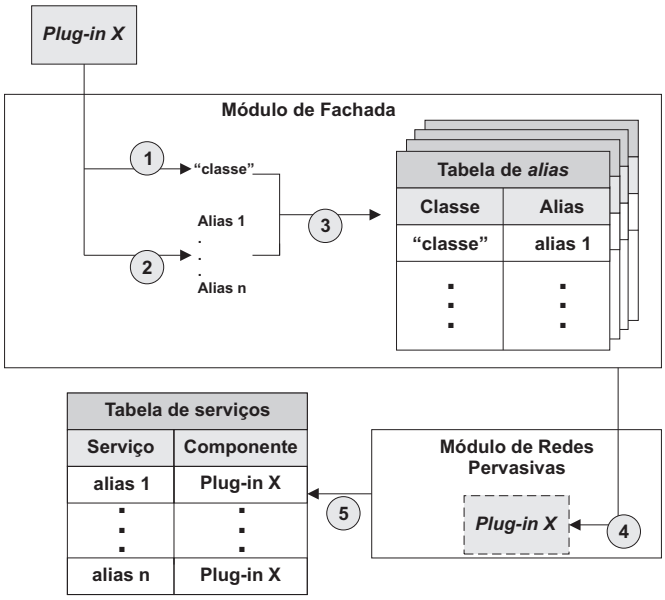


Figura 5.8: Adição de *PDNs* e *PDSs* através do módulo de fachada.

Adição e Remoção de *PCCs*

No que se refere adição e remoção de *PCCs*, além do problema de sobrescrita de serviços que descrevemos anteriormente, um outro problema causou impacto nessas duas operações. Como falamos anteriormente, cada *PCC* encapsula um conjunto de informações de contexto, acessíveis através do serviço *retrieveContextInformation*, utilizando suas respectivas chaves. Como também falamos, a recuperação de informações de contexto se dá através do módulo de fachada, mais precisamente através do método *retrieveContextInformation* da classe *MiddlewareFacade*. O fato é que, já que diferentes *PCCs* podem coexistir no *middleware*, cada um com seu próprio conjunto de informações de contexto, o módulo de fachada não sabe inicialmente qual *plug-in* provê uma determinada informação, e portanto, nem qual o serviço deve ser invocado para recuperá-la.

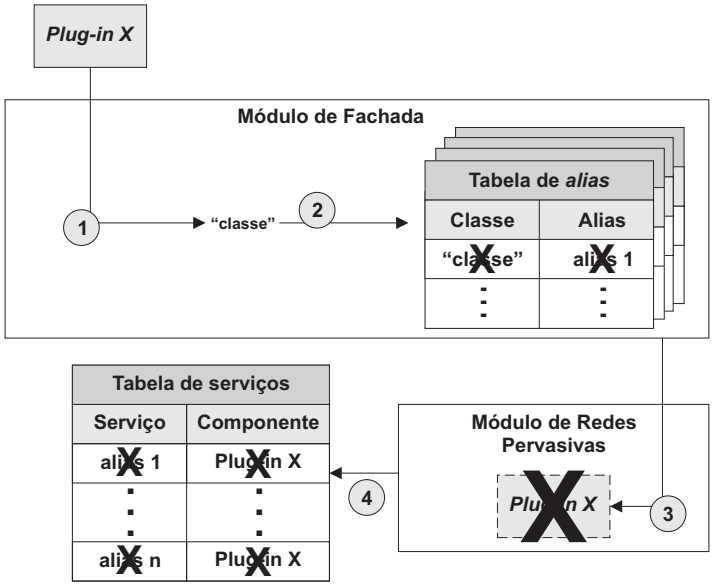


Figura 5.9: Remoção de *PDNs* e *PDSs* através do módulo de fachada.

Uma opção, nesse caso, seria procurar em todos os *PCCs* instalados pela informação desejada. Isso, no entanto, pode ser bastante lento em alguns casos, dependendo do número de *PCCs* inseridos no *middleware*. Em vez disso, a abordagem que utilizamos consiste em registrar as chaves das informações de contexto de um *PCC* no momento em que o mesmo é adicionado. Ou seja, sempre que um desses *plug-ins* é adicionado, guarda-se em uma tabela, chamada de *tabela de informações de contexto*, uma entrada para cada informação de contexto que o mesmo provê. As chaves dessas entradas são as chaves de cada informação de contexto do *PCC*, recuperadas através do método *getContextInformationKeys* da classe *ContextAwarenessPlugin*, e seu valor é o nome do serviço *retrieveContextInformation*, já considerando a alteração do seu nome original por um *alias*. Esse processo pode ser melhor entendido através da Figura 5.10, a qual ilustra todos os passos envolvidos na adição de *PCCs* através do módulo de fachada (i.e., implementação do método *addContextAwarenessPlugin* da classe *MiddlewareFacade*). Nessa figura, inicialmente um *PCC* da classe *PCC_1* é adicionado ao *middleware*. Estamos considerando, nesse exemplo, que tal *PCC* provê apenas uma informação de contexto, cuja chave é *informação 1*. Após isso, os nomes dos serviços desse *plug-in* serão alterados utilizando um conjunto de *aliases* (passo 2). Dessa forma, considerando que o *plug-in* adicionado faz parte do pacote "*wings*", os serviços do mesmo serão renomeados para *retrieveContextInformationAt_wings.PCC_1*, *re-*

gisterContextListenerAt_wings.PCC_1 e *unregisterContextListenerAt_wings.PCC_1*. Feito isso, será adicionada uma entrada na tabela de informações de contexto para cada informação provida pelo *PCC*. No nosso exemplo, isso significa adicionar apenas uma entrada, a qual terá como chave *informação 1* e como valor *retrieveContextInformationAt_wings.PCC_1* (passo 3). No quarto passo, o *PCC* é finalmente adicionado ao módulo de ciência de contexto, sendo seus serviços registrados na tabela de serviços (passo 5).

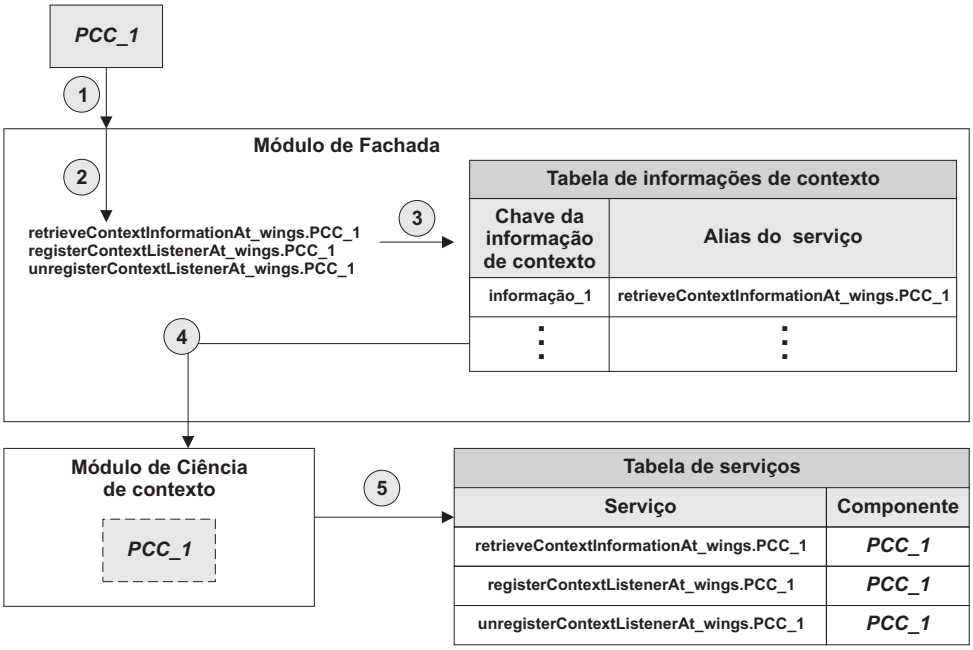


Figura 5.10: Adição de *PCCs* através do módulo de fachada.

Já que durante a inserção de *PCCs*, um conjunto de entradas é inserido na tabela de informações de contexto, sempre que os mesmos são removidos, tais entradas precisam, portanto, ser removidas também. De uma forma geral, o algoritmo para a remoção de *PCCs* a partir do módulo de fachada, implementado pelo método *removeContextAwarenessPlugin* da classe *MiddlewareFacade*, deve: 1) remover as entradas na tabela de informações contexto associadas ao *plug-in* que será removido e 2) remover o mesmo do módulo de ciência de contexto. Mais precisamente, como ilustrado no passo 1 da Figura 5.11, deve-se primeiro recuperar as chaves das informações de contexto do *plug-in* a ser removido. Como falamos anteriormente, isso é realizado através do método *getContextInformationKeys* da classe *ContextAwarenessPlugin*. Posteriormente, as entradas referentes a essas chaves serão removidas da tabela de informações de contexto (passo 2). Neste momento, o *plug-in* pode então ser

removido do módulo de ciência de contexto (passo 3), fazendo com que as informações de seus serviços sejam excluídas da tabela de serviços desse módulo (passo 4).

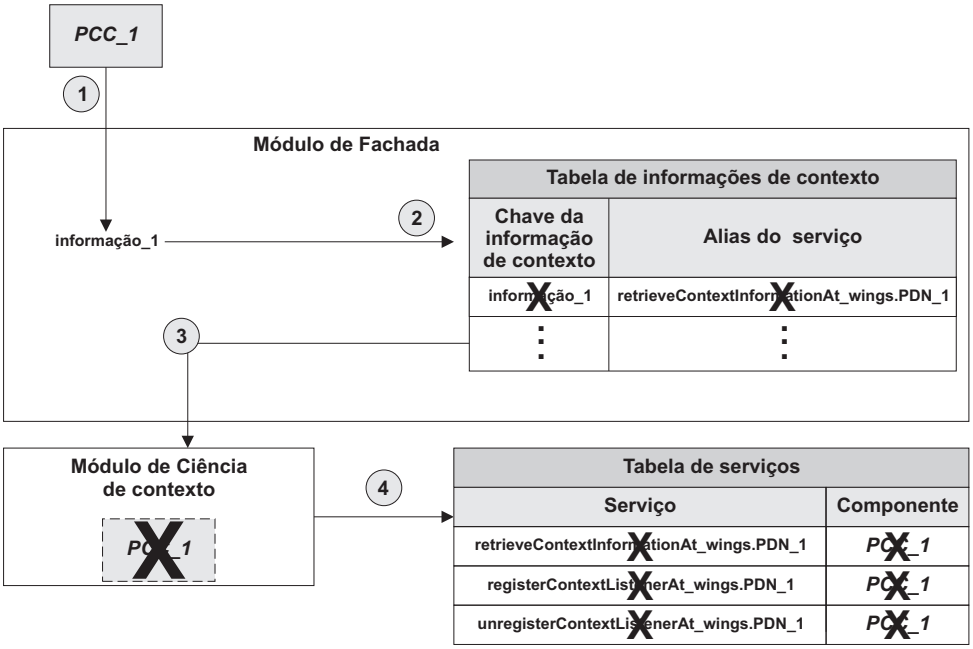


Figura 5.11: Remoção de *PCCs* através do módulo de fachada.

5.4.2 Iniciando e Cancelando uma Busca por Nós

Como visto na Seção 5.3.3, tanto o início quanto o cancelamento de busca por nós é feito através do módulo de fachada, utilizando-se o método *discoverHosts* da classe *Middleware-Facade*. Com esse propósito, o módulo de fachada invoca o serviço de descoberta de nós (i.e., *discoverHosts*) de cada *PDN* instalado, já considerando a alteração do nome original desse serviço por um *alias*. Nesse ponto, existe um problema que vale a pena ser discutido. Conforme apresentado na Seção 5.3.1, esse serviço de descoberta de nós retorna um identificador a cada nova invocação. O que se pode concluir a partir disso é que, sempre que a descoberta de nós for requisitada através da fachada, a mesma obterá tantos identificadores quantos forem os *PDNs* instalados. Por exemplo, se dois *PDNs* estiverem instalados no *middleware*, a fachada obterá dois identificadores, pois o serviço de descoberta de nós será invocado nesses dois *plug-ins*. No entanto, note que, como resposta da invocação do método *discoverHosts*, apenas um identificador deve ser retornado pelo módulo de fachada.

Para contornar esse problema, a cada requisição de descoberta de nós, o módulo de fa-

chada gera um identificador próprio e o associa aos identificadores retornados de cada invocação do serviço *discoverHosts*. Esse identificador gerado é, portanto, retornado à aplicação. Em termos mais práticos, o mesmo será armazenado em uma tabela (*tabela de identificadores*) juntamente com uma lista contendo os identificadores retornados das invocações dos serviço *discoverHosts*. As entradas dessa tabela tem o identificador gerado como chave e a lista de identificadores como valor. Com isso, a descoberta de nós através do módulo de fachada pode ser ilustrada através dos passos exibidos na Figura 5.12. Tudo começa quando uma aplicação requisita o serviço de descoberta de nós, como ilustrado no primeiro passo da figura, especificando o ouvinte que receberá os eventos acerca da busca (i.e., uma instância de *HostDiscoveryListener*). Feito isso, o módulo de fachada primeiramente gera um identificador para a busca (passo 2). Após isso, os *aliases* do serviço *discoverHosts* são recuperados da sua tabela de *aliases*, e a partir dos mesmos são feitas as invocações de serviço em cada *PDN* instalado (passo 3). Vale lembrar que, como mencionamos na Seção 5.3.1, os *PDNs* iniciam a busca por nós em uma linha de execução separada. Ou seja, nesse passo, o método *discoverHosts* da classe *HostDiscoveryPlugin*, o qual é invocado para a execução do serviço de descoberta de nós, cria e inicia uma nova linha execução. Nessa linha de execução, o método *discoverHostsImpl*, também da classe *HostDiscoveryPlugin*, será executado, iniciando de fato a busca por nós. Os identificadores retornados de cada invocação de serviço são mantidos em uma lista (passo 4), a qual é armazenada na tabela de identificadores, juntamente com o identificador gerado pela fachada (passo 5). Esse identificador é então retornado à aplicação (passo 6). Por fim, quando um nó for descoberto por algum dos *PDNs* ou houver algum erro durante a busca, o mesmo notificará o ouvinte, respectivamente através dos métodos *discoverHosts* e *errorSearching* da interface *HostDiscoveryListener* (passo 7).

O cancelamento de uma busca por nós através do módulo de fachada, realizado através do método *stopHostDiscovery* da classe *MiddlewareFacade*, deve invocar o serviço *stopHostDiscovery* de cada um dos *PDNs* instalados no *middleware*. Além disso, as entradas na tabela de identificadores referentes à busca em questão devem ser removidas. Como ilustrado na Figura 5.13, o primeiro passo no processo de cancelamento de uma busca por nós é recuperar e remover da tabela de identificadores, a partir do identificador passado pela aplicação, a entrada contendo a lista de identificadores específicos das buscas associadas a cada *PDN*. Feito isso, recupera-se os *aliases* do serviço *stopHostDiscovery*, utilizando-os para cancelar

a busca por nós em cada *PDN*, passando para os mesmos o respectivo identificador da busca (passo 2). Vale lembrar que, nesse passo, o *plug-in* tem que verificar a linha de execução associada à busca em questão, para então interrompê-la. Isso significa, portanto, que essas linhas de execução devem chamar o método *stopHostDiscoveryImpl* da classe *HostDiscoveryPlugin*.

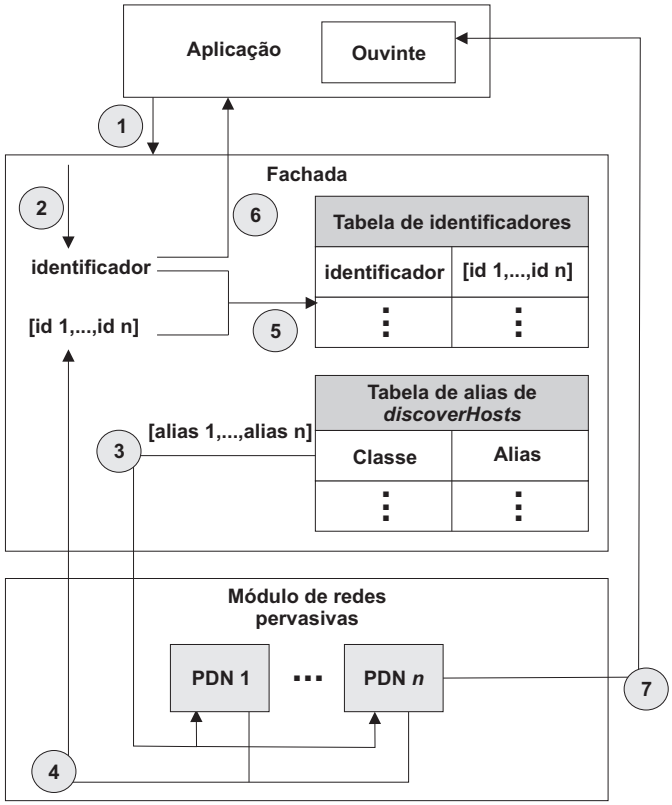


Figura 5.12: Descoberta de nós através do módulo de fachada.

5.4.3 Publicação, "Despublicação" e Busca de Serviços

De acordo com o que foi apresentado na Seção 5.3.3, assim como acontece com a descoberta e cancelamento de busca por nós, a publicação, "despublicação", descoberta de serviços e cancelamento de busca de serviços também é realizada através do módulo de fachada, utilizando-se respectivamente os métodos *advertiseService*, *unadvertiseService*, *discoverServices* e *stopServiceDiscovery*. Mais especificamente, sempre que uma dessas operações for requisitada, o mesmo deve invocar o serviço correspondente em cada *PDS* instalado, assim como acontece com os serviços dos *PDNs*.

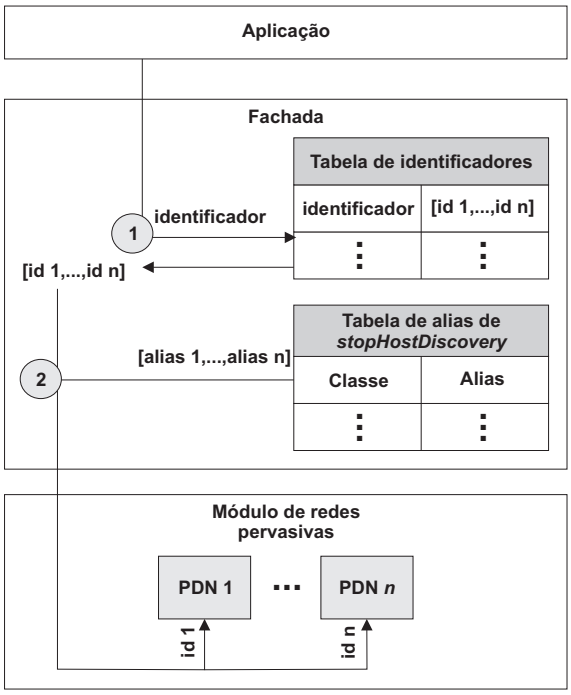


Figura 5.13: Cancelamento de uma busca por nós através da fachada.

Em relação à publicação de serviços, do ponto de vista da fachada, tudo que se precisa fazer é recuperar os *aliases* do serviço *advertiseService*, armazenados em uma das tabelas de *aliases*, e utilizá-los para publicar um determinado serviço *Wings* em cada *PDS* instalado no *middleware*. Isso pode ser melhor percebido através da Figura 5.14, na qual é apresentado todo o processo para a publicação de serviços através da fachada. Esse processo se inicia quando uma aplicação invoca o método *advertiseService* da classe *MiddlewareFacade*, passando o serviço *Wings* a ser publicado (i.e., uma instância de *LocalService*) (passo 1). A partir daí, o módulo de fachada irá recuperar os *aliases* do serviço *advertiseService*, para então invocá-lo nos *PDSs* atualmente instalados, passando como parâmetro o serviço *Wings* provido pela aplicação (passo 2).

Do ponto de vista dos *PDSs*, para a publicação de serviços, os mesmos devem ser capazes de mapear o serviço *Wings* recebido para um serviço específico da solução ao qual estão relacionados. Considerando um *PDS* implementado sobre *SLP*, por exemplo, para publicar um serviço *Wings*, o *plug-in* teria que encapsular o mesmo em uma implementação de serviço específica do *SLP*. Para isso, o mesmo deve então recuperar as informações sobre o serviço *Wings*, através dos métodos *getName*, *getDescription*, *getParameters* e *getReturnType*, utilizando-as para publicar o serviço através de *SLP*.

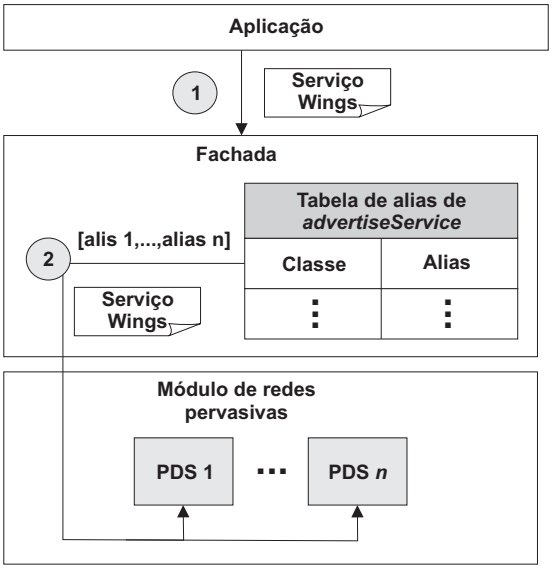


Figura 5.14: Publicação de serviços através da fachada.

A operação de "despublicação", por sua vez, possui exatamente os mesmos passos que descrevemos para a publicação de serviços. A única diferença é que no passo 2, os *aliases* são recuperados da tabela de *aliases* do serviço *unadvertiseService*. Com relação aos *PDSs*, essa tarefa envolve, de forma geral, identificar qual dos serviços publicados corresponde ao serviço *Wings* passado no parâmetro, para então cancelar sua publicação.

No que se refere à descoberta de serviços através do módulo de fachada, não existe muito a se comentar, pois o processo de descoberta de serviços em si é idêntico ao de descoberta de nós, apresentado na seção anterior. As diferenças entre essas duas operações são referentes aos seus parâmetros e ao fato de que os *aliases* dos serviços associados às mesmas são recuperados de tabelas diferentes. Em relação aos parâmetros, como vimos, a descoberta de nós requer apenas um ouvinte, passado como uma instância de *HostDiscoveryListener*, o qual recebe os eventos acerca da busca. A descoberta de serviços, por sua vez, requer, além do ouvinte, dessa vez como instância de *ServiceDiscoveryListener*, um conjunto de *strings* representando as características do serviço a ser encontrado. Por fim, em se tratando da recuperação dos *aliases*, na descoberta de nós isso é feito através da tabela de *aliases* do serviço *discoverHosts*. Por outro lado, na descoberta de serviços, os mesmos são recuperados da tabela de *aliases* do serviço *discoverServices*. Uma vez recuperado tais *aliases*, estes serão então utilizados para invocar o serviço de descoberta de serviços em cada um dos *PDSs*, passando o ouvinte e o conjunto de palavras-chave como parâmetro. Assim, quando um

serviço for encontrado ou um erro ocorrer durante a busca em algum dos *PDSs*, o ouvinte será notificado, utilizando, para isso, os métodos *serviceDiscovered* e *errorSearching* da interface *ServiceDiscoveryListener*.

5.4.4 Recuperando Informações de Contexto e Registrando-se a Eventos de Contexto

Como vimos na Seção 5.2.3, a recuperação de informações de contexto é feita utilizando-se pares do tipo chave-valor, e possivelmente um parâmetro no caso das informações parametrizadas. Mais especificamente, isso é possível através do módulo de fachada, invocando o método *retrieveContextInformation* da classe *MiddlewareFacade*. Com esse propósito, deve-se fornecer a esse método a chave da informação de contexto a ser recuperada e seu parâmetro, se assim for necessário, como ilustrado no primeiro passo da Figura 5.15. Utilizando essa chave, o módulo de fachada consulta a tabela de informações de contexto (passo 2), para assim recuperar o *alias* do serviço associado à informação desejada (passo 3). De posse desse *alias*, é possível então invocar o serviço no *PCC* correto (passo 4), passando a chave da informação requisitada e possivelmente seu parâmetro, para por fim retornar o valor obtido para a aplicação requisitante (passo 5).

Internamente nos *PCCs*, a recuperação de informação de contexto é implementada através da *API* de reflexão da linguagem *Java*. Mais precisamente, esses *plug-ins* definem, para cada informação de contexto, qual o método responsável por retorná-la, o que é realizado através do método *addProvidedContextInformation* da classe *ContextAwarenessPlugin*. Como visto na Seção 5.3.2, esse método recebe como parâmetro a chave de uma das informações de contexto do *plug-in* e o método utilizado para recuperá-la. Essas informações são guardadas em uma tabela gerenciada pelo *PCC* (*tabela de informações providas*), cujas entradas tem como chave a chave da informação de contexto e como valor o método a ser utilizado para recuperá-la. Assim, sempre que uma informação de contexto é requisitada em um *PCC*, tudo que o mesmo precisa fazer é obter da tabela de informações providas o método correspondente, utilizando a chave recebida no parâmetro. Esse método será então invocado, e seu retorno repassado ao módulo de fachada, que, por sua vez, o repassa novamente à aplicação requisitante. Todo esse processo de invocação do método correspondente

a uma informação é completamente gerenciado pelo *middleware*, de forma que o desenvolvedor de um *PCC* não precisa se preocupar com esses detalhes.

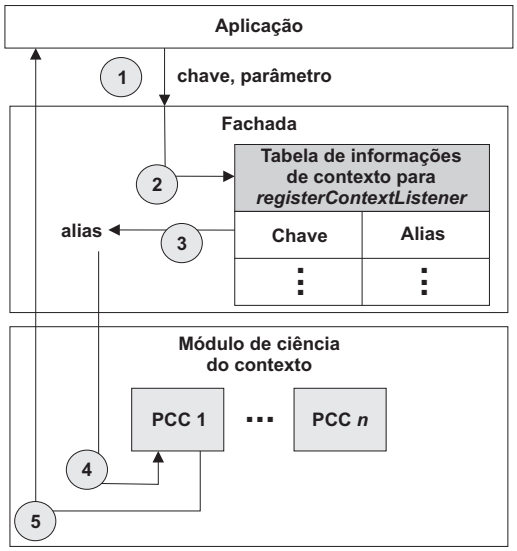


Figura 5.15: Recuperação de informações de contexto através do módulo de fachada.

A outra forma disponibilizada pelo *Wings* para a obtenção de informações de contexto é o mecanismo de eventos de contexto. Em linhas gerais, esse mecanismo permite associar uma condição a uma informação de contexto e especificar um ouvinte que será notificado sempre que a condição se tornar verdadeira. Para isso, cada condição (i.e., instância de *ContextCondition*) possui a chave da informação associada a mesma e um parâmetro, se a informação for parametrizada. Ambos são utilizados para, de tempos em tempos, verificar se a condição está satisfeita ou não. O processo completo envolvido na notificação de eventos de contexto do *Wings* é exibido na Figura 5.16. Tudo começa quando uma aplicação deseja registrar uma condição a um ouvinte de contexto (i.e., instância de *ContextEventListener*) (passo 1). A chave associada à condição especificada é utilizada para consultar a tabela de *aliases* do serviço *registerContextListener* (passo 2). Com isso obtém-se o *alias* para o serviço que permitirá registrar o ouvinte à condição (passo 3). Esse serviço é então invocado no *plug-in* correto, passando o ouvinte e a condição como parâmetros (passo 4). O *plug-in* irá então iniciar uma nova linha de execução (i.e., uma instância de *ContextEventThread*, explicada na Seção 5.3.2), na qual a condição será verificada regularmente, mais precisamente a cada segundo, de forma que quando for satisfeita o ouvinte será notificado (passo 5). Nesse processo de verificação, cada linha de execução recupera o valor atual da informação através do

método *retrieveContextInformation* da classe *Middleware*.

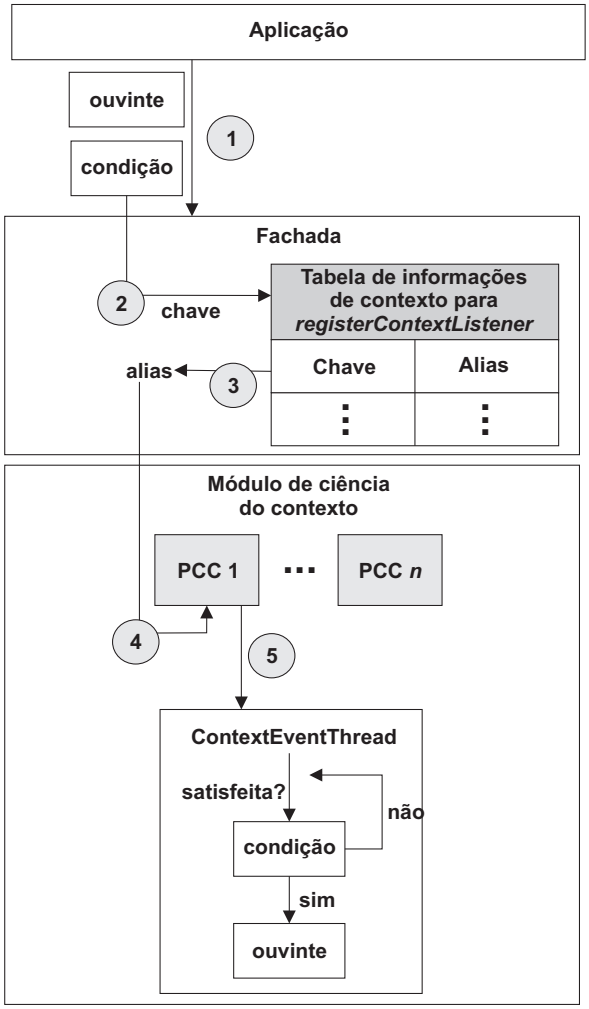


Figura 5.16: Notificação de eventos de contexto através da fachada.

O processo de remover o cadastro de um ouvinte de contexto é similar ao que acabamos de apresentar, de forma que podemos exemplificá-lo através da figura anterior. Da mesma forma como no processo de cadastro, o primeiro passo é passar para o módulo de fachada um ouvinte de contexto e a condição da qual seu cadastro será removido. Após isso, o *alias* do serviço *unregisterContextListener* será recuperado utilizando a chave da informação de contexto associada à condição (passo 2). A diferença nesse passo, com relação à figura, é que esse *alias* será recuperado da tabela do serviço *unregisterContextListener*. Tendo recuperado o *alias* para o serviço (passo 3), o mesmo será então invocado no *plug-in* correspondente (passo 4). Este, ao receber a requisição do serviço, irá verificar qual é a linha de execução associada ao evento e condição passados, para então interrompê-la.

5.5 Análise de Performance

Nesta seção, iremos apresentar as análises realizadas sobre o *Wings*, tendo em vista duas características: custo de processamento e utilização de memória. Discutiremos também alguns problemas encontrados durante a realização dessas análises bem como os resultados obtidos ao longo desse processo. A análises aqui apresentadas foram realizadas utilizando os seguintes dispositivos:

- *Smart phone Nokia 9500*:
  - Processador: *OMAP TI ARM RISC* de 150 *MHz*
  - Memória *RAM*: 64 *MB*
  - Sistema operacional: *Symbian* 7.0
  - Máquina virtual: *J9 Virtual Machine*
- *PDA HP iPAQ hx4700*:
  - Processador: *Intel PXA270* de 624 *MHz*
  - Memória *RAM*: 64 *MB*
  - Sistema operacional: *Windows Mobile 2003*
  - Máquina virtual: *CrE-Me Virtual Machine* versão 4.0 [2]

5.5.1 Custo de Processamento

Nesta análise, nosso interesse é medir o tempo de processamento das operações básicas do *Wings*. Com esse propósito, as seguintes tarefas foram realizadas:

1. Medir o tempo para adicionar e remover *PCCs*: nesta tarefa foi observado o tempo para adicionar e remover *PCCs* no *middleware*. Como apresentado na Seção 5.4.1, essas as operações exigem, respectivamente, a inserção e remoção de entradas na tabela de informações de contexto, referentes às informações de contexto disponibilizadas pelo *PCC*. Dessa forma, o número de tais informações deve ser levado em conta ao medirmos os tempos de inserção e remoção desses *plug-ins*. Com esse propósito, variamos a

[2] http://www.nsicom.com/Default.aspx?tabid=138

quantidade de informações de contexto providas pelo *PCC* a ser adicionado/removido, para assim termos uma idéia do seu efeito sobre as operações de inserção e remoção. Mais precisamente, consideramos a inserção de *PCCs* com 5, 10 e 15 informações de contexto.

2. Medir o tempo para recuperar informações de contexto: o objetivo dessa tarefa foi medir o tempo para recuperar uma informação de contexto a partir do módulo de fachada.

3. Medir o tempo para adicionar e remover *PDSs*: esta tarefa consistiu em obter os tempos para adicionar e remover um *plug-in* de disponibilização de serviços.

4. Medir o tempo para iniciar a descoberta de serviços: nessa tarefa medimos o tempo que se leva para iniciar uma busca por serviços a partir do módulo de fachada. Essa tarefa foi realizada variando-se o número de *PDSs* instalados no *middleware*, pois, como a descoberta de serviços deve ser iniciada em todos os *PDS* instalados, o tempo que queremos obter varia à medida que mais e mais desses *plug-ins* são inseridos no *middleware*.

5. Medir o tempo para adicionar e remover *PDNs*: essa tarefa teve como objetivo obter os tempos de adição e remoção de um *plug-in* de descoberta de nós.

6. Medir o tempo para iniciar a descoberta de nós: nessa tarefa, o objetivo é obter o tempo necessário para iniciar uma busca por nós a partir do módulo de fachada. Assim como na descoberta de serviços, esse tempo é afetado pelo número de *PDNs* inseridos no *middleware*. Dessa forma, de maneira similar à tarefa descrita no item 4, esse número deve ser levado em consideração ao obtermos o tempo de inserção de um *PDN*.

7. Medir o tempo para registrar/desregistrar um ouvinte de contexto: nessa última tarefa, procuramos obter os tempos para registrar e desregistrar um ouvinte de contexto através do módulo de fachada.

Durante a realização das tarefas acima, o principal problema encontrado foi o fato de que os tempos obtidos variavam bastante entre si. Era comum, por exemplo, obter diferenças de até 16 milisegundos em execuções sucessivas de uma mesma tarefa. Isso pode ser percebido no gráfico da Figura 5.17, que exibe o tempo para inserir um *PCC* com 5 informações de contexto no *Nokia 9500*, ao longo de 20 execuções.

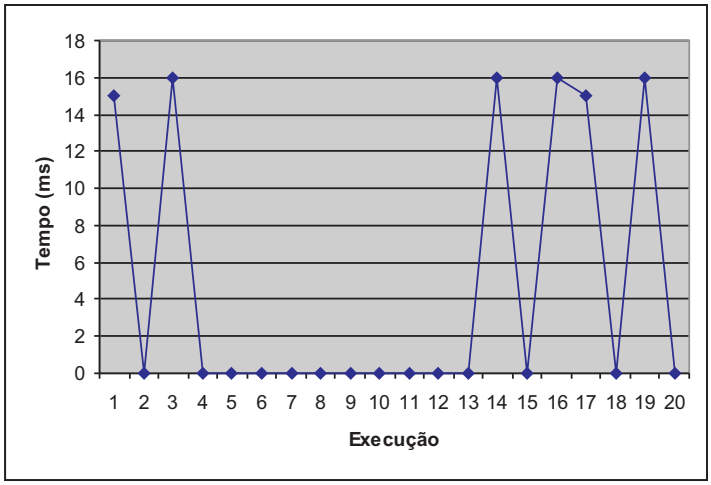


Figura 5.17: Valores do tempo de inserção de um *PCC* com 5 informações de contexto no *Nokia 9500*.

Resultados desse tipo eram, para nós, pouco conclusivos. De forma a obter valores mais precisos, nossa idéia foi executar cada tarefa 100 vezes, calcular a média, repetir essa operação um certo número de vezes, para finalmente observar a variação das médias obtidas. Essa idéia se mostrou bastante interessante, pois, as médias que obtivemos apresentaram pouquíssimas variações entre si. Por exemplo, para a inserção de um *PCC* com 5 informações de contexto, obtivemos tempos de 2 a 3 milisegundos, no *Nokia 9500*, e de 1 a 2 milisegundos, no *HP iPAQ hx4700*. Dessa forma, utilizamos essa abordagem para todas as tarefas, obtendo por fim os resultados exibidos nas tabelas 5.1 à 5.7. Todos os tempos apresentados nessas tabelas estão expressos em milisegundos.

No. de informações de contexto	Tempo de Inserção (ms)		Tempo de Remoção (ms)	
	Nokia 9500	HP iPAQ hx4700	Nokia 9500	HP iPAQ hx4700
5	2 a 3	1 a 2	1 a 2	0 a 1
10	3 a 4	2 a 3	2 a 3	0 a 1
15	4 a 5	3 a 4	3 a 4	1 a 2

Tabela 5.1: Tempos obtidos para a adição e remoção de *PCCs* através do módulo de fachada

Dispositivo	Tempo (ms)
Nokia 9500	0 a 1
HP iPAQ hx4700	0 a 1

Tabela 5.2: Tempos obtidos para a recuperação de informação de contexto através do módulo de fachada.

Dispositivo	Tempo de inserção (ms)	Tempo de remoção (ms)
Nokia 9500	2 a 4	1 a 3
HP iPAQ hx4700	1 a 2	0 a 1

Tabela 5.3: Tempos obtidos para a inserção e remoção de *PDSs* através do módulo de fachada.

No. de PDNs no middleware	Nokia 9500	HP iPAQ hx4700
5	73 a 75	4 a 6
10	147 a 148	10 a 11

Tabela 5.4: Tempos obtidos, em milisegundos, para iniciar uma busca por serviços através do módulo de fachada.

Dispositivo	Tempo de inserção (ms)	Tempo de remoção (ms)
Nokia 9500	1 a 2	1 a 2
HP iPAQ hx4700	0 a 1	0 a 1

Tabela 5.5: Tempos obtidos para a inserção e remoção de *PDNs* através do módulo de fachada.

No. de PDSs no middleware	Nokia 9500	HP iPAQ hx4700
5	73 a 75	3 a 6
10	147 a 148	10 a 11

Tabela 5.6: Tempos obtidos, em milisegundos, para iniciar uma busca por nós através do módulo de fachada.

	Tempo para registrar ouvinte (ms)	Tempo para remover um ouvinte (ms)
Nokia 9500	32 a 34	0 a 1
HP iPAQ hx4700	3	0 a 1

Tabela 5.7: Tempos obtidos para registrar e desregistrar um ouvinte de contexto através do módulo de fachada.

Dos resultados obtidos, consideramos que a grande maioria encontra-se numa faixa de tempo que não compromete seriamente a performance do *middleware*. No entanto, em três situações, descoberta de nós, descoberta de serviços e registro de ouvintes de contexto, todas envolvendo o *Nokia 9500*, os resultados obtidos mostraram-se elevados. Perceba, por exemplo, que para ambos os tipos de descoberta, os tempos foram de 73 a 75 milisegundos para 5 *PDSs* e *PDNs* instalados no *middleware*. Para se ter uma idéia melhor, fazendo alguns cálculos, pode-se concluir que o tempo dessas operações considerando apenas 1 *PDS*/*PDN* no *middleware* é de aproximadamente 15 milisegundos. Note que esse tempo está bem acima dos obtidos para as demais operações, os quais variaram entre 0 e 4 milisegundos (excluindo-se o tempo para registro de ouvintes de contexto). Embora valores como esses não cheguem a ser um desastre para o desempenho do *Wings*, os mesmos são, certamente, pontos a serem melhorados.

5.5.2 Utilização de Memória

Nosso objetivo com esta análise é medir a utilização de memória *RAM* do *Wings*. Para isso, iremos submeter o *middleware* aos seguintes cenários:

- Nenhum *plug-in* instalado: neste cenário, o *middleware* não terá nenhum *plug-in* instalado.

- Número variável de *PCCs*: este cenário consiste em variar o número de *PCCs* no *middleware* e observar as diferenças em sua utilização de memória *RAM*. Para este cenário, consideraremos situações de 1, 5, 10 e 15 *PCCs* inseridos no *middleware*. Além disso, o número de informações de contexto também será variado em cada situação.

- Número variável de *PDSs*: neste cenário o *Wings* será submetido a diferentes quantidades de *PDSs* instalados. Assim como no cenário anterior, observaremos as variações em sua utilização de memória considerando 1, 5, 10 e 15 *PDSs* instalados.

- Número variável de *PDNs*: assim como nos dois cenários anteriores, queremos aqui observar a variação na utilização de memória do *middleware*, desta vez, submetendo-o a diferentes números de *PDNs* (novamente, 1, 5, 10 e 15).

Para obter a utilização de memória em cada um dos cenários acima, foram realizados os seguintes passos: 1) recuperar a quantidade de memória *RAM* livre, 2) carregar o *middleware* de acordo com o cenário em questão, 3) recuperar novamente a quantidade de memória *RAM* livre e, por fim, 4) subtrair os valores obtidos nos passos 1 e 3. Para recuperar a quantidade de memória livre, utilizou-se o método *getFreeMemory* da classe *Runtime*.

Nesse contexto, um ponto interessante a ser comentado é o fato de que, embora tenhamos realizado com sucesso as análises no *Nokia 9500*, ao repeti-las no *HP iPAQ hx4700* os resultados obtidos não eram condizentes. Para se ter uma idéia do que estamos falando, em algumas situações obtivemos valores negativos para a utilização de memória do *Wings*. Acreditando ser esse um problema da máquina virtual que estávamos utilizando (*CrE-Me virtual machine*), decidimos, para essa análise, utilizar uma outra máquina virtual, a *Mysaifu JVM* [3]. No entanto, os resultados que obtivemos foram, para nós, insatisfatórios. Por exemplo, ao

[3] http://www2s.biglobe.ne.jp/ dat/java/project/jvm/index_en.html

medirmos a utilização de memória do *middleware* com 5 *PCCs*, cada um com 5 informações de contexto, obtivemos, nessa máquina virtual, um valor de 1355 KB, ou 1.3 MB aproximadamente. Para nós, os valores obtidos através dessa máquina virtual não representavam o uso real de memória *RAM* do *Wings* no *HP iPAQ hx4700*, principalmente quando os comparávamos à utilização obtida no *Nokia 9500* e em computadores pessoais (executamos também os diferentes cenários em um computador pessoal apenas para compararmos com os resultados do *HP iPAQ hx4700)*. Dessa forma, não realizamos a análise de utilização de memória no *HP iPAQ hx4700*. Os resultados que obtivemos para o *Nokia 9500*, no entanto, podem ser vistos nas Tabelas 5.8, 5.9 e 5.10, nas quais estão exibidos os valores para a utilização de memória dos três últimos cenários descritos. Para o primeiro cenário, o resultado obtido foi de 231 KB. Em todas as tabelas, a utilização de memória está expresso em *kilobytes*.

No. de PCCs	5 informações de contexto	10 informações de contexto	15 informações de contexto
1	235	235	235
5	243	281	281
10	325	328	376
15	376	376	426

Tabela 5.8: Utilização de memória *RAM* baseado no número de *PCCs* instalados no *middleware* e nas informações de contexto de cada um.

No. de PDNs	Utilização de memória
1	244
5	285
10	330
15	331

Tabela 5.9: Utilização de memória *RAM* baseado no número de *PDNs* instalados no *middleware*.

No. de PDSs	Utilização de memória
1	235
5	279
10	326
15	374

Tabela 5.10: Utilização de memória *RAM* baseado no número de *PDSs* instalados no *middleware*.

Os resultados referentes à utilização de memória foram, de certa forma elevados. Perceba que, com 5 *PDNs* instalados, por exemplo, a utilização de memória é na faixa de 285 KB. Embora não pareça muito, à primeira vista, devemos lembrar que os *PDNs* utilizados nas análises eram, na verdade, implementações vazias. Isso quer dizer que, implementações reais desse tipo de *plug-in* terão, certamente, uma utilização de memória consideravelmente maior. Ao estendermos essa consideração para para os outros tipos de *plug-ins* (i.e., *PDSs* e *PCCs*), não é difícil concluir que a utilização de memória do *middleware* possa facilmente ultrapassar 1 MB.

Vale a pena frisar que, dependendo da situação, isso pode ser facilmente contornado apenas removendo-se *plug-ins* que não estejam sendo utilizados. Isso não será possível, no entanto, em situações em que muitos *plug-ins* estejam instalados e sendo realmente utilizados. Nesse escopo, um comentário a se fazer é que, pelo menos em se tratando de *PDSs* e *PDNs*, acreditamos serem raras a situações em que, por exemplo, 5 de cada um desses *plug-ins* estarão em uso. Essas situações tornam-se mais raras ainda se aumentarmos o número de cada um desses *plug-ins* para 10 ou 15. Ainda assim, nessas raras situações, tal utilização de memória é um preço justo a ser pago pelas características de heterogeneidade e flexibilidade que o *middleware* provê. Outro ponto a ser considerado é que, olhando o ritmo de crescimento no poder de memória dos equipamentos eletrônicos, é provável que a capacidade de memória dos dispositivos móveis aumente ao ponto de a utilização de memória do *Wings* não se tornar mais um problema. Além disso, é importante frisar que esses resultados podem vir a ser melhorados, através de um refatoramento na implementação do *Wings*.

Capítulo 6

Estudo de Caso

"A lógica da validação permite nos movermos entre os limites do dogmatismo e do ceticismo" (Paul Ricoeur)

Como uma forma de validar a implementação do *Wings*, foram desenvolvidas duas aplicações como estudo de caso. Essas aplicações têm como foco o Laboratório de Sistemas Embarcados e Computação Pervasiva (http://embedded.dee.ufcg.edu.br), da Universidade Federal Campina Grande. Mais precisamente, as seguintes aplicações foram desenvolvidas:

- Biblioteca Pervasiva: essa aplicação provê duas funcionalidades, busca por livros na biblioteca do laboratório e registro de livros de interesse. A primeira delas permite ao usuário procurar por livros a partir de um conjunto de palavras-chave, as quais serão comparadas com o título dos livros cadastrados na biblioteca. Já a segunda funcionalidade, permite que se registre o interesse em um livro que esteja emprestado, com o intuito de que, quando este for devolvido, o usuário seja notificado.

- Porteiro Eletrônico: a aplicação de Porteiro Eletrônico permite ao usuário abrir a porta do laboratório através do seu celular.

Dada as descrições iniciais acerca de cada aplicação, vamos apresentar agora os detalhes relacionados ao estudo de caso em questão. Para esse fim, iremos descrever inicialmente o ambiente no qual essas aplicações executam. A partir disso, apresentaremos a arquitetura das principais aplicações executando em tal ambiente, para finalmente, mostraremos como a Biblioteca Pervasiva e o Porteiro Eletrônico implementam suas funcionalidades.

6.1 Configuração do Ambiente

O cenário do estudo de caso em questão encontra-se configurado como ilustrado na Figura 6.1, na qual os seguintes elementos podem ser destacados: *clientes*, *ponto de acesso*, *servidor da biblioteca*, *dispositivo Bluetooth*. Os clientes representam qualquer nó que faça uso dos serviços e informações disponíveis no ambiente, seja através de uma interface de rede sem fio (no nosso caso *Bluetooth* ou *Wi-Fi*) ou cabeada. O ponto de acesso, por sua vez, provê conectividade *Wi-Fi* para os demais dispositivos no ambiente. Já o servidor da biblioteca é responsável por manter a base de dados da biblioteca e por implementar os serviços para acesso às suas informações. É através desses serviços, portanto, que a aplicação da Biblioteca Pervasiva implementa suas funcionalidades. Com esse propósito, os serviços do servidor da biblioteca recebem as requisições dos clientes através da rede cabeada. Assim, além dos clientes ligados diretamente a mesma, dispositivos móveis equipados com interface de rede *Wi-Fi* podem também fazer uso desses serviços, através do ponto de acesso. Por fim, o dispositivo *Bluetooth* é responsável por abrir a porta do laboratório. Com esse propósito, tal dispositivo envia sinais diretamente à fechadura eletrônica da porta, sempre que são recebidas mensagens de dispositivos clientes.

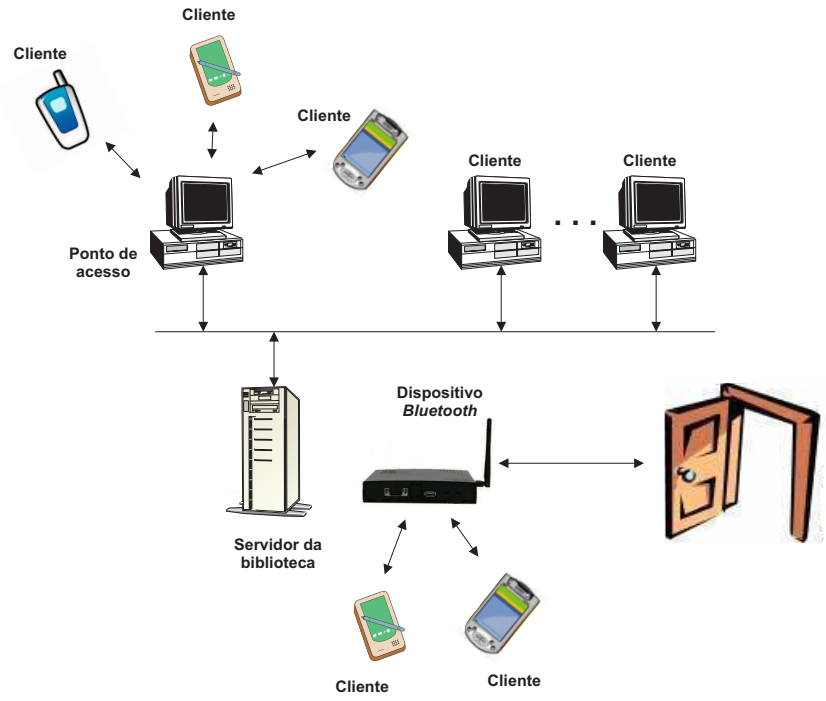


Figura 6.1: Configuração do ambiente no qual as aplicações irão executar.

6.2 Aplicações Executando no Ambiente

Dado o ambiente descrito na seção anterior, vale a pena destacar duas das entidades apresentadas, o servidor da biblioteca e os clientes, descritos nas seções seguintes.

6.2.1 O Servidor da Biblioteca

Como apresentado, o servidor da biblioteca é responsável por manter o banco de dados da biblioteca e implementar os serviços para acesso à mesma. Com esse propósito, o mesmo utiliza o *SGBD MySQL* [1], para a base de dados, e *web services*, para a implementação dos serviços. Para estes últimos, mais especificamente, foram utilizadas duas *APIs* da *Apache* [2], *Axis* [3] e *JUDDI* [4], ambas executando sobre o servidor *Web Tomcat* [5]. A primeira é utilizada para o *deployment* e execução dos serviços. Mais especificamente, a *API Axis* é utilizada para tornar um serviço acessível através de uma *URL*, para que assim o mesmo possa receber invocações dos clientes. No entanto, para se acessar um serviço através dessa *API*, é necessário saber *a priori* o endereço do mesmo. Como queremos descobrir os serviços da biblioteca em tempo de execução, essa abordagem não é interessante para o estudo de caso em questão. É devido a isso que utilizamos também a *API JUDDI*, a qual permite publicar serviços e descobri-los dinamicamente. Com esse propósito, essa *API* faz uso das informações acerca dos serviços atualmente gerenciados pela *API Axis*, permitindo que uma aplicação os descubra através de alguns critérios. A implementação padrão da *API JUDDI*, por exemplo, implementa um mecanismo para a busca de serviços que consiste em comparar palavras-chave com os nomes dos serviços publicados. Para mecanismos mais complexos, no entanto, faz-se necessário extender a *API*.

Dado isso, a arquitetura do servidor da biblioteca encontra-se organizada como ilustrado na Figura 6.2. Nessa figura, a camada de mais alto nível, *serviços da biblioteca*, representa os dois serviços implementados pelo servidor da biblioteca: *serviço de busca de livros* e *serviço de consulta de livro*. O primeiro deles permite que uma aplicação busque por um conjunto

[1] http://www.mysql.com
[2] http://www.apache.org
[3] http://ws.apache.org/axis
[4] http://ws.apache.org/juddi
[5] http://tomcat.apache.org

de livros cadastrados na base de dados do servidor da biblioteca. Para isso, o serviço deve receber um conjunto de palavras-chave, as quais serão combinadas com os nomes dos livros cadastrados. Os livros cujos nomes contêm as palavras-chave passadas terão, portanto, suas informações retornadas à aplicação requisitante. Mais especificamente, retorna-se o nome do livro, sua descrição e *ISBN*. O serviço de consulta de livro, por sua vez, permite verificar a disponibilidade de um livro específico. Com esse propósito, uma aplicação deve fornecer o nome do livro a ser consultado, de forma que o serviço irá então retornar verdadeiro se o mesmo estiver disponível e falso caso contrário. Esses dois serviços são, portanto, utilizados pela aplicação de Biblioteca Pervasiva na implementação de suas funcionalidades. Mais detalhes sobre as mesmas serão dados respectivamente nas Seções 6.4 e 6.5.

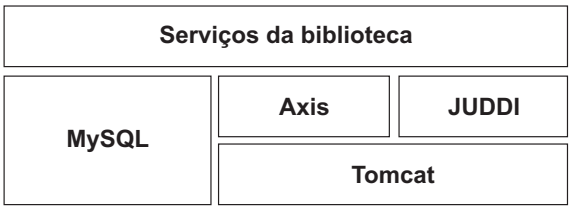


Figura 6.2: Arquitetura do servidor da biblioteca.

6.2.2 Aplicações Cliente

Para as aplicações cliente, foram implementados três *plug-ins*, mais precisamente, um *PCC*, um *PDSs* e um *PDN*. O *PCC* disponibiliza apenas uma informação de contexto, parametrizada. Essa informação tem como chave a *string isAvailable*, e refere-se à disponibilidade de livros na biblioteca. Para recuperá-la, o nome do livro que se quer verificar deve ser passado como parâmetro. É a partir dessa informação, obtida através do serviço de consulta de livros do servidor da biblioteca, que a aplicação da Biblioteca Pervasiva permite ao usuário registrar livros de interesse. Um outro *plug-in* implementado foi um *PDS*, o qual utiliza *web services*, através da *API kSOAP* [6], e permite descobrir os serviços publicados pelo servidor da biblioteca. É através desse *PDS*, portanto, que a aplicação da Biblioteca Pervasiva realiza a busca de livros e que o *PCC* recupera sua informação de contexto, como mostraremos respectivamente nas Seções 6.4 e 6.5. Por fim, o último dos *plug-ins* implementados, um

[6] http://ksoap.org

PDN, é responsável por encontrar nós *Bluetooth* nas proximidades. Dessa forma, esse *plug-in* é utilizado pela aplicação de Porteiro Eletrônico para descobrir o dispositivo *Bluetooth*, de forma a enviar ao mesmo o comando para que a porta do laboratório seja aberta.

Para o estudo de caso em questão, as aplicações cliente foram executadas em um único dispositivo, um *smart phone Nokia9500* equipado com interfaces de rede *Wi-Fi* e *Bluetooth*. Com isso, dado os *plug-ins* descritos acima, a configuração do *Wings* no dispositivo em questão pode ser ilustrada através da Figura 6.3.

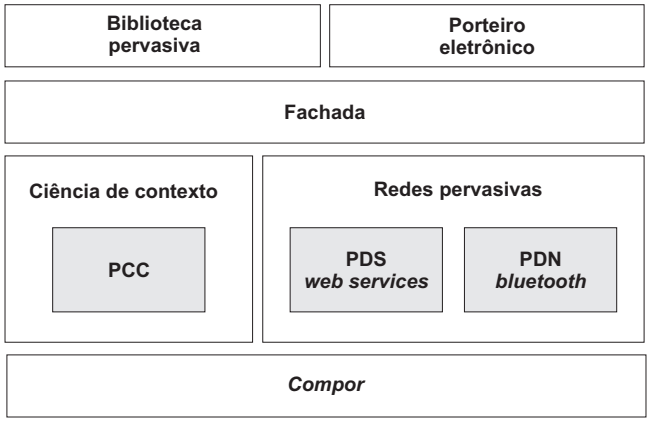


Figura 6.3: Configuração do *Wings* no *Nokia 9500* para o estudo de caso em questão.

6.3 *Deployment* e Publicação dos Serviços da Biblioteca

Como mencionado, o servidor da biblioteca implementa dois serviços, um para a busca de livros e outro para a consulta sobre a disponibilidade dos mesmos. Para que esses serviços sejam utilizados pela aplicação da Biblioteca Pervasiva, no entanto, os mesmos precisam ser publicados. O primeiro passo rumo a esse objetivo é realizar o *deployment* desses serviços, através da *API Axis*. Com esse propósito, cada serviço do servidor da biblioteca possui um arquivo de descrição no formato *XML*, com extensão *.wsd* (de *Web Services Descriptor*), os quais contêm informações como o nome do serviço e a classe que o implementa. Para os serviços do servidor da biblioteca, esses arquivos de descrição apresentam as informações exibidas nas Listagens 6.1 e 6.2. Esses arquivos são então passados para um *servlet* da *API Axis*, chamado de *AdminService*. Esse *servlet* utiliza, portanto, as informações contidas em tais arquivos para assim realizar o *deployment* dos serviços do servidor

da biblioteca. Ao término desse processo, cada serviço é associado a uma *URL*, no formato *http://endereco:porta/axis/nomeDoServico*, através da qual os clientes podem utilizar o serviço. Nessa *URL*, os campos *endereço* e *porta* referem-se ao endereço de rede e porta da máquina na qual a *API Axis* está executando. O campo *axis* refere-se ao contexto da *API Axis* dentro servidor *Tomcat*. Por fim, o campo *nomeDoServiço* indica o nome do serviço acessível através da *URL*, o qual é especificado no seu arquivo de descrição, mais especificamente através do atributo *name* da *tag service*.

Listing 6.1: Arquivo de descrição do serviço de busca de livros.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <service name="SearchBooks" provider="java:RPC">
        <parameter name="className"
        value="org.percomp.wings.services.webservices.pervasivelibrary.SearchBooksService"/>
        <parameter name="allowedMethods" value="*"/>
    </service>
</deployment>
```

Listing 6.2: Arquivo de descrição do serviço de consulta de livro.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <service name="CheckBook" provider="java:RPC">
        <parameter name="className"
        value="org.percomp.wings.services.webservices.pervasivelibrary.CheckBookService"/>
        <parameter name="allowedMethods" value="*"/>
    </service>
</deployment>
```

6.4 Procurando Livros na Biblioteca

A busca de livros através do *Nokia 9500* é realizada através da aplicação da Biblioteca Pervasiva, envolvendo três passos: 1) encontrar o serviço de busca de livros, 2) coletar as palavras-chave a serem utilizadas na busca e finalmente 3) invocar o serviço passando as mesmas. Para o primeiro passo, como já mostramos, a aplicação deve invocar o método *discoverServices* da classe *MiddlewareFacade*, passando um conjunto de palavras-chave representando a funcionalidade desejada. Com esse objetivo, a aplicação fornece duas palavras-chave, “search” e “book”. O *middleware* então se encarregará de buscar os serviços com as funcionalidades

desejadas, os quais serão recebidos pela aplicação como instâncias de *ServiceProxy*. Vale lembrar que, em nosso exemplo, apenas um serviço com essas propriedades existe no ambiente. Dessa forma, para facilitar a implementação da aplicação cliente, o primeiro (e único) serviço que for descoberto com essas propriedades é utilizado pela mesma. Note, portanto, que o passo de seleção foi desconsiderado em nosso estudo de caso. Mapeando esses passos para as configurações do servidor da biblioteca e do cliente, obtemos a seqüência ilustrada na Figura 6.4. Como ilustrado em tal figura, a aplicação da Biblioteca Pervasiva requisita a busca por serviços através do módulo de fachada, o qual irá então repassar essa requisição para os *PDSs* instalados, nesse caso, apenas para *PDS web services* (passo 1). Este, ao receber a requisição, irá se comunicar com o servidor da biblioteca, mais precisamente, enviando uma *query* à *API JUDDI*, encapsulada em uma mensagem *SOAP* (passo 2). Essa query é formatada da seguinte forma, "%search%book%", indicando que se deseja qualquer serviço cujo nome contenha as palavras "*search*" e "*book*", nessa ordem. Nesse momento, a *API JUDDI* irá então consultar os serviços disponíveis no servidor da biblioteca através da *API Axis*, comparando seus nomes com as palavras-chave recebidas. Feito isso, a *API JUDDI* irá então retornar ao *PDS* requisitante as descrições de cada serviço que contenha essas palavras no nome (passo 3). Essas descrições serão então utilizadas pelo *PDS* para criar as instâncias de *ServiceProxy* relacionadas aos serviços descobertos, para que assim o mesmo possa passá-las à aplicação da Biblioteca Pervasiva (passo 4).

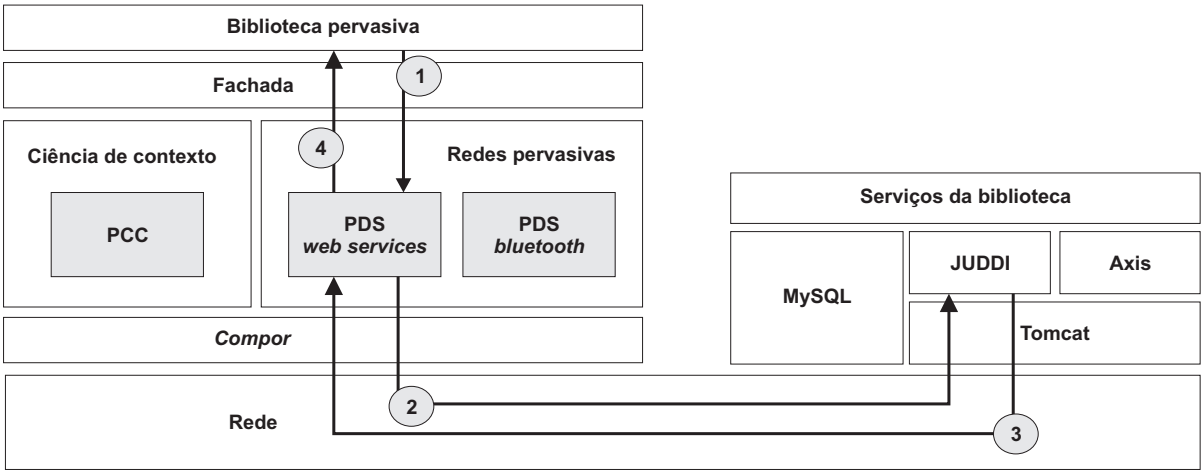


Figura 6.4: Processo de descoberta do serviço de busca de livros no servidor da biblioteca.

Uma vez vez que a aplicação está de posse do serviço para busca de livros, a mesma irá então coletar as palavras-chave que o usuário deseja utilizar na busca. Feito isso, as mesmas serão passadas para o método *invoke* do objeto *ServiceProxy* referente ao serviço de busca de livros. Como vimos na Seção 6.2.1, ao serem recebidas remotamente pelo mesmo, tais palavras-chave serão comparadas com os nomes dos livros cadastrados na base de dados do servidor da biblioteca. As informações dos livros cujos nomes contêm as palavras-chave passadas serão, portanto, retornadas como resposta à invocação do serviço. A aplicação então exibe em uma lista o nome de cada um dos livros retornados, permitindo ao usuário selecionar um item em particular, para assim obter mais detalhes sobre o livro associado. As telas da aplicação da Biblioteca Pervasiva, executando em um emulador do *Nokia 9500*, referentes à coleta das palavras-chave, exibição dos livros encontrados e dos seus detalhes são exibidas respectivamente nas Figuras 6.5, 6.6 e 6.7.

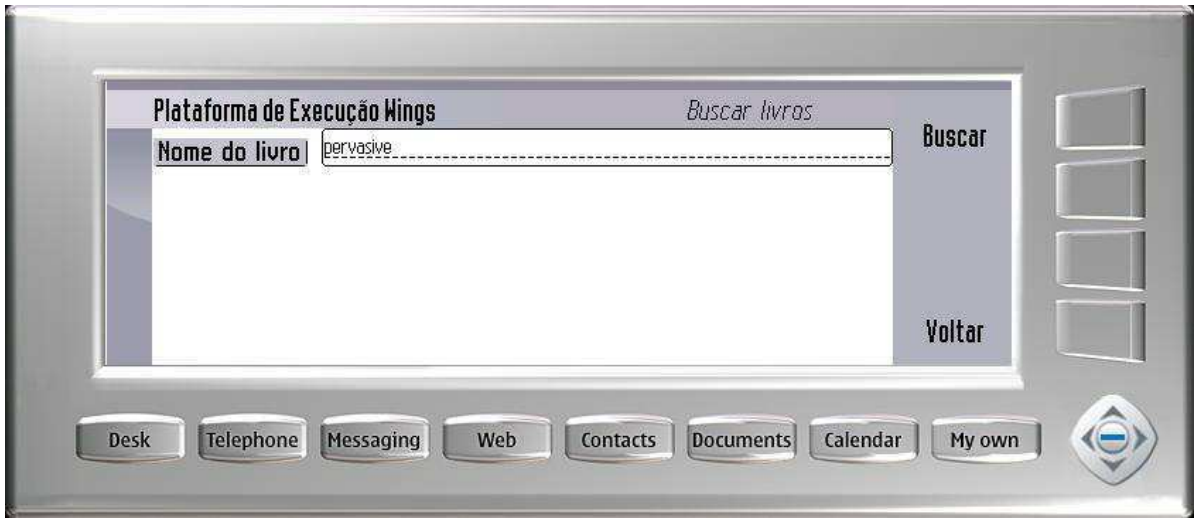


Figura 6.5: Tela da aplicação da Biblioteca Pervasiva para coletar as palavras-chave a serem usadas na busca por livros.

Figura 6.6: Tela da aplicação da Biblioteca Pervasiva para exibição dos livros encontrados durante a busca.

Figura 6.7: Tela da aplicação da Biblioteca Pervasiva para exibição dos detalhes de um livro selecionado.

6.5 Recebendo Notificações Sobre um Livro de Interesse

A outra funcionalidade da Biblioteca Pervasiva é o registro de livros de interesse. Com esse propósito, o usuário deve inicialmente buscar por um livro, através do processo descrito na seção anterior. Dessa forma, para cada livro encontrado, o usuário pode, além de exibir seus detalhes, registrar seu interesse no mesmo. Com esse propósito, a aplicação da Biblioteca Pervasiva define uma condição, *BookAvailable*, a qual implementa a interface *Condition*,

explicada na Seção 5.3.2. Essa condição tem associada a si a chave "*isAvailable*", que conforme apresentado na Seção 6.2.2, é a chave da informação de contexto provida pelo *PCC* implementado para esse estudo de caso. Essa informação, no entanto, requer um parâmetro, o qual é representado pelo título do livro selecionado pelo usuário. Uma vez que a condição é cadastrada, portanto, o *PCC* se encarregará de verificar quando a mesma é satisfeita, descobrindo e invocando, de tempos em tempos, o serviço de consulta de livros implementado pelo servidor da biblioteca. Quando isso acontece, a aplicação da Biblioteca Pervasiva é então notificada, de forma a avisar ao usuário que um dos seus livros de interesse já está disponível na biblioteca. Esse processo é ilustrado na Figura 6.8, na qual inicialmente a aplicação da Biblioteca Pervasiva cadastra a condição através da fachada (passo 1) (utilizando o método *registerContextCondition* da classe *MiddlewareFacade*). Ao receber a requisição, o *PCC* então inicia uma linha de execução, na qual o serviço de consulta de livros será descoberto (passos 2 e 3), através da *API JUDDI*, e posteriormente invocado, através da *API Axis* (passo 4), passando o título do livro associado à condição como parâmetro. O resultado de cada invocação é então verificado (passo 5), e sendo o mesmo verdadeiro, indica que o livro está disponível, e que, portanto, a aplicação da Biblioteca Pervasiva pode notificar o usuário (passo 6). Na Figura 6.9 está ilustrada a tela utilizada para notificar ao usuário que um dos seus livros de interesse está disponível na biblioteca.

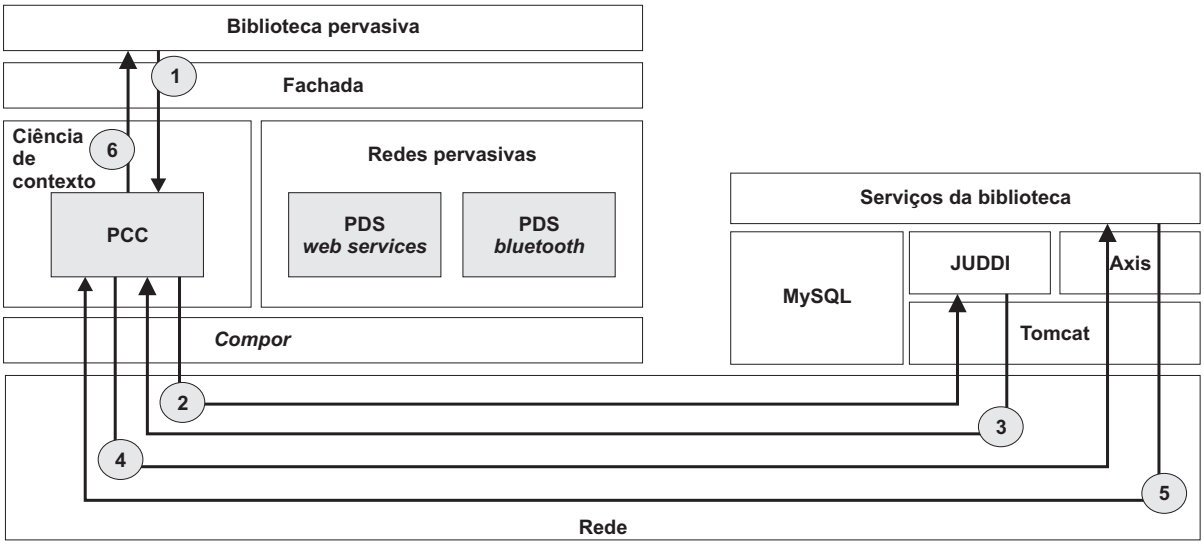


Figura 6.8: Verificação da disponibilidade de livros no servidor da biblioteca.

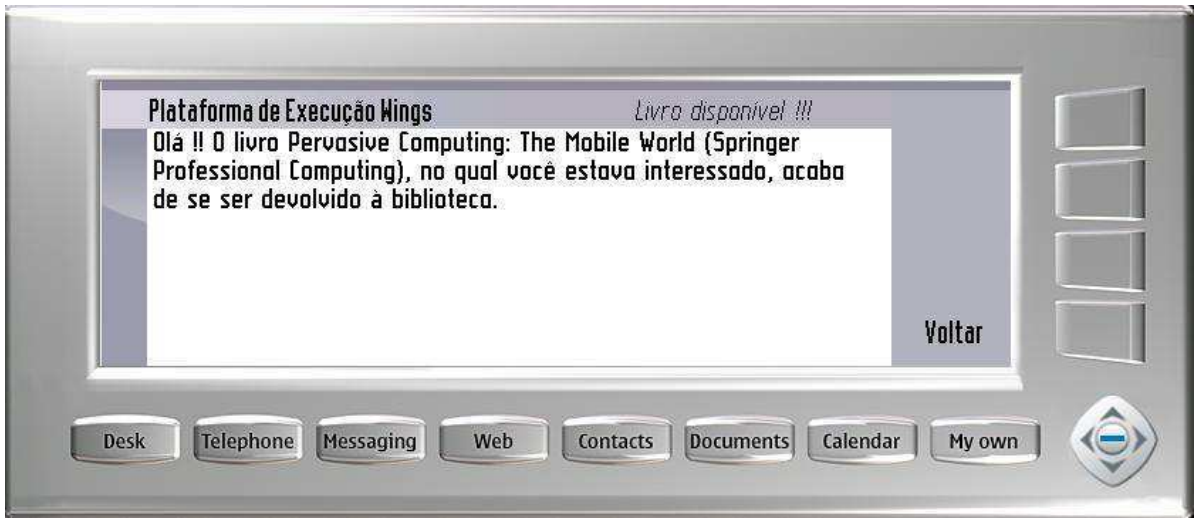


Figura 6.9: Tela da Biblioteca Pervasiva para notificação de que um livro de interesse encontra-se disponível.

6.6 Abrindo a Porta do Laboratório

A abertura da porta do laboratório, como foi visto, é realizada pelo Porteiro Eletrônico. Para isso, dois passos são realizados. No primeiro deles, a aplicação deve encontrar o dispositivo *Bluetooth* no ambiente, o que é feito através da fachada, conforme apresentado na Seção 5.4.2. Mais precisamente, a mesma invoca o método *discoverHosts* da classe *MiddlewareFacade*, passando um ouvinte com parâmetro (i.e., uma instância de *HostDiscoveryListener*). Essa requisição será então repassada aos *PDNs* instalados no *middleware*, nesse caso, somente *PDN Bluetooth*. Caso o dispositivo cliente esteja no raio de alcance da interface *Bluetooth* do dispositivo, o *PDN* irá, portanto, descobrí-lo e dessa forma encapsular suas informações (i.e., endereço, nome, dentre outras) em uma instância de *RemoteHost*, a qual será passada ao ouvinte da aplicação. De posse dessa instância, a aplicação de Porteiro Eletrônico pode, finalmente, abrir uma conexão com o dispositivo *Bluetooth*, através do método *openConnection*, para assim enviar ao mesmo o comando de abertura da porta do laboratório.

Capítulo 7

Trabalhos Relacionados

"A idéia é tentar dar todas as informações que ajudem os outros a julgar o valor da sua contribuição; não apenas as informações que levem o julgamento a uma direção em particular." (Richard Feynman)

Dado o crescente interesse da comunidade científica no campo da computação pervasiva, não é de se surpreender que muitas soluções já tenham sido desenvolvidas nesse contexto. Dessa forma, não seria possível, ou na melhor das hipóteses muito difícil, listar todas essas soluções aqui neste trabalho. Assim, neste capítulo iremos apresentar o que acreditamos ser os principais trabalhos relacionados ao *middleware Wings*. Para esse fim, os dividiremos em dois grupos. No primeiro deles, descreveremos algumas soluções exclusivamente voltadas à disponibilização de serviços. No segundo grupo, por outro lado, os principais *middlewares* para computação pervasiva, relacionados a este trabalho, serão apresentados.

7.1 Soluções para Disponibilização de Serviços

Nesta seção estamos considerando soluções concentradas especificamente na publicação, descoberta e utilização de serviços. A idéia é apresentar algumas das soluções que poderiam ser utilizadas na implementação dos *plug-ins* de disponibilização de serviço do *Wings*. Além disso, a apresentação dessas soluções serve para dar uma visão geral acerca dos esforços atuais em relação a esse aspecto em particular do *middleware*.

7.1.1 *Bluetooth*

Bluetooth é um tecnologia para comunicação sem fio entre dispositivos eletrônicos, definindo protocolos para a descoberta de nós e serviços. A descoberta de serviços é realizada pelo protocolo *SDP* [1], o qual permite enumerar os dispositivos próximos e recuperar os serviços publicados pelos mesmos. *Bluetooth* utiliza a abordagem *pull* distribuída para a disponibilização de serviços. Com esse propósito, cada dispositivo mantém um Banco de Dados de Descoberta de Serviços, ou *SDDB* [2], local no qual seus serviços são publicados. Dessa forma, para um dispositivo *Bluetooth* descobrir serviços, o mesmo deve procurá-los nos *SDDBs* dos nós próximos. Esse processo de descoberta de serviços é ilustrado na Figura 7.1. Note que, inicialmente, todos os dispositivos publicam seus serviços nos respectivos *SDDBs*, ilustrado no quadro 1. A seguir, um cliente procura por todos os nós *Bluetooth* dentro da sua área de alcance (quadro 2). Para cada nó encontrado, o cliente envia uma *query* referente à disponibilidade de um serviço de seu interesse (quadro 3). Os nós então respondem a essas *queries* informando se possuem ou não um serviço com as características desejadas (quadro 4). Uma vez que um serviço de interesse tenha sido localizado, o cliente pode, portanto, comunicar-se diretamente com o nó provedor para enfim utilizar o serviço, como ilustrado no quadro 5.

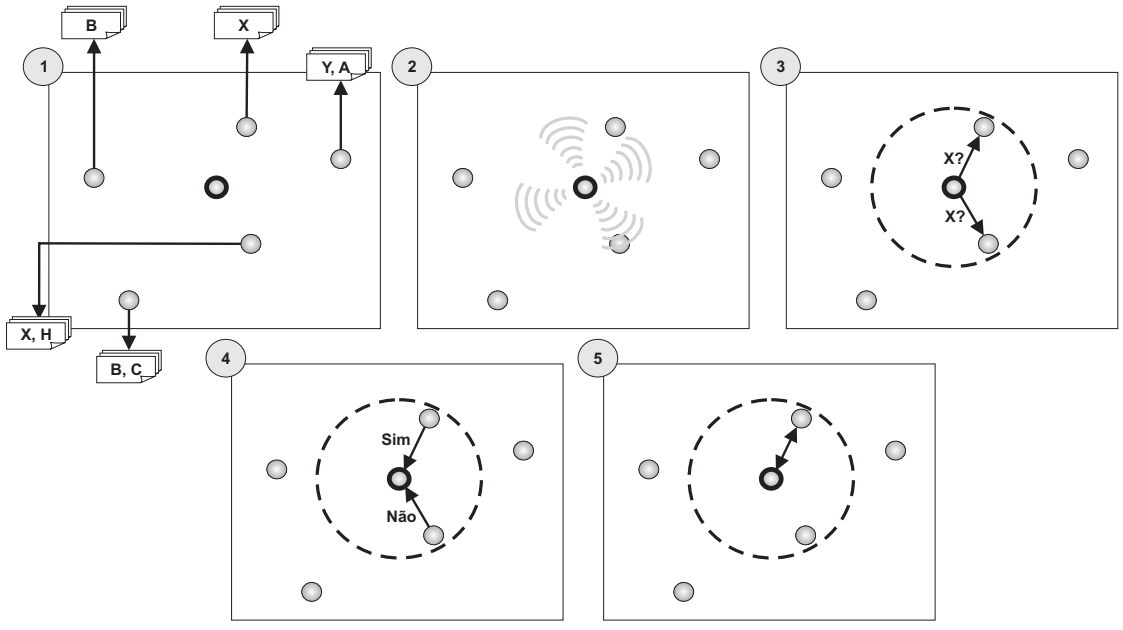


Figura 7.1: Descoberta de serviços na tecnologia *Bluetooth*.

[1] *Service Discovery Protocol*
[2] *Service Discovery Database*

7.1.2 *Web Services*

Definido pela *W3C* [3], *Web Services* (Vogels, 2003) é uma solução para computação orientada a serviços baseada em *XML*. Utilizando a abordagem *pull* centralizada, três especificações constituem o centro dessa solução: *UDDI*, *WSDL* e *SOAP*. A primeira delas, *UDDI*, ou *Universal Description, Discovery, and Integration*, é utilizado tanto na publicação quanto na descoberta de serviços. Com esse propósito, são definidas um conjunto de funções, que devem implementadas pelos registros de serviços. Exemplos de tais funções são "save_service", utilizada pelos provedores na publicação de serviços, e "find_service", utilizada pelos clientes durante a descoberta de serviços. A especificação *WSDL*, de *Web Services Description Language*, por sua vez, é utilizada na descrição de serviços. Tais descrições são acessadas no momento em que um serviço é descoberto, para se recuperar informações como suas funcionalidades específicas, parâmetros das mesmas, protocolos de acesso, dentre outras. Por fim, *SOAP*, acrônimo para *Simple Object Access Protocol*, é o protocolo utilizado em todas as interações em *Web Services* (i.e., publicação, descoberta, ligação e utilização).

7.1.3 *UPnP*

UPnP [4], ou *Universal Plug and Play*, é um protocolo aberto, baseado nos protocolos da *Internet*, com o objetivo de prover configuração automática de rede, conectividade ponto a ponto e disponibilização de serviços aos nós. Para esse fim, o protocolo *UPnP* define o seguinte conjunto de passos: *descrição*, *endereçamento*, *descoberta*, *controle*, *apresentação* e *evento*. No primeiro passo, é onde se realiza a descrição de um nó, em um documento *XML*. Nesse documento estão contidas as propriedades do nó (e.g., nome e descrição) bem como os serviços que o mesmo disponibiliza. O passo de endereçamento consiste em determinar o endereço de rede de um nó que esteja entrando na rede, utilizando o protocolo *DHCP* [5]. Após isso, o passo de descoberta irá notificar aos demais nós sobre os serviços que o recém chegado disponibiliza. Este pode também descobrir os outros nós bem como seus serviços, baseando-se no modo de disponibilização *pull* distribuído. Ao se descobrir um serviço, um nó pode então utilizá-lo, o que é feito pelo passo de controle. Por outro lado, ao se descobrir

[3] *World Wide Web Consortium*

[4] *http://www.upnp.org*

[5] *Dynamic Host Configuration Protocol*

um nó, pode-se carregar uma *URL* [6], através do passo de apresentação, a qual exibe uma interface para o mesmo. É através dessa URL, especificada na descrição do nó descoberto, que se pode, portanto, controlar o mesmo. Por fim, com o passo de evento, os nós podem receber notificações sobre mudanças ocorridas nos demais, como por exemplo, novos serviços que tenham sido publicados.

7.1.4 *Jini*

Jini (Waldo, 1999) é uma tecnologia de serviços baseada na linguagem Java que utiliza a abordagem *pull* centralizada. Dessa forma, a publicação de serviços é realizada em servidores centrais, chamados de *lookup servers*. A disponibilização de serviços em *Jini* envolve três protocolos: *discovery*, *join* e *lookup*. O protocolo *discovery* permite a um provedor de serviço descobrir os servidores centrais disponíveis. Uma vez descoberto tais servidores, o protocolo *join* se responsabiliza por publicar o serviço nos mesmos. Para isso, tal protocolo envia ao servidor uma interface Java contendo os métodos que os clientes devem invocar para utilizar o serviço. Por fim, o protocolo *lookup* permite que os clientes descubram os serviços publicados nos servidores centrais. Quando isso acontece, os clientes recebem, como *proxy* do serviço, uma cópia de sua interface Java. Na terminologa *Jini*, esses *proxies* são chamados de Objetos de Controle Remoto [7]. As operações de invocação de serviço e mobilidade de código são todas realizadas sobre o protocolo *RMI* [8].

O acesso aos serviços *Jini* é controlado pelo conceito de *lease*. Esse conceito pode ser visto como uma garantia que os clientes possuem para utilizar um serviço, sendo a mesma válida apenas por um determinado intervalo de tempo. Esse intervalo é determinado pelo provedor do serviço, no momento em que o mesmo é publicado. Assim, quando o *lease* expira, os clientes precisam então renová-lo, caso desejem continuar utilizando o serviço. Esse mecanismo impede, por exemplo, que os serviços de um nó que tenha saído da rede continuem válidos indefinidamente, pois quando o *lease* expira, os servidores centrais estão livres para descartar o serviço.

[6] *Universal Resource Locator*

[7] Do inglês, *Remote Control Objects*

[8] *Remote Method Invocation* (http://java.sun.com/products/jdk/rmi)

7.1.5 *Zeroconf*

Sendo desenvolvido pelo *IETF Zeroconf Working Group*, *Zeroconf* (Guttman, 2001) é uma solução para configuração automática de redes *IP*. Mais especificamente, *Zeroconf* permite que um dispositivo se configure automaticamente de forma a fazer parte de uma rede IP. Com esse propósito, o *IETF Zeroconf Working Group* define quatro requisitos envolvidos nesse processo de configuração automática [9]: *autoconfiguração de endereço*, *tradução de nomes para endereços*, *descoberta de serviços* e *alocação de endereço multicast*. O primeiro requisito está associado à configuração das interfaces de rede de um nó. Isso envolve, por exemplo, a determinação da máscara de rede a ser utilizada e a detecção de endereços duplicados. O requisito de tradução de nomes para endereços permite obter o endereço *IP* de um nó a partir de seu nome e vice-versa. A descoberta de serviços, por sua vez, tem o papel de encontrar serviços disponíveis na rede, sem no entanto requerer administração centralizada. Ou seja, a disponibilização de serviços é realizada utilizando-se a abordagem *pull* distribuída. Por fim, o último requisito, alocação de endereços *multicast*, está relacionado à atribuição de endereços *multicast* às aplicações.

7.1.6 *SLP*

O protocolo *SLP* (Guttman, 1999), de *Service Location Protocol*, é uma tentativa da *IETF* de prover uma solução para a descoberta automática de serviços. Para isso, foram definidos três tipos de agentes [10]: *agentes de usuário*, *agentes de serviços* e *agentes de diretório*. Os agentes de usuário são responsáveis por descobrirem serviços para as aplicações cliente. Já os agentes de serviços cuidam de publicar serviços. Por fim, os agentes de diretório têm como função armazenar informações de serviços que tenham sido publicados.

A descoberta de serviços em *SLP* se dá através das abordagens *pull* centralizada e distribuída, apresentadas na Seção 3.2.2. A primeira é utilizada quando um agente de diretório está presente na rede. Assim, os mesmos recebem os serviços a serem publicados e as requisições de descoberta, respectivamente, dos agentes de serviços e de usuários. Por outro lado, quando nenhum agente de diretório existe na rede, a abordagem distribuída é utilizada.

[9] No original, *address auto-configuration*, *name-to-address translation*, *service discovery* e *multicast address allocation*

[10] No original, *user agents*, *service agents* e *directory agents*

Portanto, nessas situações, as requisições de descoberta dos agentes de usuário são recebidas diretamente pelos agentes de serviço, que enviam respostas aos mesmos caso tenham o serviço requisitado.

Baseando-se nessas características, define-se também três modos de cenários de operação do *SLP*, os quais levam em consideração a quantidade de nós no ambiente. Basicamente, em ambientes com poucos nós, a abordagem *pull* distribuída é utilizada, excluindo-se, portanto, os agentes de diretório. Diferentemente, em ambientes de médio e grande porte, utiliza-se a abordagem *pull* centralizada. Mais precisamente, nos primeiros, apenas um agente de diretório é utilizado, enquanto que nos segundos, vários desses agentes podem existir na rede.

7.1.7 *SDIPP*

O *Protocolo de Descoberta, Interação e Pagamento de Serviços* [11] (Ravi, Stern, Desai, & Iftode, 2005), ou *SDIPP*, é como nome indica, um protocolo para disponibilização de serviços, com foco em ambientes de computação pervasiva. *SDIPP* é um protocolo dividido em duas camadas, como ilustrado na Figura 7.2. A camada de mais baixo nível tem como uma de suas funções armazenar, no módulo *Cache*, algumas informações pessoais do usuário, como seu nome, idade e endereço. Além disso, essa camada também é responsável pela descoberta de serviços, tanto em redes sem fio de curta distância, através do *Mecanismo Bluetooth* [12], quanto na Internet, através do *Mecanismo GPRS* [13]. Já a camada superior, encarrega-se de implementar os três passos básicos de disponibilização de serviços do protocolo *SDIPP*, *descoberta*, *interação* e *pagamento*, implementados respectivamente pelos módulos *Protocolo de Descoberta*, *Protocolo de Interação* e *Protocolo de Pagamento*. No passo de descoberta, os serviços de interesse são procurados, primeiramente nos nós vizinhos (i.e., um único *hop* de busca), utilizando *Bluetooth*. Caso nenhum serviço seja encontrado, a busca prosseguirá, agora em nós não accessíveis diretamente através do cliente (i.e., vários *hops* de busca). Isso pode ser realizado utilizando-se *UPnP*, *DEAPspace*, *SLP*, *Secure SDS* e agentes móveis. Se ainda assim o serviço de interesse não for encontrado, um *web service* irá então realizar a

[11] No original, *Service Discovery, Interaction, and Payment Protocol*
[12] No original, *Bluetooth Engine*
[13] No original, *GPRS Engine*

busca, através do *Mecanismo GPRS*. Na fase seguinte, de interação, os serviços descobertos são listados para que apenas um seja selecionado. Quando isso acontece, a interface do serviço é baixada para o dispositivo local, para que o cliente possa então utilizar o serviço. Por fim, na fase de pagamento, é realizada uma espécie de cobrança eletrônica pelo uso do serviço.

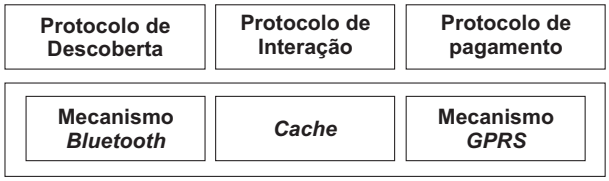


Figura 7.2: Arquitetura do protocolo *SDIPP*.

7.1.8 *PDP*

O protocolo *PDP* (Campo, Rubio, López, & Almenárez, 2006), ou *Pervasive Discovery Protocol*, é uma solução para a descoberta de serviços em redes *ad hoc* móveis. Um dos objetivos desse protocolo é minimizar o consumo de energia dos nós e as mensagens enviadas no processo de descoberta de serviços. *PDP* foi desenvolvido para ambientes descentralizados, podendo disponibilizar serviços através da abordagem *push* e da *pull* distribuída.

Na disponibilização de serviços do *PDP*, cada nó possui um *agente de usuário PDP* [14] e um *agente de serviços PDP* [15]. Agentes de usuário são responsáveis por descobrir serviços publicados pelos demais nós da rede. Os agentes de serviço, por outro lado, se encarregam de publicar os serviços de um nós aos demais. Quando um serviço é publicado, independentemente da abordagem utilizada, são necessárias duas informações: a descrição do serviço e seu tempo de validade. É através deste último que os registros de serviço podem controlar quais serviços são válidos e quais não são. Dessa forma, quando um serviço tem seu tempo de validade expirado, o mesmo é removido do registro. Isso ajuda, portanto, a garantir consistência nas informações referentes aos serviços atualmente publicados na rede.

[14] No original, *user agent PDP*.

[15] No original, *service agent PDP*.

7.1.9 *Salutation*

Definida pelo Consórcio *Salutation*, a arquitetura *Salutation* tem como propósito permitir a disponibilização de serviços independentemente do protocolo de transporte. Ilustrada na Figura 7.3, tal arquitetura é composta por três componentes [16]: *unidade funcional*, *gerenciador salutation* e *gerenciador de transporte*. A unidade funcional define um serviço em execução. Para cada tipo de serviço da arquitetura, deve existir uma especificação associada ao mesmo. Alguns dos serviços atualmente especificados são fax, impressão e armazenamento de documento. A especificação de um serviço define, dentre outras coisas, os atributos que o caracterizam. Um outro componente da arquitetura *Salutaiton*, o gerenciador *salutation*, implementa as funções de publicação e descoberta de serviços (i.e., unidades funcionais). Por fim, no nível mais baixo dessa arquitetura residem os gerenciadores de transporte, responsáveis por implementar o transporte de dados entre clientes e servidores. É possível, dessa forma, adicionar e remover gerenciadores de transporte sempre que necessário. Além disso, ambas as operações podem ser efetuadas sem modificações no gerenciador *salutation*. Tendo em vista o mercado de dispositivos móveis, definiu-se também uma arquitetura mais leve, chamada de *Salutation-Lite*.

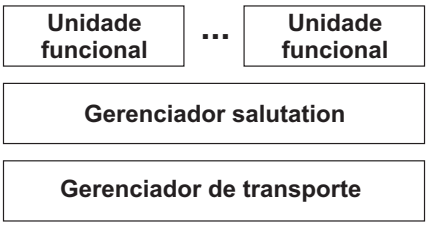


Figura 7.3: A arquitetura *Salutation*.

7.1.10 *Considerações Sobre as Soluções de Disponibilização de Serviços*

Como foi possível observar, existe atualmente um bom número de soluções voltadas à disponibilização de serviços, ainda mais se levarmos em consideração que nem todas foram aqui apresentadas. Nesse escopo, um ponto a ser observado é que nenhuma das soluções cobre todas as abordagens de disponibilização de serviços. Como foi possível perceber, apenas em algumas delas, como o protocolo *SDIPP*, mais de uma abordagem é suportada. Dessa forma,

[16] No original, *functional unit*, *salutation manager* e *transport manager*

ao adotarmos a estratégia de encapsular essas soluções em *plug-ins*, cobrimos portanto toda a gama de abordagens para a disponibilização de serviços, bastando, para isso, que os *plug-ins* corretos estejam instalados.

7.2 *Middlewares* para Computação Pervasiva

Nessa seção, iremos apresentar algum dos atuais *middlewares* para computação pervasiva. Mais especificamente, descreveremos aqui soluções que apresentem pelo menos uma das seguintes características, as quais são descritas abaixo: *extensibilidade*, *evolução dinâmica*, *publicação de serviços*, *descoberta de serviços*, *utilização de serviços*, *ciência de contexto*, *descoberta de nós* e *independência de protocolo*. Além disso, apresentaremos também, ao final desta seção, um resumo das características de cada *middleware*, além de uma comparação com as apresentadas pelo *Wings*.

- *Extensibilidade*: indica se o *middleware* pode ser estendido.
- *Evolução dinâmica*: indica se o *middleware* pode ser atualizado em tempo de execução.
- *Publicação de serviços*: indica se o *middleware* provê suporte à publicação de serviços.
- *Descoberta de serviços*: indica se o *middleware* suporta a descoberta de serviços.
- *Utilização de serviços*: indica se o *middleware* permite que serviços sejam utilizados, uma vez que foram descobertos.
- *Ciência de contexto*: indica se o *middleware* provê mecanismos para a ciência de contexto.
- *Descoberta de nós*: indica se o *middleware* permite a descoberta de nós.
- *Independência de protocolo*: indica se o *middleware* é independente de protocolo, no que se refere à disponibilização de serviços e descoberta de nós.

7.2.1 *Green*

Green (Sivaharan, Blair, & Coulson, 2005) é um *middleware* de computação pervasiva que provê mecanismos para a publicação, registro e notificação de eventos. Uma das principais características desse mecanismo é sua flexibilidade para suportar diferentes tipos de notificação de eventos, inclusive através de redes heterogêneas. Tendo em vista essa característica, os criadores desse *middleware* o basearam no modelo de componentes *OpenCOMM* (Clarke, Blair, Coulson, & Parlavantzas, 2001), o qual provê a flexibilidade necessária para que componentes sejam adicionados e removidos do mesmo em tempo de execução.

A arquitetura do *middleware Green* é constituída por duas camadas, como ilustrado na Figura 7.4, ambas implementadas como um arcabouço de componentes (i.e., *component frameworks* (Szypersky, 1997)). A camada de mais alto nível, *Arcabouço de Componentes para Publicação e Registro* [17], é responsável pelos diferentes tipos de interação envolvidos na publicação e registro de eventos. Com esse propósito, esses tipos de interação são encapsulados em *plug-ins*, de forma a serem adicionados à camada em tempo de execução. Já a camada de mais baixo nível, *Arcabouço de Componentes para Overlay de Eventos* [18], está associada à notificação de eventos através da rede, a qual está relacionada diretamente com os protocolos de comunicação. Portanto, os diferentes mecanismos de notificação de eventos são também encapsulados em *plug-ins*, permitindo ao *middleware Green* englobar uma ampla gama de protocolos e infra-estruturas de rede.

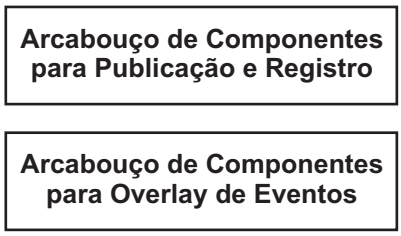


Figura 7.4: Arquitetura do *middleware Green*.

[17] No original, *Publish-subscribe Interaction Component Framework*
[18] No original, *Event Broker Overlay Component Framework*

7.2.2 *Plug-in ORB*

Como o nome indica, *Plug-in ORB* (d'Acierno, Pietro, Coronato, & Gugliara, 2005) é um *middleware* baseado em *plug-ins*, voltado à computação pervasiva. Mais precisamente, esse *middleware* permite a utilização de recursos remotos heterogêneos, utilizando para isso diferentes *middlewares*, os quais são encapsulados em *plug-ins*. Assim, aplicações poderiam, por exemplo, acessar objetos remotos através de *RMI*, *CORBA* [19], dentre outros *middlewares*, apenas instalando os *plug-ins* necessários.

O *middleware Plug-in ORB* está baseado em dois tipos de *plug-ins* [20]: *Plug-ins de Meta ORB* e *Plug-in de Middleware*. O primeiro abstrai para o programador detalhes da interação entre os *plug-ins* e as aplicações, provendo funcionalidades de *busca*, a qual retorna referências para objetos remotos, e *invocação remota*, para a invocação de métodos desses objetos. O *Plug-in* de *Middleware*, por sua vez, implementa essas funcionalidades básicas para um *middleware* específico. Dessa forma, teríamos, por exemplo, *Plug-ins* de *Middleware* para *RMI*, *CORBA*, *Jini*, dentre outros *middlewares*. Esses *plug-ins* são carregados e descarregados dinamicamente pelo *Plug-in de Carregamento de Middleware* [21], o qual é gerenciado pelo *Plug-in* de *Meta ORB*. Assim, quando uma das funcionalidades deste último é requisitada por uma aplicação, o *Plug-in* de Carregamento de *Middleware* irá carregar o *Plug-in* de *Middleware* correto para realizá-la, descarregando-o quando não mais necessário.

7.2.3 *RUNES*

Estando baseado em *plug-ins*, *RUNES* [22] (Costa, Coulson, Mascolo, Picco, & Zachariadis, 2005) é mais um *middleware* voltado à computação pervasiva com foco em reconfiguração. Tal característica é provida pelo modelo de componentes *RUNES*, no qual o *middleware* se baseia. A idéia básica por trás do *middleware RUNES* é a utilização de diferentes arcabouços de componentes, cada um com um propósito específico. Esses arcabouços definem, portanto, que tipos de extensões, isto é, *plug-ins*, podem ser integradas aos mesmos. Com esse propósito, são definidos diferentes arcabouços de componentes, dos quais destacamos [23]: *Ser-*

[19] http://www.omg.org/corba

[20] No original, *Meta ORB Plug-ins* e *Middleware Plug-ins*

[21] No original, *Biding Middleware Plug-in*

[22] *Reconfigurable, Ubiquitous, Networked Embedded Systems*

[23] No original, *Network Services*, *Location Services* e *Advertising and Discovery Services*

viços de Rede, *Serviços de Localização* e *Publicação e Descoberta de Serviços*. O primeiro desses arcabouços provê comunicação tanto para redes *ad-hoc* quanto para as baseadas em infra-estrutura fixa. Já o arcabouço de Serviços de Localização é responsável por disponibilizar informações sobre a localização do dispositivo no qual o *middleware* está executando. Isso pode ser realizado, por exemplo, via sensores *GPS* [24]. O último dos arcabouços é o de Publicação e Descoberta de Serviços, o qual permite a publicação e descoberta tanto de componentes (e.g., componentes de *codec*) quanto de serviços (e.g., serviços de impressão). A publicação é realizada pelos chamados *Componentes Publicadores* [25] enquanto que a descoberta é feita através de *Componentes de Descoberta* [26]. Portanto, adicionando diferentes componentes publicadores e de descoberta é possível descobrir componentes e serviços através de diferentes protocolos.

7.2.4 *TyniObj*

TyniObj (Poupyrev et al., 2005) é uma solução para a disponibilização de serviços em ambientes pervasivos, definindo, para esse propósito, quatro passos [27]: 1) *Inicialização de Dados*, 2) *Troca de Dados sem Fio*, 3) *Comparação de Serviços* e 4) *Alertas de Descoberta*. O primeiro passo consiste em inicializar os dados referentes aos provedores e clientes de serviço. Enquanto que os provedores irão inicializar as descrições dos serviços que os mesmos implementam, os clientes, por sua vez, irão inicializar os dados referentes aos serviços de que precisam. No próximo passo, o da Troca de Dados sem Fio, mensagens de difusão serão enviadas pela rede. Com esse propósito, os participantes, isto é, clientes e provedores de serviços, podem assumir um papel ativo ou passivo. No papel ativo, o participante envia as mensagens de difusão para os demais. No passivo, por outro lado, o participante espera pelas mensagens de difusão vindas da rede. Uma vez decidido se o participante atuará no modo ativo ou no passivo, entra em ação então o passo de Comparação de Serviços. Dessa forma, quando um provedor de serviços atua no modo ativo, este envia aos outros nós as descrições dos serviços que o mesmo deseja publicar. Essas descrições serão, portanto, recebidas pelos clientes que estejam no modo passivo. Por outro lado, quando o cliente atua no modo ativo, o

[24] *Global Positioning System*

[25] No original, *Advertiser Components*

[26] No original, *Discoverer Components*

[27] No original, *Data Initialization*, *Wireless Data Exchange*, *Service Matchmaking* e *Discovery Alerts*.

mesmo irá enviar, aos demais nós, mensagens informando sobre os serviços de sua preferência. Essas mensagens serão recebidas pelos provedores de serviços que estiverem no modo passivo, os quais irão verificar se possuem algum dos serviços requisitados. Finalmente, no quarto e último passo, o usuário é então notificado de que um serviço de seu interesse foi encontrado.

7.2.5 *Split Smart Messages*

Split Smart Messages (Ravi, Borcea, Kang, & Iftode, 2004) é um *middleware* para computação pervasiva voltado às redes *ad-hoc* baseando-se no conceito de *Mensagens Inteligentes* (*SMs* [28]). Estas mensagens são, na verdade, aplicações cujas execuções são distribuídas em diversos nós de uma rede, através da migração das mesmas. Essa migração é realizada através da descoberta de nós para os quais uma *SM* pode migrar, utilizando para isso a função *migrate*, provida pelo *middleware*.

Cada *SM* é dividida em três partes [29]: *código*, *dados* e *estados de execução de controle*. O primeiro são os arquivos binários *Java* da mensagem inteligente. Já os dados representam, como seu nome indica, os dados atualmente utilizados pela mensagem inteligente. Os mesmos são encapsulados em objetos *Java*, os quais são serializados através da rede sempre que a mensagem inteligente associada migra de um nó para outro. Finalmente, os estados de execução de controle fornecem informações sobre a execução de uma mensagem inteligente, como o ponto no qual a mesma foi interrompida e a próxima instrução a ser executada. Isso permite, portanto, que o *middleware* seja capaz de migrar mensagens inteligentes juntamente com seus dados, parando as mesmas na origem e reiniciando-as no destino.

Dentre as demais funções do *Split Smart Messages*, gostaríamos de destacar a migração e descoberta de serviços. A primeira permite a um serviço migrar de um nó para outro, levando consigo sua descrição. Assim, aplicações executando em outros dispositivos da rede podem descobrir serviços recentemente migrados, bem como os já presentes, através da função de descoberta de serviços. Para a realização de ambas as funções o *middleware Split Smart Messages* considera tanto redes sem fio de curto alcance, através de *Bluetooth*, quanto a Internet, através de *GPRS*.

[28] *Smart Messages*

[29] No original, *code bricks*, *data bricks* e *execution control state*

7.2.6 *PHolo*

Pervasive Holo (Bonatto, Barbosa, Cavalheiro, & Ramos, 2005), também chamada de *PHolo*, é uma arquitetura de software para computação pervasiva baseada no *Holoparadigma* (Barbosa, 2002). O *Holoparadigma* propõe um modelo para o desenvolvimento de aplicações distribuídas, sendo estas implementadas utilizando uma linguagem chamada de *Hololinguagem*. As aplicações escritas nessa linguagem executam sobre um máquina virtual, a *HoloVM*. Sendo assim, o propósito da arquitetura *PHolo* é acrescentar características de computação pervasiva ao ambiente no quais as aplicações do *Holoparadigma* executam.

A arquitetura *PHolo*, ilustrada na Figura 7.5, encontra-se estruturada em duas camadas. A camada de mais alto nível, *Execução*, consiste da *HoloVM* e do módulo *HNS*, este último voltado à execução distribuída das aplicações. Abaixo dessa camada localiza-se a camada de *Serviços*, composta por cinco módulos, responsáveis por prover características de computação pervasiva à arquitetura *PHolo*. O primeiro deles, o serviço de *Localização*, fornece informações de localização para a *HoloVM*. Assim, utilizando esse serviço em conjunto com o de *Contexto*, as aplicações podem ter acesso as informações de contexto que precisam, de acordo com suas respectivas localizações. Um outro serviço da arquitetura é o de *Mobilidade*, o qual fornece às aplicações diferentes graus de mobilidade, permitindo as mesmas movimentar-se, por exemplo, entre diferentes dispositivos. Já o serviço *Bonjour* provê mecanismos para a configuração automática de rede, utilizando, para isso, o protocolo *Zeroconf*. Além disso, esse serviço também provê a descoberta automática de nós na rede. Por fim, através do serviço de *Descoberta*, é possível ainda realizar a publicação, descoberta e utilização de serviços, o que também é feito sobre *Zeroconf*.

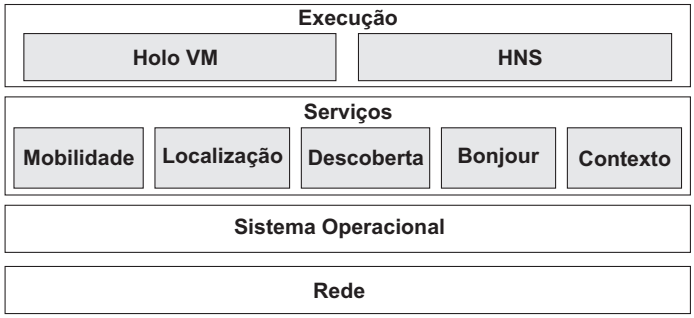


Figura 7.5: A arquitetura *PHolo*.

7.2.7 *Nexus*

Aparecendo como mais uma solução concentrada em mesclar a abordagem de serviços à computação pervasiva, o *middleware Nexus* (Kaveh & Hercock, 2004) se utiliza da abordagem de *plug-ins* para contornar a heterogeneidade de protocolos de serviços existentes. Sua arquitetura, ilustrada na Figura 7.6, é composta por três camadas. A primeira delas, *Camada de Descoberta* [30], é responsável pela publicação e descoberta de serviços, independente de protocolo. Dessa forma, cada protocolo a ser utilizado nessa camada é encapsulado em um *plug-in*, podendo ser adicionado ou mesmo removido do *middleware* em tempo de execução. A próxima camada, *Camada de agente*, tem como propósito adicionar funcionalidades de busca de serviços mais inteligentes ao *middleware*. Mais precisamente, a idéia é adicionar agentes a essa camada, para que estes realizem tarefas específicas em favor do usuário. A *Camada de Agente*, no entanto, é opcional. Por fim, na camada de mais alto nível do *middleware Nexus*, *Camada de Serviços*, residem os serviços a serem publicados na rede. Dessa forma, serviços de diferentes tecnologias podem executar nessa camada, podendo assim ser publicados através de diferentes protocolos, possivelmente ao mesmo tempo.

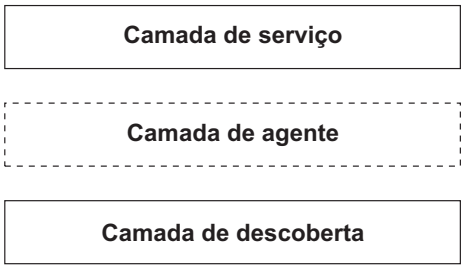


Figura 7.6: Arquitetura do *middleware Nexus*.

7.2.8 *ReMMoC*

O *middleware ReMMoC* (Grace, Blair, & Samuel, 2005) utiliza-se do modelo de componentes *OpenCOM* e de diferentes arcabouços de componentes de forma a poder ser reconfigurado em tempo de execução. Sua arquitetura possui três camadas [31], *Concreto*, *Abstrato para Concreto* e *Abstrato*, como ilustrado na Figura 7.7. A primeira camada é composta por

[30] No original, *Discovery Layer*.

[31] No original, *Concrete*, *Abstract to Concrete* e *Abstract*.

dois arcabouços de componentes, um para os diferentes tipos de ligação entre serviços (e.g., *RPC*, *publish-subscribe*), chamado de *Arcabouço de Ligação* [32], e outro para a descoberta de serviços, chamado de *Arcabouço de Descoberta de Serviços* [33]. Neste último residem diferentes *plug-ins* para a descoberta de serviços, os quais são baseados no modelo *OpenCOM* e permitem, portanto, não só a reconfiguração dinâmica do arcabouço como também a descoberta de serviços através de diferentes protocolos. A camada Abstrato, por sua vez, implementa uma *API* para realizar a descoberta de serviços e ligações entre os mesmos, independentemente do protocolo. Por fim, a camada Abstrato para Concreto é responsável por mapear as requisições abstratas da camada superior (Abstrato) para as implementações da camada inferior (Concreto).

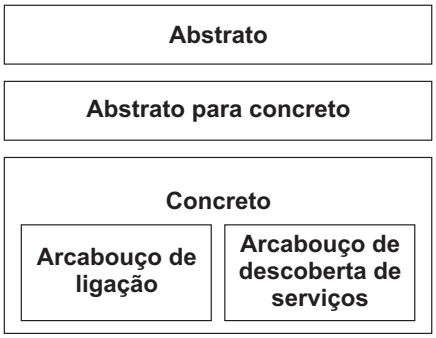


Figura 7.7: Arquitetura do *middleware ReMMoC*.

Um característica interessante do *middleware ReMMoC* é o fato de que o mesmo pode ser configurado para utilizar apenas um dos arcabouços de componente da camada Concreto. Isso permite, portanto, diminuir significativamente a utilização de memória do *middleware*, o que vem a ser útil para dispositivos móveis possuindo uma maior limitação de recursos. Além disso, é possível também plugar novos arcabouços de componentes ao *middleware* para lidar com requisitos não funcionais, como segurança e desempenho.

7.2.9 *JCAF*

Baseado em *Java*, *JCAF* (Bardram, 2005), acrônimo para *Java Context Awareness Framework*, é um *middleware* para computação pervasiva baseado em eventos e serviços, cons-

[32] No original, *Binding Framework*.

[33] No original, *Service Discovery Framework*.

tituído de um *Arcabouço de Programação para Ciência de Contexto* [34] e uma *Infra-estrutura de Ciência de Contexto em Tempo de Execução* [35]. O arcabouço de programação provê um conjunto de classes e interfaces *Java* para o desenvolvimento de aplicações pervasivas e *serviços de contexto* [36], estes últimos constituindo a Infra-estrutura de Ciência de Contexto em Tempo de Execução. Mais precisamente, cada serviço de contexto deve lidar com as informações de um ambiente em particular, por exemplo, uma sala ou um quarto, estando os mesmos dispostos em uma topologia ponto a ponto, para que possam assim se comunicar e trocar informações de contexto entre si. As informações providas por cada serviço de contexto podem ser recuperadas através de requisições e respostas (i.e., *request-response*) ou registrando-se como ouvintes de mudanças no contexto.

A modelagem das informações de um serviço de contexto envolve três elementos [37]: *contexto*, *entidade*, e *item de contexto*. Enquanto que o contexto representa o ambiente sendo modelado, a entidade refere-se aos objetos existentes no mesmo, como cadeira, mesa, cama, dentre outros. Por fim, os items de contexto representam as informações que podem ser obtidas das diversas entidades de um ambiente (e.g., localização de uma entidade). Uma aspecto interessante a ser observado sobre a modelagem de contexto no *JCAF* é que novas entidades e items de contexto podem ser adicionados a um serviço de contexto sempre que necessário, mesmo em tempo execução.

7.2.10 *SOCAM*

Concebido pelo Departamento de Computação da Universidade de Cingapura, o *middleware SOCAM* (Gu, Pung, & Zhang, 2005) (*Service Oriented Context Aware Middleware*) propõe uma solução baseada em ontologias e serviços para computação pervasiva. Ontologias são utilizadas na modelagem do contexto, mais precisamente através de *OWL* [38]. Uma das vantagens da utilização de *OWL* é o fato desta facilitar o compartilhamento de informações de contexto, por ser baseada em *XML*, e a inferência sobre as mesmas (i.e., raciocínio de contexto), dado que já existem *APIs* com esse propósito.

[34] No original, *Context Awareness Programming Framework*.
[35] No original, *Context Awareness Runtime Infrastructure*.
[36] No original, *context services*
[37] No original, *entity*, *context* e *context item*.
[38] *Ontology Web Language*

Dado isso, a arquitetura do *middleware SOCAM*, ilustrada na Figura 7.7, define os seguintes componentes: *Provedores de Contexto*, *Serviço de Localização de Serviços*, *Interpretador de Contexto*, *Banco de Dados de Contexto* e *Serviços Cientes do Contexto*. Provedores de Contexto são responsáveis por adquirir informações de contexto, dividindo-se em *Provedores de Contexto Externo* e *Provedores de Contexto Interno*. Enquanto que os primeiros obtém informações a partir de fontes externas, como servidores remotos, os Provedores Internos as recuperam de através de sensores. Os Provedores de Contexto são disponibilizados no ambiente como serviços, para que assim nós remotos possam recuperar as informações que os mesmos provém. Devido a isso, cada Provedor de Contexto precisa ser publicado em um registro de serviços local, o que é realizado através do Serviço de Localização de Serviço. É através dele também que nós remotos podem descobrir os Provedores de Contexto disponíveis no ambiente. As informações de contexto adquiridas a partir dos provedores são então convertidas para *OWL* e repassadas para o Interpretador de Contexto, para serem interpretadas. Além disso, as informações previamente utilizadas são mantidas no *Banco de Dados de Contexto*, podendo, portanto ser recuperadas e utilizadas posteriormente. Todos esses componentes servem de base para a execução dos chamados Serviços Cientes de Contexto, ou seja, aplicações, agentes ou serviços que fazem uso de informações de contexto e reagem às mudanças no mesmo.

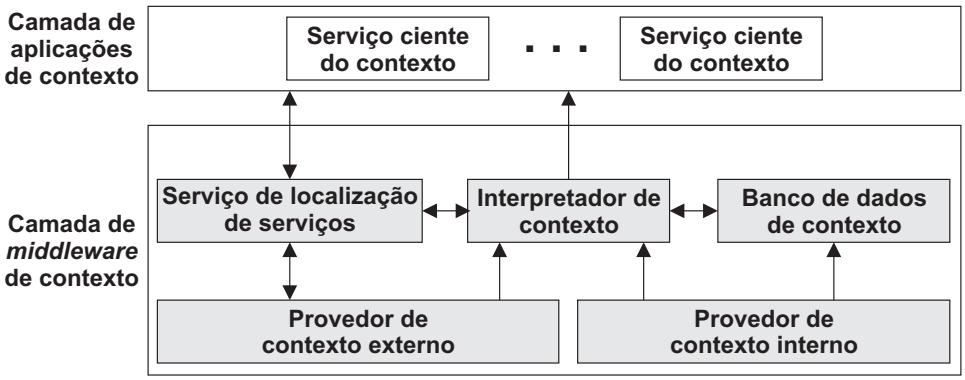


Figura 7.8: Arquitetura do *middleware SOCAM*.

7.2.11 Considerações Sobre as Soluções de Computação Pervasiva

Como foi possível perceber, as características que estamos considerando no *middleware Wings* já têm sido implementadas em várias outros *middlewares*. É fato, portanto, que a

inserção de tais características no *Wings* não representa nenhuma inovação. No entanto, é importante frisar que, como se pode perceber através da Tabela 7.1, nenhuma das atuais soluções contempla todas as características consideradas neste trabalho. Portanto, isso abre espaço para a definição e implementação de uma solução mais completa.

Middleware	Extensibilidade	Evolução dinâmica	Publicação de serviços	Descoberta de serviços	Utilização de serviços	Independência de protocolo	Ciência de Contexto	Descoberta de nós
Green	✓	✓						
Plug-in ORB	✓	✓		✓	✓	✓		
RUNES	✓	✓	✓	✓	✓	✓		
TyniObj			✓	✓	✓			
Split Smart Messages				✓				
PHolo			✓	✓	✓		✓	✓
Nexus	✓	✓	✓	✓	✓	✓		
ReMMoC	✓	✓		✓	✓	✓		
JCAF	✓	✓					✓	
SOCAM							✓	
Wings	✓	✓	✓	✓	✓	✓	✓	✓

Tabela 7.1: Comparativo das diferentes soluções de computação pervasiva.

Capítulo 8

Considerações Finais

"Isso não é o fim. Isto não é sequer o começo do fim. Isto é, talvez,
o fim do começo." (Winston Churchill)

Neste trabalho, foi apresentado um *middleware* para a disponibilização de serviços em ambientes pervasivos, denominado *Wings*. Como foi detalhado, na concepção do *middleware* os principais aspectos abordados relacionam-se a heterogeneidade de protocolos para a disponibilização de serviços, ciência de contexto, descoberta de nós, extensibilidade e finalmente a evolução dinâmica do *middleware*.

Como foi apresentado, a abordagem adotada para resolver tais problemas foi basear o *middleware* em uma arquitetura de *plug-ins*, implementando a mesma sobre uma solução de evolução dinâmica, o Modelo de Componentes *Compor*. Com isso, foi possível encapsular as funcionalidades básicas do *Wings* em *plug-ins*, permitindo ainda adicioná-los e removê-los do *middleware* em tempo de execução. Dessa forma, foi possível desenvolver um *middleware* com extensibilidade suficiente para a execução em cenários de computação pervasiva. Além disso, a possibilidade de inserir diferentes *plug-ins* para a disponibilização de serviços e descoberta de nós possibilita que o *Wings* possa, paralelamente, descobrir serviços e nós disponíveis através de protocolos variados.

Além disso, com base nos resultados de análise apresentados no Capítulo 5, pode-se concluir que o custo de processamento não foi seriamente comprometido. Ou seja, do ponto de vista de sobrecarga de processamento o impacto *Wings* é negligenciável. Nesse contexto, é fato que a utilização de memória foi decididamente elevada. Isso, no entanto, não chega a

ser preocupante. Primeiro porque, como os *plug-ins* podem ser removidos sempre que possível, esse consumo tende a ser reduzido bastante em algumas situações. Segundo porque, mesmo que isso não seja possível em alguns casos, com a crescente capacidade de memória dos dispositivos móveis, o consumo do *Wings* certamente deixará de ser um problema. Finalmente, é importante enfatizar que não foi executada nenhuma otimização na implementação do *Wings*. Entretanto, essa otimização, quando realizada, deverá resultar em ganhos consideráveis de desempenho, notadamente com respeito ao uso de memória.

8.1 Trabalhos em Andamento

Como resultado da implementação do *middleware*, pode-se destacar três trabalhos em andamento. O primeiro deles consiste em implementar o *Wings* na linguagem *C++* para o sistema operacional *Symbian* [1]. Para essa versão do *Wings*, o Modelo de Componentes *Compor* foi substituído pelo arcabouço *ECom* [2], o qual permite o carregamento e descarregamento dinâmico de *plug-ins*, sendo parte integrante da *API* do *Symbian* desde a versão 7.0. Atualmente, os módulos de Redes Pervasivas e Ciência do Contexto já se encontram implementados, e com isso, os esforços estão agora concentrados na implementação do Módulo de Fachada e no desenvolvimento de *plug-ins* de disponibilização de serviços (PDSs). Nesse contexto, dois *plug-ins* estão em desenvolvimento, sobre *Bluetooth* e *UPnP*, este último utilizando a *API Sphinx* [3].

Outro trabalho em desenvolvimento com base no *Wings* é a definição e implementação de uma infra-estrutura para raciocínio de contexto para aplicações pervasivas (Bublitz, 2006), estendendo assim o módulo de Ciência de Contexto do *middleware*. Com esse propósito, a mesma fará uso de ontologias para a modelagem de contexto e de *plug-ins* para encapsular os diferentes mecanismos de raciocínio de contexto. Essa infra-estrutura está sendo desenvolvida como um trabalho de mestrado no Curso de Pós-graduação em Informática da Universidade Federal de Campina Grande.

Por fim, o último dos trabalhos envolvendo o *middleware Wings* consiste no desenvolvimento de uma solução para *handoff* horizontal de serviços em ambientes pervasivos (Oli-

[1] http://wiki.percomp.org/index.php/Wings

[2] http://www.newlc.com/ECOMPLUS.html

[3] http://www.protosys.com

veira, 2006). O objetivo central desse trabalho é fornecer ao usuário continuidade na execução de um serviço em ambientes de computação pervasiva quando ocorre a mudança de ponto de acesso para uma determinada tecnologia sem fio. Assim como o trabalho anterior, este também está sendo conduzido no âmbito de uma dissertação de mestrado no Curso de Pós-graduação em Informática da Universidade Federal de Campina Grande.

8.2 Trabalhos Futuros

No que se refere aos possíveis trabalhos futuros, um dos que se incluem nessa categoria é o desenvolvimento de um ambiente de execução para a *middleware Wings* e suas aplicações. Mais precisamente, a idéia é definir e implementar um ambiente que gerencie a execução e instalação de aplicações e *plug-ins* no dispositivo. Tal ambiente de execução implementaria também um mecanismo para detectar os protocolos de disponibilização de serviços e descoberta de nós disponíveis em um ambiente, para assim baixar e instalar os *plug-ins* necessários automaticamente.

Nesse contexto, vale a pena comentar também a possibilidade de se conceber um repositório de *plug-ins*, acessível através da *Web*. Um dos propósitos de tal repositório seria, por exemplo, permitir ao ambiente de execução procurar por implementações específicas de *plug-ins* do *Wings*, para assim baixar e instalar as mesmas sempre que fosse necessário.

Um outro possível trabalho é o desenvolvimento de um *plug-in* para a plataforma *Eclipse* [4] para dar suporte ao desenvolvimento de aplicações voltadas ao *Wings*. Mais precisamente, esse *plug-in* deveria fornecer suporte a nível de implementação, testes (e.g., através de emuladores) e *deployment* de aplicações em dispositivos móveis.

Outros trabalhos que podem também ser incluídos nessa categoria são, por exemplo, a implementação de um mecanismo para a composição dinâmica de serviços em ambientes de computação pervasiva, modelagem formal do *middleware*, desenvolvimento de serviços multimídia sobre o *Wings* e ainda a implementação de um mecanismo de segurança e privacidade, para regular o acesso aos serviços de um dispositivo e às informações pessoais dos usuários.

[4] http://www.eclipse.org

Bibliografia

Almeida, H., Loureiro, E., Ferreira, G., Paes, R., Perkusich, A., & Costa, E. (2003, Outubro). Ambiente Integrado para o Desenvolvimento de Sistemas Multiagentes. In *10ª Sessão de Ferramentas do 17º Simpósio Brasileiro de Engenharia de Software* (pp. 55–60). Manaus, Brasil.

Almeida, H., Perkusich, A., Costa, E., Ferreira, G., Loureiro, E., Oliveira, L., et al. (2006). A Component Based Model to Develop Software Supporting Dynamic Unanticipated Evolution. In *21º Simpósio Brasileiro de Engenharia de Software.* (Aceito para publicação)

Barbosa, J. L. V. (2002). *Holoparadigma: um Modelo Multiparadigma Orientado ao Desenvolvimento de Software Distribuído*. Unpublished doctoral dissertation, Universidade Federal do Rio Grande do Sul, Rio Grande do Sul, Brasil.

Bardram, J. E. (2005, Maio). The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications. In H. Gellersen, R. Want, & A. Schmidt (Eds.), *Proceedings of the 3rd International Conference on Pervasive Computing* (Vol. 3468, pp. 98–115). Munique, Alemanha: Springer Verlag.

Bellur, U., & Narendra, N. C. (2005, Abril). Towards Service Orientation in Pervasive Computing Systems. In *International Conference on Information Technology: Coding and Computing - Volume II* (pp. 289–295). Las Vegas, NV, EUA.

Birsan, D. (2005, Março). On Plug-ins and Extensible Architectures. *ACM Queue*, *3*(2), 40–46.

Bonatto, D., Barbosa, J., Cavalheiro, G., & Ramos, J. D. (2005, Outubro). PHolo: Uma Arquitetura para a Computação Pervasiva Utilizando o Holoparadigma. In *VI Workshop em Sistemas de Alto Desempenho.* Rio de Janeiro, Brasil.

Boyera, S., & Lewis, R. (2005, Maio). *Device Independence Activity.* URL: http://www.w3.org/2001/di. (Acessado em 12/11/2005)

Bray, J., & Sturman, C. F. (2000). *Bluetooth: Connect Without Cables* (1 ed.). Prentice Hall.

Bublitz, F. (2006, Abril). *Arquitetura para o Desenvolvimento de Aplicações em Ambientes Pervasivos com Suporte a Ciência de Contexto.* (Proposta de Mestrado, Universidade Federal de Campina Grande, Campina Grande, Brasil)

Campo, C., Rubio, C. G., López, A. M., & Almenárez, F. (2006). PDP: A Lightweight Discovery Protocol for Local-Scope Interactions in Wireless Ad-hoc Networks. *Computer Networks*. (Aceito para publicação)

Chen, & Kotz, D. (2000). *A Survey of Context-Aware Mobile Computing Research* (Tech. Rep.). Hanover, NH, EUA: Dartmouth College.

Chen, H., Finin, T., & Joshi, A. (2003, Maio). An Ontology for Context-aware Pervasive Computing Environments. *Knowledge Engineering Review*, *18*(3), 197–207.

Chlamtac, I., Conti, M., & Liu, J. J.-N. (2003, Julho). Mobile Ad hoc Networking: Imperatives and Challenges. *Ad Hoc Networks*, *1*(1), 13-64.

Cirne, W., & Neto, E. S. (2005, Maio). *Grids Computacionais: Da Computação de Alto Desempenho a Serviços sob Demanda.* Mini-curso no 23º Simpósio Brasileiro de Redes de Computadores.

Clarke, M., Blair, G. S., Coulson, G., & Parlavantzas, N. (2001, Novembro). An Efficient Component Model for the Construction of Adaptive Middlewares. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms* (Vol. 2218, pp. 160–178). Heidelberg, Alemanha: Springer-Verlag.

Costa, P., Coulson, G., Mascolo, C., Picco, G. P., & Zachariadis, S. (2005, Setembro). The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In *Proceedings of the 16th IEEE International Symposium on Personal Indoor and Mobile Radio Communications.* Berlin, Alemanha: IEEE Communications Society.

d'Acierno, A., Pietro, G. D., Coronato, A., & Gugliara, G. (2005, Junho). Plugin-Orb for Applications in a Pervasive Computing Environment. In *Proceedings of the 2005 International Conference on Pervasive Systems and Computing* (pp. 140–146). Las Vegas, NV, EUA: CSREA Press.

Dey, A. K. (2001, Fevereiro). Understanding and Using Context. *Personal and Ubiquitous Computing*, *5*(1), 4–7.

Ebraert, P., Vandewoude, Y., Cazzola, W., D'Hondt, T., & Berbers, Y. (2005, Julho). Pitfalls in Unanticipated Dynamic Software Evolution. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution.* Glasgow, Escócia.

Esler, M., Hightower, J., Anderson, T., & Borriello, G. (1999, Agosto). Next Century Challenges: Data-centric Networking for Invisible Computing - The Portolano Project at the University of Washington. In *Proceedings of the 5th International Conference on Mobile Computing and Networking* (pp. 24–35). Seattle, WA, EUA: ACM Press.

Ferreira, G., Loureiro, E., Nogueira, W., Gomes, A., Almeida, H., & Frery, A. (2004, Julho). Uma Abordagem Baseada em Componentes para a Construção de Edifícios Virtuais. In *Proceedings of the 7th Symposium on Virtual Reality* (pp. 279–290). São Paulo, Brasil.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns* (1 ed.). Boston, MA, EUA: Addison-Wesley Professional.

Garlan, D., Siewiorek, D., Smailagic, A., & Steenkiste, P. (2002, Abril). Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, *1*(2), 22-31.

Gong, L. (2001, Maio). JXTA: A Network Programming Environment. *IEEE Internet Computing*, *5*(3), 88–95.

Grace, P., Blair, G. S., & Samuel, S. (2005, Janeiro). A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments. *Mobile Computing and Communications Review*, *9*(1), 2–14.

Gu, T., Pung, H. K., & Zhang, D. Q. (2004, Abril). A Bayesian Approach for Dealing with Uncertain Contexts. In G. Kotsis (Ed.), *Proceedings of the 2nd International Conference on Pervasive Computing.* Viena, Austria: Austrian Computer Society.

Gu, T., Pung, H. K., & Zhang, D. Q. (2005, Janeiro). A Service-oriented Middleware for Building Context-aware Services. *Journal of Network and Computer Applications*, *28*(1), 1–18.

Guttman, E. (1999, Julho). Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, *3*(4), 71–80.

Guttman, E. (2001, Maio). Autoconfiguration for IP Networking: Enabling Local Communication. *IEEE Internet Computing*, *5*(3), 81–86.

Gwizdka, J. (2000, Abril). What's in the Context? In *Workshop on the What, Who, Where, When, Why, and How of Context-Awareness.* Hague: Holanda.

Henricksen, K., Indulska, J., & Rakotonirainy, A. (2002, Agosto). Modeling Context Information in Pervasive Computing Systems. In *Proceedings of the First International Conference on Pervasive Computing* (pp. 167–180). Zurique, Suíça: Springer-Verlag.

Henricksen, K., Livingstone, S., & Indulska, J. (2004, Setembro). Towards a Hybrid Approach to Context Modeling, Reasoning, and Interoperation. In *Proceedings of the First International Workshop on Advanced Context Modeling, Reasoning, and Management* (pp. 54–61). Nottingham, Inglaterra: ACM Press.

Huhns, M. N., & Singh, M. P. (2005, Janeiro). Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, *9*(1), 75–81.

Kaveh, N., & Hercock, R. G. (2004, Julho). NEXUS: Resilient Intelligent Middleware. *BT Technology Journal*, *22*(3), 209–215.

Kephart, J. O., & Chess, D. M. (2003, Janeiro). The Vision of Autonomic Computing. *Computer*, *36*(1), 41–50.

Lehman, M., & Ramil, J. F. (2006, Junho). Software Evolution. In N. H. Madhavji, J. Fernandez-Ramil, & D. Perry (Eds.), *Software Evolution and Feedback: Theory and Practice* (1 ed., pp. 7–40). John Willey & Sons.

Loureiro, E., Bublitz, F., Barbosa, N., Perkusich, A., Almeida, H., & Ferreira, G. (2006, Junho). A Flexible Middleware for Service Provision Over Heterogeneous Pervasive Networks. In *4th International Workshop on Mobile and Distributed Computing.* Niagara Falls, NY, EUA: IEEE Computer Society. (Aceito para publicação)

Loureiro, E., Bublitz, F., Oliveira, L., Barbosa, N., Perkusich, A., Almeida, H., et al. (2006). Service Provision for Pervasive Computing Environments. In D. Taniar (Ed.), *Encyclopedia of Mobile Computing and Commerce* (Vol. 1). Hershey, PA, EUA: Idea Group Inc. (Aceito para publicação)

Loureiro, E., Ferreira, G., Almeida, H., & Perkusich, A. (2006). Pervasive Computing: What is it Anyway? In M. Lytras & A. Naeve (Eds.), *Ubiquitous and Pervasive Knowledge and Learning Management: Semantics, Social Networking and New Media to Their*

Full Potential. Hershey, PA, EUA: Idea Group. (Aceito para publicação)

Loureiro, E., Oliveira, L., Almeida, H., Ferreira, G., & Perkusich, A. (2005, Novembro). Improving Flexibility on Host Discovery for Pervasive Computing Middlewares. In *Proceedings of the 3rd International Workshop on Middleware for Pervasive and Ad hoc Computing*. Grenoble, França: ACM Press.

Ma, L. (2005, Março). *Develop P2P Applications With Device Discovery Technologies.* URL: http://www-128.ibm.com/developerworks/java/library/wi-peerapp. (Acessado em 17/02/2006)

Ma, T., Kim, Y. D., Ma, Q., Tang, M., & Zhou, W. (2005, Agosto). Context-Aware Implementation Based on CBR for Smart Home. In *Proceedings of IEEE International Conference on Wireless and Mobile Computing, Networking and Communications* (pp. 112–115). Montreal, Canadá: IEEE Computer Society.

Mascolo, C., Capra, L., & Emmerich, W. (2002, Maio). Middleware for Mobile Computing (A Survey). In E. Gregori, G. Anastasi, & S. Basagni (Eds.), *Advanced Lectures on Networking* (Vol. 2497, pp. 20–58). Springer-Verlag.

Masuoka, R., Labrou, Y., Parsia, B., & Sirin, E. (2003, Setembro). Ontology-Enabled Pervasive Computing Applications. *IEEE Intelligent Systems*, *18*(5), 68–72.

McGovern, J., Tyagi, S., Stevens, M., & Mathew, S. (2003, Abril). Service Oriented Architecture. In *Java Web Services Architecture* (1 ed., p. 35-63). Morgan Kaufmann.

Mostéfaoui, G. K., Rocha, J. P., & Brézillon, P. (2004, Julho). Context-Aware Computing: A Guide for the Pervasive Computing Community. In F. Mattern & M. Naghshineh (Eds.), *Proceedings of the 2004 IEEE/ACS International Conference on Pervasive Services* (pp. 39–48). Beirute, Líbano: IEEE Computer Society.

Nickull, D. (2005, Fevereiro). *Service Oriented Architecture* (White Paper). San Jose, CA, EUA: Adobe Systems Incorporated.
URL: http://www.adobe.com/enterprise/pdfs/Services_Oriented_Architecture_from_Adobe.pdf.

Nishigaki, K., Yasumoto, K., Shibata, N., Ito, M., & Higashino, T. (2005, Junho). Framework and Rule-based Language for Facilitating Context-Aware Computing Using Information Appliances. In *Proceedings of the First International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet* (pp. 345–351). Colum-

bia, OH, EUA: IEEE Computer Society.

Nogueira, W., Loureiro, E., & Almeida, H. (2005, Maio). Plataformas para Desenvolvimento de Jogos para Celulares. *INFOCOMP Journal of Computer Science*, *4*(1), 53–61.

Oliveira, L. (2006, Abril). *Uma Infra-estrutura para Diponibilização Dinâmica de Serviços em Ambientes Pervasivos.* (Proposta de Mestrado, Universidade Federal de Campina Grande, Campina Grande, Brasil)

Oliveira, L., Loureiro, E., Almeida, H., & Perkusich, A. (2006). Issues on Bridging Mobile and Service Oriented Computing. In D. Taniar (Ed.), *Encyclopedia of Mobile Computing and Commerce* (Vol. 2). Hershey, PA, EUA: Idea Group Inc. (Aceito para publicação)

Papazoglou, M. P. (2003a, Dezembro). Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering* (pp. 3–12). Roma, Itália: IEEE Computer Society.

Papazoglou, M. P. (2003b, Março). Web Services and Business Transactions. *World Wide Web*, *6*(1), 49–91.

Papazoglou, M. P., & Georgakopoulos, D. (2003, Abril). Service Oriented Computing. *Communications of the ACM*, *46*(10), 24–28.

Perkins, C. E. (1997, Maio). Mobile IP. *IEEE Communications*, *37*(5), 66–81.

Petrelli, D., Not, E., Strapparava, C., Stock, O., & Zancanaro, M. (2000, Abril). Modeling Context is Like Taking Pictures. In *Workshop on the What, Who, Where, When, Why, and How of Context-Awareness.* Hague: Holanda.

Poupyrev, P., Sasao, T., Saruwatari, S., Morikawa, H., Aoyama, T., & Davis, P. (2005, Maio). Service Discovery in TinyObj: Strategies and Approaches. In *Proceedings of the Workshop on Pervasive Mobile Interaction Devices* (pp. 19–22). Munique, Alemanha: Springer Verlag.

Ranganathan, A., Muhtadi, J., & Campbell, R. (2004, Setembro). Reasoning about Uncertain Contexts in Pervasive Computing Environments. *IEEE Pervasive Computing*, *3*(2), 62–70.

Ravi, N., Borcea, C., Kang, P., & Iftode, L. (2004, Agosto). Portable Smart Messages for Ubiquitous Java-Enabled Devices. In *First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services* (pp. 412–421). Cambridge,

MA, EUA: IEEE Computer Society.

Ravi, N., Stern, P., Desai, N., & Iftode, L. (2005, Março). Accessing Ubiquitous Services Using Smart Phones. In *Proceedings of 3rd IEEE International Conference on Pervasive Computing and Communications* (pp. 383–393). Ilhas Kauai, Havaí: IEEE Computer Society.

Reid, N. P., & Seide, R. (2002). *Wi-Fi (802.11) Network Handbook* (1 ed.). McGraw-Hill.

Ryan, N. (1999, Agosto 6). *ConteXtML: Exchanging Contextual Information between a Mobile Client and the FieldNote Server.*
URL: http://www.cs.kent.ac.uk/projects/mobicomp/fnc/ConteXtML.html. (Acessado em 12/11/2005)

Sales, L., Pontes, F., Costa, E., Luna, H. P., Loureiro, E., & Sales, M. (2006, Janeiro). Set your Multimedia Application Free With Arcolive: An Open Source Component-based Toolkit to Support E-learning Environments. In *Proceedings of The Fifth IASTED International Conference on Web-based Education.* Puerto Vallarta, Mexico: Acta Press.

Satyanarayanan, M. (1996, Maio). Fundamental Challenges in Mobile Computing. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (pp. 1–7). Filadélfia, PA, EUA: ACM Press.

Satyanarayanan, M. (2001, Agosto). Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, *8*(4), 10-17.

Schilit, B., & Theimer, M. (1994, Setembro). Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, *8*(5), 22–32.

Sivaharan, T., Blair, G., & Coulson, G. (2005, Outubro). GREEN: A Configurable and Re-Configurable Publish-Subscribe Middleware for Pervasive Computing. In *Proceedings of the International Symposium on Distributed Objects and Applications* (Vol. 3760, p. 732-749). Agia Napa, Chipre: Springer Verlag.

Stanford, V., Garofolo, J., Galibert, O., Michel, M., & Laprun, C. (2003, Abril). The NIST Smart Space and Meeting Room Projects: Signals, Acquisition, Annotation and Metrics. In *Proceedings of the 2003 IEEE Conference on Acoustics, Speech, and Signal Processing.* Hong Kong: IEEE Computer Society.

Sun, J. Z., & Sauvola, J. (2002, Agosto). Mobility and Mobility Management: A Conceptual

Framework. In *Proceedings of the 10th IEEE International Conference on Networks* (pp. 205–210). Cingapura: IEEE Computer Society.

Szypersky, C. (1997). *Component Software: Beyond Object-Oriented Programming*. Boston, MA, EUA: Addison-Wesley Professional.

Tiengo, W., Costa, E., Tenório, L. E., & Loureiro, E. (2006, Janeiro). An Architecture for Personalization and Recommendation System for Virtual Learning Community Libraries. In *Proceedings of The Fifth IASTED International Conference on Web-based Education.* Puerto Vallarta, Mexico: Acta Press.

Vogels, W. (2003, Novembro). Web Services Are Not Distributed Objects. *IEEE Internet Computing*, *7*(6), 59–66.

Waldo, J. (1999, Julho). The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, *42*(7), 76–82.

Weiser, M. (1991, Setembro). The computer for the 21st century. *Scientific American*, *265*(3), 94–104.

Weiser, M., & Brown, J. (1995, Dezembro). Designing Calm Technology. *PowerGrid Journal*, *1*(1). (Sem número de página pois o artigo foi aceito em um periódico eletrônico)

Yang, J., Papazoglou, M. P., & Heuvel, W.-J. V. den. (2002, Fevereiro). Tackling the Challenges of Service Composition in E-Marketplaces. In *12th International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems.* San Jose, CA, EUA.

Zhu, F., Mutka, M. W., & Ni, L. M. (2005, Outubro). Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, *4*(4), 81–90.

Esta dissertação foi composta na tipologia
Times New Roman, em corpo 10-25, e impressa
em papel Sulfite 75g/m^2.